1. Circle each of the following statements which are TRUE. **[12 points, -2 per incorrectly circled/not circled answer, minimum 0]** (correct answers in red, yellow means both taken)

    a) *Source encoding* generally involves adding error correction bits and is dependent on the characteristics of the "source" data.

    b) In the USA the FCC regulates the use of the frequency spectrum. A very small percentage of the overall frequency is available for use by most consumer goods such as wireless routers.

    c) For medium-to-high power applications where there are fairly large voltage drops, a switching power supply is generally more efficient than a linear regulator, but the switching supply tends to be more noisy.

    d) Shannon's Limit provides an upper-bound on the amount of useful data we can send over a given channel. (This was meant to be true, but it should the *rate* of useful data…)

    e) Digital Signal Processors are a specialized form of a general purpose processor that is optimized for doing digital signal processing at extremely high speed, but at the cost of using a bit more power than a general purpose processor doing the same task.

    f) A buck converter which converts 20V to 15V can potentially waste less than 15% of the input power as heat, while an LDO would certainly waste more than that.

    g) A LIPO battery is being used with a constant current draw of "10C". You would expect that the battery would last at least 6 minutes, but probably not much longer.

    h) Busybox is a single Linux application which can perform the role of many standard command-line utilities such as "ls" and "pwd".

    i) Fixed-point arithmetic is generally preferred over floating-point arithmetic in embedded systems because fixed-point is easier to work with, uses less power, and can represent a more dynamic range of numbers than floating point.

    j) When designing a power distribution network, very low frequency noise (say 1KHz) is primarily handled by the power supply.

    k) When designing a power distribution network high frequency noise (say 900MHz) is primarily handled by the power ground plane / bulk capacitors / parasitic inductors.

    l) 16-QAM sends 256 bits of information in a single encoding.

2. For most forms of wireless communication, we communicate using sinusoids at some fixed frequency and then vary the phase and amplitude of the signal. **[6 points]**
   - Why do we use a sinusoid rather than some other function (say a square wave for example)? **[4]**

     Lots of valid answers here. The big one is the fact that sinusoids only generate power at one frequency and so will interfere less with other frequencies than a squarewave or something else might. It is incorrect to say that a squarewave can't have frequency, phase, or amplitude (which a lot of people said).

   - If we instead vary only the frequencies over a handful of specific frequencies, that is called [✓ ***Frequency Shift Keying***] / ***Frequency Modulation*** / ***Frequency Table*** / ***Local QAM.*** (circle one answer) **[2]**

3. Say you have a hamming(7,4)-code where you have 4 data elements (d_1 to d_4) followed by 3 parity bits (p_1 to p_3) where the parity bits are defined as follows (the function P() returns the even one's parity as we did in class, so e.g., P(1,1,1) =1 and P(0,1,1)=0.
   - $p_1=P(d_2, d_3, d_4)$
   - $p_2=P(d_1, d_2, d_4)$
   - $p_3=P(d_1, d_2, d_3)$

   [6 points]

   a) If $d[1:4]=1001_2$, what should be the value of p[1:3]? **[2]**

   P_1 = 1 P_2 = 0 P_3 = 1

   b) Assuming no more than one bit of data will go bad at a time, if the receiver receives the packet "1100111" what data bit, if any, is in error? Justify your answer. **[4]**

   Would expect p[1:3]=100, instead got 111. So p2 and p3 and wrong. And the only way that could happen with a single bit flip is if d1 were wrong. So **0100111**

4. Consider the following code found as the read function member of the file_operations struct for a Linux kernel module. It is associated with the device file "/dev/txx2" (so a read of the file /dev/txx2 will result in this function being called). Assume that everything is set up appropriately beforehand and that count will be 1024 any time the function is called.

```
ssize_t memory_read(struct file *filp, char *buf,
                   size_t count, loff_t *f_pos) {

      /* Transferring data to user space */
      int x=count;
      if(x>5)
            x=5;
      copy_to_user (buf,"abcdefghijk",x);
      printk("<1> fpos= %d\n",*f_pos);
      /* Changing reading position as best suits */
      if (*f_pos <10) {
         *f_pos+=x;
         return x;
      } else {
         return 0;
      }
}
```

Now say the user types "cat /dev/txx2". What will happen? In addition to filling in the boxes below indicating what's on the screen and what's in the log file, but sure to clearly explain why. **[12 points]**

Cat calls the read function continuously until the function returns 0. Due to this memory_read gets called 3 times but on the 3rd time nothing is printed out to the terminal since 0 was returned as the number of characters read.

Appears on the screen:

ABCDEABCDE

Appears in the log file:

Fpos = 0

fpos = 5

fpos = 10

5. Answer the following questions about fixed-point numbers. For purposes of this question, assume that “largest value” means “closest to positive infinity” and “smallest value” means “closest to negative infinity”. **[14 points]** **Question was clarified during the exam to be for signed numbers. Which is what Q numbers generally are, but we clarified anyways.**

- What is the largest *value* that can be represented as an 8-bit Q5 number? Be exact, you can give your answer in decimal or as a fraction. **[2]**

  3 31/21

- What is the binary representation for the largest value that can be represented as an 8-bit Q5 number? **[2]**

  01111111

- What is the smallest *value* that can be represented as an 8-bit Q5 number? Be exact, you can give your answer in decimal or as a fraction. **[2]**

  -4

- Write a function named q5mult, which takes in two signed 8-bit Q5 numbers as input and outputs a signed 8-bit Q5 which is the product of the two inputs. If the exact result cannot be represented either due to overflow (the value is outside the range of representation) or due to precision/rounding issues, you should set the output to the nearest output. **[8]**

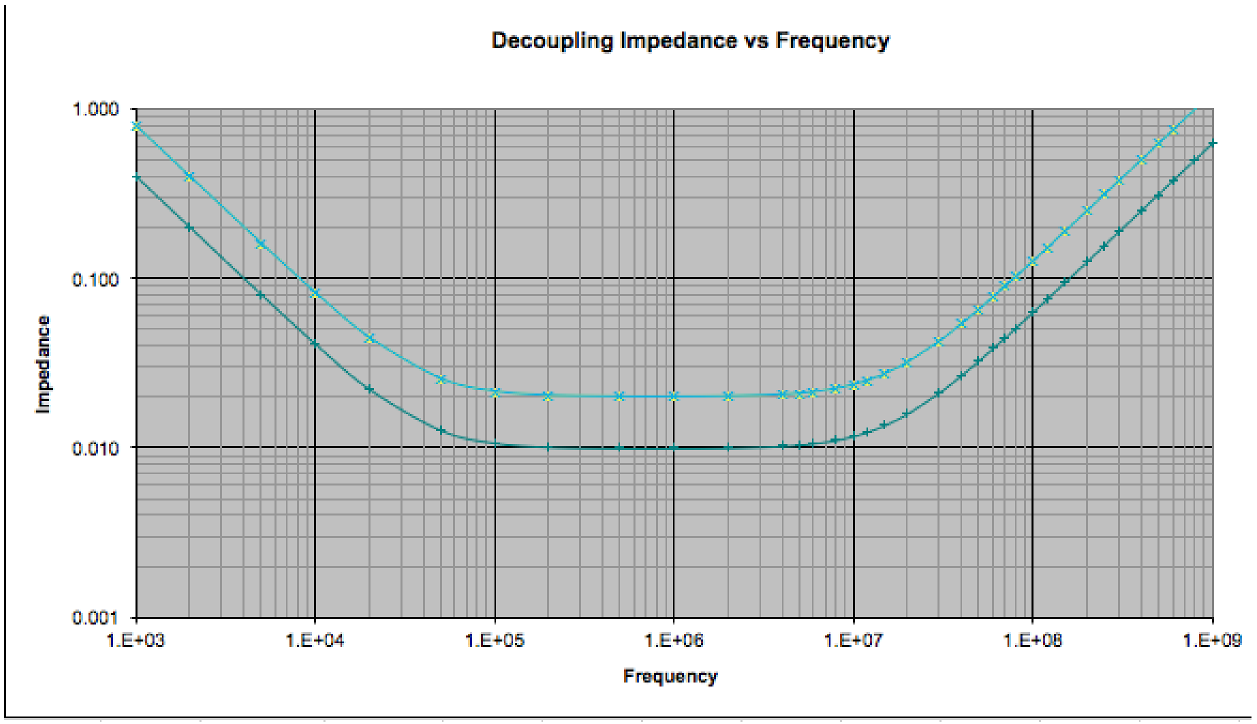
```
char q5mult(char a, char b) {
      short res = (a*b + 0x10) >> 5;
      if (res > 127)
            return 127;
      if (res < -128)
            return -128;
      return res;
}
```

6. Consider a 2.4GHz radio/receiver pair where the transmitter broadcasts at 20mW and uses an antenna with a gain of 4x over an isotropic antenna. The receiver has a sensitivity of -85 dBm and uses the same antenna as the transmitter. What is the expected range? Show your work **[5]**

dBm	mW	dBm	mW	dBm	mW
-3	0.5	9	8	21	126
-2	0.6	10	10	22	158
-1	0.8	11	13	23	200
0	1.0	12	16	24	250
1	1.3	13	20	25	316
2	1.6	14	25	26	398
3	2.0	15	32	27	500
4	2.5	16	40	28	630
5	3.2	17	50	29	800
6	4	18	63	30	1000
7	5	19	79	33	2000
8	6	20	100	36	4000

$$r = \frac{10^{(p_t + g_t + g_r - p_r)/20}}{41.88 \times f}$$

$$\frac{10^{(13+6+6+85)/20}}{41.88 * 2400} = 3.146\ km$$



7. The above graph shows the frequency vs. impedance for a given capacitor. Redraw the graph showing the same information for 2 of these capacitors in parallel. **[5 points]**

Design Problem: Engine Air Intake Control System

(You will want to read the whole problem before starting)

You are given the task to control the air flow of an automotive intake system by monitoring pressure and temperature and then adjusting a flow control valve accordingly.

- In particular, if the engine pressure is above 180psi OR the temperature is above 220 degrees F the control voltage should be set to 3.0V, otherwise that voltage should be set to 1.0V.

That task must be done every 10ms in order to keep the engine running correctly. In addition, there is a device ("Pressure Level Detect Logic") that continuously monitors the pressure. If it ever exceeds some fixed threshold you need to capture the temperature and pressure values immediately and start a check engine process. That process takes a long time to run, but is not essential to keep the engine running.

This experimental system uses a Raspberry Pi (RPI) running FreeRTOS. You will need to create a task that runs every 10ms to control the pressure. There needs to be another low-priority task to handle the engine check process.

There are a number of parts to this problem.

1. Connecting the various components.
2. Write a function that uses the ADC to get the pressure and temperature.
3. Write a function that writes a DAC to control the valve.
4. Write a task that reads the sensors and adjusts the control valve every 10ms using the functions written in parts 2 and 3.
5. Write a low-priority task that executes when the interrupt function tells it to
6. Create an interrupt function that starts the check engine task, reads the pressure and temperature values, and places those two values in global variables.

We have supplied a main program with the references. Again, we recommend you read over the entire problem before beginning.

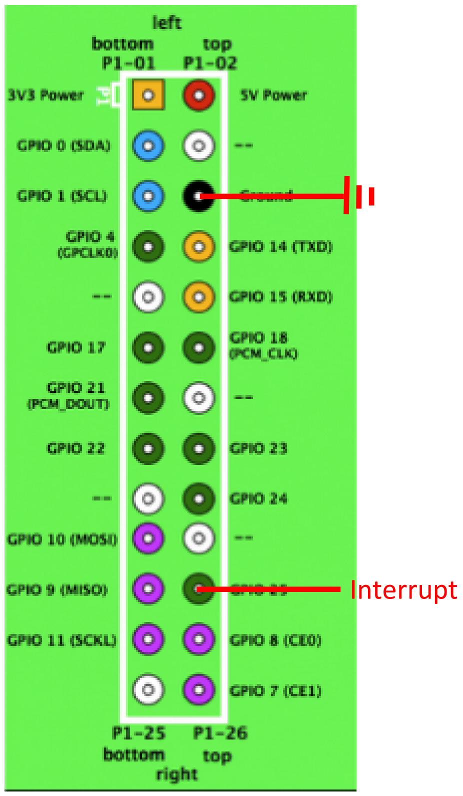
Processor, Sensor and Detector Characteristics

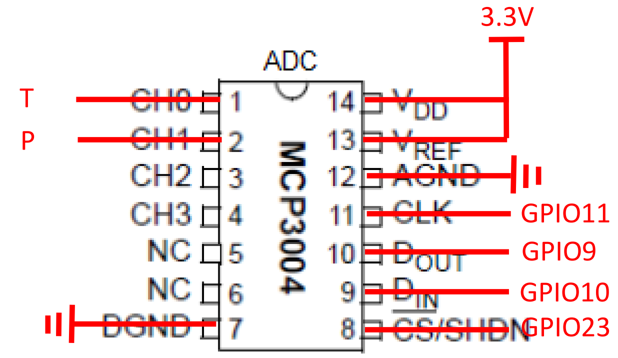
- **RPI:** RTOS tick is every 1ms.
- **Temperature Sensor:** Provides a voltage proportional to temperature.
- **Pressure Sensors**: Provides voltage proportional to pressure.
- **ADC**: Four channel 10 bit ADC with SPI interface. Data sheet excerpts in appendix.
- **DAC**: Two channel 8 bit DAC with SPI interface. Data sheet excerpts in appendix.
- **Pressure Level Detect Logic**: Is provided.

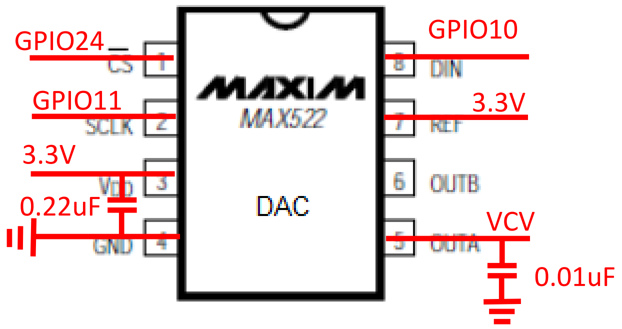
One of the references in the question listed a voltage reference to be 1.024V rather than 3.3V. That error was fixed during the exam.

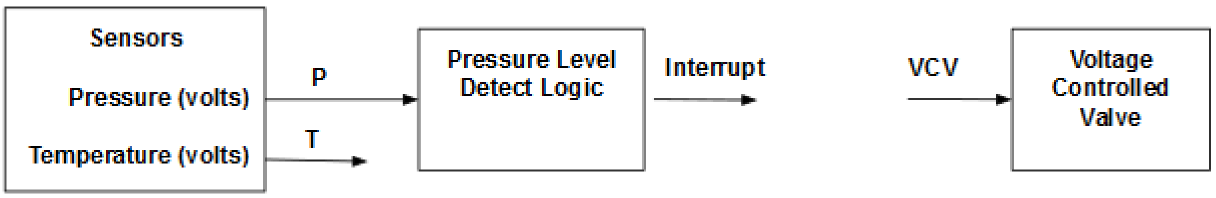
Part 1: Connections [7 points]

Connect the ADC, DAC and pressure detect/interrupt detection circuit to the RPI. Use GPIOs 23 for the MCP3004 ADC SPI select, GPIO 24 for the MAX522 DAC SPI select and GPIO 25 for the interrupt. Assume all the devices run on 3.3 volts and that the ADC and DAC reference voltages are also 3.3V. Pinouts and SPI interface details are provided in an appendix at the end of the test. CHO and CH1 of the MCP3004 will be used to read the temperature and pressure sensors respectively. OUTA of the MAX522 will be used to control the valve.









Part 2: Read Pressure and Temperature [8 points]

For this part you need to read the temperature and pressure using the ADC. The function is to provide the 10-bit values read from the temperature and pressure sensors as values pointed to by the function parameters.

```
void get_engine_values(unint32 *temp, uint32 *pressure);
```

SPI function

The following SPI function is available. You can assume the clock speed, mode, and frame size are already initialized and this SPI device is a master. Remember, SPI devices are full duplex. That is they send and transmit at the same time.

```
void spi_transaction(uint32 transmit frame, uint32* data,
                     char length)
```

- **transmit frame**: A 32-bit unsigned integer that contains the data you wish to transmit
- **data**: Is a pointer to a 32-bit unsigned integer where the received data will be placed
- **length:** is the number of bytes to transmit/receive. It must be either 1, 2, 3 or 4.

Sensor and control voltage formula

- The temperature sensor has a range from 0 degrees to 660 degrees Fahrenheit with 0.2 degrees/mV.
- The pressure sensor has a range from 0 to 330 psi 0.1psi/mV.

GPIO function

The following function can be used to set the GPIO to low or high.

```
void set_gpio(uint8 gpio_number, uint8 value);
```

Where **gpio_number** is the gpio pin number to be set and **value** is the value it is set to.

(Part 2 continued)

Write the C code for your API below. Please don’t provide answers on the back.

```
void get_engine_values(unint32 *temp, uint32 *pressure) {
     uint32_t frame1 = 0x018000;
     uint32_t frame2 = 0x019000;
     set_gpio(23,0);
     spi_transaction(frame1, temp, 3);
     set_gpio(23,1);
     set_gpio(23,0);
     spi_transaction(frame2, pressure, 3);
     set_gpio(23,1);
}
```

Part 3: Write DAC for Valve Control Voltage API [6 points]

For this part you need to write a function that will provide the voltage for the control valve. The function prototype follows with the control voltage passed as an unsigned 32-bit integer with the value in millivolts (mV).

```
void set_valve_voltage(uint32 mV){

        uint32_t frame = 0x2100;

        uint32_t data;

        set_gpio(24,0);

        spi_transaction(frame | (mV * 255/3300), &data, 2);

        set_gpio(24,1);

}
```

Grading comment: a lot of people did something like 256/3300*mV which would be wrong. Doesn't integer math in C just make you so happy? No, we didn't take off much for that (.25 points in fact). But people really did have problems scaling the voltage (not just integer math). People largely got the 0x2100, but struggled with lots of other things. Also, this version doesn't have rounding to the nearest value, which might be a fine idea.

Part 4: Valve Task [7 points]

Provide a task that will run every 10ms using the functions you wrote above to set the valve's input correctly. The specification for what value the valve input should be set to are found on the first page of the design problem.

```
Void task1(void *pvParameters) {
    TickType_t xLastWake;
    const TickType_t xFrequency = 10;
    uint32_t pres, temp;
    xLastWake = xTaskGetTickCount();
    for(;;) {
        vTaskDelayUntil(&xLastWake, xFrequency);
        get_engine_values(&temp, pres);
        if (temp*0.2 * 1024/3300 > 220 || pres * 0.1* 1024/3300) {
            write_control_valve(3000);
        } else {
            write_control_valve(1000);
        }
    }
}
```

Part 5: Interrupt [6 points]

For this part you need to read the pressure and temperature, storing the values in the globals TEMP and PRES. You then need to signal the check engine task to start.

```
void rupt(int num, void* parm){
        get_control_values(&TEMP, &PRES); //Used global variables defined in reference

        xSemaphoreGiveFromISR(lock1, false);

}
```

Part 6: Check Engine Task [6 points]

For this part you will write the low-priority task that will only run when signaled to do so by the interrupt handler you wrote in part 5. The function you are to invoke is `do_engine_check()`. It has a prototype as follows:

```
void do_engine_check(uint32 temperature, unit32 pressure)
```

This function may take a long time (100ms or more) to run. It takes the temperature (in degrees Fahrenheit) and the pressure (in psi) as arguments. Those should be the values as they existed as close as possible to the event that triggered the interrupt.

```
Void task2 (void *pvParameters) {
        for(;;) {
                xSemaphoreTake(lock1, portMAX_DELAY);
                do_engine_check(TEMP * 0.2 * 1024/3300, PRES * 0.1 * 1024/3300);
        }
}
```