EECS 473 Final Exam

Fall 2016

Name: ______________________________________ unique name: ____________________

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

____

NOTES:

1. Closed book and Closed notes
2. <u>Do not write anything you want graded on the back pages of the exam</u>.
3. There are <u>**14**</u> pages total for the exam. There is also a references handout.
4. Calculators are allowed, but no PDAs, Portables, Cell phones, etc. Using a calculator to store notes is not allowed nor is a calculator with any type of wireless capability.
5. You have about 120 minutes for the exam.

Be sure to show work and explain what you've done when asked to do so. That may be very significant in the grading of this exam.

1. Circle each of the following statements which are TRUE. **[12 points, -2 per incorrectly circled/not circled answer, minimum 0]**

   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~~ rate of useful data we can send over a given channel. (The struck part was an error that wasn’t addressed during the exam. We took all answers.)

   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~~. (The struck part was an error that wasn’t addressed during the exam. We took all answers.)

   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]**
   a. Why do we use a sinusoid rather than some other function (say a square wave for example)? **[4]**

   b. If we instead vary only the frequencies over a handful of specific frequencies, that is called ***<u>Frequency Shift Keying / Frequency Modulation / Frequency Table / Local QAM</u>***. (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]**

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

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]**

Appears on the screen:	Appears in the log file:

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”. Assume all values are signed. **[14 points]**
   a. 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]**

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

   c. 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]**

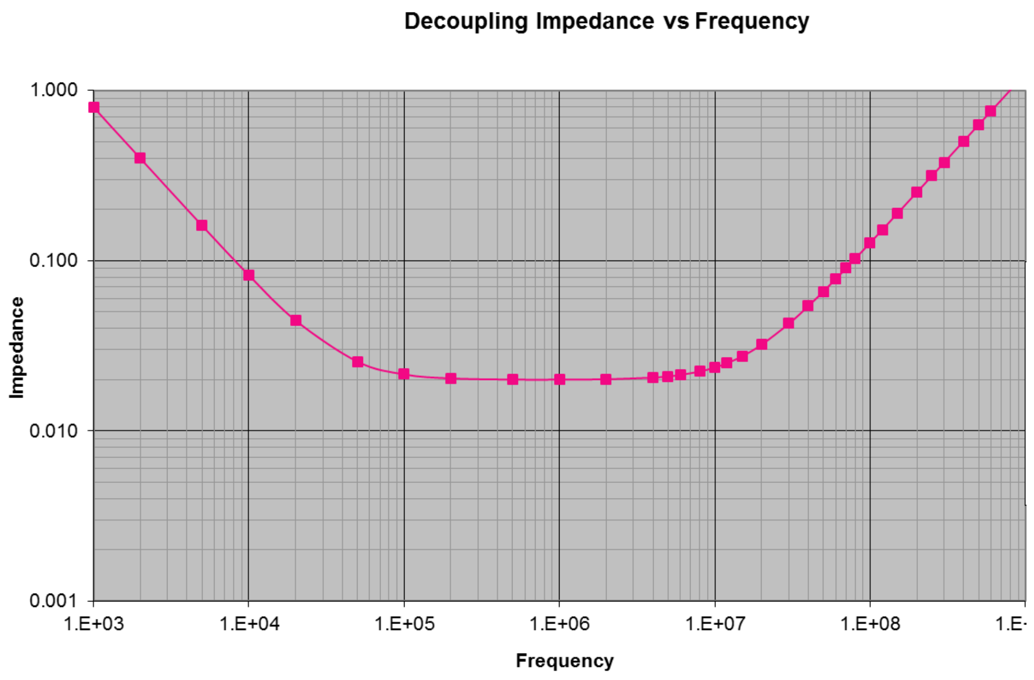
   d. Write a function named q5mult, which takes in two signed 8-bit Q5 numbers as input and outputs a signed 8-bit Q5 which is the product of the two inputs. If the exact result cannot be represented either due to overflow (the value is outside the range of representation) or due to precision/rounding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

```
char q5mult(char a, char b) {
```

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

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



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.

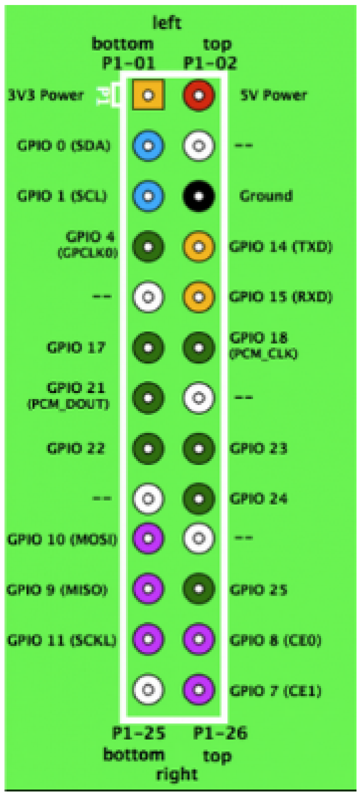
We have supplied a main program with the references. Again, we recommend you read over the entire problem before beginning.

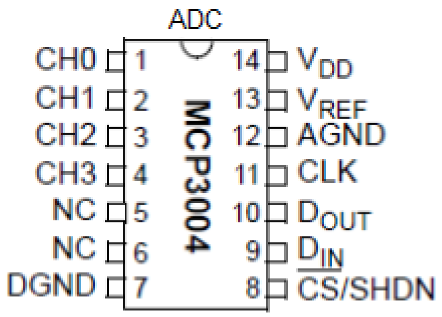
Processor, Sensor and Detector Characteristics

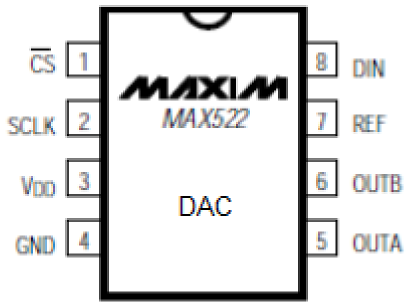
- **RPI:** RTOS tick is every 1ms.
- **Temperature Sensor:** Provides a voltage proportional to temperature.
- **Pressure Sensors**: Provides voltage proportional to pressure.
- **ADC**: Four channel 10 bit ADC with SPI interface. Data sheet excerpts in appendix.
- **DAC**: Two channel 8 bit DAC with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

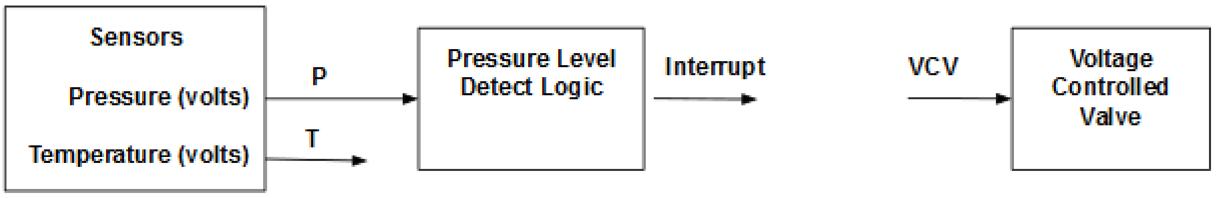
Part 1: Connections [7 points]

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) {
```

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){
```

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.

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){
```

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.

References

MAIN (provided)

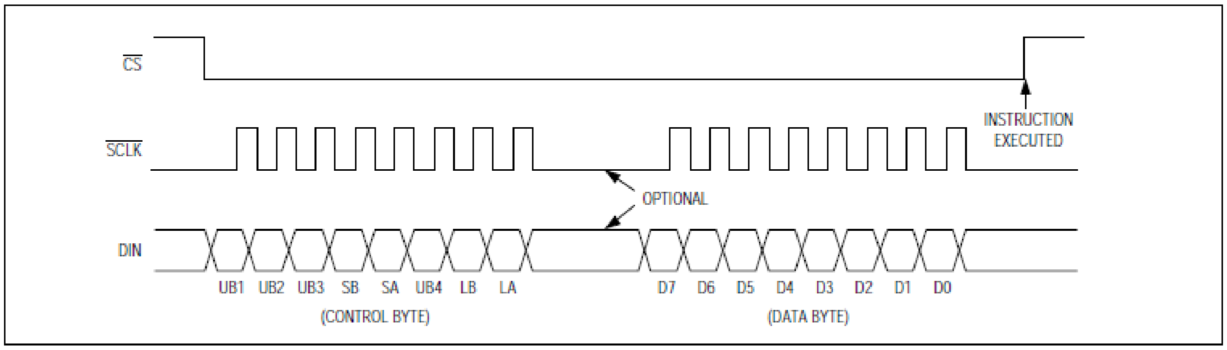
```
#define IRQn 49       // Interrupt number (pin 25)
#define IRQ_TYPE DETECT_RISING   // Detection type
xSemaphoreHandle lock1;
uint32_t TEMP, PRES;

void main(void) {
      DisableInterrupts();
      InitInterruptController();
      RegisterInterrupt( IRQn, rupt, NULL );
      EnableGpioDetect( INT_PIN, IRQ_TYPE );
      ConfigureGPIO();      // configures GPIO pins (direction etc.)
      ConfigureADC();       // configures ADC (including turning it on)
      ConfigureDAC();       // configures DAC (including turning it on)
      vSemaphoreCreateBinary (lock1);
      xTaskCreate(task1, "voltage control", 128, NULL, 2, NULL);
      xTaskCreate(task2, "check engine", 128, NULL, 1, NULL);
      EnableInterrupt( IRQn );
      EnableInterrupts();
      vTaskStartScheduler();
      while(1) {};

}
```

MAX522 DAC Details

- It is a 8 bit device with a reference voltage of 3.3 V.
- It has two DACs. Use DACA to control the valve.
- It is a SPI interface with following transaction example



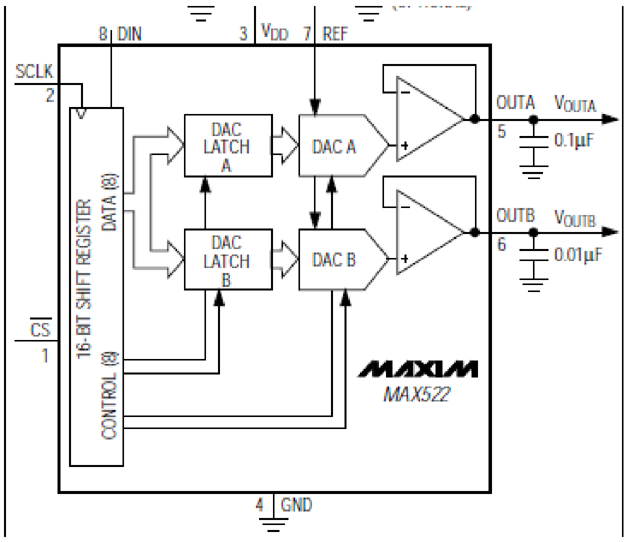
CONTROL								DATA								FUNCTION
UB1	UB2	UB3	SB	SA	UB4	LB	LA	B7 MSB	B6	B5	B4	B3	B2	B1	B0 LSB	
X	X	1	*	*	0	0	0	X	X	X	X	X	X	X	X	No Operation to DAC Registers
X	X	1	*	*	0	0	0									Unassigned Command
X	X	1	*	*	0	1	0	8-Bit DAC Data								Load Register to DAC B
X	X	1	*	*	0	0	1	8-Bit DAC Data								Load Register to DAC A
X	X	1	*	*	0	1	1	8-Bit DAC Data								Load Both DAC Registers
X	X	1	0	0	0	*	*	X	X	X	X	X	X	X	X	All DACs Active
X	X	1	0	0	0	*	*	X	X	X	X	X	X	X	X	Unassigned Command
X	X	1	1	0	0	*	*	X	X	X	X	X	X	X	X	Shut Down DAC B
X	X	1	0	1	0	*	*	X	X	X	X	X	X	X	X	Shut Down DAC A
X	X	1	1	1	0	*	*	X	X	X	X	X	X	X	X	Shut Down All DACs

X = *Don't care.*
* = *Not shown, for the sake of clarity. The functions of loading and shutting down the DACs and programming the logic can be combined in a single command.*

Notes:

- **Assume the * values are 0.**
- **Set the don't care values (X's) to 0.**

MAX522 DAC Pinout and Functional Diagram



PIN	NAME	FUNCTION
1	$\overline{CS}$	Chip Select (active low). Enables data to be shifted into the 16-bit shift register. Programming commands are executed at the rising edge of $\overline{CS}$.
2	SCLK	Serial Clock Input. Data is clocked in on the rising edge of SCLK.
3	V_{DD}	Positive Power Supply (2.7V to 5.5V). Bypass with 0.22µF to GND.
4	GND	Ground
5	OUTA	DAC A Output Voltage (Buffered). **Connect 0.1µF capacitor or greater to GND.**
6	OUTB	DAC B Output Voltage (Buffered). **Connect 0.01µF capacitor or greater to GND.**
7	REF	Reference Input for DAC A and DAC B
8	DIN	Serial Data Input of the 16-bit shift register. Data is clocked into the register on the rising edge of SCLK.

MCP3004 ADC Details

- It is a 10 bit device with a reference voltage of ~~1.024 volts or 1 mv/LSB~~. (the struck part was an error addressed during the exam)
- It has an analog multiplexer allowing for the selection of one of for inputs: CH0, CH1, CH2 and CH3.
- Only one channel can be converted at a time.
- It is a SPI interface with following transaction example.

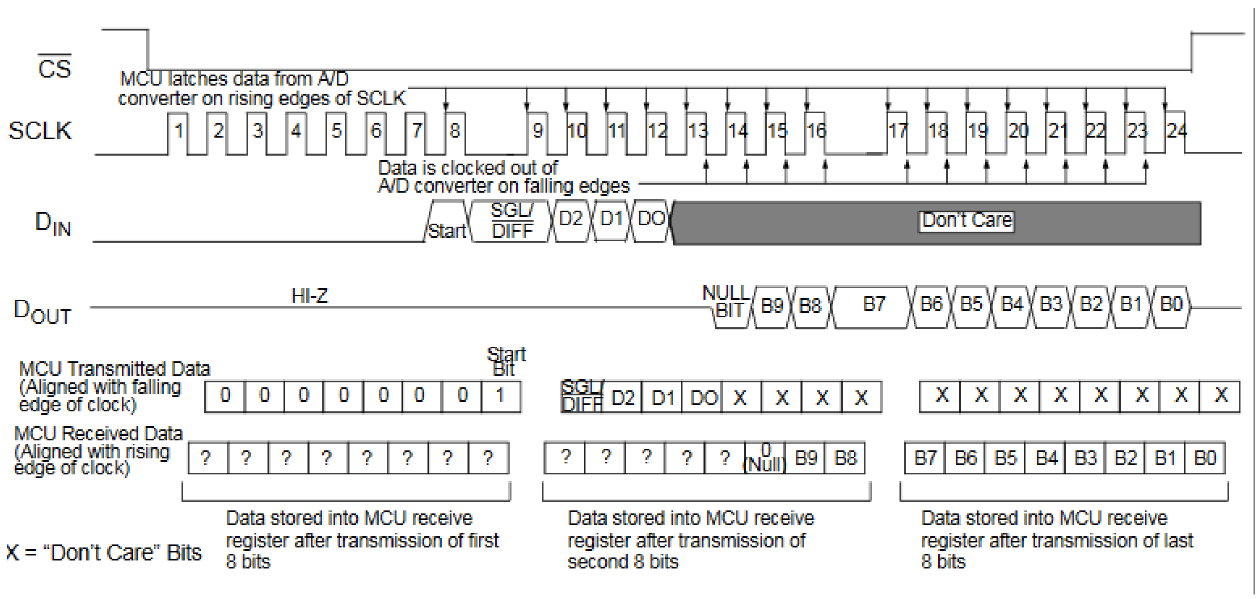

CS
SCLK
MCU latches data from A/D converter on rising edges of SCLK
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Data is clocked out of A/D converter on falling edges
DIN
Start SGL/DIFF D2 D1 DO
Don't Care
DOUT
HI-Z
NULL BIT B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
MCU Transmitted Data (Aligned with falling edge of clock)
Start Bit
0 0 0 0 0 0 0 1
SGL/DIFF D2 D1 DO X X X X
X X X X X X X X
MCU Received Data (Aligned with rising edge of clock)
? ? ? ? ? ? ? ?
? ? ? ? ? 0 (Null) B9 B8
B7 B6 B5 B4 B3 B2 B1 B0
Data stored into MCU receive register after transmission of first 8 bits
Data stored into MCU receive register after transmission of second 8 bits
Data stored into MCU receive register after transmission of last 8 bits
X = "Don't Care" Bits

TABLE 5-1: CONFIGURE BITS FOR THE MCP3004

Control Bit Selections				Input Configuration	Channel Selection
Single/ Diff	D2*	D1	D0		
1	X	0	0	single-ended	CH0
1	X	0	1	single-ended	CH1
1	X	1	0	single-ended	CH2
1	X	1	1	single-ended	CH3
0	X	0	0	differential	CH0 = IN+ CH1 = IN-
0	X	0	1	differential	CH0 = IN- CH1 = IN+
0	X	1	0	differential	CH2 = IN+ CH3 = IN-
0	X	1	1	differential	CH2 = IN- CH3 = IN+

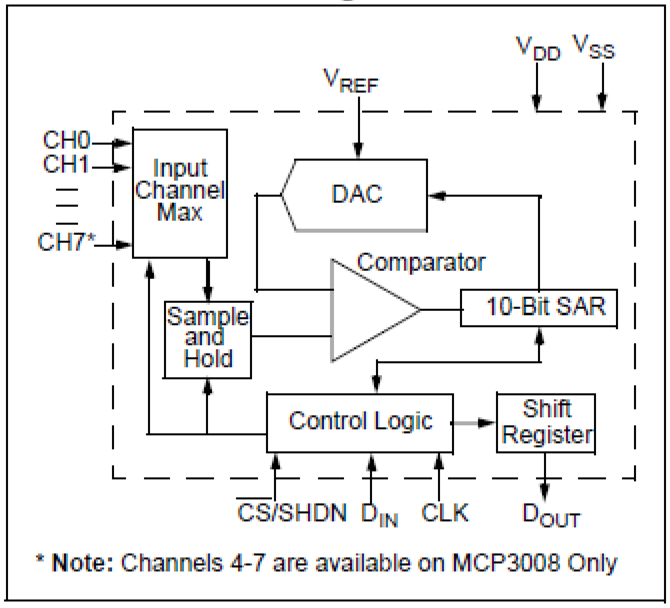
* D2 is "don't care" for MCP3004

Some Configuration Notes:

- Assume we are using Single-ended mode.
- Temperature is on CH0.
- Pressure is on CH1.
- CS is the SPI chip select line.

- **<u>Assume the * values are 0.</u>**
- **<u>Set the don't care values (X's) to 0.</u>**

MPC3004 ADC Pinout and Functional Diagram



*** Note:** Channels 4-7 are available on MCP3008 Only

MCP3004	MCP3008	Symbol	Description
PDIP, SOIC, TSSOP	PDIP, SOIC		
1	1	CH0	Analog Input
2	2	CH1	Analog Input
3	3	CH2	Analog Input
4	4	CH3	Analog Input
–	5	CH4	Analog Input
–	6	CH5	Analog Input
–	7	CH6	Analog Input
–	8	CH7	Analog Input
7	9	DGND	Digital Ground
8	10	$\overline{CS}$/SHDN	Chip Select/Shutdown Input
9	11	D_{IN}	Serial Data In
10	12	D_{OUT}	Serial Data Out
11	13	CLK	Serial Clock
12	14	AGND	Analog Ground
13	15	V_{REF}	Reference Voltage Input
14	16	V_{DD}	+2.7V to 5.5V Power Supply
5,6	–	NC	No Connection

Sample FreeRTOS functions and examples.

```
// Perform an action every 10 ticks.
 void vTaskFunction( void * pvParameters )
 {
 TickType_t xLastWakeTime;
 const TickType_t xFrequency = 10;

     // Initialise the xLastWakeTime variable with the current time.
     xLastWakeTime = xTaskGetTickCount();

     for( ;; )
     {
         // Wait for the next cycle.
         vTaskDelayUntil( &xLastWakeTime, xFrequency );

         // Perform action here.
     }
 }
```

```
xSemaphoreGiveFromISR
      (
        SemaphoreHandle_t xSemaphore,
        signed BaseType_t *pxHigherPriorityTaskWoken
      )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.
Parameters:

xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.
pxHigherPriorityTaskWoken	xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

Returns:
pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(), and obtained using sSemaphoreTake().

This must not be used from an ISR. See xSemaphoreGiveFromISR() for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Parameters:

xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns:

pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

```
xSemaphoreTakeFromISR
      (
        SemaphoreHandle_t xSemaphore,
        signed BaseType_t *pxHigherPriorityTaskWoken
      )
```

A version of xSemaphoreTake() that can be called from an ISR. Unlike xSemaphoreTake(), xSemaphoreTakeFromISR() does not permit a block time to be specified.

Parameters:

xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.
pxHigherPriorityTaskWoken	It is possible (although unlikely, and dependent on the semaphore type) that a semaphore will have one or more tasks blocked on it waiting to give the semaphore. Calling xSemaphoreTakeFromISR() will make a task that was blocked waiting to give the semaphore leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE.

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                 TickType_t xTicksToWait );
```

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

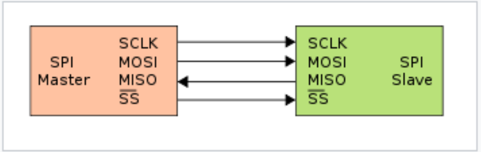
This macro must not be called from an ISR. xQueueReceiveFromISR() can be used to take a semaphore from within an interrupt if required, although this would not be a normal operation. Semaphores use queues as their underlying mechanism, so functions are to some extent interoperable.
Parameters:

xSemaphore	A handle to the semaphore being taken - obtained when the semaphore was created.
xTicksToWait	The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if xTicksToWait expired without the semaphore becoming available.

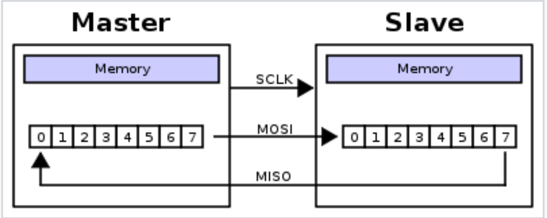
The **Serial Peripheral Interface** (**SPI**) bus is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The interface was developed by Motorola in the late eighties and has become a *de facto* standard. Typical applications include Secure Digital cards and liquid crystal displays.



Single Master to Single Slave : basic SPI bus example.

SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave devices are supported through selection with individual slave select (SS) lines.

Data transmission [edit]



A typical hardware setup using two shift registers to form an inter-chip circular buffer

To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. The master then selects the slave device with a logic level 0 on the select line. If a waiting period is required, such as for analog-to-digital conversion, the master must wait for at least that period of time before issuing clock cycles.

During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended.

Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a virtual ring topology. Data is usually shifted out with the most-significant bit first, while shifting a new least-significant bit into the same register. At the same time, Data from the counterpart is shifted into the least-significant bit register. After the register bits have been shifted out and in, the master and slave have exchanged register values. If more data needs to be exchanged, the shift registers are reloaded and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.

Transmissions often consist of 8-bit words. However, other word sizes are also common, for example, 16-bit words for touchscreen controllers or audio codecs, such as the TSC2101 by Texas Instruments, or 12-bit words for many digital-to-analog or analog-to-digital converters.

Every slave on the bus that has not been activated using its chip select line must disregard the input clock and MOSI signals, and must not drive MISO.