# ECGR 4101/5101, Spring 2012: Lab 6

High Speed UART, PWM, Microphone, and Streaming.

## Learning Objectives:

This lab will test your ability to apply the knowledge you have obtained from the last five labs and expand that knowledge further by introducing you to atypical UART speeds, the on board microphone, PWM signal generation, time critical UART communications, and a few DSP concepts.

This lab like all the labs before it expects that you are very comfortable with Lab 1 thru 5 and some steps will be skipped as it is expected that you already know how to do this.

Note: you should be creating a code repository of all of your past projects to allow you to quickly implement similar functionality in future projects.

Additionally, this lab will not provide as much step by step detail as lab five did so be prepared to spend additional time researching datasheets, reading over your class notes, and using the age old process of trial and error. I also recommend that you get acquainted with the HEW debugger as being able to add breakpoints and variable watches is very useful when trying to figuring out why your code isn't working correctly.

## General Information:

1. Review the steps provided by the lab 1 supplemental information document
2. Review the steps provided by the lab 2/3/4/5 Activity
3. Copy the necessary files into your new workspace and import them
4. Import the necessary mathematical and string libraries (string.h or math.h) if required
5. Implement the lab requirements.
6. Demonstrate your working project to the TA, and turn in a lab report.

## Prelab Activity:

You may use the PCs in Woodward 203 or your own PC to do this lab experiment. If you want to work on lab assignments on your own PC, then you will need to load the necessary tools on your PC in order to perform this exercise.

In this lab you will need to create a new project called lab 6 that requires the custom LCD functions, CMT and TMR Timers, ADC, Queue, and UART functions from your prior labs.

You will also need to review your class notes, the Renesas API document, and possibly look at the available sample applications throughout this lab.

## Some Helpful Advice:

Start on this ASAP!

## Prelab Questions:

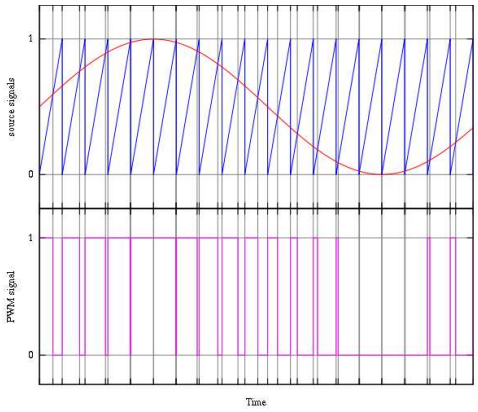
1. What is SCI2.SEMR.BIT.ABCS? And what effect occurs as a result of changing this value?
2. What are the equations needed to find the value of SCI2.BRR (keep question 1 in mind)
3. What are the error equations for the value of SCI2.BRR (keep question 1 in mind)
4. Based on your answers from 1 2 and 3 use Matlab or a similar program to determine valid SCI2.BRR values that have an absolute error less than 1 percent for baud rates between 2400 to the maximum asynchronous baud rate listed in the datasheet in increments of 2400*n

   Your table might look something like this but longer (Note values above 1 percent error are not listed)

| Baud Rate | BRR | % Error <1% |
|---|---|---|
| 2400 | 1302 | -0.0703504732668225 |
| 4800 | 651 | -0.146983640081799 |
| 7200 | 434 | -0.22349936143039 |

5. From 4 determine the required modifications needed to your UART code from the last lab to increase the Baud rate to 626401, also determine the chance of error and if this communication rate is possible.
6. Look at the RX Development board datasheet and determine what pin the Microphone is connected at and the function of the AMP_SHDN pin.
7. Look at the RX Development board datasheet and determine what pin the Left and Right speaker jack is connected too. Also determine what PWM peripheral is mapped to those pins.

## Laboratory Assignment Overview:




## Audio Signal Processing

In the last two labs we explored UART communication and various applications in which UART can be utilized and this lab will further your education about UART along with introduce a new topic of Pulse Width Modulation (PWM).

Since PWM is a key topic in this lab, lets being by quickly summarizing that PWM is simply the process of creating an average time signal by switching between two states (On or Off) at a given rate.

Because physical systems cannot react instantly to stimulation they tend to ignore rapid changing transients and respond to a net average signal.

Typically this physical characteristic is modeled as a LPF (Low Pass Filter) in electronics or mathematically as an integration over time operation which implies that the higher frequency transition between (ON or OFF) is filtered out and the average value from the transitions will remain.

As long as the PWM signal is periodic the average value should remain static once transient effects have subsided and this aspect is typically used to create defined DC voltage values that are used in numerous applications.

Typically a PWM voltage signal is generated by comparing a desired average voltage signal with a periodic ramp function and when the ramp is below the desired signal the PWM output is high and when the ramp is above the desired signal the PWM output is low.

This produces a PWM output that when filtered by a LPF results in an average waveform that resembles the desired signal.

If the desired signal is a DC value the PWM output will take on a periodic appearance in which part of the signal will remain high for a defined length of time while part of the signal will remain low, and this special case is typically defined in terms of duty ratio. (Typically when you use PWM controlled motors you create a DC value using this method and a internal controller inside the PWM motor converts this DC value into a defined position)

While the DC case of PWM is useful, modulated PWM is far more interesting and is easy to understand once you learn the basic concept behind PWM signal generation. In the case of a modulated PWM signal the desired signal is continually changed and the filtered PWM output results in a time varying average that resembles the changing desired signal.

So let's begin!

## Laboratory Assignment:

1. Do the pre-lab activity described above.
2. Once you have your new project up and running preform a quick test to ensure that your project is working as intended.
3. Initialize your LCD screen and test that it is working correctly by writing some text to the screen.
4. Initialize your Screen buffer LCD custom font code and test that it is working correctly by drawing a few pixels on the screen.
5. Create a 100ms timer that will set a LCD global draw flag and add code to your main loop to render the Screen buffer to the LCD screen when the draw flag is set. (note be sure to clear your flag after you render the screen in the main loop)
6. Create a global integer array called window that is the size of the number of horizontal pixels on the LCD screen.
7. Create a function that will iterate thru each window element from above and create an algorithm to translate the value of the integer into vertical pixel coordinates and then plot all the values on the LCD screen.

   Note: you only have to worry about vales from 0 to 4095 (2^12 ADC)

   Note: a value of 0 will appear at the bottom of the screen while a value of 4095 will appear at the top of the screen.

   When your function is complete it should plot a line across the LCD screen since all of the values should be initialized to 0

8. Once you have figured out what pin the microphone is on and any additional pins that need to be set to a desired value to turn the microphone on, Use the R_ADC_12_Create and R_TMR_CreatePeriodic commands to sample the microphone and the potentiometer ADC channels every 1E-4 seconds (10000Hz)

9. In your ADC sample done function create a global ADCDone flag and add code to your main loop to detect when a new ADC sample is available.

10. In the main loop in the ADCDone code you created above, create an algorithm that will shift the window matrix to the right by 1 element (it should drop the last element in the index) and set the first index equal to the new microphone ADC value.

    Note: be sure you reset the ADC done flag in the main loop.

11. At this point your LCD should display the signals captured by the microphone, if you are unsure that your window and ADC code is working correctly, try plotting the value of the potentiometer to the screen as a debugging step.

12. At this point you will need to create a global variable called state and detect when button 1,2 and 3 are pressed and change the value of state to (1,2,3) according to the button that is pressed (note the default state should be 1)

    Pressing button 1 results in state being set to 1
    Pressing button 2 results in state being set to 2
    Pressing button 3 results in state being set to 3

13. Inside the LCD draw function after you have drawn the screen buffer to the LCD, switch the LCD font mode back to standard text and depending on the value of state write to LCD_LINE1 the following

    If state=1 then write Microphone
    If state=2 then write Alarm
    If state=3 then write Stream

14. At this point we need to setup the PWM module, while I believe that it is important to understand how to accomplish this at register level, for this lab we will be using the RX API and a few registers commands for convenience.

    I will warn you that you should obtain the RX API reference manual that I have mentioned before in prior labs as this is a very important document that discusses what all the parameters are in the RX API (if you do not have a copy of this available you are about to become very confused)

    To begin, because the PWM module is so massive and has so many different modes of operation Renesas cleverly created a custom configuration structure that must be used to configured the PWM modules correctly.

This structure is called R_MTU_Create_structure and you will need to create two R_MTU_Create_structure local variables in a new function called void PWM_SETUP() by

```
R_MTU_Create_structure  PWM_Driver;
R_MTU_Create_structure  PWM_Modulator;
```

We need two PWM structures, because our audio system we are trying to configure has two channels of audio which requires us to have two PWM outputs thus two PWM modules are required (but not for the reason you might think), (the reasoning behind this will become clear as we continue)

Because structures do not automatically construct them self (IE they don't load default parameters into memory) we must initialize them by using the R_MTU_Create_load_defaults function by

```
R_MTU_Create_load_defaults( &PWM_Driver );
R_MTU_Create_load_defaults( &PWM_Modulator );
```

We also need to enable the PWM modules ( this is a similar process that you have used to enable the UART, DAC, and ADC by

```
R_MTU_Set( PDL_MTU_PIN_CLKABCD_B|PDL_MTU_PIN_CLKABCD_A );
```

At this point we need to have a little discussion about PWM signals and the PWM module.

Recall from the laboratory assignment overview that in order to create a PWM signal you need to have two things, a periodic signal that is typically a ramp and a desired signal that the ramp with be compared to that determines the output value based on if the ramp is above or below the desired signal.

In the case of the PWM module the ramp signal is produced by a counter that is incremented at some user defined rate that rests back to zero when it reached a user specified value or when the counter reaches a 16 bit 2^16-1 boundary.

We can control the frequency of this periodic counter by changing the point at which it will reset back to zero. To increase the frequency we shorten the reset at value decrease the frequency we increase the reset at value.

To provide some real world number, the PCLK is around 50MHz and the PWM module supports a 16 bit counter thus the counter will increment by 1 every 1/50MHz seconds or 20 nano seconds and will rest when the counter increments 2^16 or 65536 times which will happen every 20ns*65536=1.31072ms which in turn creates a periodic ramp signal of 762.939453125Hz

This PWM rate happens to be in the middle of our audio range which means if we attempted to modulate a PWM signal with this periodic waveform we would get a nasty buzzing sound at 762 Hz out of our speaker.

To avoid this problem if we configured the PWM module to reset its counter back to zero when it counted to 100 this results in a periodic waveform that occurs every 2us or creates a periodic ramp signal of 500000Hz

This periodic signal is much higher than what the human ear can hear and since the RX development board has a 10 KHz LPF filter on the audio amplifier this component of the PWM signal will be removed leaving only the desired average value coming out of the speaker.

It is important to note that the periodic signal we are currently discussing is not a measureable quantity but a numeric value inside a microprocessor (TCNT) register.

In fact PWM comparison are all done between two or more 16 bit registers in memory and the result of those comparisons modify the output voltage on the physical pin which creates a physical PWM signal (this concept is somewhat similar to how a DAC works in a pseudo similar way)

Now that we have disused the periodic part of the PWM generation process we need to discuss the desired waveform and its role in PWM generation.

As we mentioned earlier, TCNT represents the periodic ramp signal while additional registers (TGRA and TGRB) represent the signal that TCNT is being compared with.

When we disused PWM creation earlier we only mention comparing two values to each other yet there are now a total of three registers involved, this is because each PWM module has two PWM outputs pins that are independent in terms of TGRA and TGRB but are dependent on the current value of TCNT

For example if TCNT resets every 500khz and TGRA=20 while TGRB=50 the output PWM signal controlled by comparing TCNT with TGRA would result in a PWM signal that is on for 400ns and off for 1.6us while the output PWM signal controlled by comparing TCNT with TGRB would result in a PWM signal that is on for 1us and off for 1us.

One problem that arises from using two PWM signals simultaneously is typically in a one PWM output system TGRB is used to reset the TCNT counter while TGRA is used to define the output PWM signal as disused above. In the event that you want to use both TGRB and TGRA for both PWM outputs at the same time you will need to have another PWM module that will trigger a reset in the TCNT counter at the specified rate.

This is why we have two PWM configuration variables defined above; the PWM_Driver will only be used to reset our TCNT counter in the double output PWM driver while PWM_Modulator creates our PWM modulated signal by comparing TGRB to TCNT and TGRA to TCNT.

To configure the PWM Driver we will need to define the following parameters

```
PWM_Driver.data2  = PDL_MTU_MODE_NORMAL|PDL_MTU_SYNC_ENABLE;
PWM_Driver.data3  = PDL_MTU_CLK_PCLK_DIV_1|PDL_MTU_CLEAR_TGRB;
PWM_Driver.data9 = 0;    //TCNT Start Value
PWM_Driver.data10 = 10;  //TGRA Start Value
PWM_Driver.data11 = 100; //TGRB Start Value (Resets TCNT at 100)
```

Note you will need to include "r_pdl_mtu.h" if you have not included it already

We will also need to initialize the driver by

```
R_MTU_Create( 7, &PWM_Driver );
```

Note that the 7 is the PWM module that we are setting up

We will also need to configure the PWM modulator by

```
PWM_Modulator.data2  = PDL_MTU_MODE_PWM2|PDL_MTU_SYNC_ENABLE;
PWM_Modulator.data3  = PDL_MTU_CLK_PCLK_DIV_1|PDL_MTU_CLEAR_SYNC;
PWM_Modulator.data6  = PDL_MTU_A_OC_LOW_CM_INV|PDL_MTU_B_OC_LOW_CM_INV;
PWM_Modulator.data9 = 0;  //TCNT Start Value
PWM_Modulator.data10 = 10; //TGRA Start Value
PWM_Modulator.data11 = 10; //TGRB Start Value
PWM_Modulator.func1=Modulator_Changed; // a ISR called when TCNT=TGRA or TCNT=TGRB
PWM_Modulator.data19 = 1; // ISR priority
```

Note you will need to define a function called void Modulator_Changed()

We will also need to initialize the modulator by

```
R_MTU_Create( 8, &PWM_Modulator );
```

Note that / is the PWM module that the left and right audio channels are connected on

Because two PWM modules have to communicate to each other in order for the counter TCNT to be reset the following code must be added

```
MTUB.TSYR.BIT.SYNC1 = 1;
MTUB.TSYR.BIT.SYNC2 = 1;
```

In order to synchronize the reset operation rate and both PWM modules have to be enabled by

```
MTUB.TSTR.BYTE = 0x06;
```

15. Created a global integer variable called Output and in In the function Modulator_Changed function you created earlier add the following code

```
MTU8.TGRA=Output;
MTU8.TGRB=Output;
```

Note that this function is updating our desired average function for both the left and right audio PWM outputs and that TGRA and TGRB can only be updated inside this ISR call without odd things happening.

16. Because we are not trying to create an average DC PWM output but rather a modulated PWM output we will need to create another timer using R_CMT_Create that will call the function Update_Sound every 0.000125 seconds (note that this function calling rate 8000Hz)

At this point you will need to create a update_audio global flag that is set to 1 inside the Update_Sound function and create a if then statement inside the main loop that detect when this flag is set to 1 and resets it back to zero

17. Inside the main loop update sound if then statement add either a switch statement or another if than else statement that will check for the current state

If the state is in the microphone mode (state 1) convert the current ADC microphone value from a 12 bit adc value that has a range between (0 to 4095) to a PWM modulation value that has a range between (0 to 100) and place this converted value into the global Output variable

Note conversion can be done in a manner similar to the way you converted ADC and DAC voltages in lab 5 or similar to the way you converted ADC to LCD coordinates

18. At this point you can plug either a speaker or headphones into the audio jack of the RX board (Note do not put ear buds in your ears or headphones on your head at this point in time unless

you want to hurt your eardrums) power on the board and you should be able to hear yourself talk into the microphone

19. At this point because the volume out of the PWM is rather loud some volume control would be extremely nice (thus using the on board potentiometer) figure out how to convert the 12 bit ADC value from (0 to 4095) to a floating point (0 to 1) value

    Once you have the conversion factor figured out inside the Modulator_Changed function multiply the global output variable by this factor before writing the data to the TGRA and TGRB registers.

    Note be sure that you cast your floating point number back to a integer prior to preforming the write operation

20. At this point running the program should result in the microphone being sent to the audio speaker and you can now control the volume using the potentiometer.
21. Because listing to the microphone is not necessarily the most interesting thing in the world to do lets expand our boards sound production functionality by adding an alarm when state 2 is active. to do this you will need to go back to the update sound if then statement in main and write a mathematical function to change the Output value between (0 to 100) as a function of time and sample rate (800hz)

    To provide an example consider

```
freq+=1;
if(freq>3000)
{
        freq=200;
}
t=t+0.000125;
if(t>=1/freq)
{
        t=0;
}
Output=(int)(50.0*sin(2*PI*freq*t)+50.0);
```

    While this sounds interesting, I would like you to write your own alarm sound, to provide you some ideas creating a Motorola two tone paging alarm effect might earn you some extra credit or a sound from the public safety sector.

22. At this point the only part of the project that remains is the streaming RS232 audio portion of the project, in order to accomplish this part you will need to take the UART SETUP code from lab 5 and modify it to operate at a baud rate of 626401 bps

You will also need to find the void Interrupt_SCI2_TXI2(void) function and replace it with

```
void Interrupt_SCI2_TXI2(void)
{
}
```

Furthermore you should modify Interrupt_SCI2_RXI2(void) to look like this

```
void Interrupt_SCI2_RXI2(void)
{
        char data=0;
        data=SCI2.RDR;
        Queue_Enqueue(&RX, data);
}
```

You should remove the Queue TX variable from your project

And in the queue.h change

```
#define QUEUE_SIZE 50
```

To

```
#define QUEUE_SIZE 50000
```

Note if you get memory errors try modifying this to 30000

At this point you might want to try testing out your UART by sending a character by

```
SCI2.TDR='A';
```

And ensure you are receiving data by having a break point code inside the Interrupt_SCI2_RXI2 and sending data to the RX board.

Note HyperTerminal might not like this odd UART rate, other HyperTerminal like programs are available or you can simply do

```
import serial
rs232 = serial.Serial(
   port="COM1",
   baudrate=626401,
   parity=serial.PARITY_EVEN,
   stopbits=serial.STOPBITS_ONE
)
print rs232.readline()
rs232.write("HELLO")
```

in python version before version 3 if you have the pyserial module installed.
Note all lab computers have python and pyserial installed simply open the command line prompt and type python to run python
Note that exit() will allow you to exit from python or ctrl C will force exit out of python

Once you have your UART working at its now time to add code to handle streaming audio data when state 3 is active

Because of the time critical nature of streaming audio state 3 cannot be processed inside the main loop without a noticeable delay in audio quality. To get around this you should add an if statement inside the Update_Sound() such that your function resembles

```
void Update_Sound()
{
	update_audio=1;
	if(state==3)
	{
		if(RX.Size<1000)
		{
			SCI2.TDR='\n';
		}
		if(!Queue_Empty(&RX))
		{
			Output=Queue_Dequeue(&RX);
		}
	}
}
```

Once you have this code implemented you are ready to test out the streaming audio part, to accomplish this you will need to download the Stream.py file and the Data1.txt file from the embedded systems website and place both in a folder you can navigate to quickly from the command line.

next run the python application by navigating to the location of the python file in the command prompt window and type the following command

python Stream.py COM5

Note if your COM port is not located on COM5 then you will need to change the COM port to your file

Upon running the python scrip the application will say

Playing File

At this point you can press button 3 and you should hear streaming audio over out of the speaker.

Extra credit might be given if you can identify the streaming sound.

Note because the sample rate is set to 8000hz and the human ear typically enjoys a range of sounds from 20hz to 20khz the quality is not going to be (great) but hopefully this lab demonstrates the very cool things you can do with PWM and UART.

## Lab Report:

| | | |
|---|---|---|
| 1 | LCD Plots Window correctly | |
| 2 | Window Reflects microphone Value | |
| 3 | Pressing SW1,2,3 changes Text on the LCD screen | |
| 4 | PWM works | |
| 5 | Pressing SW 1,2,3 changes the PWM output | |
| 6 | UART works at correct speed | |
| 7 | Streaming Works | |

*Include in your lab report observations and procedure like the following:*

*The general learning objectives of this lab were . . .*

*The general steps needed to complete this lab were . . .*

*Some detailed steps to complete this lab were . . . .*

1. *Step one*
2. *Step Two*
3. *. . . .*

*Code generated or modified to complete this lab...*

*No need to include all the files for the lab. Just include the modified code.*

*Some important observations while completing/testing this lab were . . .*

*In this lab we learned . . . .*