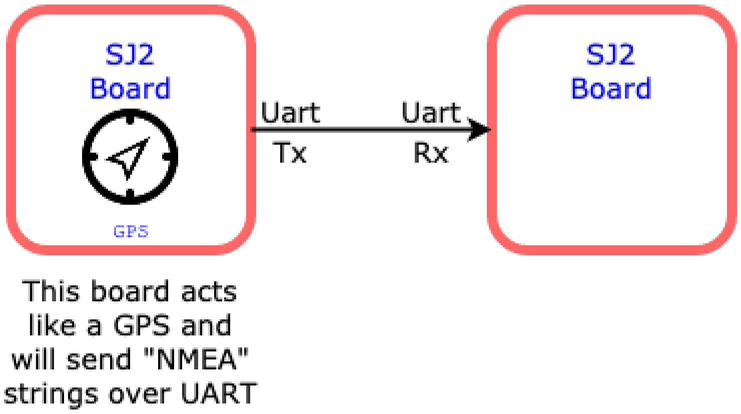
LAB: GPS and UART

Objective

- Use existing drivers to communicate over UART (GPS module will utilize it)
- Design a line buffer library that may be useful with the GPS module
- Reinforce how to design software structured around the periodic callbacks

Background

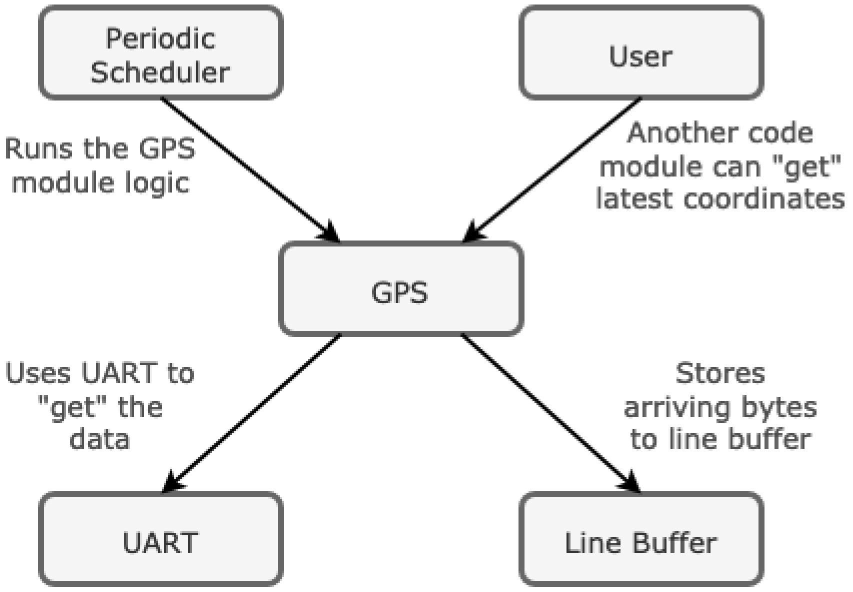
A GPS typically operates by sending "NMEA" strings over UART in plain ASCII text that is readable by humans. Here is a good reference article. What you will do is use one of the SJ2 boards to send "fake" GPS string, and have another board parse the input and extract latitude and longitude.


SJ2 Board
GPS
Uart Tx
Uart Rx
SJ2 Board
This board acts like a GPS and will send "NMEA" strings over UART

Overall Software Design

What we are designing is a GPS module which exposes simple API for the periodic scheduler to run its logic, and another API for a user to query GPS coordinates.

This module internally (at its `gps.c` file) has other module dependencies, but it does not introduce these dependencies to the user and in fact keeps them hidden. **This is useful because any code module that `#includes` the GPS module should not need to know or mock the UART or the line buffer code module.**


Periodic Scheduler
User
Runs the GPS module logic
Another code module can "get" latest coordinates
GPS
Uses UART to "get" the data
Stores arriving bytes to line buffer
UART
Line Buffer

```c
#pragma once

// Notice the simplicity of this module
// This module is easily mockable and provides a very simple API interface
// UART driver and line buffer module will be hidden inside of gps.c

void gps__run_once(void);
float gps__get_latitude(void);
```

Lab

Part 0: Familiarize with MCU Pins

The LPC (SJ2) microcontroller has dedicated pins that can be used for serial communication such as UART. The first thing to do is identify the pins that you will be using (or compromising) for UART communication. Please reference this article.

At this point, put your SJ2 board away, and perform test-driven development of the code modules and we will test it on the board at the last step of this lab.

Part 1: Create `line_buffer` code module

In this part of the lab, you will create a new code module that will remove data from the UART driver, and buffer it inside of this code module. Since this should be a group lab, please **pair program** and **do not work on this code module alone.** Notice the minimal API because according to our tests below, we simply will not need anything further than this.

```c
#pragma once

#include <stdint.h>
#include <stdbool.h>
```

```c

typedef struct {
  void * memory;
  size_t max_size;
  size_t write_index;
} line_buffer_s;

// Initialize *line_buffer_s with the user provided buffer space and size
void line_buffer__init(line_buffer_s *buffer, void *memory, size_t size);

// Adds a byte to the buffer, and returns true if the buffer had enough space to add the by
bool line_buffer__add_byte(line_buffer_s *buffer, char byte);

/**
 * If the line buffer has a complete line, it will remove that contents and save it to "cha
 * @param line_max_size This is the max size of line
 */
bool line_buffer__remove_line(line_buffer_s *buffer, char * line, size_t line_max_size);
```

Here are the unit-tests that are already designed for you. You should use this to ensure that the line buffer code module is working correctly. Feel free to also add more tests to this.

```c
#include "unity.h"

// Include the source we wish to test
#include "line_buffer.h"

static line_buffer_s line_buffer;
static char memory[8];

void setUp(void) { line_buffer__init(&line_buffer, memory, sizeof(memory)); }

void tearDown(void) {}

void test_line_buffer__nominal_case(void) {
  line_buffer__add_byte(&line_buffer, 'a');
  line_buffer__add_byte(&line_buffer, 'b');
  line_buffer__add_byte(&line_buffer, 'c');
  line_buffer__add_byte(&line_buffer, '\n');

  char line[8];
  TEST_ASSERT_TRUE(line_buffer__remove_line(&line_buffer, line, sizeof(line)));
  TEST_ASSERT_EQUAL_STRING(line, "abc");
}

void test_line_buffer__overflow_case(void) {
  // Add chars until full capacity
  for (size_t i = 0; i < sizeof(memory) - 1; i++) {
    TEST_ASSERT_TRUE(line_buffer__add_byte(&line_buffer, 'a' + i));
  }


  // Buffer should be full now
  TEST_ASSERT_FALSE(line_buffer__add_byte(&line_buffer, 'b'));

  // Retreive truncated output
  char line[8];
  TEST_ASSERT_TRUE(line_buffer__remove_line(&line_buffer, line, sizeof(line)));
  TEST_ASSERT_EQUAL_STRING(line, "abcdefg");
}
```

Part 2: Create `gps` code module

The GPS code module will glue the UART driver, and the `line_buffer` module and this will be the single module that needs to be integrated with the periodic callbacks.

The starter code for `gps.h` and `gps.c` is given to you already. This is not to spoil your fun, but to provide a guideline of how the GPS code module should be structured. But we have not spoiled all the fun, since you need to build the unit-test for the GPS module: `test_gps.c`

```c
// gps.h
#pragma once

typedef struct {
  float latitude;
  float longitude;
} gps_coordinates_t;

void gps__init(void);
void gps__run_once(void);

gps_coordinates_t gps__get_coordinates(void);
```

```c
// gps.c
#include "gps.h"

#include "uart.h"
#include "line_buffer.h"

#include "clock.h" // needed for UART initialization

// Change this according to which UART you plan to use
static uart_e gps_uart = UART__1;

// Space for the line buffer, and the line buffer data structure instance
static char line_buffer[256];
static line_buffer_s line;

static gps_coordinates_t parsed_coordinates;

static void gps__absorb_data(void) {
  char byte;
  while (uart__get(gps_uart, &byte, 0)) {
    line_buffer__add_byte(&line, byte);
  }
}

static void gps__handle_line(void) {
  char gps_line[100];
  if (line_buffer__remove_line(&line, gps_line, sizeof(gps_line))) {
    // TODO: Parse the line to store GPS coordinates etc.
    // TODO: parse and store to parsed_coordinates
  }
}

void gps__init(void) {
  line_buffer__init(&line, line_buffer, sizeof(line_buffer));
  uart__init(gps_uart, clock__get_peripheral_clock_hz(), 38400);

  // RX queue should be sized such that can buffer data in UART driver until gps__run_once(
  // Note: Assuming 38400bps, we can get 4 chars per ms, and 40 chars per 10ms (100Hz)
  QueueHandle_t rxq_handle = xQueueCreate(50, sizeof(char));
  QueueHandle_t txq_handle = xQueueCreate(8, sizeof(char));  // We don't send anything to t
  uart__enable_queues(gps_uart, txq_handle, rxq_handle);
}

/// Public functions:
///
void gps__run_once(void) {
  gps__absorb_data();
  gps__handle_line();
}

gps_coordinates_t gps__get_coordinates(void) {
  // TODO return parsed_coordinates
}
```

```c
// TODO:
// test_gps.c
#include "unity.h"

// Mocks
#include "Mockclock.h"
#include "Mockuart.h"

// Use the real implementation (not mocks) for:
#include "line_buffer.h"

// Include the source we wish to test
```

```c
#include "gps.h"

void setUp(void) {}
void tearDown(void) {}

void test_gps_init(void) {}

void test_gps_other_stuff(void) {}
```

Part 3: Integrate and test

Once you have your GPS and line buffer code module fully tested, this part might be the simplest part because your code may simply work the first time (which usually never happens). This is of course only possible becuase you have already unit-tested your code.

```c
void periodic_callbacks__initialize(void) {
  // This method is invoked once when the periodic tasks are created
  gps__init();
}

/**
 * Depending on the size of your UART queues, you can probably
 * run your GPS logic either in 10Hz or 100Hz
 */
void periodic_callbacks__100Hz(uint32_t callback_count) {
  gpio__toggle(board_io__get_led2());
  gps__run_once();
}
```

One assumption is that the second SJ2 board is already interfaced to your primary SJ2 board and is sending fake GPS data:

```c
// @file: fake_gps.c
#include "fake_gps.h" // TODO: You need to create this module, unit-tests for this are opti

#include "uart.h"
#include "uart_printf.h"

#include "clock.h" // needed for UART initialization

// Change this according to which UART you plan to use
static uart_e gps_uart = UART__1;

void gps__init(void) {
  uart__init(gps_uart, clock__get_peripheral_clock_hz(), 38400);

  QueueHandle_t rxq_handle = xQueueCreate(4,   sizeof(char)); // Nothing to receive
  QueueHandle_t txq_handle = xQueueCreate(100, sizeof(char)); // We send a lot of data
  uart__enable_queues(gps_uart, txq_handle, rxq_handle);
```

```
}

/// Send a fake GPS string
/// TODO: You may want to be somewhat random about the coordinates that you send here
void gps__run_once(void) {
  uart_printf(gps_uart, gps_uart, "$GPGGA,230612.015,3907.3815,N,12102.4634,W,0,04,5.7,508
}
```