1 - Questions:

a. *Explain the SPI communication protocol with a timing diagram.*

Figure 1 below shows a generic timing diagram for an 8-bit transfer using the SPI protocol.

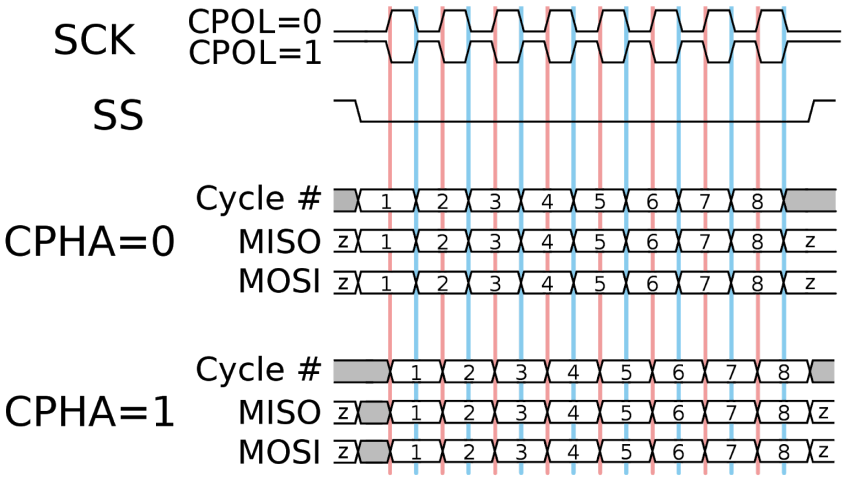

Figure 1: SPI timing diagram.

SPI can be used to interface with several different slave devices using the same communication pins on the master device. The master selects the desired slave device using the chip select pin SS (which is active when low). For the purpose of explanation, only a master and single slave device is shown in Figure 1 above. The master device is responsible for generating a clock signal to ensure the transmission is synchronous. Before data transmission, the clock is held in an idle state (either high or low, depending on the value of CPOL. CPOL is the clock polarity). The CPHA (clock phase) determines whether the data is sampled on the first or second clock edge.

In the example shown in Figure 1, there are 8 bits being transferred, each bit is changed in between clock pulses to ensure it is in the correct state when sampled.

b. *Define interrupt and threaded call-back in the context of an embedded system.*

If an embedded system is required to perform a specific task on an asynchronous trigger (e.g. a button push), the system could continually monitor the pin responsible for the trigger, however this takes time and would slow the system. A better way to implement this is to use an interrupt which uses a separately threaded interrupt controller. This has no effect on the main thread until an interrupt is triggered in which case it halts the main thread to perform the desired task before resuming where the program left off. A threaded call-back functions in a similar way, however the interrupt method is run in a second thread having no effect on the main thread.

c. Write a function that converts a 10-bit ADC reading from the potentiometer to a 3V3 limited voltage output.

```
def convertPot(value):
    voltage = value * (3.3/1023) # Scale ADC value to a fraction of 3.3V
    return "{:.2f} V".format(voltage)
```

Figure 2: Potentiometer conversion function.

d. Write a function that converts a 10-bit ADC reading from the temperature sensor to a reading in degree Celsius.

```
def convertTemp(value):
    # Convert to degrees
    voltage = value * (3.3/1023)
    degrees = (voltage - V0) / Tc # From datasheet
    return "{:.1f} C".format(degrees)
```

Figure 3: Temperature conversion function.

e. Write a function that converts a 10-bit ADC reading from the LDR to a percentage representing the amount of light received by the LDR.

```
def convertLDR(value):
    # Convert to percentage
    percentage = 100 * (value - LDR_MIN) / (LDR_MAX - LDR_MIN)

    if (percentage > 100):
        percentage = 100
    if (percentage < 0):
        percentage = 0

    return "{:.0f}%".format(percentage)
```

Figure 4: LDR conversion function.

f. Draw a flowchart of the system.

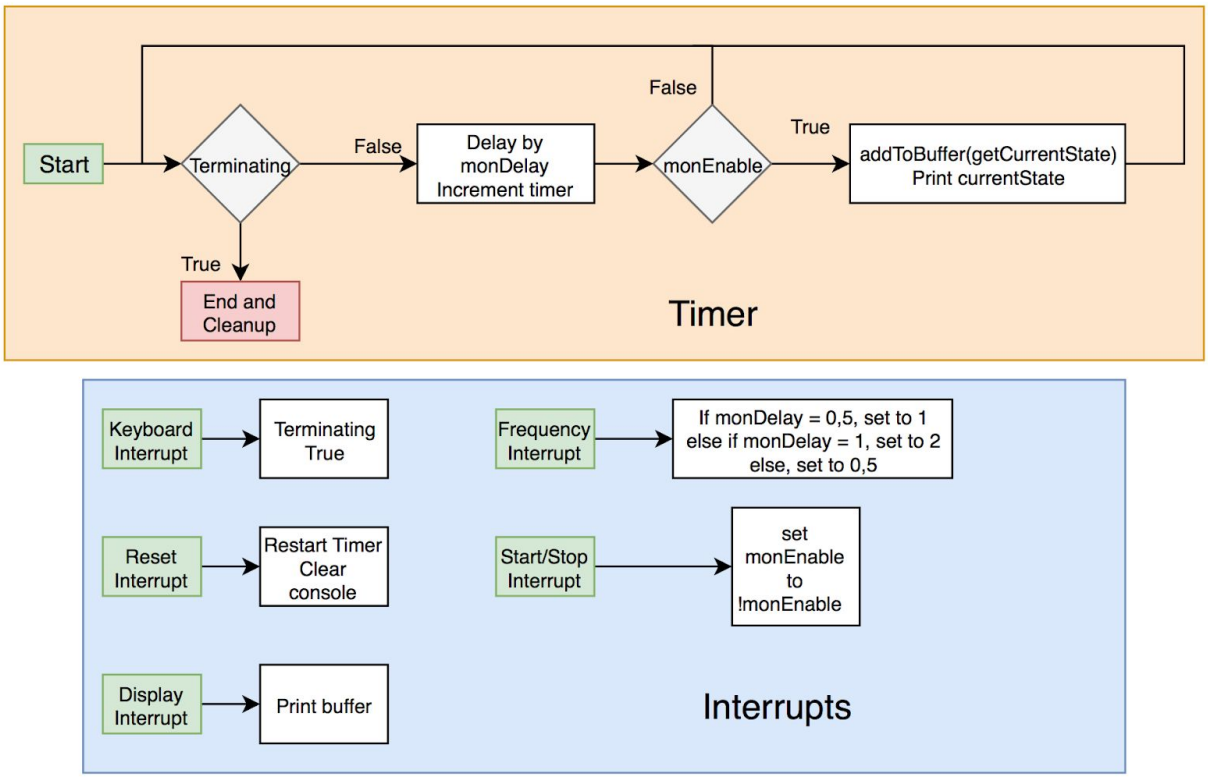

Figure 5: System flowchart.

2 - Demonstration:

a. *Default.*

```
def timer():
    if (not terminating): # Only continue if the parent thread is still running
        global upTime
        if (monEnabled):
            #print("Testing... UpTime = ", upTime) # DEBUG
            # Add store current state
            entry = getCurrentState()
            addToBuffer(entry)
            # Print the current state to the console
            print("{:<15}{:<15}{:<15}{:<15}{:<15}".format(entry[0], entry[1], entry[2], entry[3], entry[4]))
        # Start timer in new thread, delay and recall function
        threading.Timer(monDelay, timer).start()
        upTime += monDelay
```

Figure 6: Threaded timer method.

```
try:
    os.system('clear')
    print("Ready!")
    timer()
    # Keep the program running, waiting for button presses
    while(1):
        pass
except KeyboardInterrupt:
    print("losing")
    # Release all resources being used by this program
    terminating = True
    GPIO.cleanup()
```

Figure 7: Program start, and exit handler.

b. *Reset.*

```
def resetPush(channel):
    if (GPIO.input(channel) == GPIO.LOW): # Avoid trigger on button realease
        # Reset the timer
        global upTime
        upTime = 0
        # Clean the console
        os.system('clear')
        print("Reset button pushed. Timer has been reset.") # DEBUG
```

Figure 8: Reset button callback method.

c. *Frequency.*

```
def frequencyPush(channel):
    if (GPIO.input(channel) == GPIO.LOW): # Avoid trigger on button realease
        # Change the monitoring frequency
        global monDelay
        if (monDelay == 0.5): monDelay = 1
        elif (monDelay == 1): monDelay = 2
        elif (monDelay == 2): monDelay = 0.5
        # Clean the console
        os.system('clear')
        print("Frequency button pushed. Monitoring interval is now {}s.".format(monDelay)) # DEBUG
```

Figure 9: Frequency change button callback method.

d. *Stop.*

```
def stopPush(channel):
    if (GPIO.input(channel) == GPIO.LOW): # Avoid trigger on button realease
        # Start/Stop monitoring, leave timer alone
        global monEnabled
        monEnabled = not monEnabled
        # Clean the console
        os.system('clear')
        if (monEnabled): print("Start/Stop button pushed. Monitoring resumed.") # DEBUG
        else: print("Start/Stop button pushed. Monitoring stopped") # DEBUG
```

Figure 10: Stop button callback method.

e. *Display.*

```
def displayPush(channel):
    if (GPIO.input(channel) == GPIO.LOW): # Avoid trigger on button realease
        # Clean the console
        os.system('clear')
        #print("Display button pushed") # DEBUG
        # Display recent entries
        print("{:<15}{:<15}{:<15}{:<15}{:<15}".format("Time", "Timer", "Pot", "Temp", "Light"))
        for entry in readBuffer:
            # each entry is a state
            print("{:<15}{:<15}{:<15}{:<15}{:<15}".format(entry[0], entry[1], entry[2], entry[3], entry[4]))
```

Figure 11: Display button callback method.

3 - Git:

a. You must have a git repository containing meaningful commit messages. (3 per group member).

Figure 12: A selection of commits with comments (CTTKIE001).

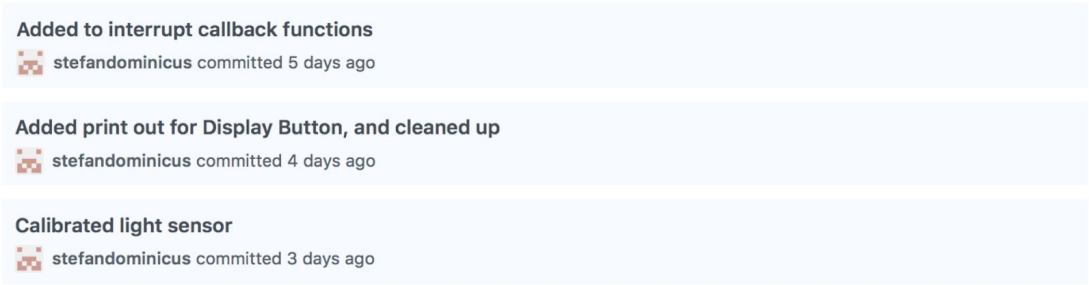

Figure 13: A selection of commits with comments (DMNSTE001).

b. Each member must work on different branches, different from the master branch.

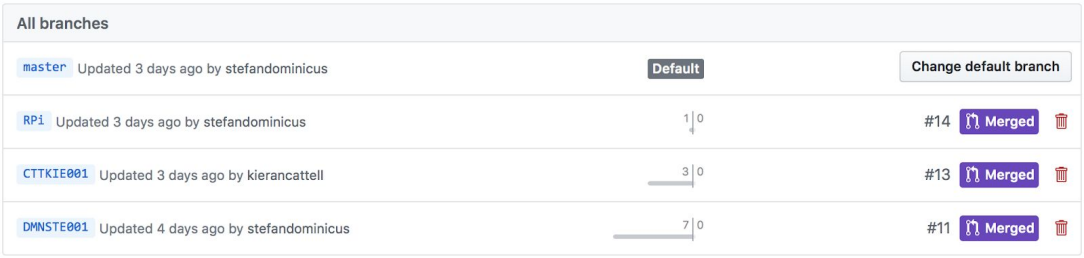

Figure 14: Screenshot showing each member's branch, as well as a branch for edits made on the RPi.

c. *Your final submission must be on your master branch.*

Figure 15: Screenshot showing the final python file in the repository (see figure 17).

d. *Each member must merge their branch to the master branch.*

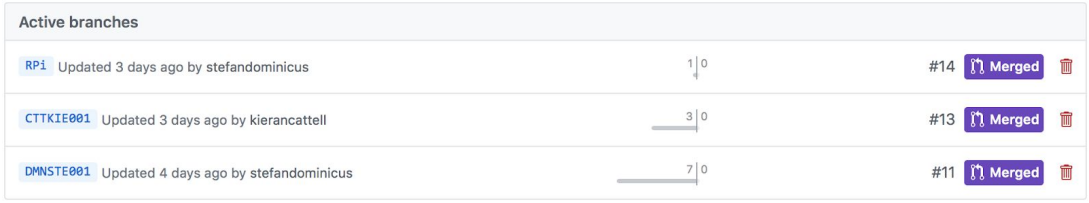

Figure 16: Screenshot showing that each member has merged their branch into the master branch.

e. *Your repository must contain any files used for the assignment.*

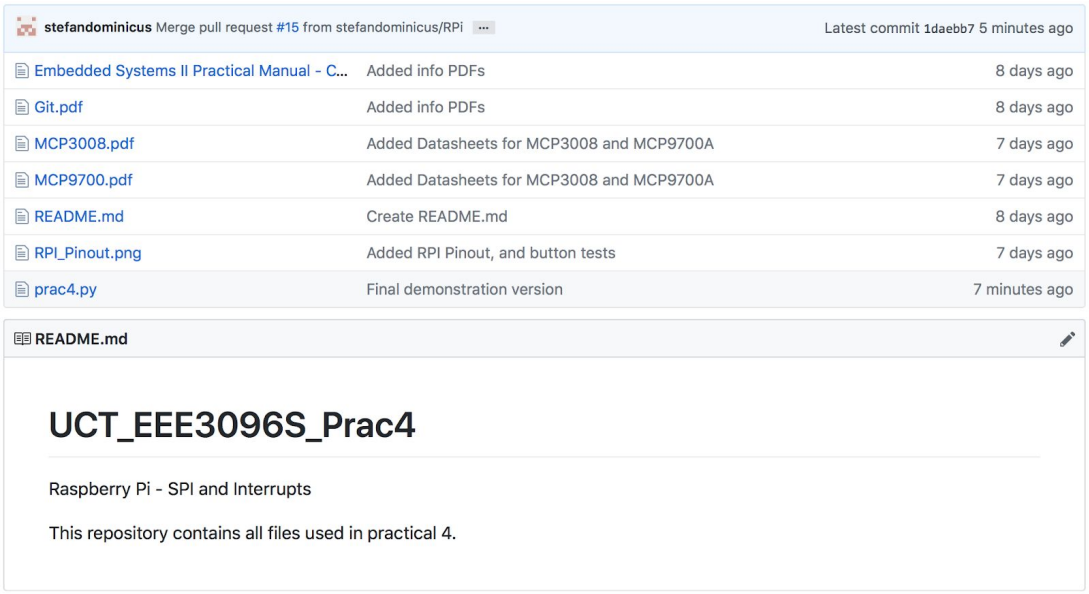

Figure 17: Screenshot showing that all files used for the prac are contained in the Github repository.