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 5 2/3/2020

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	Holiday – enjoy!	
2/17	L6	
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 (L4) Overview

- uCOS Internals (Labrosse ch 3, 6, 9)
  - Task States
  - Task Control Blocks (TCBs)
  - Ready List
  - Task Scheduling
  - Event Control Blocks
    - Placing a task in the ECB Wait List
    - Removing a Task from the ECB Wait List
    - Find the Highest Priority Task Waiting on the ECB
    - List of Free ECBs
    - Initializing an ECB
    - Making a Task Ready
    - Making a Task Wait for an Event
    - Making a task Ready Because of a Timeout
  - Event Flags

- Task Synchronization Techniques (Tanenbaum, Wikipedia, etc.)
  - Sharing data between ISRs and tasks
  - Producer-Consumer Problem
  - Readers and Writers Problem
  - Deadlock
  - Dining Philosophers Problem
- Event Driven Systems (Wikipedia)
  - Characteristics of Event Driven Systems
  - Data Flow Diagrams
- Assignment 4 – Task Synchronization

Stopped here last lecture.
Next slide: 46

Current Lecture 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

Music Player Project

- Walk through assignment spec
- Demo of starter app

SPI – Serial Peripheral Interface

SPI – Serial Peripheral Interface

- Developed by Motorola (1980s)
- A four-wire serial bus
- Synchronous serial communication – meaning bits are transmitted only on clock pulses
- De facto standard – meaning no official standard for usage

- SCLK : Serial Clock (output from master)
- MOSI : Master Output, Slave Input (output from master)
- MISO : Master Input, Slave Output (output from slave)
- SS : Slave Select (active low, output from master)

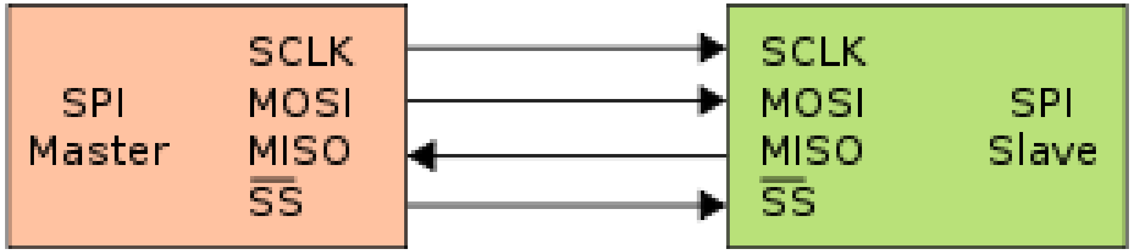


Diagram source: Wikipedia

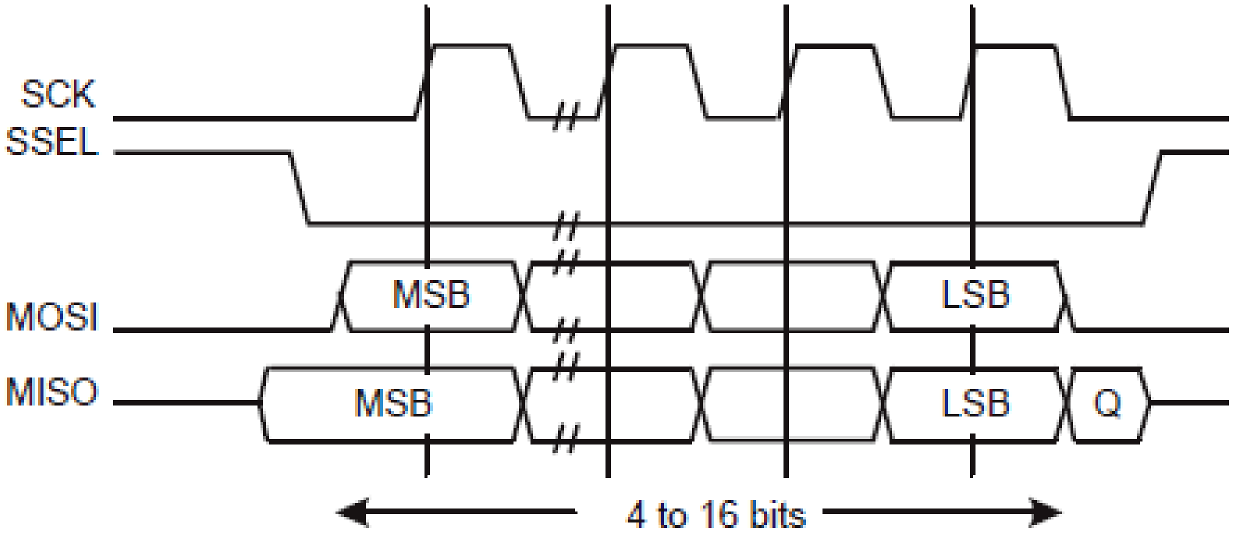
SPI – Serial Peripheral Interface

- Full duplex communication
  - 1 bit of data is simultaneously shifted out from master to slave and slave to master on each clock pulse.
  - Data moves from the master to the slave on the MOSI line
  - Data moves from the slave to the master on the MISO line
  - Allows for high speed data transfers determined by clock rate
- Master controls the clock which controls data transfer protocol
  - When clock is not pulsing no data is exchanged
  - 2 parameters combined give 4 variations for data transfer
    - CPOL – need to choose the clock-idle polarity, either high or low
    - CPHA – need to choose clock phase when data are captured – either rising or falling clock edge

SPI – Serial Peripheral Interface

CPOL == 0, CPHA == 0

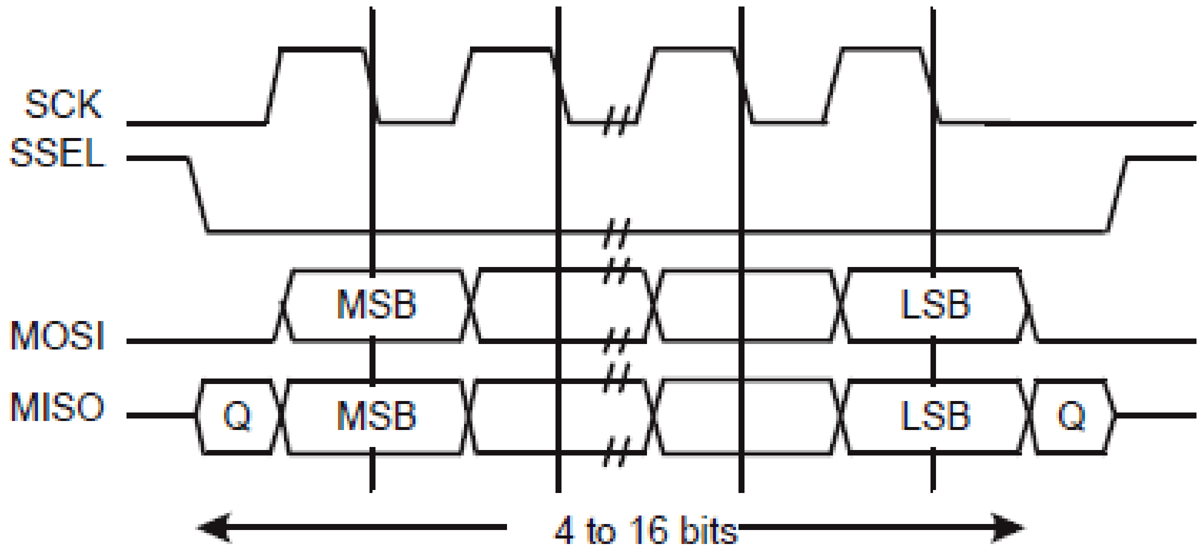
- Clock polarity idle state is **low**
- Data capture occurs on clock **rising** edge
- Diagram is for LPC23xx which provides for frame sizes between 4 and 16 bits



SPI – Serial Peripheral Interface

CPOL == 0, CPHA == 1

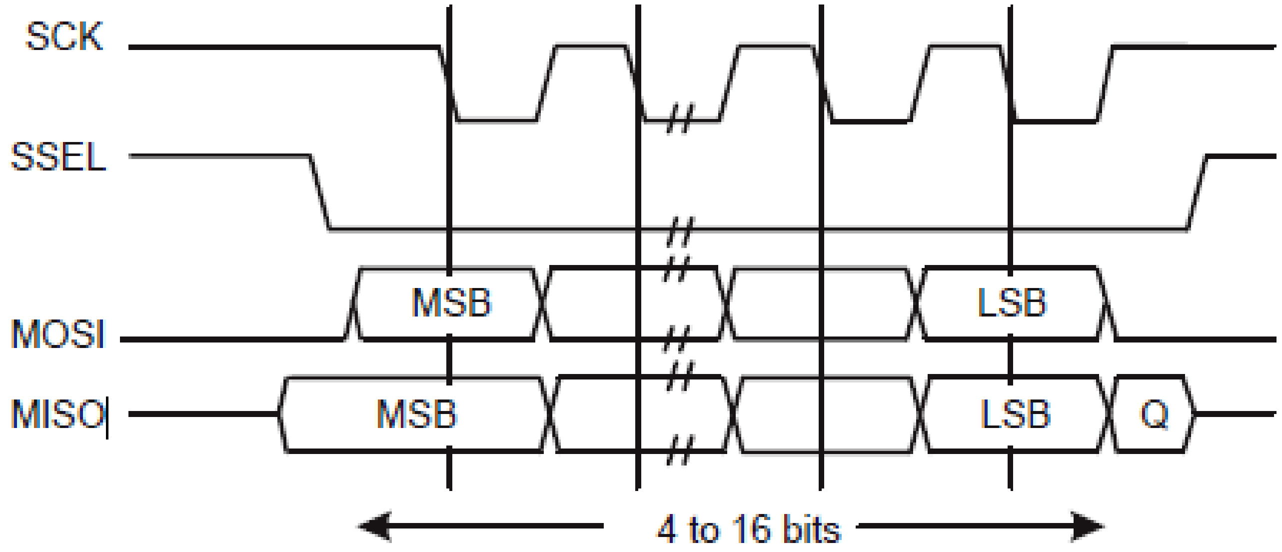
- Clock polarity idle state is **low**
- Data capture occurs on clock **falling** edge
- Diagram is for LPC23xx which provides for frame sizes between 4 and 16 bits



SPI – Serial Peripheral Interface

CPOL == 1, CPHA == 0

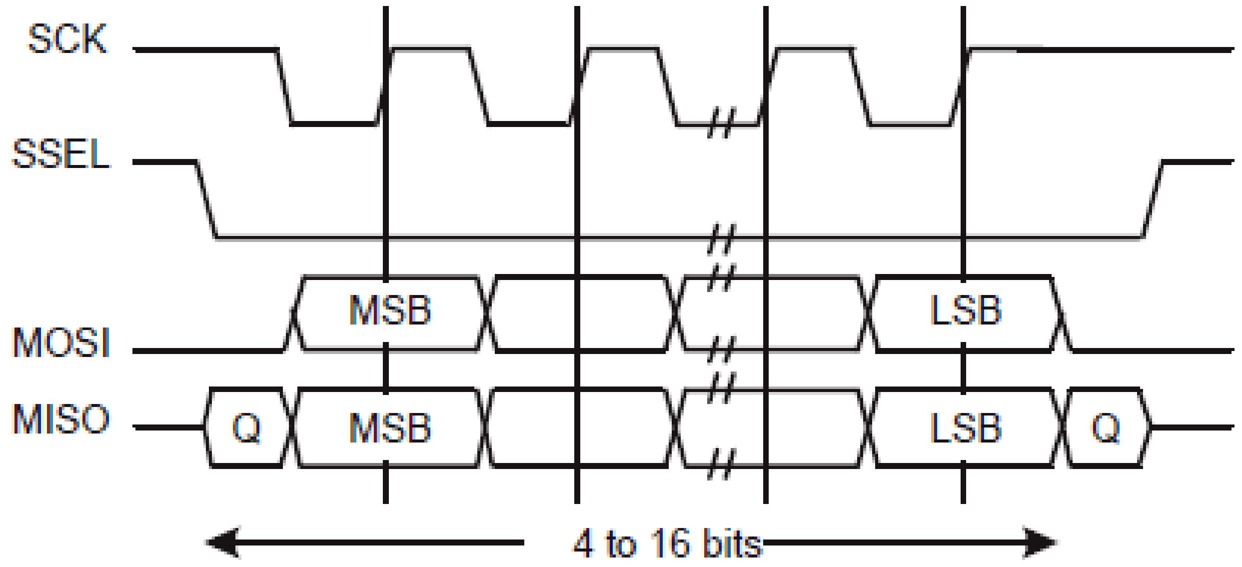
- Clock polarity idle state is **high**
- Data capture occurs on clock **falling** edge
- Diagram is for LPC23xx which provides for frame sizes between 4 and 16 bits



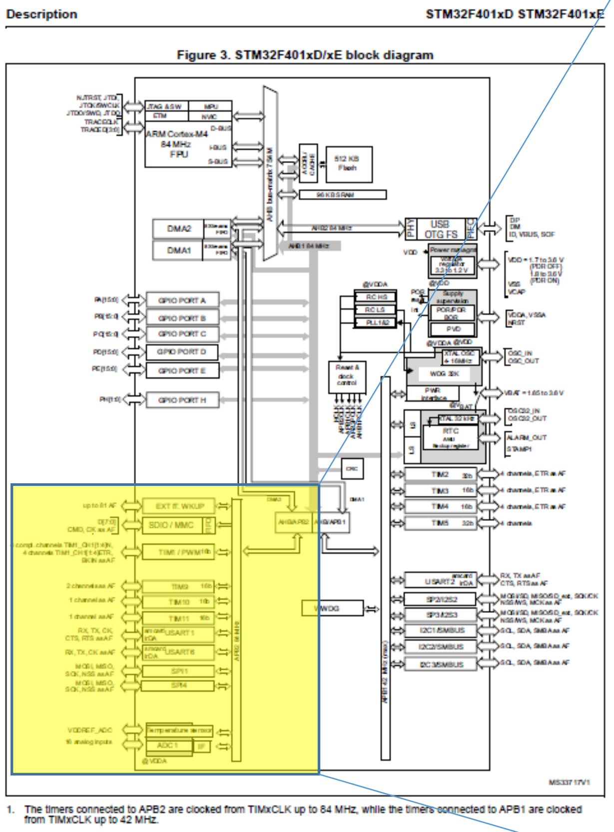
SPI – Serial Peripheral Interface

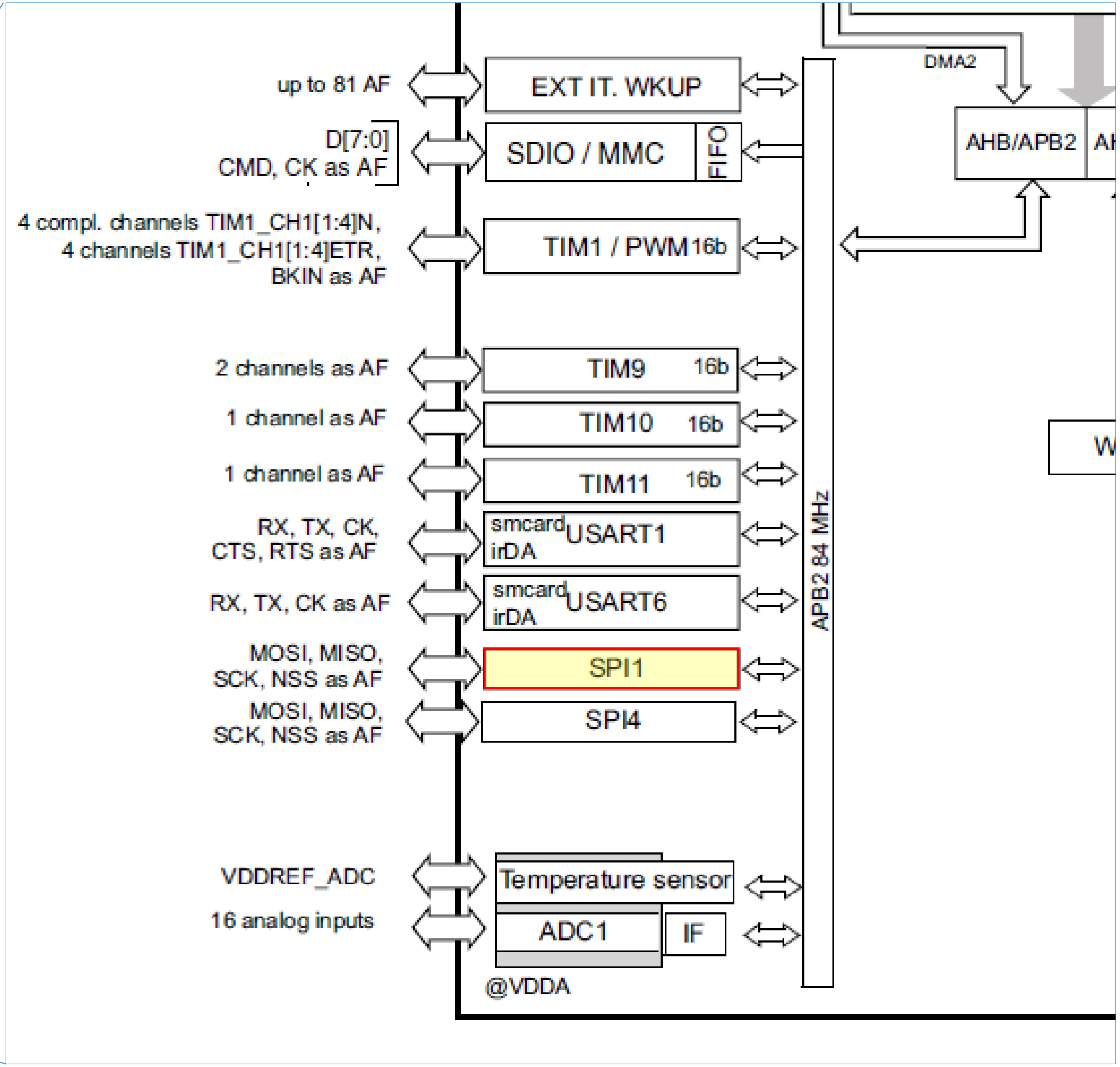
CPOL == 1, CPHA == 1

- Clock polarity idle state is **high**
- Data capture occurs on clock **rising** edge
- Diagram is for LPC23xx which provides for frame sizes between 4 and 16 bits



Block diagram showing SPI1





Reference: http://www.st.com/web/en/resource/technical/document/datasheet/DM00102166.pdf

SPI – Serial Peripheral Interface

```
void BspSPI1Init()
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

    SPI_InitTypeDef SPI_InitTypeDefStruct;

    SPI_InitTypeDefStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitTypeDefStruct.SPI_Mode = SPI_Mode_Master;
    SPI_InitTypeDefStruct.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitTypeDefStruct.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitTypeDefStruct.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitTypeDefStruct.SPI_NSS = SPI_NSS_Soft;
    SPI_InitTypeDefStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_32;
    SPI_InitTypeDefStruct.SPI_FirstBit = SPI_FirstBit_MSB;

    SPI_Init(SPI1, &SPI_InitTypeDefStruct);
```

continued …

SPI – Serial Peripheral Interface

```
 /*-------- Configure SCK, MISO, MOSI --------*/
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;

    GPIO_Init(GPIOA, &GPIO_InitStruct);

    /*--------- Configure alternate GPIO functions to SPI1 ------*/
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);

    SPI_Cmd(SPI1, ENABLE);
}
```

I2C Inter-Integrated Circuit

I2C Inter-Integrated Circuit

- I2C-bus is a de facto world standard
- Implemented in over 1000 different ICs manufactured by more than 50 companies
- V1 1982
  - **100-kHz clock – default used in our MP3 project**
  - **7-bit slave addresses – used in our MP3 project**
  - 1992 added **400-kHz Fast-mode (try it!)** and 10-bit slave addresses
- V2 1998 added **3.4-MHz High-speed mode**
- …
- V6 2014 latest with **5-MHz Ultra Fast-mode** and other features
- Compare to **SPI** which can go up to **42 MHz** on our STM32.

References: Wikipedia, and http://www.nxp.com/documents/user_manual/UM10204.pdf

I2C

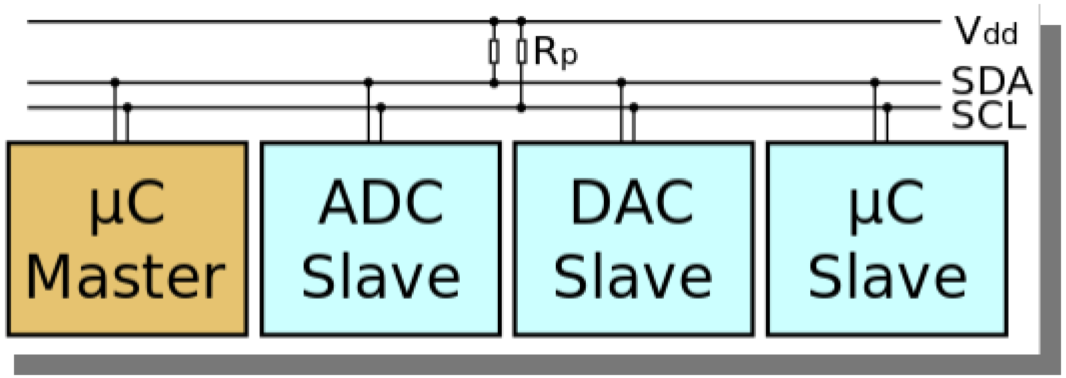
Design

- Uses only two bidirectional open-drain lines
  - SDA, Serial Data Line
  - SCL Serial Clock Line
  - pulled up with resistors, i.e. **idle state is high**.
- **Master** device initiates data transfers between **Slave** device(s) and memory
- Each slave has a unique address
- Compare to **SPI** where each slave has a chip-select line instead of an address

I2C

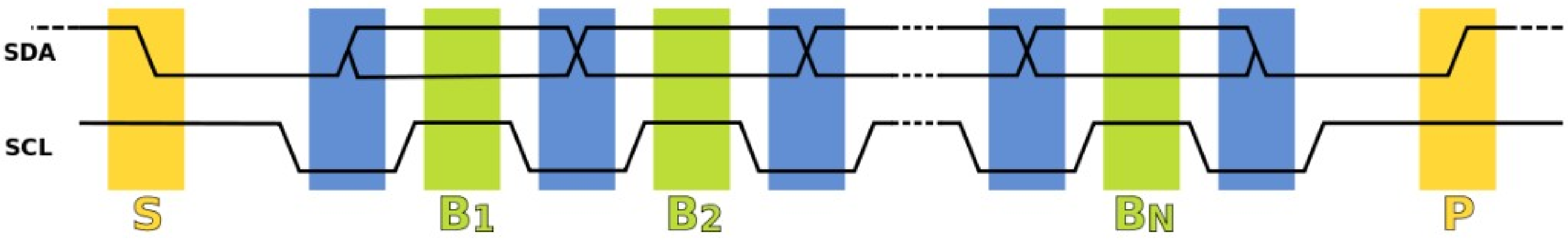
Example circuit

- Master is a microcontroller
- 3 Slave devices
  - ADC
  - DAC
  - Another microcontroller



- SDA and SCL are pulled high by Vdd, i.e. to transmit “0”, device pulls the SDA line down, to transmit “1” device does **not** pull the SDA line down.

I2C



Overview of I2C data transmission

1. **Start Bit S**: Master initiates transmission with a start bit:
   - while clock (SCL) is idle (high), Master pulls data (SDA) low
2. Clock pulses are then initiated
3. **Set data**: Falling SCL edge triggers first SDA bit to be **set while clock is low** (**blue region**)
4. **Sample data**: Clock pulses high at which point bit B1 is sampled (received)
5. Clock continues to pulse and SDA transmits 1 bit per pulse. Bits are set in **blue regions** when clock drops low and sampled in green regions when clock rises high.
6. **Stop Bit P**: Master concludes transmission with a stop bit (**yellow region P**):
   - while clock (SCL) is idle (high), Master allows data (SDA) to go high

I2C

Write to Slave

S	SLAVE ADDRESS	R/$\overline{W}$	A	DATA	A	DATA	A/$\overline{A}$	P

S=Start bit
SLAVE ADDRESS=7-bit address
Command bit = **0 for write**
A=ACK=0 returned by Slave to Master
DATA= 8 bits … multiple bytes may be written by master
A/$\overline{A}$= ACK/NACK, NACK indicates problem, can resend
P=Stop bit

Master to Slave

Slave to Master

I2C

Read from Slave – direct read

S	SLAVE ADDRESS	$R/\overline{W}$	A	DATA	A	DATA	$\overline{A}$	P

S=Start bit
SLAVE ADDRESS=7-bit address
$R/\overline{W}$ = Command bit = **1 for read**
DATA= 8 bits … multiple bytes may be read
$A/\overline{A}$ = ACK/NACK from receiver
P=Stop bit

Master to Slave

Slave to Master

I2C

Combined format

S	SLAVE ADDRESS	$R/\overline{W}$	A	DATA	$A/\overline{A}$	Sr	SLAVE ADDRESS	$R/\overline{W}$	A	DATA	$A/\overline{A}$	P

- **Sr is a Repeated Start Bit** which serves to keep the connection open with the Slave
- DATA may consist of any number of bytes from sender with corresponding ACK/NACK from receiver
- Example: to read touch points from our touch panel we first send the address of the register we want to read, then we read the register contents.

Master to Slave

Slave to Master

Direction depends on preceding R or W

Communicating with the FT6206 (touch)

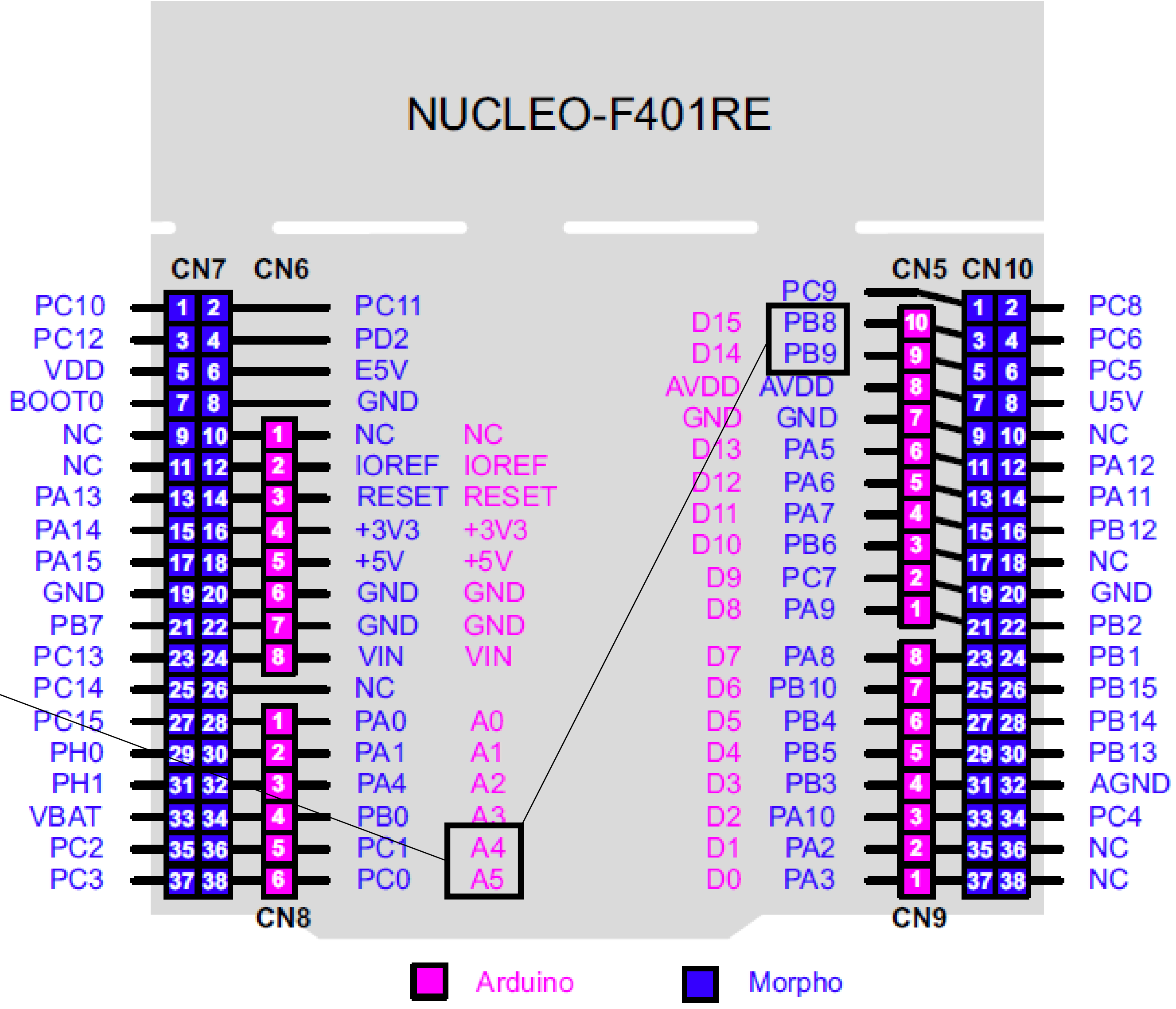
Capacitive Touch Pins (Adafruit)

- SDA - This is the I2C data pin used by the FT6206 capacitive touch controller chip. It can be shared with other I2C devices. On Arduino UNO this pin is also known as **Analog 4**.
- SCL - This is the I2C clock pin used by the FT6206 capacitive touch controller chip. It can be shared with other I2C devices. On UNO this pin is also known as **Analog 5**.

Communicati with the FT6206 (touc

SDA – A4 PB9
SCL – A5 PB8

- Alternate wiring on our boards has A4/A5 connected to PB9/PB8



Intro to C++

Class

- We'll limit our intro to C++ features that are used in our project code
- **Class: definition:** the **encapsulation** of a data structure together with the functions or "**methods**" to operate on it
- In C++, a **class** is defined in a **header** file
- A C++ **class** definition can be understood as an enhanced **struct** definition
- Key enhancements include
  - **private** and **public** tags on data fields and functions (**class members**) to determine which parts of the class a **client application** can access
  - **Function overload:** a function can be defined multiple times with the same name but different parameter lists
  - **Inheritance**: a **subclass (**or **derived class)** can inherit from a **base** class and add more features to make a new hybrid **subclass**.
  - **Virtual functions:** a **subclass** can **override** the **base class** definition of a function tagged as **virtual**. The function prototype is preserved but the function behavior is modified in the **subclass.**

Intro to C++

Example header file Rectangle.h:

```
class Rectangle {
public: // accessible to applications
    Rectangle(int x, int y, int w, int h); // constructor
    boolean IsInside(int x, int y); // is (x, y) inside the rectangle?
private: // not accessible to applications
    int x, y; // top left corner
    int w, h; // width, height
};
```

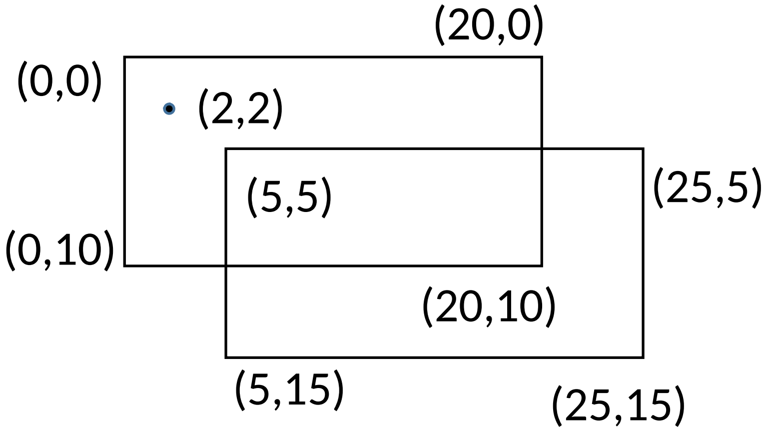
Intro to C++

Corresponding class implementation in file Rectangle.cpp:

```
Rectangle::Rectangle(int x, int y, int w, int h) { // Constructor initializes the class fields
    this->x = x; // this is an implicit first parameter in every method definition
    this->y = y;
    this->w = w;
    this->h = h;
}

boolean Rectangle::IsInside(int x, int y) {
    if (x >= this->x && x <= this->x + w && y >= this->y and y <= this->y + h)
    return true; else return false;
}
```

Intro to C++



Client code that uses **class Rectangle:**

Rectangle r1 = Rectangle(0, 0, 20, 10); // class **instance** r1

Rectangle r2 = Rectangle(5, 5, 20, 10); // class **instance** r2

r1.x = 5; // compiler error, x is private

r1.IsInside(2, 2); // true

r2.IsInside(2, 2); // false

Intro to C++

Base class example. Header file Shape.h:

```
class Shape {
public:
    Shape(char *name) {this->name = name}; // constructor
    virtual float Area() = 0; // pure virtual function to compute area of shape
    char *GetName() {return name;}; // accessor for applications to retrieve protected
                                    // member
protected:  // protected means accessible to subclasses, not applications
    char *name; // name of shape
};
```

Intro to C++

Subclass example header file Rectangle.h:

```
class Rectangle : public Shape {  // inherits from Shape
public:
    Rectangle(int x, int y, int w, int h); // constructor
    float Area() override;  // base class virtual function override
    boolean IsInside(int x, int y); // is (x, y) inside the rectangle?
private:
    int x, y; // top left corner
    int w, h; // width, height
};
```

Intro to C++

Corresponding class implementation in file Rectangle.cpp:

```
static const char *name = “rectangle”;

Rectangle::Rectangle(int x, int y, int w, int h) {
    this->x = x;
    this->y = y;
    this->w = w;
    this->h = h;
    this->name = name; // base class member
}
```

```
boolean Rectangle::IsInside(int x, int y) {
    if (x >= this->x && x <= this->x + w
            && y >= this->y and y <= this->y + h)
        return true; else return false;
}

float Rectangle::Area() {
    return (float) w * h;
}
```

Intro to C++

Subclass example header file Circle.h:

```
class Circle: public Shape {  // inherits from Shape
public:
	Circle(int x, int y, int r); // constructor
	float Area() override;  // base class virtual function override
private:
	int x, y, r; // center and radius
};
```

Intro to C++

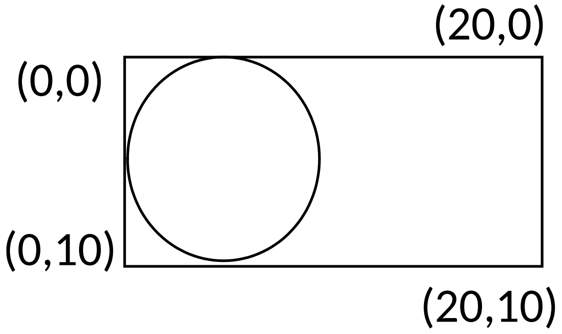
Corresponding class implementation in file Circle.cpp:

```
static const char *name = "circle";

Circle::Circle(int x, int y, int r) { // constructor
    this->x = x;
    this->y = y;
    this->r= r;
    this->name = name; // base class member
}
```

```
float Circle::Area() {
    return (float) 3.1416 * r * r;
}
```

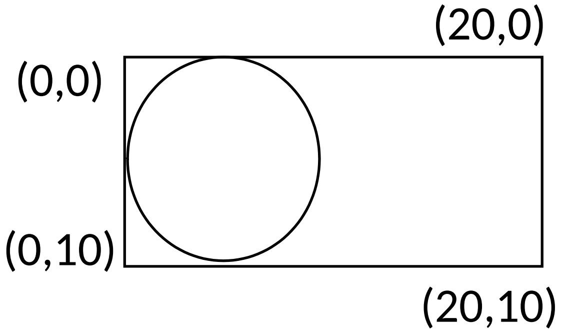
Intro to C++



Client code using subclasses**:**

```
Rectangle r1 = Rectangle(0, 0, 20, 10); // Rectangle class instance r1
Circle c1 = Circle(5, 5, 5);  // Circle class instance c1
char *nameR = r1.name; // compiler error: name is protected
char *nameR = r1.GetName(); // “rectangle”
char *nameC = c1.GetName(); // “circle”
float aR = r1.Area(); // 200
float aC = c1.Area(); // 78.54
```

Intro to C++



Client code using **polymorphism:**

```
Rectangle r1 = Rectangle(0, 0, 20, 10); // Rectangle class instance r1
Circle c1 = Circle(5, 5, 5);  // Circle class instance c1
Shape* pShape1 = dynamic_cast<Shape*>(&r1);
Shape* pShape2 = dynamic_cast<Shape*>(&c1);
char *nameR = pShape1->GetName(); // “rectangle”
char *nameC = pShape2->GetName(); // “circle”
float aR = pShape1->Area(); // 200
float aC = pShape2->Area(); // 78.54
```

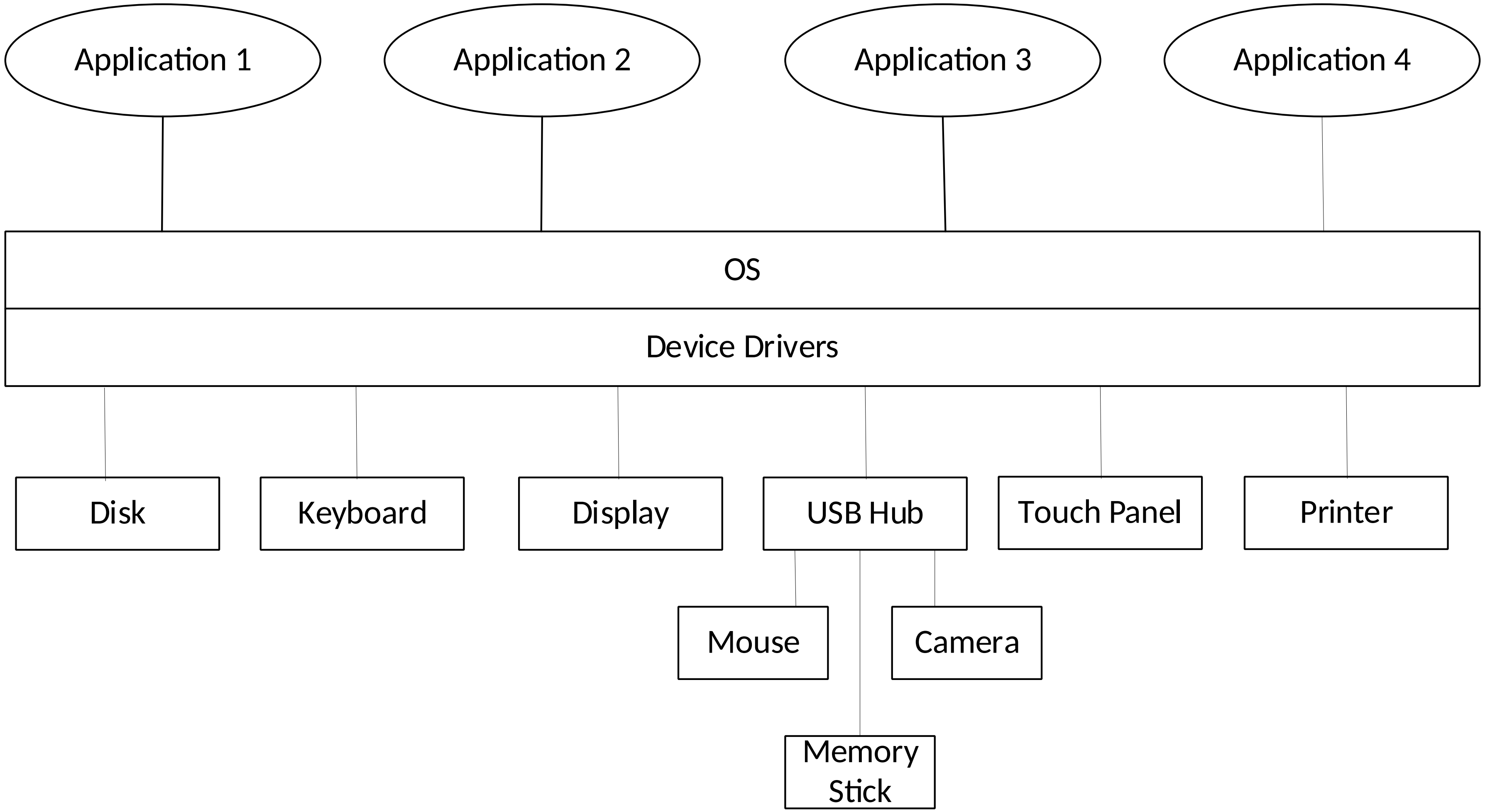
Introduction to device Drivers

Device Drivers

What is a Device Driver?

- Also referred to simply as a Driver
- Software that controls or operates a hardware device
- Simplifies control and operation of the device by exposing only a high level interface to client software that uses the device
- Aims to standardize the operation procedures for wide ranges of devices by providing a uniform interface across a class of devices e.g. printers, keyboards, etc.
- Enables a device to be attached to an OS for use by applications via the driver interface
- Examples
  - Disk Drive driver
  - Keyboard driver
  - Video Display driver
  - USB driver
  - Touch Panel driver
  - Printer

Device Drivers



Class Driver

What is a Class Driver?

- Supports a wide range of devices with similar characteristics
- Uses a common protocol
- Hardware makers just need to implement the class interface for their device and any class compliant OS and application can operate the device.
- USB has the most prolific examples of class drivers
  - The USB-IF (Universal Serial Bus Implementers Forum) defines a couple of dozen class specifications such as
    - Mass Storage Class (memory sticks)
    - HID Device Class (Keyboards, mice, game controllers, etc.)
    - Printer Class
    - Smart Card Class
    - Video Class (streaming video)
    - Battery Charging Class

Device Driver Model

What is a Device Driver Model or Framework?

- Provides standardized interfaces and protocols for ***applications*** to control and operate a device or range of devices
- Provides standardized interfaces and protocols for ***driver developers*** to enable the device to interact with the OS

Device Driver Model

Two common device driver models:

- Block Device Driver Model
  - Applies to devices that transfer data in fixed size blocks of bytes
  - Blocks may be accessed in **random** order
  - Examples: hard disk, SD card
- Character (Char) Device Driver Model
  - Applies to devices that transfer data in variable length streams of bytes
  - Access is **sequential**
  - Examples: mouse, keyboard

Char Device Driver Model

Char Device Driver Model

- Let’s look more closely at the Char Driver model
- The Char Driver Model consists of
  - A standard interface allowing applications to use the device
  - A standard protocol to follow when using the interface
- The Char Device Driver Model follows the Unix file I/O API
  - Unix chose to unify the interface across devices by treating them like sequential access files
- A compliant driver implements the following functions
  - open(): get a Handle to a specified device
  - close(): close a Handle to a specified device
  - read(): read bytes from a specified device
  - write(): write bytes to a specified device
  - ioctl(): send a control request to a device – device dependent

Char Device Driver Model

Protocol for using a device under the model

- To begin using the device, call ***open()*** to obtain an access token to the device (descriptor or handle)
- You may then use the handle to call ***read()*** to read bytes from the device, or ***write()*** to write bytes to the device
- To control any special device-dependent features, call ***ioctl()***
- When done with the device, call ***close()*** to release the device

Char Device Driver Model

Device Handle

HANDLE h = open(“/dev/devx”, flags)

- The open function obtains a handle (descriptor in Linux) to a device specified the same way as a file path
- The handle is a reference (pointer or index) to a device connection struct which is allocated to the calling application
- In our implementation it actually indexes into the table of device drivers directly
- Once a handle has been assigned to an application, the application may control and operate the Char device bound to the handle

Char Device Driver Model

ioctl

- In the Char Driver Model, ioctl is used to control all the device-specific features of a particular device
- open, close, read, write can be expected to work the same way across all devices that follow the Char Driver Model
- ioctl is the catch-all function for controlling everything else about a Char device

Example of a Char Driver Model:

PJ Driver Framework

used in M3Player starter app

PJ Driver Framework – Interface Spec

- Open
- Close
- Read
- Write
- Ioctl

PJ Driver Framework – Interface Spec

Open

```
HANDLE Open(char *pName, INT8U flags)
```

Open a handle to a specified device

pName – name of device within devices directory, eg. /dev/SAM0 (as a C string "/dev/SAM0")
flags – or'ed flags modeled on Linux file I/O – not currently used in our implementation – pass the value 0. Linux examples:

- O_APPEND – append subsequent writes to a file
- O_CREAT – create a file
- O_TRUNC – truncate the file i.e. prepare to rewrite the file

Returns – a small positive integer or a PJDF error code if something went wrong

PJ Driver Framework – Interface Spec

Close

```
INT8U Close(HANDLE h)
```

Close a handle to a device – release the device

h – handle of a device previously returned by Open()

Returns – PJDF_ERR_NONE if successful otherwise an error code

PJ Driver Framework – Interface Spec

Read

```
INT8U Read(HANDLE h, PVOID pBuffer, INT32U* pLength)
```

Read data from a device

h – handle of a device previously returned by Open()
pBuffer – pointer to memory buffer of bytes to receive data
pLength – pointer to length of buffer on input, length of transfer on output

Returns – PJDF_ERR_NONE if successful

PJ Driver Framework – Interface Spec

Write

```
INT8U Write(HANDLE h, void* pBuffer,INT32U* pLength)
```

Write data to a device

h – handle of a device previously returned by Open()
pBuffer – pointer to memory buffer of bytes to write
pLength – pointer to length of buffer on input, length of transfer on output

Returns – PJDF_ERR_NONE if successful

PJ Driver Framework – Interface Spec

Ioctl

```
INT8U Ioctl(HANDLE h, INT8U request, void* pArgs,
      INT32U* pSize)
```

Perform device I/O control on a device

h – handle of a device previously returned by Open()
request – code number of a request – available codes depend on the device
pArgs– pointer to memory buffer of bytes to send/receive data
pLength – pointer to length of buffer on input, length of transfer on output

Returns – PJDF_ERR_NONE if successful

PJ Driver Framework – Internals simplified

- Driver management
- Internal representation of drivers
- Mechanism for accessing a specific driver function via the application interface

PJ Driver Framework – Internals simplified

Application code

Calls generic driver functions in pjdf.c to perform Open, Close, Read, Write, Ioctl for a selected device.

pjdf.c

Generic code to open, close, read, write, ioctl for any of N devices.

driversInternal[]

0	*pName= *Open= *Close= *Read= *Write= *Ioctl=	"/dev/devA" OpenA() CloseA() ReadA() WriteA() IoctlA()
1	*pName= *Open= *Close= *Read= *Write= *Ioctl=	"/dev/devB" OpenB() CloseB() ReadB() WriteB() IoctlB()
2	…	…
N-1	…	…

pjdf.c manages the drivers internally with an array of structs where each (simplified) struct contains:

- Pointer to name (string) of device i
- Pointer to Open() for device i
- Pointer to Close() for device i
- Pointer to Read() for device i
- Pointer to Write() for device i
- Pointer to Ioctl() for device i

PJ Driver Framework – Internals simplified

Application code

hB = Open("/dev/devB", 0);
// hB is returned as 2

pjdf.c

Generic code for **Open():**

- Search for driver "/dev/devB"
- Call **Open()** of the found driver
- Return driversInternal index+1 i.e. **2**

driversInternal[]

0	*pName=	"dev/devA"
	*Open=	OpenA()
	*Close=	CloseA()
	*Read=	ReadA()
	*Write=	WriteA()
	*Ioctl=	IoctlA()
1	***pName=**	**"/dev/devB"**
	***Open=**	**OpenB()**
	*Close=	CloseB()
	*Read=	ReadB()
	*Write=	WriteB()
	*Ioctl=	IoctlB()

Code for **OpenB():**

- Do stuff to open device B

PJ Driver Framework – Internals simplified

Application code

```
char *buf = “data”;
INT32U len = 4;
Write(hB, buf, &len); // hB is 2
```

pjdf.c

Generic code for **Write():**

- Driver index = hB – 1 = **1**
- Call **Write()** for **driver 1**

driversInternal[]

0	*pName=	“/dev/devA”
	*Open=	OpenA()
	*Close=	CloseA()
	*Read=	ReadA()
	*Write=	WriteA()
	*Ioctl=	IoctlA()
1	*pName=	“/dev/devB”
	*Open=	OpenB()
	*Close=	CloseB()
	*Read=	ReadB()
	***Write=**	**WriteB()**
	*Ioctl=	IoctlB()

Code for **WriteB():**

- Do stuff to write buf to device B

PJ Driver Framework – Internals simplified

Application code

```
char buf[4];
INT32U len = 4;
Read(hB, buf, &len); // hB is 2
```

pjdf.c

Generic code for **Read():**

- Driver index = hB – 1 = **1**
- Call **Read()** for driver **1**

driversInternal[]

0	*pName=	"/dev/devA"
	*Open=	OpenA()
	*Close=	CloseA()
	*Read=	ReadA()
	*Write=	WriteA()
	*Ioctl=	IoctlA()
1	*pName=	"/dev/devB"
	*Open=	OpenB()
	*Close=	CloseB()
	***Read=**	**ReadB()**
	*Write=	WriteB()
	*Ioctl=	IoctlB()

Code for **ReadB():**

- Do stuff to read from device B into buf

MP3Player PJDF Device Drivers

- Multiple drivers are implemented in the MP3Player starter app

MP3Player PJDF Device Drivers

- The MP3Player starter app contains three or more PJDF device drivers. Names of the drivers are defined in pjdf.h:

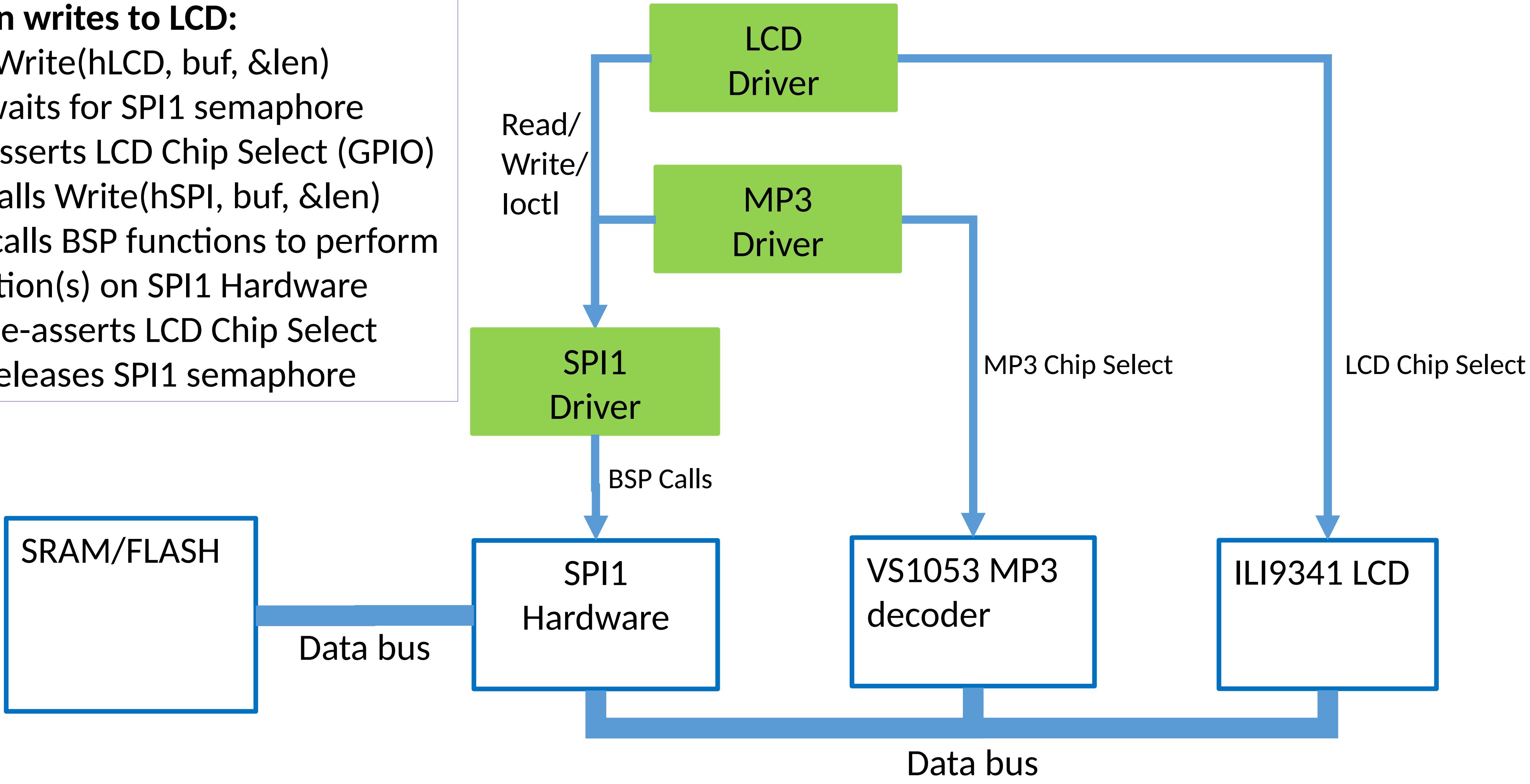
```
#define PJDF_DEVICE_ID_SPI1  "/dev/spi1"
#define PJDF_DEVICE_ID_MP3_VS1053   "/dev/mp3_vs1053"
#define PJDF_DEVICE_ID_LCD_ILI9341   "/dev/lcd_ili9341“
```

- The SPI driver is used to send/receive data between the memory and a peripheral device through a Serial Peripheral Interface (SPI)
- The MP3 driver is used to control the peripheral VS1053 MP3 decoder chip. The MP3 driver can be viewed as a simple wrapper around a SPI driver which is used to send commands and data to the MP3 decoder.
- The LCD driver is used to control the peripheral ILI9341 display. Like the MP3 driver it is also a wrapper around a SPI driver used to send commands and data to the display.
- It so happens that both the VS1053 and the ILI9341 are connected to the same SPI GPIO pins on the NUCLEO (SPI1). Since they share the SPI resource their accesses to it have to be serialized (using a semaphore).

MP3Player PJDF Device Drivers

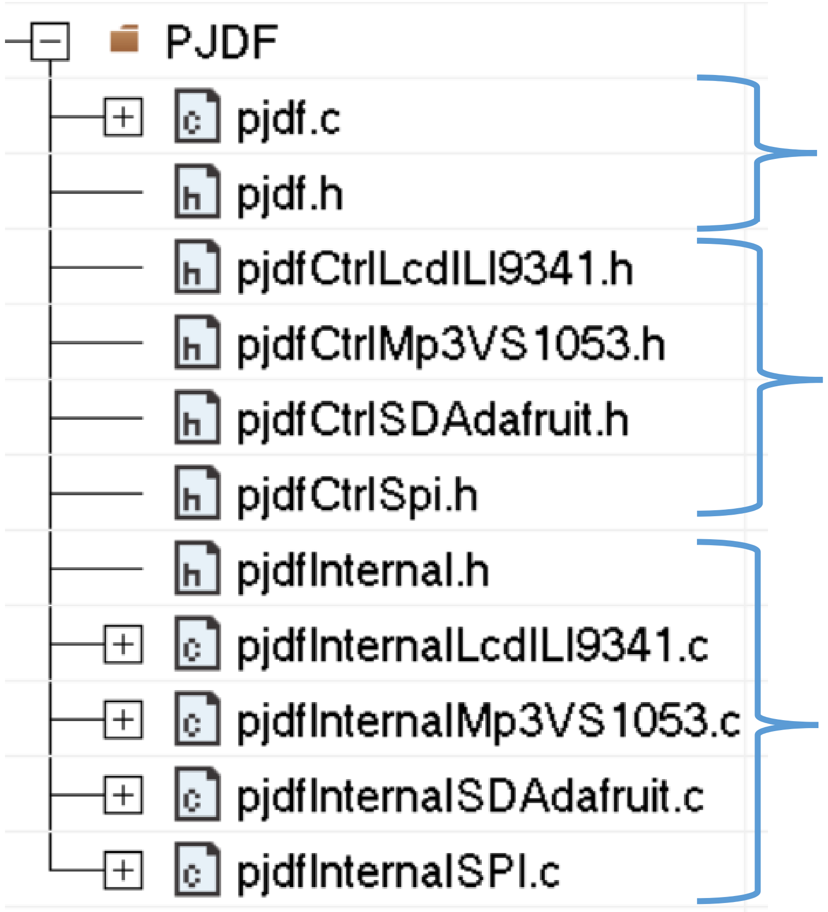
Example: application writes to LCD:

- Application calls Write(hLCD, buf, &len)
  - LCD Driver waits for SPI1 semaphore
  - LCD Driver asserts LCD Chip Select (GPIO)
  - LCD Driver calls Write(hSPI, buf, &len)
  - SPI1 Driver calls BSP functions to perform Write operation(s) on SPI1 Hardware
  - LCD Driver de-asserts LCD Chip Select
  - LCD Driver releases SPI1 semaphore



MP3Player PJDF Device Drivers

PJFD source code organization



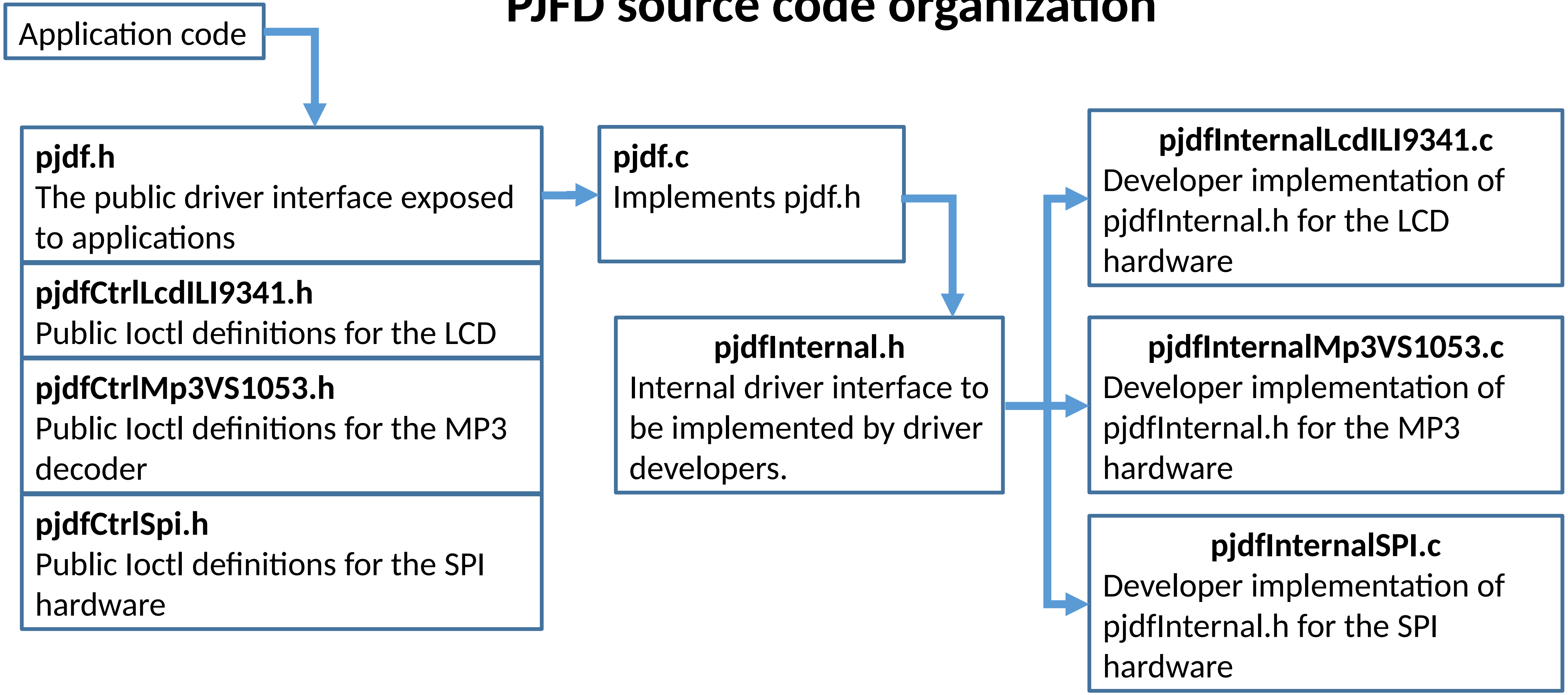
Interface (.h) exposed to applications, and its implementation (.c)

Ioctl interface (.h) per device

Internal interface (.h), and implementation per device (.c)

MP3Player Device Drivers

PJFD source code organization



MP3Player Device Drivers

Adding a new driver

MP3Player Device Drivers

Adding a new driver

pjdfInternal.h

- pjdfInternal.h is the interface to be implemented by device driver developers
- It consists of the 5 standard functions to implement per device:
  - Open(), Close(), Read(), Write(), Ioctl()
- And one additional function per device:
  - **Init()**: function for initializing the device before it is exposed to applications i.e. before multitasking begins
  - To make this work, function InitPjdf() is provided by the framework and is called before multitasking begins. It calls the Init() function for each PJDF driver.

MP3Player Device Drivers

Adding the touch driver to the PJDF framework

- This driver should be simpler to add than the MP3 or LCD drivers
- The touch controller does not use SPI, instead it uses I2C
- Currently, the MP3Player starter app directly calls BSP functions when it wants to read or write to the I2C bus.
- Your job will be to replace those direct BSP calls with calls to Open(), Close(), Read(), Write() and Ioctl() as necessary. Of course you will need to implement those functions as well.

MP3Player Device Drivers

Steps to add the I2C driver to the PJDF framework

- Start in pjdf.h
  - Observe the “TODO LIST” for developers
  - Add the new driver ID:
    - #define PJDF_DEVICE_ID_I2C1   "/dev/sd_I2C1"
  - etc – see the TODO LIST

Assignment 5

- Touch Driver