featherlib Documentation

Release 0.2.2

K. Isom

Apr 10, 2019

Contents:

CHAPTER 1

Introduction

featherlib is a set of tools for quickly building programs on the Adafruit Feather ecosystem without sacrificing readability and modularity. It's the same sort of hardware abstraction layer (or HAL) that I find myself either copying over or reimplementing in new projects, so I decided to package it up as a PlatformIO library.

One of the examples that I use regularly is the program for setting the RTC in an Adalogger using the time from an Ultimate GPS wing:

```
// rtcgps is a small sketch that sets an Adalogger's RTC to the
// current GPS time. It uses a Feather M0, the Ultimate GPS
// Featherwing, and the Adalogger Featherwing.
#include <Arduino.h>
#include <RTClib.h>

#include <feather/feather.h>
#include <feather/scheduling.h>
#include <feather/wing/adalogger.h>
#include <feather/wing/gps.h>


#if defined(FEATHER_M0)
FeatherM0        board(INPUT, A1);
#elif defined(FEATHER_M4)
FeatherM4        board;
#else
#error Unknown board.
#endif


// The default GPS constructor uses Serial1 for the
// connection.
GPS              gps;

// NB: setting the Adalogger's CS pin to 0 disables the SD card,
// which isn't used in the sketch and therefore doesn't require an
```

(continues on next page)

(continued from previous page)

```
// SD card to be inserted.
Adalogger        logger(0);


void
setup()
{
        // Start the serial port at 9600 baud but don't wait for a
        // serial connection to continue booting.
        board.setup(9600, false);

        // Registering wings allows them to be set up in one pass and
        // allows any update tasks to be started later on.
        registerWing(&gps);
        registerWing(&logger);

        if (!initialiseWings()) {
                // If a wing fails to initialise, a message will be
                // printed to serial.
                while (true) ;
        }

        // This starts a background thread that runs the update tasks
        // for the featherwings. For example, in this sketch, the GPS
        // needs to be updated in the background. Another scheduler
        // can be used that calls `runWings`, too.
        scheduleWingTasks();
}


void
loop()
{
        // rtcSet will be set to true when the GPS is used to set the
        // RTC.
        static bool        rtcSet = false;
        DateTime           dateTime;

        // when rtcSet is true, the program will stop.
        while (rtcSet) ;

        if (!gps.getDateTime(dateTime)) {
                return;
        }

        if (!logger.adjustRTC(dateTime)) {
                Serial.println("Failed to adjust the RTC.");
                return;
        }

        rtcSet = true;
        Serial.println("RTC is set; halting.");
}
```

I find this is relatively easy to read; organising the functionality under a wing is debatable (and arguably makes this not a true HAL) but it seems to be working out well for the projects I've been using it in.

1.1 Overhead

As a test, I've compiled a basic Arduino sketch for the Feather M0:

```
#include <Arduino.h>


void
setup()
{
	Serial.begin(9600);
	while (!Serial) ;
	Serial.println("boot OK");
}


void
loop()
{

}
```

Building this with PlatformIO shows the following sizes:

```
Building .pioenvs/adafruit_feather_m0/firmware.bin
Memory Usage -> http://bit.ly/pio-memory-usage
DATA:    [=         ]   8.0% (used 2620 bytes from 32768 bytes)
PROGRAM: [          ]   4.2% (used 10992 bytes from 262144 bytes)
```

and the equivalent using the featherlib library:

```
#include <Arduino.h>
#include <feather/feather.h>


FeatherM0	board;


void
setup()
{
	board.setup(9600, true);

	Serial.println("BOOT OK");
}


void
loop()
{

}
```

yields the following sizes:

```
Building .pioenvs/adafruit_feather_m0/firmware.bin
Memory Usage -> http://bit.ly/pio-memory-usage
```

(continues on next page)

(continued from previous page)

```
DATA:    [=         ]   8.1% (used 2648 bytes from 32768 bytes)
PROGRAM: [=         ]   5.2% (used 13568 bytes from 262144 bytes)
```

The additional program space is taken up by the random number seeding. It's worse in this case because a fair amount of additional setup is done, but once more peripherals are added, the tradeoff is generally useful to me.

As additional examples for the Feather M0:

Example	Data (bytes)	Program (bytes)	Components (plus Feather)
calamity	3496 (10.7%)	24784 (9.5%)	OLED
rtcgps	4844 (14.8%)	48576 (18.5%)	Adalogger, GPS
loraspy	5216 (15.9%)	48336 (18.4%)	Adalogger, OLED, RFM95, Trigger
lorabcn	3832 (11.7%)	35136 (13.8%)	RFM95, Trigger

CHAPTER 2

Changelog

This is a list of the release versions of this library.

2.1 v0.4.1 - 2019-04-10

- Add missing namespace on thread scheduling in the wing helpers.

2.2 v0.4.0 - 2019-03-08

- Remove scheduling functions as part of cross-platform effort.
- Add more examples to Travis.

2.3 v0.3.4 - 2019-03-05

- Fix trigger logic.
- Add test target to Makefile.

2.4 v0.3.3 - 2019-03-04

- Convert GPS results from DDMM.SSSS to decimal.
- Support enable, disable, and reset on the RFM95.

2.5 v0.3.2 - 2019-02-28

- Add Trigger class.
- Add more examples.

2.6 v0.3.1 - 2019-02-28

- Fix RFM95 constructor.

2.7 v0.3.0 - 2019-02-28

- Fix subclassing issue with RFM95.
- Support formatting a DateTime object directly.

2.8 v0.2.5 - 2019-02-28

- Fix Travis CI build.

2.9 v0.2.4 - 2019-02-28

- Switch to fork of RadioHead library that supports the M4 boards.

2.10 v0.2.3 - 2019-02-28

- The OLED now supports disabling buttons.
- The GPS supports overriding the default mode, update frequency, and baudrate.
- Add docs.

2.11 v0.2.2 - 2019-02-26

- Fix RadioHead dependency name.
- Remove unused variable from the Feather M4 source.

2.12 v0.2.1 - 2019-02-26

- Adalogger supports an RTC-only mode.
- Add startThread function so users don't have to think about schedulers if it's not a concern.

2.13 v0.2.0 - 2019-02-26

- Add support for RFM95 LoRa.
- Rename AdaLogger class to Adalogger.

v0.1.0 - 2019-02-25

- Initial release.
- Supported Feathers: M0 (tested against basic and LoRa feathers), M4.
- Supported Featherwings: Adalogger, OLED, Ultimate GPS.

CHAPTER 3

Organisation

The code is roughly organised into three pieces:

- boards, which are the computing base (e.g. the Feather M0),
- featherwings and peripherals, and
- everything else - scheduling, etc.

CHAPTER 4

Boards

Support for a board is loaded by including `feather/feather.h` - this will pull in support for the appropriate board. The boards that are currently supported are

- Feather M0-based boards; this has been tested with the Feather M0 basic and the Feather M0 with RFM95 LoRa radio.
- The Feather M4 Express.

It may work on other M0 or M4-based boards, but these haven't been tested.

4.1 The `Board` abstract base class

The `Board` class defines a few virtual methods common to all Feathers:

- `double voltage()` returns the current battery voltage as read from the onboard voltage divider.
- `setup(int baudrate, bool wait)` starts the serial console at the given baudrate; if `wait` is `true`, it will wait for a serial connection before continuing with the boot process. It will also load support for the `%f` verb in `printf`, and seed the random number generator.
- `uint32_t random()` returns a random number; for boards that support a true random number generator, it will be a cryptographically valid random number generator; otherwise, it's a best-effort random number generated by repeatedly sampling the unused analog pin.
- `seed` uses a random number from `Board::random` to seed the Arduino random number generator.

4.2 Feather M0

The `FeatherM0` class is instantiated using one of three constructors:

- The default constructor, `FeatherM0()`, is a wrapper for `FeatherM0(INPUT, UNUSED_ANALOG)`.
- `FeatherM0(int pin9Mode)` is a wrapper for `FeatherM0(pin9Mode, UNUSED_ANALOG)`.

- `FeatherM0(int pin9Mode, int unusedAnalog)` explicitly defines the pin 9 mode and the unused analog port.

The Feather M0's voltage divider is on pin 9; in order to be read, it has to be put in the `INPUT` mode. Afterwards, it will be reset to whatever the `pin9Mode` is. For example, when using the OLED featherwing (described later), it should be set to `INPUT_PULLUP`.

The unused analog pin is used for the random number generator as described above.

4.3 Feather M4

The `FeatherM4` class is instantiated using a default constructor. It has a true random number generator that is used for the seeding process and for returning random numbers.

CHAPTER 5

Featherwings and Peripherals

Support for peripherals (e.g. Featherwings or the RFM95 radio on the radio-enabled Feathers) is done using the appropriate header under `feather/wing`. Generally, there are four steps to setting up wings:

- Define all the wings as global variables to make them available throughout the code.
- Register the wings using `registerWing`.
- Call `initialiseWings` to run the wings' setup functions. If any of these fails, they will print a message about the fault to the serial console and booting will halt.
- Regularly run `runWings`; support for the Arduino SAMD scheduler is provided using `scheduleWingTasks`.

5.1 The `FeatherWing` base class

Every supported peripheral is an instance of the `FeatherWing` class. They provide three functions:

- `bool setup()` runs any setup tasks; if setup fails, this will return false and a message about the fault is printed to the serial console.
- `void task()` runs regular update tasks. For example, a GPS needs to regularly update itself to check for new data. These are written with the intent that they can be used in a cooperative scheduler; they won't take over and block execution.
- `const char *name()` returns the name of the FeatherWing.

The `void registerWing(FeatherWing &)` function, included in `feather/wing/wing.h`, will add the FeatherWing to the global registry. This registry is used for setting up the wings later on and for running update tasks.

Once the wings are all registered, `bool initialiseWings()` should be called to run the setup function on all the wings, called in the order they are registered. If any of the setup tasks fails, this will return false. The intent is to call something like the following in the `setup` function:

```
if (!initialiseWings()) {
        Serial.println("BOOT FAILED");
        while (true) ;
}
```

Finally, near the end of the `setup` function, `void scheduleWingTasks()` should be called - this will use the Arduino scheduler to run the wing update tasks in the background. Alternately, the function `runWings()` can be called regularly, e.g. in the main loop so long as there aren't long delays. It may also be used with another scheduler or task management system; wing tasks are designed to be cooperative and each run of the function will run through the update tasks once. For schedulers that don't treat tasks as `loop` equivalents, a wrapper function should be used, such as:

```
void
wingThread()
{
        while (true) {
                runWings();
                yield();
        }
}
```

5.2 The `Clock` abstract base class

Another base class is implemented in `feather/wing/wing.h` is `Clock`, which is meant to be used in defining realtime clocks (RTCs). A `Clock` provides two functions:

- `bool isClockReady()` should return true if the clock has a valid time.
- `bool getDateTime(DateTime &dateTime)` will return false if the clock isn't ready or if an error getting the time occurs. Otherwise, the `DateTime` instance (this type is defined in RTClib) will be filled in with the current time from the `Clock`.

A helper function is also provided in `feather/wing/wing.h` for use with a `Clock`: `bool clockFormatTime(Clock &clock, char *buf)` wraps `getDateTime`, and if successful, fills `buf` with the time formatted as YYYY-MM-DD hh:mm:ss. The buffer must be large enough to support this, which is a minimum of 19 bytes.

CHAPTER 6

Adalogger

The Adalogger is a Featherwing with a PCF8523 RTC and a microSD card slot.

- Header: `feather/wing/adalogger.h`
- Link: Adalogger RTC + SD
- Update task: nothing is done

The `Adalogger` is instantiated with one of two constructors:

- `Adalogger()` is a wrapper for `Adalogger(10)`, and sets up the Featherwing with support for the onboard RTC and SD.
- `Adalogger(uint8_t cs)` sets up the Featherwing with an alternate SD CS pin if the default has been changed. Alternatively, using a CS pin of 0 will disable SD card support.

The `setup` method doesn't check whether the RTC has a valid date and time; this is provided by other functions.

Note: if SD support is enabled, the card must be inserted and ready by the time `setup()` is called. There isn't support for hotswapping SD cards right now.

SD support is provided by the *SdFat* external library; only FAT support is provided at this time.

The `Adalogger` class provides the following methods for interacting with the SD card:

- `File openFile(const char *path, bool write)` opens a file; the `File` type is provided by the SdFat library. You don't need to include anything extra if you're not using the SdFat library anywhere else.
- `bool exists(const char *path)` returns true if the file or directory named by `path` exists on the SD card.
- `bool remove(const char *path)` returns true if the file named by `path` was successfully removed.
- `bool mkdir(const char *path)` returns true if the directory named by `path` was successfully created.

This class is also an instance of the `Clock` class; in addition to the standard `Clock` methods, it has `void adjustRTC(DateTime &dateTime)` to set the date and time in the RTC.

CHAPTER 7

OLED Featherwing

- Header: `feather/wing/oled.h`
- Link: FeatherWing OLED
- Update task: sample the buttons

The `OLED` class is instantiated with one of two constructors:

- The default constructor `OLED()` is a wrapper around `OLED(9, 6, 5)`, and sets up the Featherwing with the default pin mappings.
- The `OLED(uint8_t a, uint8_t b, uint8_t c)` allows overriding the button pin assignments. If a button's pin is set to 0, that button will be disabled.

The `setup` method will do the necessary work to set up the OLED and clear the display; the update `task` will regularly check and update the buttons.

Graphics being a complex thing, this class has a lot of methods that broadly fall into three categories: graphics primitives, text display, and button handling.

7.1 Graphics primitives

Note: `clear` and `show` are the only functions in this group that immediately affect the display. For the sake of efficiency, the other functions write to a backing buffer that is sent to the display when `show` is called. The x values must be less than the constant `OLED::WIDTH` and the y values must be less than the constant `OLED::HEIGHT` - these are currently 128 and 32, respectively.

- `void clear()` will clear both the display and the backing buffer.
- `void pixel(uint16_t x, uint16_t y)` draws a pixel at the x, y coordinates.
- `void clearPixel(uint16_t x, uint16_t y)` unsets the pixel at the x, y coordinates.
- `void circle(uint16_t x, uint16_t y, uint16_t r, bool fill)` draws a circle whose origin is at (x, y) and whose radius is r. If `fill` is true, the circle will be filled in, otherwise it will be just the outline.

- *void line(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1)*‘ draws a line from (*x0*, *y0*) to (*x1*, *y1*).
- `void show()` sends the graphics buffer to the display.

7.2 Text display

For printing text, this display supports three lines of 20 characters. The line is indexed starting from 0. The text display functions take effect immediately; there is no need to call `show` after calling these. Note that using these functions will clear any drawing that has been done. Calling `clear` will also erase the text from the display, but will preserve the text buffers so that the next call to one of these methods will restore any previously printed text.

- `void clearText()` will clear all the text from the display and reset the text buffers.
- `void print(uint8_t line, const char *text)` prints normal text on the normal line.
- `void iprint(uint8_t line, const char *text)` prints inverse text on the line.
- `void clearLine(uint8_t line)` clears the text for the given line.

7.3 Button handling

The OLED Featherwing has three buttons that are, by default, on pins 9, 6, and 5. The three buttons are called A, B, and C in the Adafruit docs, and are indexed starting from 0: button A, the topmost button, is at index 0 and button C, the bottom button, is at index 2. The buttons will be setup in the `INPUT_PULLUP` mode during setup. Note that pin 9 conflicts with the voltage divider on the Feather M0; this is handled by this library so that both can coexist. Buttons can be disabled by explicitly passing a 0 pin value to the full constructor, in which case they won’t be setup in the `INPUT_PULLUP` mode.

- `void sample()` will check the buttons for updates. This is called in the display’s update `task`.
- `void registerCallback(uint8_t button, void (*callback)())` registers a function to be called when a button is pressed.
- `void unregisterCallback(uint8_t button)` clears the callback for the given button.
- `void unregisterAllCallbacks()` clears the callbacks for all buttons.

There is an example of using buttons in `examples/calamity`.

CHAPTER 8

RFM95 support

- Header: `feather/wing/rfm95.h`
- Link (Featherwing): RFM95W 900 MHz Radiofruit
- Link (Feather M0 with RFM95): Adafruit Feather M0 with RFM95 LoRa Radio - 900MHz
- Update task: nothing is done

The `RFM95` class is used both for the Featherwing and the onboard radio. There are a few defines that can be overridden, which should be done in the `platformio.ini` config. These default to valid values for the Feather M0 with RFM95 for use in the US.

- `LORA_FREQ` defaults to `915.0`, which is valid in the US.
- `RFM95_CS` defaults to `8`.
- `RFM95_RST` defaults to `4`.
- `RFM95_INT` (which is the radio interrupt or IRQ pin) defaults to `3`.

The wiring instructions for the Featherwing are on Adafruit's site.

There are two constructors:

- `RFM95()` uses the values from the three `RFM95_` defines above.
- `RFM95(uint8_t cs, uint8_t irq, uint8_t rst)` allows setting the pins explicitly.

The `setup` method will initialise the radio, set it to the appropriate frequency, and set it to maximum transmit power. The transmit power can be set using the `setPower` method, described below.

The class provides the following methods:

- `bool available()` returns true if the radio has received data.
- `void setPower(uint8_t)` changes the transmit power. Valid values are in the range 5 to 23, inclusive, with higher values providing more transmit power. As per the docs, this uses the `PA_BOOST` pin to provide higher transmit power.
- `void transmit(uint8_t *buf, uint8_t len, bool blocking)` sends the message contained in `buf`; if `blocking` is true, the method will block until transmission is complete.

- `bool receive(uint8_t *buf, uint8_t *len, int16_t *rssi)` returns true if a message is available. `buf` should have at least 251 bytes available, which is the maximum message length for an RFM95 message. If `rssi` is not NULL, it will be set to the received signal strength. The message length will be returned via `len`, which must be set to the size of `buf` before being passed to this method.

Finally, three radio control methods are provided:

- `void disable()` pulls the reset pin low.
- `void enable()` pulls the reset pin high.
- `void reset()` performs a manual reset, pulling the reset pin low for 10 ms, then pulling the reset pin high.

CHAPTER 9

Ultimate GPS Featherwing

- Header: `feather/wing/gps.h`
- Link: *Adafruit Ultimate GPS FeatherWing <https://www.adafruit.com/product/3133>*
- Update task: checking the GPS for new data and updating the fix and position data

The GPS Featherwing is a standard serial-based GPS. It is instantiated using one of two constructors:

- `GPS()` will use `Serial1` for communicating with the GPS.
- `GPS(HardwareSerial *)` will use the given hardware serial port.

GPS position data is returned using the following structures:

```
typedef struct {
        uint16_t        year;
        uint8_t         month;
        uint8_t         day;
        uint8_t         hour;
        uint8_t         minute;
        uint8_t         second;
} Time;

typedef struct {
        uint8_t quality;
        uint8_t satellites;
} Fix;

typedef struct {
        float   latitude;
        float   longitude;
        Time    timestamp;
        Fix     fix;
} Position;
```

The `setup` method will set up the serial connection to the GPS and tell it to return the standard position data (aka `RMCGGA`) and to send updates every second. This can be overriden using the `GPS_MODE` and `GPS_UPDATE_FREQ` defines, which should be set in `platformio.ini`. It also expects the GPS to be communicating at a baudrate of 9600; this can be overridden with the `GPS_BAUDRATE` define.

The `GPS` class provides the following methods for working with position data:

- `bool haveFix()` returns true if the GPS has a fix.
- `bool position(Position &pos)` returns true if the GPS has a valid fix and fills in the `Position` struct with the most recent fix data.
- `void dump()` will block and echo data from the GPS serial port to the serial console. This might be useful for debugging GPS issues.

The `GPS` class is also an instance of the `Clock` virtual class, and therefore provides the relevant RTC methods.

Note that the returned GPS coordinates have a precision of six decimal degrees at most. According to *this <https://en.wikipedia.org/wiki/Decimal_degrees>* article, that should be sufficient for fairly precise locations.

CHAPTER 10

Miscellaneous functions

There are few additional utility functions provided that don't fall under direct hardware support, but that I find myself using often.

10.1 Scheduling

Scheduling used to be done in this library, but it has been moved to KASL.

10.2 Triggers

Including `feather/trigger.h` makes the `Trigger` class available. This class is constructed with a millisecond delta, and its `ready` method will return true if at least that long has passed since the last call to `ready`. By default, it will be ready immediately; if a second optional true argument is passed to the constructor, it will require waiting `delta` milliseconds before being ready for the first time.

The `ready` method has two forms:

- `bool ready()` calls `ready(millis())`.
- `bool ready(unsigned long now)` allows the same `millis` value to be reused in multiple places to avoid calling the function multiple times. When the allotted time is up, the trigger will reset to the last update time plus the delta.

Finally, there is a `reset` method:

- `void reset()` calls `reset(millis())`.
- `void reset(unsigned long now)` resets the trigger to fire next in `now` + `delta` milliseconds.

10.3 Util

The `feather/util.h` contains functions that don't really fit elsewhere:

- `void swap_u8(uint8_t &a, uint8_t &b)` will swap its arguments.
- `void swap_ul(unsigned long &a, unsigned long &b)` will swap its arguments.

CHAPTER 11

Links

- Github repo
- PlatformIO library
- Docs
- Feather notes

CHAPTER 12

Indices and tables

- genindex
- modindex
- search