Keep learning.
It's the Washington Way.

EMBSYS 105
Programming with Embedded & Real-Time Operating Systems

Instructor: Nick Strathy, nstrathy@uw.edu

TA: Gideon Lee, gideonhlee@yahoo.com

Lecture 6 2/10/2020

PROFESSIONAL & CONTINUING EDUCATION

UNIVERSITY of WASHINGTON

Looking ahead

Date	Lecture number	Assignment
1/6	L1	A1 due* before L2
1/13	L2	A2 due before L3
1/20	L3	A3 due before L4
1/27	L4	A4 due before L5
2/3	L5	A5 due before L7, Project due before L10
2/10	L6	
2/17	Holiday – enjoy!	
2/24	L7	
3/2	L8	
3/9	L9	
3/16	L10 – Student presentations	

* Assignments are due Sunday night at 11:59 PM

Previous Lecture (L5) Overview

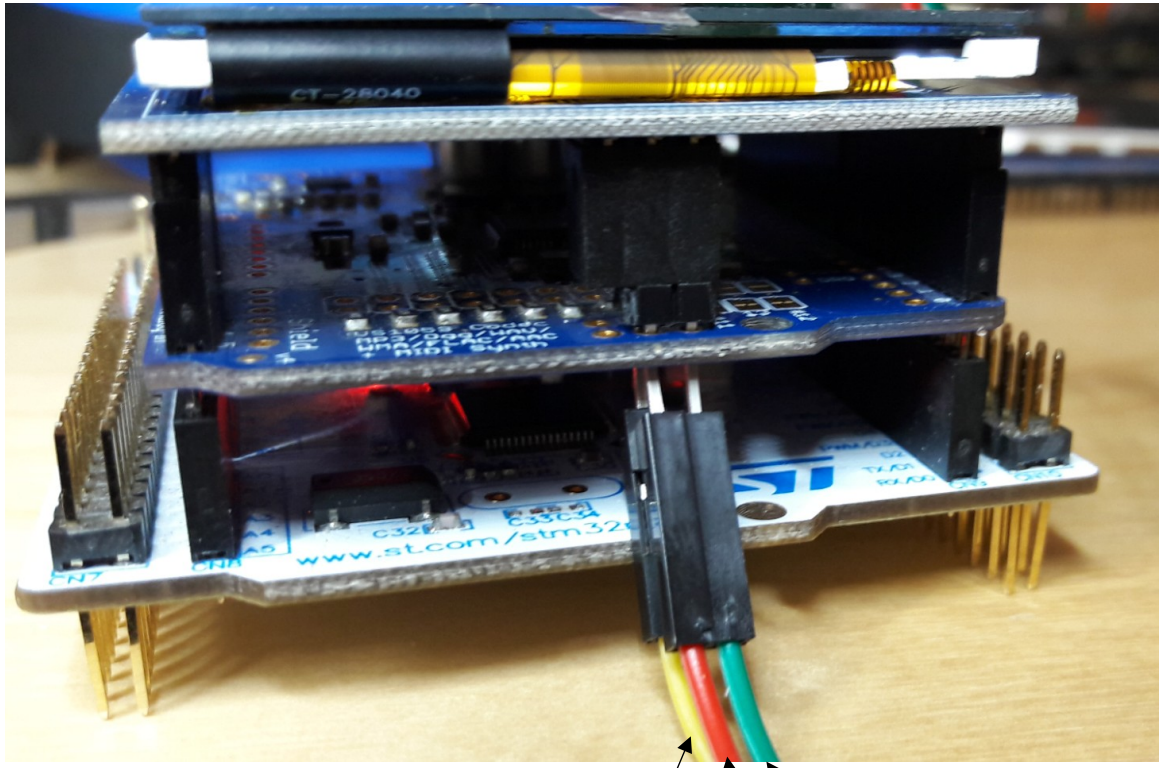
- Music Player Project
  - Assignment spec
  - Demo skeleton code
- SPI protocol
- I2C protocol
- Intro to C++
- Device Drivers
  - Intro
  - CHAR model
  - PJ driver framework
- Assignment 5 – Touch driver

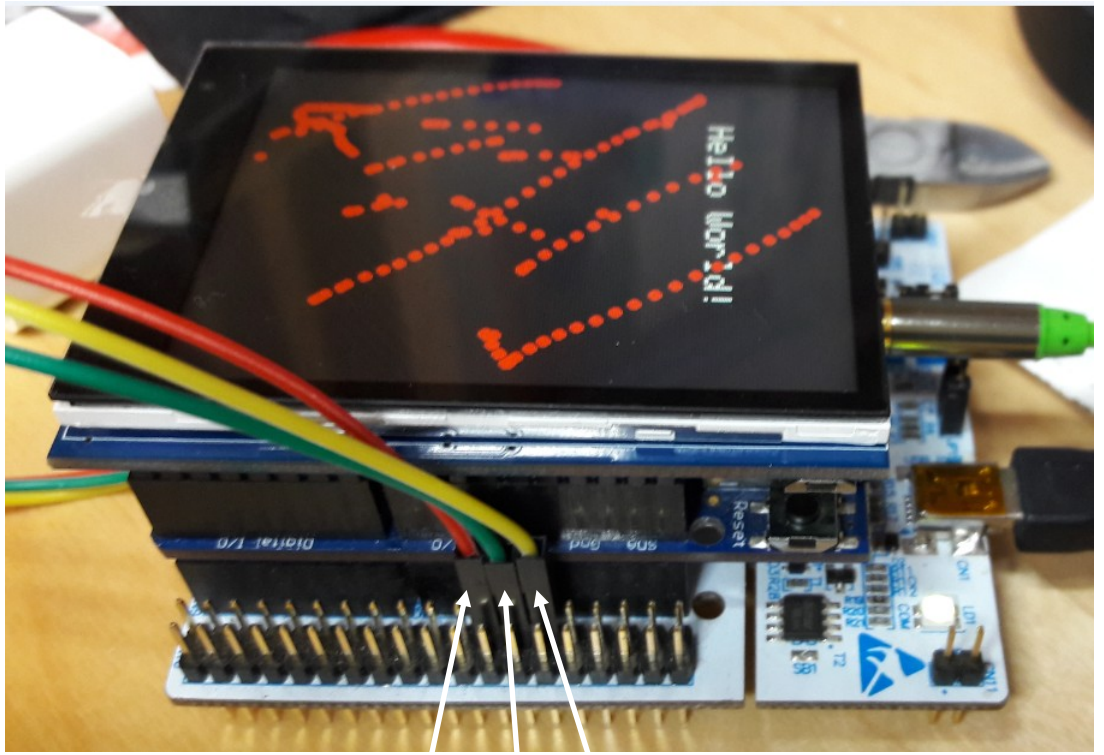
Current Lecture (L6) Overview

- Connecting jumper wires to make the LCD work
- Event driven MP3 player
- Task priorities - Rate monotonic scheduling
- Adding I2C driver to PJDF
- Tracing uCOS message queue
- Explore graphics library – Button class

Connecting jumper wires to make the LCD work

Watch the video in the lecture to learn how to connect the jumper wires to the ICSP header





Event driven MP3 player

- Terminals:
  - Input touch panel:
    - Event stream of x-y points from the touch panel
    - Button ideas: can have buttons for “play/pause”, “stop”, “fast forward”, “reverse”, volume up, volume down, etc.
    - Can have a “message loop” to filter each touch event to determine if it is within a button
  - UART: status and debugging information
  - LCD: display buttons, song list, current status, current song, etc.
  - MP3 decoder: need task to stream data to the decoder. Start/stop/pause streaming.

Event driven MP3 player

Review: what to implement

- Required features:
  - Initial state: no music playing
  - Start: start playing an MP3 audio file from the beginning in response to a screen button touch
  - Stop: stop playing an MP3 audio file in response to a screen button touch
  - Indicate play in progress or stopped on the LCD
  - Add one additional functionality
  - UART: use for more detailed (debug) information
- Optional features (suggestions):
  - Pause/play toggle: toggle to pause or resume playing an MP3 audio file in response to a button touch
  - Volume up/down in response to button touch
  - Move to next song in response to a button touch
  - Move to previous song in response to a button touch
  - display song list on LCD
  - Challenge: Read the MP3 data from the SD card instead of flash memory

Event driven MP3 player

- Event Channel
  - Translate x-y touch coordinates into a known command event (enum)
  - Forward event to appropriate queue/mailbox

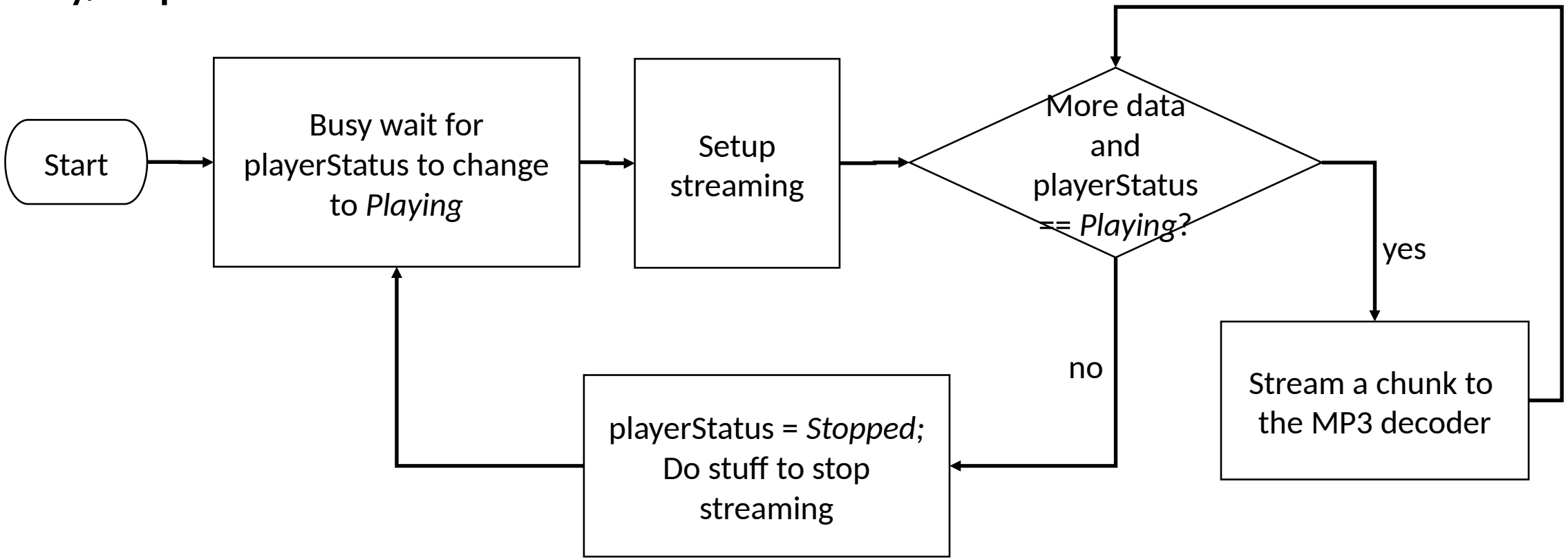
```
typedef enum
{
    INPUTCOMMAND_NONE,
    INPUTCOMMAND_FASTFORWARD,
    INPUTCOMMAND_NEXTSONG,
    INPUTCOMMAND_PLAY,
    INPUTCOMMAND_PREVSONG,
    INPUTCOMMAND_REWIND,
    INPUTCOMMAND_VOLUP,
    INPUTCOMMAND_VOLDOWN,
} InputCommandEnum;
```

Event driven MP3 player

- Task suggestions
  - Touch polling task: polls the touch panel and determines if a button has been touched and forwards an event stream of button press events to the command handler task
  - Command handler task: receives an event stream of commands from the touch polling task such as “play”, “pause”, “next song” etc.
  - Streaming task: waits for a signal to begin streaming data to the MP3 decoder; checks for status updates periodically during streaming to determine whether to stop, pause, etc.
  - LCD task: receives events from other tasks containing information for updating the display, e.g. change status to “play”

Streaming task

Play/Stop flow chart – one solution



Choosing task priorities

- Experiment with reversing task priorities in MP3Player

Rate Monotonic Scheduling (RMS)

- Questions:
  - With priority based scheduling, how can we determine the relative priority order for our tasks?
  - Does our CPU have enough speed to guarantee hard deadlines for our application?
- Liu & Layland (1973) proved that for a set of n periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is below a specific bound (depending on the number of tasks).
- A periodic task is one that does its job at a set time interval, e.g. the MP3 streaming task takes x milliseconds to send a chunk of 32 bytes to the decoder and it needs to do that every y milliseconds to maintain sound quality.
- RMS applies to an ideal system and thus serves as a good starting point for a real world system.

Sources: Labrosse, and Wikipedia

Rate Monotonic Scheduling (RMS)

The scheduling test for CPU utilization *U for n tasks* is

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

- Where C_i is computation time for task *i*, T_i corresponds to the execution period of task *i*.
- In other words, C_i/T_i is the fraction of CPU time required to execute task *i*.
- For 2 tasks, U = 2(sqrt(2) – 1) = 0.8284 i.e. we're OK as long as our 2 tasks don't use more than around 82% of the CPU time.

Rate Monotonic Scheduling (RMS)

Example for 3 tasks

Task	Execution time	Period	Priority
T1	2	5	1
T2	2	10	2
T3	1	8	3

- RMS states that the highest rate task gets the highest priority
- CPU utilization = 2/5 + 2/10 + 1/8 = 0.725
- $U = 3(2^{1/3} - 1) = 0.78$
- Since 0.725 <= 0.78 then our system should meet all hard deadlines

Rate Monotonic Scheduling (RMS)

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

- Rule of thumb is to keep CPU utilization below 70% for tasks with hard deadlines.
- RMS is proven for an ideal theoretical system however it is a good starting point for real world systems.

Adafruit_FT6206 touch driver

- The Adafruit driver is designed for polling the touch panel without being interrupted.
- Adafruit_FT6206.h
  - #define FT6206_ADDR 0x38
    - The 7-bit I2C slave address of the touch hardware
    - Assigned by NXP
  - Has #defines for register addresses in the touch hardware

  Defines class AdaFruit_FT6206
  - Constructor:
    - Adafruit_FT6206(void);

Adafruit_FT6206 touch driver

Adafruit_FT6206 methods:

- boolean begin(uint8_t thresh = FT6206_DEFAULT_THRESSHOLD);
  - Initializes the touch hardware
- void writeRegister8(uint8_t reg, uint8_t val);
  - Writes a value to a register in the touch hardware
- uint8_t readRegister8(uint8_t reg);
  - Reads a byte from a register in the touch hardware
- void readData(uint16_t *x, uint16_t *y);
  - Reads a touch (x,y) point from the touch hardware
- void autoCalibrate(void);
  - Don’t use this
- boolean touched(void);
  - Queries the touch hardware to see if a touch occurred
- TS_Point getPoint(void);
  - Wrapper for readData() that returns an (x,y,z) class

Adafruit_FT6206 touch driver

Adafruit_FT6206 usage:

- Create an instance of the device in tasks.c:
  - Adafruit_FT6206 touchCtrl = Adafruit_FT6206();
- Initialize:
  - touchCtrl.begin(40);
- See if a touch occurred and get the touch point:

```
TS_Point rawPoint;
If (touchCtrl.touched()) {
    rawPoint = touchCtrl.getPoint();
}
```

Adding I2C driver to PJDF

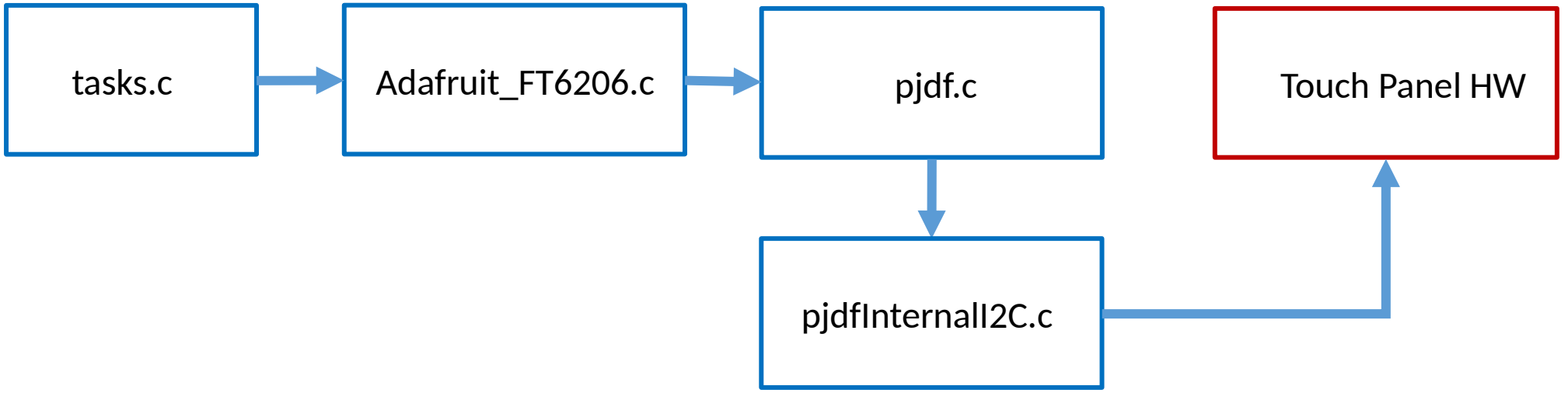
Goal:

- Make class Adafruit_FT6206 hardware independent by using the PJDF Char model.
- Result (ideally) will be that to port class Adafruit_FT6206 to a new microcontroller the only code we will need to change will be PJDF code and not Adafruit_FT6206 code.

Adding I2C driver to PJDF

Before:

After:



Adding I2C driver to PJDF

Steps: Begin in pjdf.h

// PJDF DEVELOPER TODO LIST FOR ADDING A NEW DRIVER:

// - define a new PJDF_DEVICE_ID_<MYDEVICE> below

// - reference it under PJDF_DEVICE_IDS below

// - add a new pjdfInternal<mydevice>.c module to implement the pjdfInternal.h interface

// - reference the Init() function of your driver in the driversInternal array in pjdf.c

// - add a new pjdfCtrl<mydevice>.h interface to define the Ioctl() functionality of your device

// - #include your pjdfCtrl<mydevice>.h in the present header file above

// - add modules as needed to the BSP folder to keep board-dependent code out of your PJDF implementation

Adding I2C driver to PJDF

pjdf.h

```
// Master list of device drivers.
// These are the identifiers used by applications to Open device drivers.
#define PJDF_DEVICE_ID_SPI1  "/dev/spi1"
#define PJDF_DEVICE_ID_MP3_VS1053   "/dev/mp3_vs1053"
#define PJDF_DEVICE_ID_LCD_ILI9341   "/dev/lcd_ili9341"
#define PJDF_DEVICE_ID_SD_ADAFRUIT   "/dev/sd_adafruit"
#define PJDF_DEVICE_ID_I2C1   "/dev/i2c1"

#define PJDF_DEVICE_IDS \
        PJDF_DEVICE_ID_SPI1, \
        PJDF_DEVICE_ID_MP3_VS1053, \
        PJDF_DEVICE_ID_LCD_ILI9341, \
        PJDF_DEVICE_ID_SD_ADAFRUIT, \
        PJDF_DEVICE_ID_I2C1, \
```

Adding I2C driver to PJDF

Changes to file Adafruit_FT6206.c

Modify the following functions:

- Adafruit_FT6206::readRegister8()
- Adafruit_FT6206::readData()
- Adafruit_FT6206::writeRegister8()

1. Don’t call I2C_* functions from here anymore; instead call pjdf.h functions
2. Move the I2C_* function calls to pjdfInternalI2C.c

Adding I2C driver to PJDF

- Adafruit_FT6206 will now become a client of pjdf
- Replace the code in the following methods with calls to your driver
  - Adafruit_FT6206::readRegister8()
  - Adafruit_FT6206::readData()
  - Adafruit_FT6206::writeRegister8()
- Adafruit_FT6206 will need a handle to your driver
  - You can follow a pattern similar to that of Adafruit_ILI9341 where

    HANDLE hLcd is defined in the header file and initialized with setPjdfHandle()

Adding I2C driver to PJDF

What is being supplied to you (see PjdfI2c.zip on Canvas):

- pjdf.h – already completed for you
- pjdfCtrlI2C.h – already completed for you
- pjdfInternalI2C.c – partially completed for you

What's left for you to do:

- Finish pjdfInternalI2C.c – just the <your code here> portions
- Handle initialization of the new I2C PJDF driver
- Modify Adafruit_FT6206.c and Adafruit_FT6206.h as needed to stop calling I2C_* functions and start calling PJDF functions.

Explore Touch specs

- See handy reference links page https://canvas.uw.edu/courses/1347174/pages/embsys-105-resources
- Application Note contains register spec
  - Data Sheet contains pin spec and electrical schematics

Tracing uCOS message queue

- Download uCosQueue.zip from Canvas
- Requires your context switch
- Task one sends lower case letters
- Task two sends upper case letters
- Task three consumes and prints the received letters
- Experiment with changing the priorities of the tasks
  - Consumer low pri
  - Consumer high pri
- Set breakpoints and look at the data returned by OSQQuery()
  - Notice the priority of the blocked task is stored as a bit
  - Look at the queue in memory

Explore

Graphics library

```
Adafruit_GFX_Button button1 = Adafruit_GFX_Button(); // Declare a button
 // Initialize a button for clearing the screen
    button1.initButton(
                    &lcdCtrl,
                    ILI9341_TFTWIDTH-30, ILI9341_TFTHEIGHT-20, // x, y center of button
                    60, 40, // width, height
                    ILI9341_YELLOW, // outline
                    ILI9341_BLACK, // fill
                    ILI9341_YELLOW, // text color
                    "Clear", // label
                    1); // text size
```

Explore

Graphics library

```
 // Get a touch point p
if (button1.contains(p.x, p.y)) {
    // clear screen and start over
    DrawLcdContents();
}
```