FlexC Language Reference

5.9.14

Total Spectrum Software

07/22/2022

Contents

Flex C

Introduction

FlexC is the C dialect implemented by the FlexProp compiler. It eventually will implement the C99 standard with some C++ extensions. The "natve" front end is `flexcc`, although the other FlexProp tools (like `flexspin`) can also compile C files.

flexcc recognizes the language by the extension of the file being compiled. If a file ends in `.c` it is treated as a C file. If a file ends in `.cpp`, `.cc`, or `.cxx` then it is treated as a C++ file; this enables a few keywords not available in C, but otherwise is very similar to C mode (FlexC is not a fully featured C++ compiler).

This document assumes that you are familiar with programming in C and with the Parallax Propeller chip. It mostly covers the differences between standard C and FlexC.

DEVELOPMENT STATUS

The C compiler is mostly implemented and could probably be considered "beta" software now, but there are a few missing features.

Known Bugs

There are several known bugs and deviations from the C99 standard:

Name Spaces

The namespaces for types and variable names are not separated as they should be, so some C code that uses the same identifiers for types and variables or struct members may not work properly.

Doubles

The `double` type is implemented as a 32 bit IEEE single precision float (the same as `float`). This doesn't meet the requirements in the C99 and later standards for the range available for double.

long long

The 64 bit integer type ("long long") is only partially implemented at this time, and does not work properly.

Designated initializers

C99 desginated initializers are supported only in their simplest form, that is, for only one level of initializer. So for example a statement like:

```
struct point c = { .x = 1, .y = 2 };
```

will work, but a designated initializer for a sub structure field like:

```
struct person p = { .address.streetnum = 10 };
```

will not work: the double levels of ".address.streetnum" will fail.

Variable length arrays

Variable length arrays are not supported. A work-around is to use the `__builtin_alloca()` function to allocate memory on the stack.

Preprocessor

Flex C uses the open source mcpp preprocessor (originally from mcpp.sourceforge.net), which is a very well respected and standards compliant preprocessor.

Predefined symbols

Symbol	When Defined
`__propeller__`	always defined to 1 (for P1) or 2 (for P2)
`__FLEXC__`	always defined to the flexspin/flexcc major version number
`__FLEXSPIN__`	always defined to the flexspin/flexcc major version number
`__P2__`	only defined if compiling for Propeller 2 (obsolete)
`__propeller2__`	only defined if compiling for Propeller 2
`__ILP32__`	always defined; some programs use this to determine pointer size
`__HAVE_FCACHE__`	if the FCACHE optimization is enabled
`__VERSION__`	defined to a string containing the full flexspin version
`__OUTPUT_ASM__`	if PASM code is being generated
`__OUTPUT_BYTECODE__`	if bytecode is being generated

Symbol	When Defined
`__OUTPUT_C__`	if C code is being generated
`__OUTPUT_CPP__`	if C++ code is being generated

Runtime Environment

P1 Clock Frequency

In C code, the P1 clock frequency defaults to 80 MHz, assuming a 5 MHz crystal and `xtal1 + pll16x` clock mode. This is a common configuration. You may override it with the builtin `clkset` function, which works the same as the Spin `clkset` builtin.

P2 Clock Frequency

The P2 has a default clock frequency of 160 MHz in C mode. You may set up a different frequency with the loader (loadp2), but it is probably best to explicitly set it, either with a `_clkfreq` enum, or by using `_clkset(mode, freq)`. This is similar to the P1 `clkset` except that `mode` is a P2 `HUBSET` mode.

_clkfreq

If an enumeration constant named `_clkfreq` is defined in the top level file (near the `main` function) then its value is used for the clock frequency instead of 160 MHz. For example:

```
enum { _clkfreq = 297000000 };
```

may be used to specify 297 MHz.

_clkset

Header files `sys/p2es_clock.h` and `sys/p2d2_clock.h` are provided for convenience in calculating a mode. To use these, define the macro P2_TARGET_MHZ before including the appropriate header file for your board. The header will calculate and define macros `_SETFREQ` (containing the mode bits) and `_CLOCKFREQ` (containing the frequency; this should normally be `P2_TARGET_MHZ * 1000000`). So for example to set the frequency to 180 MHz you would do:

```
#define P2_TARGET_MHZ 180
#include <sys/p2es_clock.h>
...
_clkset(_SETFREQ, _CLOCKFREQ);
```

The macros `_SETFREQ` and `_CLOCKFREQ` are not special in any way, and this whole mechanism is just provided as a convenience. You may completely ignore it and calculate the mode bits and frequency setting to pass to `_clkset` yourself.

Extensions to C

Inline Assembly (C Style)

The inline assembly syntax is similar to that of MSVC. Inline assembly blocks are marked with the keyword `__asm`. For example, a function to get the current cog id could be written as:

```
int getcogid() {
   int x;
   __asm {
      cogid x
   };
   return x;
}
```

The `__asm` keyword must be followed by a `{` or else `const` (or `volatile`) and then a `{`; everything between that and the next `}` is taken to be assembly code. `__asm volatile` suppresses optimization of the assembly code (see below) and forces the code to be placed into FCACHE memory. `__asm const` is similar, but does not force the code into FCACHE (it will execute from HUB as usual).

For inline assembly inside a function, any instructions may be used, but the only legal operands are integer constants (preceded by `#`) and local variables, including parameters, of the function which contains the inline assembly. Labels may be defined, and may be used as the target for `goto` elsewhere in the function.

Some conditional execution directives (like `if_c_and_z`) are not accepted in inline assembly. In general, inline assembly is restricted, and is no substitute for full assembly in top level `__pasm` blocks.

Inline assembly inside a function is normally optimized along with the generated code; this produces opportunities to improve the generated code based on knowledge of the assembly. This may be suppressed by using `__asm const` (or `__asm volatile`) instead of `__asm`. Generally this will hurt the output code, but may be necessary if there is inline assembly with very sensitive timing.

Inline assembly may also appear outside of any function. In this case the inline assembly block is similar to a Spin `DAT` section, and creates a global block of code and/or data.

The syntax of expressions inside inline assembly is the same as that of C, and only C style constants may be used (so write `0xff` instead of `$ff`). Comments must be made in C style, not Spin style (so use `//` to begin a comment that extends to the end of line).

Inline Assembly (Spin Style)

Because much existing assembly code is written in the Spin language, FlexC supports inline assembly that (mostly) uses the Spin rules for expression evaluation and comments. These blocks are like C style inline assembly, but start with the keyword `__pasm` instead of `__asm`. Inside `__pasm` blocks comments start with a single quote, and expressions are evaluated as they are in Spin.

`__pasm` blocks may only appear at top level (outside of any function). Inside a function only `__asm` blocks are supported, for now. Also note that the Spin language compatibility inside `__pasm` blocks is still a work in progress, and there are probably many missing pieces.

Note that FlexC supports calling Spin methods directly, so to adapt existing Spin code it may be easier to just include the Spin object with `struct __using` (see below).

Using Inline Assembly in Macros

Note that the C preprocessor replaces all newlines in macros with spaces. This makes it impossible to support the traditional assembly language layout with newlines at the end of each line. To support the use of assembly in macros, you may use a semicolon character `;` to mark the end of line. This also allows you to place multiple assembly instructions on the same text line in code you write.

Example:

```
#define ADD_LONG(desthi, destlo, srchi, srclo)      \
    __asm {                                         \
        add destlo, srclo  wc;                      \
        addx desthi, srchi;                         \
    }
```

Note the necessity of placing a semicolon at the end of each line, even before the final bracket!

Classes

The C `struct` declaration is extended slightly to allow functions (methods) to be declared. All methods must be defined in the struct definition itself; there is at present no way to declare a method outside of the definition.

For example, a simple counter class might be implemented as:

```
typedef struct counter {
  int val;
  void setval(int x) {
      val = x;
  }
  int getval() { return val; }
  int incval() { return ++val; }
} Counter;
```

```
Counter x;
...
x.setval(0);
x.incval();
```

In C++ mode (that is, if the file being compiled has an extension like `.cpp` or `.cc`) then the keyword `class` may be used instead of `struct`. The two are the same, except that the default for `class` is for variables and methods to be private rather than public.

Note that FlexC does not automatically create typedefs for classes, unlike real C++.

External Classes (e.g. Spin Objects)

It is possible to use classes written in other languages. The syntax is similar to the BASIC `class using`, but in C this is written `struct __using`. For example, to use the FullDuplexSerial Spin object you would do:

```
struct __using("FullDuplexSerial.spin") fds;

void main()
{
    fds.start(31, 30, 0, 115200);
    fds.str("hello, world!\r\n");
}
```

This declares a struct `fds` which corresponds to a Spin OBJ, using the code in "FullDuplexSerial.spin". Spin, BASIC, and even C code may be used. In the case of C code, something like:

```
struct __using("myclass.c") myclass;
```

is basically equivalent to:

```
struct {
#include "myclass.c"
} myclass;
```

Note that allowing function definitions inside a struct is an extension to C (it is feature of C++).

If you plan on using a Spin object in many places, it is helpful to create a typedef for it, e.g.:

```
typedef struct __using("FullDuplexSerial.spin") FDS;

FDS ser1, *serptr;
```

Name resolution in Spin and BASIC classes

Because Spin and BASIC are case insensitive languages, their identifiers may be accessed in a case insensitive way (e.g. `x.Vga`, `x.VGA`, and `x.vga` are all equivalent if `x` is a Spin or BASIC class. It is strongly recommended to be consistent though, because this will avoid confusion for readers who are used to C being a case sensitive language.

RESTRICTIONS ON C CLASSES

Class support for C/C++ is very much incomplete, and probably will not work in all cases. In particular, calling functions outside of the class may not work (so simple self contained classes should be fine, but classes which call into other classes may not work properly).

Header file external function definitions

There is no linker as yet, so in order to use standard library functions we use a FlexC specific construct, `__fromfile`. The declaration:

```
size_t strlen(const char *s) __fromfile("libc/string/strlen.c");
```

declares the `strlen` function, and also says that if it is used and no definition is given for it, the file "libc/string/strlen.c" should be added to the build. This file is searched for along the standard include path.

C++ reference parameters

FlexC supports C++ references. These are just like pointers, but are automaticaly dereferenced upon use. They are declared with `&` in place of `*`. For example, a function to swap two integers could be written as:

```
void swap(int &a, int &b)
{
   int t = a;
   a = b;
   b = t;
}
```

and used as

```
   swap(x, y);
```

Internally this is the same as the traditional C:

```
void swap_c(int *a, int *b)
{
   int t = *a;
   *a = *b;
   *b = t;
```

```
}
...
   swap_c(&x, &y);
```

Default values for parameters

Like C++, FlexC allows function parameters to be given default values. For example, if a function is declared as:

```
int incr(int x, int v=1) { return x + v; }
```

then a call `incr(x)` where the second parameter is not given will be compiled as `incr(x, 1)`.

__this and __class

The keywords `__this` and `__class` are allowed in both C and C++ code, and mean the same as `this` and `class` in C++. These keywords are intended for use in C code which wishes to use FlexC's class features.

__catch and __throw

The limited form of exception handling supported by FlexC in C++ mode is also available in C mode, using the `__catch` and `__throw` keywords.

Range cases

In switch statements you may add a range of consecutive cases by putting three dots between them. If the case values are integers you should put spaces around the dots to ensure the parser is not confused. So for example:

```
switch (x) {
  case 1 ... 3: return 1;
}
```

would be the same as

```
switch (x) {
  case 1: case 2: case 3: return 1;
}
```

Statement expressions

A compound statement enclosed in parentheses may appear as an expression. This is a GCC extension which allows loops, switches, and similar features to appear within an expression.

Miscellaneous extensions

The `@` symbol may be used as an addressof operator in place of `&`. This is mainly useful in inline assembly (where it mimics the syntax of PASM/Spin).

Builtin functions

ABS

```
x = __builtin_abs(y)
```

Calculates the absolute value of `y`. This is not like a normal C function in that the result type depends on the input type. If the input is an integer, the result is an integer. If the input is a float, the result is a float.

ALLOCA

```
ptr = __builtin_alloca(size)
```

Allocates `size` bytes of memory on the stack, and returns a pointer to that memory. When the enclosing function returns, the allocated memory will become invalid (so do not attempt to return the result from a function!)

BITREVERSE32

```
x = __builtin_bitreverse32(y)
```

Reverse all 32 bits of the unsigned integer `y` and returns the result. This is the same as `__builtin_propeller_rev(y, 32)` and is provided for compatibility with clang.

BSWAP16

```
x = __builtin_bswap16(y)
```

Swaps the lower two bytes of `y`, clears the upper two bytes, and returns the result.

BSWAP32

```
x = __builtin_bswap32(y)
```

Swaps all four bytes of `y` and returns the result.

CLZ (Count Leading Zeros)

```
x = __builtin_clz(y)
```

Calculates the number of 0 bits at the start of the unsigned integer `y`. The result is between 0 and 32 (inclusive).

COGSTART

Starts a function running in another COG. This builtin is more of a macro than a traditional function, because it does not immediately evaluate its first parameter (which should be a function call); instead, it causes that function call to run in a new COG. For example:

```
static long stack[32];
id = __builtin_cogstart(somefunc(a, b), &stack[0]);
```

runs `somefunc` with parameters `a` and `b` in a new COG, with a stack starting at `&stack[0]` (stacks grow up in FlexC).

The amount of space required for the stack depends on the complexity of the code to run, but must be at least 16 longs (64 bytes).

`__builtin_cogstart` returns the identifier of the new COG, or -1 if no COGs are free.

EXPECT

Indicates the expected value for an expression. `__builtin_expect(x, y)` evaluates `x`, and indicates to the optimizer that the value will normally be `y`. This is provided for GCC compatibility, and the expected value is ignored by FlexC (so `__builtin_expect(x, y)` is treated the same as `(x)`).

FRAC

```
x = __builtin_frac(a, b)
```

Sets `x` to the quotient of `a << 32` and `b`; this is similar to the Spin2 FRAC operator.

MOVBYTS

```
x = __builtin_movbyts(a, m)
```

Uses bits of `m` to select bytes from `a`, similar to the P2 MOVBYTS instruction. The low byte of the result is the byte of `a` selected by the low 2 bits of `m`; the next byte of the result is selected by bits 2-3 or `m`; and so on.

MULH

```
x = __builtin_mulh(a, b)
```

Calculates the upper 32 bits of the 64 bit product of (signed) integers `a` and `b`.

MULUH

```
x = __builtin_muluh(a, b)
```

Calculates the upper 32 bits of the 64 bit product of (unsigned) integers `a` and `b`.

PARITY

```
x = __builtin_parity(y)
```

Returns the parity of the unsigned integer `y` (either 0 or 1).

POPCOUNT

```
x = __builtin_popcount(y)
```

Returns the number of bits set in the unsigned integer `y` (between 0 and 32).

REV

```
x = __builtin_propeller_rev(y, n)
```

Reverses the bits of `y` and then shifts the result right by `32-n` places. This effectively means that the bottom `n` bits of `y` are reversed, and 0 placed in the remaining bits.

SQRT

```
  x = __builtin_sqrt(y)
```

Calculates the square root of `y`. This is not like a normal C function in that the result type depends on the input type. If the input is an integer, the result is an integer. If the input is a float, the result is a float.

Builtin Constants

FlexC has many built-in constants for P1 and P2 hardware access. These constants are inherited from the Spin / Spin2 support, and so they are actually case insensitive. See the Parallax Spin2 "Built-In Symbols" sections for a list of these.

For example, the Spin2 `P_INVERT_A` smart pin symbol is available in C, and may be referred to as `P_INVERT_A`, `p_invert_a`, or `P_Invert_A`. The all-caps form is preferred, as it is most likely to be compatible with other C compilers. These built-in symbols are "weak" and may be overridden by the user, but any such override only affects the exact (case sensitive) version defined by the user. The following program illustrates this:

```
#include <stdio.h>

int main()
{
    int p_invert_a = 0xdeadbeef; // overrides the built-in symbol
```

```
    printf("P_INVERT_A = 0x%08x\n", P_INVERT_A); // prints 0x80000000
    printf("P_Invert_A = 0x%08x\n", P_Invert_A); // prints 0x80000000
    printf("p_invert_a = 0x%08x\n", p_invert_a); // prints 0xdeadbeef
}
```

propeller.h

Propeller 1 specific functions are contained in the header file `propeller.h`. Many of these work on P2 as well.

The propeller.h header file is not standardized. FlexC's library mostly follows the PropGCC propeller.h.

cogstart

```
int cogstart(void (*func)(void *), void *arg, void *stack, size_tstacksize);
```

Starts the C function `func` with argument `arg` in another COG, using `stack` as its stack. Returns the COG started, or -1 on failure.

getcnt

```
unsigned getcnt();
```

Fetches the current value of the CNT register. Using this instead of directly using CNT will make your code portable to P2.

getpin

```
int getpin(int pin);
```

Returns the current state of input pin `pin`, either 0 or 1.

setpin

```
int setpin(int pin, int val);
```

Sets the output pin `pin` to 0 if `val` is 0, or 1 otherwise.

togglepin

```
void togglepin(int pin, int val);
```

Inverts the output pin `pin`.

propeller2.h

Propeller 2 specific functions are contained in the header file `propeller2.h`. This file is usually quite portable among C compilers.

Clock and time control

__clkset

```
void _clkset(uint32_t clkmode, uint32_t clkfreq);
```

Sets the system clock to a new frequency `clkfreq`, using clock mode bits `clkmode`. Note that correct setting of the clock depends on the actual crystal frequency of the hardware the program is being run on. No validation is performed by `_clkset`. The user is responsible for ensuring that `clkmode` is a valid mode and that `clkfreq` does in fact correspond to `clkmode` on this system.

__cnt

```
uint32_t _cnt(void);
```

Returns the low 32 bits of the system clock counter.

__cnth

```
uint32_t _cnth(void);
```

Returns the upper 32 bits of the system clock counter.

__getsec

```
uint32_t _getsec(void);
```

Gets the seconds elapsed on the system timer. On the P1 this will wrap around after about 54 seconds. On the P2 a 64 bit counter is used, so it will wrap around only after many years.

__getms

```
uint32_t _getms(void);
```

Gets the time elapsed on the system timer in milliseconds. On the P1 this will wrap around after about 54 seconds. On the P2 a 64 bit counter is used for the system timer, so it will wrap around only after about 50 days.

__getus

```
uint32_t _getus(void);
```

Gets the time elapsed on the system timer in microseconds. On the P1 this will wrap around after about 54 seconds.

__waitms

```
void _waitms(uint32_t delay);
```

Waits for `delay` milliseconds.

__waitus

```
void _waitus(uint32_t delay);
```

Waits for `delay` microseconds.

__waitx

```
void _waitx(uint32_t delay);
```

Waits for `delay` clock cycles.

COG control

__cogatn

```
void _cogatn(uint32_t mask)
```

Raises the ATN signal on the COGs specified by `mask`. `mask` is a bitmask with 1 set in position `n` if COG `n` should be signalled, So for example, to signal COGs 2 and 3 you would say `_cogatn((1<<2)|(1<<3))`.

__cogchk

```
int _cogchk(int n);
```

Checks to see if cog `n` is running. Returns nonzero if it is, 0 if it is not.

__cogid

```
int _cogid();
```

Returns the ID of the currently running COG.

__coginit

```
int _coginit(int cogid, void *cogpgm, void *ptra)
```

Starts PASM code in another COG. `cogid` is the ID of the COG to start, or `ANY_COG` if a new one should be allocated. `cogpgm` points to the compiled PASM code to start, and `ptra` is a value to be placed in the new COG's `ptra` register. Returns the ID of the new COG, or -1 on failure.

Some compilers (e.g. Catalina) use `_cogstart_PASM` for this.

__cogstart__C

```
int _cogstart_C(void (*func)(void *), void *arg, void *stack_base, uint32_t stack_size
```

Starts C code in another COG. `func` is the address of a C function which expects one argument, and which will run in another COG (cpu core). `arg` is the argument to pass to the function for this invocation. `stack_base` is the base address of a block of memory to use for the stack. `stack_size` is the size in bytes of the memory.

__cogstop

`void _cogstop(int cogid);`

Stops the given COG.

__pollatn

`int _pollatn(void);`

Checks to see if ATN has been signalled for this COG by `_cogatn`. Returns nonzero if it has, 0 if not.

__reboot

`void _reboot(void);`

Reboots the P2. Needless to say, this function never returns.

__waitatn

`int _waitatn(void);`

Waits for an ATN signal to be sent by `_cogatn`. Doesn't really return any useful information at this time.

Locks

__locknew

`int _locknew(void);`

Allocate a new lock and return its value. Returns -1 if no locks are avilable.

__lockret

`void _lockret(int lockid);`

Frees a lock previously allocated by `_locknew`.

__locktry

`int _locktry(int lockid);`

Attempts to lock the lock with id `lockid`. Returns 0 on failure, non-zero on success.

__lockrel

`int _lockrel(int lockid);`

Releases a lock held due to a successful call to `_locktry`.

Math Functions

__clz

`int _clz(uint32_t val);`

Returns the number of leading zeros in `val`, e.g. 4 for 0x0fffffff, or 32 if `val` is 0.

__encod

`int _encod(uint32_t val);`

Finds 1 + the floor of log base 2 of `val`. Similar to the Spin2 `ENCOD` operator.

__isqrt

`uint32_t _isqrt(uint32_t val);`

Finds the integer square root of a 32 bit unsigned number.

__rev

`uint32_t _rev(uint32_t val);`

Returns `val` with all of its bits reversed.

__rnd

`uint32_t _rnd(void);`

Returns a 32 bit unsigned random number. On P2 this uses the built in hardware random number instruction; on P1 it uses a pseudo-random number generator.

__rotxy

`cartesian_t _rotxy(cartesian_t coord, uint32_t angle);`

`cartesian_t` is a structure containing the `x` and `y` coordinates of a point. `_rotxy` rotates this point by the given `angle`, which is specified as a 0.32 bit fraction of a full circle (so 90 degrees corresponds to 0x40000000).

__polxy

`cartesian_t _polxy(polar_t coord);`

Converts polar coordinates into Cartesian (xy) coordinates. `coord` is a structure with 2 unsigned integers: `r` (which gives the radius of the point) and `t` (which gives the angle as a 0.32 fraction of a whole circle.

_xypol

```
polar_t     _xypol(cartesian_t coord);
```

Converts Cartesian (xy) coordinates into polar coordinates. `coord` has two signed 32 bit integer fields, `x` and `y`. The result is a structure with 2 unsigned integers: `r` (which gives the radius of the point) and `t` (which gives the angle as a 0.32 fraction of a whole circle.

Regular Pin I/O

_pinf

```
void      _pinf(int pin);
```

Forces pin `pin` to float low.

_pinl

```
void      _pinl(int pin);
```

Makes pin `pin` an output and forces it low.

_pinh

```
void      _pinh(int pin);
```

Makes pin `pin` an output and forces it high.

_pinnot

```
void      _pinnot(int pin);
```

Makes pin `pin` an output and inverts it.

_pinrnd

```
void      _pinrnd(int pin);
```

Makes pin `pin` an output and sets it to a random bit value.

_pinr

```
int       _pinr(int pin);
```

Makes pin `pin` an input and returns its current value (0 or 1). Only works for single pins. For multiple pins, use `_pinread`.

_pinread

```
int _pinread(int pins);
```

Read one or more pins. `pins` can be a single pin from 0-63, or it can be a group of `num` pins starting at `base` specified as `base + ((num-1)<<6)`. For reading a single pin, the `_pinr` function is more efficient.

_pinw

```
void      _pinw(int pin, int val);
```

Makes pin `pin` an output and writes `val` to it. `val` should be only 0 or 1; results for other values are undefined (that is, only a single pin is supported). For writing multiple pins, see `_pinwrite`.

_pinwrite

```
void      _pinwrite(int pins, int val);
```

`pins` can be a single pin from 0-63, or it can be a group of `num` pins starting at `base` specified as `base + ((num-1)<<6)`. For writing a single pin, `_pinw` is more efficient.

Smart Pin controls

_akpin

```
void _akpin(int pin);
```

Acknowledge input from the given smart pin. Necessary only if you use `rqpin`.

_rdpin

```
uint32_t _rdpin(int pin);
```

Reads data from the smart pin and acknowledges the input. The value returned is the 32 bit smart pin data value.

_rqpin

```
uint32_t _rqpin(int pin);
```

Reads data from the smart pin and *without* acknowledging the input. The value returned is the 32 bit smart pin data value. `_akpin` must be called later in order to allow further smart pin input.

_wrpin

```
void _wrpin(int pin, uint32_t val);
```

Write `val` to the smart pin mode of pin `pin`.

_wxpin

```
void _wxpin(int pin, uint32_t val);
```

Write `val` to the smart pin X register of pin `pin`.

_wypin

```
void _wypin(int pin, uint32_t val);
```

Write `val` to the smart pin Y register of pin `pin`.

_pinstart

```
void _pinstart(int pin, uint32_t mode, uint32_t xval, uint32_t yval);
```

Activate a smart pin. `mode` is the smart pin mode (written with `_wrpin`), and `xval` and `yval` are the values for the smart pin X and Y registers.

_pinclear

```
void _pinclear(int pin);
```

Turn off a smart pin (writes 0 to mode).

Disk I/O routines (P2 Only)

On the P2 there are some methods available for disk I/O. The `mount` call must be made before any other calls.

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.
- `_vfs_open_sdcardx(clk, ss, di, do)` is the same, but allows explicit specifications of the pins to use.

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

Stdio

After mounting a file system, the standard FILE functions like `fopen`, `fprintf`, `fgets` and so on are available and usable.

Posix file functions

`int remove(const char *path)`

Removes the regular file specified by `path`. Returns non-zero if the removal failed for some reason.

Posix directory functions

A number of standard POSIX directory functions are available, including:

`int mkdir(const char *path)`

Creates a new directory named `path`. Returns 0 on success, non-zero on error (in which case errno is set to the specific error).

`int rmdir(const char *path)`

Removes the directory specified by `path`. Returns 0 on success, non-zero on failure (in the latter case sets `errno` to the precise error.

`int chdir(const char *path)`

Sets the current directory to `path`. Returns 0 on success, non-zero on failure.

`char *getcwd(char *buf, size_t size)`

Copies the current directory into `buf`, which must have at least `size` bytes available. Returns `buf`, or NULL if `buf` is not larget enough to hold the directory.

`DIR *opendir(const char *path)`

Opens `path` for reading with `readdir`. Returns NULL on error, otherwise a handle to use with `readdir`.

`int closedir(DIR *dir)`

Closes directory previously opened with `opendir`.

`struct dirent *readdir(DIR *dir)`

Reads the next directory entry.

Further information

For further information see the `File I/O` section of the general documentation.

Serial I/O functions

ioctl

```
#include <unistd.h>
#include <sys/ioctl.h>
...
r = ioctl(fd, TTYIOCTLGETFLAGS, &flags); // get current flags
r = ioctl(fd, TTYIOCTLSETFLAGS, &flags); // set current flags
```

These ioctls get and set flags controlling the operation of terminals (e.g. the default serial port). They may be used to turn on or off echoing of input characters, and/or to control mapping of carriage return to newline. The flags are made of a mask of the bits `TTY_FLAG_ECHO` (to turn echo on) and `TTY_FLAG_CRNL` (to turn on mapping of CR to LF).

__rxraw

Reads a single character from the default serial port, with no processing (no echoing and no CR/LF conversion). Takes a single parameter giving a timeout in milliseconds (with 0 meaning "no timeout, wait forever"). If a timeout occurs before the character is read, returns -1, otherwise returns the ASCII character read.

__txraw

Sends a single character out over the default serial port.

See Also

See also the general compiler documentation for more details on serial I/O functions.

Time Functions

The standard C99 library functions like `asctime`, `localtime`, `mktime`, and `strftime` are all available. The `time_t` type is an unsigned 32 bit integer, counting the number of non-leap seconds since midnight Jan. 1, 1970. Note that most P2 boards do not have a real time clock built in, so the time returned will not be accurate unless it is first set by `settimeofday` (see below). Also note that all of the time functions make use of an internal counter which is based on the system frequency, and hence must be called at least once every 54 seconds or so (on P1) in order to avoid losing time.

The POSIX functions `settimeofday` and `gettimeofday` are also available, in the header file `<sys/time.h>`. These use a `struct timeval` structure giving the number of (non leap) seconds and microseconds elapsed since midnight Jan.

1, 1970. settimeofday is the best way to interface with a hardware RTC. To use it, read the time from the hardware RTC periodically (e.g. once every 30 seconds) and call settimeofday to update the internal time based on it. See the example below.

Sample time program

Here is a simple example of setting the clock and then reading it repeatedly to display the time:

```
//
// simple clock program
// shows how to set the time to a specific date/time
// and then display it
//
#include <stdio.h>
#include <sys/time.h>

int main()
{
    struct timeval tv;
    struct tm tm_now;
    char dispbuf[40];

    // set the time to 2020-October-17, 1:30 pm
    // set up struct tm structure
    memset(&tm_now, 0, sizeof(tm_now));
    tm_now.tm_sec = 0;
    tm_now.tm_min = 30;
    tm_now.tm_hour = 13; // 1pm
    tm_now.tm_mon = 10 - 1;  // month is offset by 1
    tm_now.tm_mday = 17;
    tm_now.tm_year = 2020 - 1900;  // year is offset relative to 1900

    // convert to seconds + microseconds
    tv.tv_sec = mktime(&tm_now); // set seconds
    tv.tv_usec = 0;              // no microsecond offset

    // and set the time
    settimeofday(&tv, 0);

    // now continuously display the time
    for(;;) {
        // get current time
        gettimeofday(&tv, 0);
        // get an ASCII version of the time
```

```
        // uses the standard C library strftime function to format the time
        strftime(dispbuf, sizeof(dispbuf), "%a %b %d %H:%M:%S %Y", localtime(&tv.tv_sec));
        // print it; use carriage return but no linefeed so we keep writing on the same line
        printf("%s\r", dispbuf);
    }
}
```