ECGR 4101/5101, Spring 2012: Lab 3

Putting the Pieces Together: A exploration in pseudo physics

Learning Objectives:

This lab will test your ability to apply the knowledge you have obtained from the last two labs to and expand that knowledge further by introducing you to some numerical methods and there embedded implementation.

Note: be warned, this lab expects that you are very comfortable with the Lab1 and Lab2 procedures for creating a HEW project from scratch and implementing the custom LCD interface. Details on these steps will be skipped in this lab as it is expected that you already know how to do this.

Additionally, this lab will not provide as much step by step detail as Lab one and two did so be prepared to spend additional time researching datasheets, 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 Prelab Activity
3. Copy the necessary files into your new workspace and import them
4. Import the necessary mathematical library
5. Write your code to create a pseudo physics simulator
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 project workspace that is very similar to the lab2 workspace since you will be using custom LCD graphics. It is recommended that you either follow the prelab exercise provided in lab 2 or backup your lab 2 workspace and gut it for lab 3.

Tip: you will not need any of the custom LCD characters from lab2, so don't feel obligated to include the emergency logo or squad 51 in this project.

In this lab you will also be using the 3-axis accelerometer that is included on the RX62 development board. Since part of this lab assignment is to figure out how to get data from the 3-axis accelerometer detail steps about how to do this will not be provided, however, as a helpful hint you might want to use the YRDKRX62N project wizard to create a "***sample code***" project of the "**IIC Master**" and look at how that sample project communicates with the accelerometer. You might also want to consider copy some

of the IIC Master sample project files such as (accelerometer.c,ADT7420.h,ADXL345.h,iic_defines.h, r_pdl_iic.h) to your lab 3 workspace folder and add them to your Lab 3 project .

Because some mathematics are involved that require the square root and power operation you will need to include math.h. While this sounds like a relatively straight forward thing to do, by simply putting

```
#include <math.h>
```

At the top of the c file, there is an additional linker step you must do tell HEW that it needs to include math.lib support.

You will need to click the Build>RX Standard Toolchain… from the HEW menu and select the "standard library" tab on the dialog that appears. From there you will need to change the "Category" drop down box to "Standard Library" and check math.h(C89/C99) ……. in the "category" list box, after which you can simply press ok.

Note: there is a math.h and a mathf.h be sure that you select the correct one!

To expand upon your Glyph knowledge further, two commands you might want to consider adding to your lcd.c InitialiseLCD() function are

```
GlyphWrite(G_lcd, GLYPH_FRAME_RATE, 137);
GlyphWrite(G_lcd, GLYPH_CONTRAST, 255);
```

GlyphWrite is a method that writes a LCD command to the LCD driver while GLYPH_FRAME_RATE will configure the LCD to operate at a particular refresh rate and GLYPH_CONTRAST will configure the LCD pixel darkness.

Note: the max refresh rate is 137Hz for the LCD provided by the RX development kit.

Note: GLYPH_CONTRAST has a value between 0 to 255, where 255 is the darkest setting and 0 is the lightest. If you have issues with this command (the screen appears completely dark, try reducing this amount by half)

Note: please check out both the LCD datasheet and the Glyph API manual for more information about all the cool and useful things you can do to your LCD configuration.

One issue that you might have discovered in lab 2 is the LCD screen inability to correctly set the y value to a particular pixel. After further investigation, you might have discovered that the y offset was rounded to the nearest 8th pixel. So for example trying to set the LCD to a y offset between pixel 1 to pixel 7 would result in the next character being rendered at either pixel 0 or at pixel 8.

This rounding y offset property is a function of the LCD screen used, and it makes rendering graphics that required pixel by pixel movement choppy looking as you might have noticed in lab 2 when squad 51 moved up and down on the screen.

Because this lab will require pixel by pixel movement of an object, and at this point you now have some experience working with custom LCD characters it is time to discuss some interesting ways around this y offset limitation that the LCD driver has.

At this point you should know from experience that a custom LCD character the size of the LCD screen can successfully be rendered on the LCD screen if the LCD x,y offset is located at 0,0. This fact implies that we can dynamically generate a LCD character the size of the screen that contains all of our correctly located pixel information and then render this dynamically generated LCD character to the LCD screen in order to resolve the y offset issue caused by the LCD driver.

This concept could be classified as a screen buffering, in which we keep a virtual copy of the LCD screen in memory and update the LCD only when changes are detected.

Before we can create a LCD character that will function as our screen buffer we must first examine how LCD characters are defined in a font file.

From font_bitmap.c it becomes apparent that custom characters are defined as

```
const uint8_t character[]
```

or more verbally stated as constant unsigned character array, which implies that typically characters are not changeable at runtime.

However, if the const term is removed, the character can be changed at runtime and it will be uploaded to the LCD via the glyph driver when the character is written to the LCD the exact same way as a const character which thus allows us to easily create a screen buffer from a non-constant character definition.

Note: To make your life easier, a screen buffer character definition has been provided on the embedded website in a file called lab3.c that you should add to your font_bitmap.c file in a manner described by lab2.

Additionally you should also add

```
extern uint8_t Screen[];
```

to lcd.c in order to quickly access the screen buffer that needs to be modified.

Note: be sure that you don't forget to add the new character to the Bitmaps_table otherwise you will not be able to use the Set_LCD_Char command to render the character.

Note: be sure that you select the appropriate font before trying to render the character, Set_Font_Bitmap is required.

Once you have a screen buffer character, the next challenge is mapping the information in the screen buffer to pixels on the LCD screen. In lab 2 we briefly explained how characters are stored in font memory and to quickly summarize, the first two bytes are the width and height of the LCD character, while the 3 to end of array byte represent LCD character data.

Each byte in the LCD character data represents 8 vertical pixels while each byte index represents a horizontal movement to the right. Once the byte index reached the width of the LCD character the index is reset back to zero and the vertical offset is increment by 8.

For example a 3 by 16 LCD character would have the following data structure.

Byte 1 would be 3 which is the width of the character while byte 2 would be 16 which is the height of the character after this the LCD character data would follow (Byte 3 to Byte 8) and have the following LCD mapping.

<table>
<tr><td></td><td>Column 1</td><td>Column 2</td><td>Column 3</td></tr>
<tr><td>Line 1</td><td rowspan="8">Byte 3</td><td rowspan="8">Byte 4</td><td rowspan="8">Byte 5</td></tr>
<tr><td>Line 2</td></tr>
<tr><td>Line 3</td></tr>
<tr><td>Line 4</td></tr>
<tr><td>Line 5</td></tr>
<tr><td>Line 6</td></tr>
<tr><td>Line 7</td></tr>
<tr><td>Line 8</td></tr>
<tr><td>Line 9</td><td rowspan="8">Byte 6</td><td rowspan="8">Byte 7</td><td rowspan="8">Byte 8</td></tr>
<tr><td>Line 10</td></tr>
<tr><td>Line 11</td></tr>
<tr><td>Line 12</td></tr>
<tr><td>Line 13</td></tr>
<tr><td>Line 14</td></tr>
<tr><td>Line 15</td></tr>
<tr><td>Line 16</td></tr>
</table>

These 6 bytes of data represent 3*16 or 48 pixels on the LCD screen.

Note that each byte represents 8 vertical pixels in which the LSB is the top pixel and the MSB is bottom pixel in the vertical pixel array.

To correctly map the screen buffer coordinates to LCD coordinates a C routine similar to this pseudo code is required. (Note if you want to really understand how LCD characters work I recommend you try to create your own pixel algorithm and not look at the one below)

```
Function LCD_set_pixel(x, y)
        Define xindex=0
        Define yindex=0
        Define shiftby=0
        if x>=96 or y>=64 or x<0 or y<0 then
                Exit function since x y is outside of the lcd screen
        end if
        yindex=y divided by 8  // since each byte holds 8 pixels
        shiftby=y minus (yindex times 8 ) // find the bit offset in the byte
        xindex=2 plus (yindex times 96) plus x // find the current byte based on the vertical offset
        Screen[xindex]=Screen[xindex] bitwise or (1 shifted up by shiftby) // set the pixel to on state
End Function
```

You will need to implement this pseudo function in your lcd.c and if you implemented it correctly calling

```
Set_LCD_Pos(0,0);
LCD_set_pixel(0, 0)
Set_LCD_Char(3);
while(1){}
```

Should render the top left pixel on.

Once you have completed all of these steps, compile your code to ensure everything is in working order.

At this point you should be ready to begin the lab, good luck and have fun!

Some Helpful Advice:

Note is recommended that you look over the datasheet for the ST7579 found at
http://www.tianma.com/web/uploads/controller/20080316012510_ST7579_V0.9a.pdf

you should also look over the documentation from Renesas about Glyph found at
http://www.renesasrulz.com/servlet/JiveServlet/previewBody/1685-102-1-1614/Generic%20API%20for%20Graphics%20LCD%20v1.00.pdf

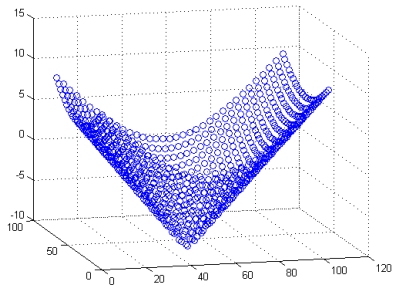
you should also look over the documentation from Renesas API found at
http://www.renesas.eu/media/products/software_and_tools/introductory_and_evaluation_tools/european_pdfs/RPDL_RX62N_API_UsersManual.pdf

and you should also look over the accelerometer datasheet found at
http://www.sparkfun.com/datasheets/Sensors/Accelerometer/ADXL345.pdf

Prelab Questions:

1. Where should your “working directory” be located when using the lab computers?
2. Where is the first place to look for help with labs in this class?
3. What is the true width and length in pixels of the LCD on the Renesas RX dev board (you will need to find this with your LCD_set_pixel command) ?
4. What is the problem with using the Glyph y offset command?

Laboratory Assignment Overview:



A Pseudo Physics Simulator

Rolling a ball around the surface of a cone

Numerical analysis is one of the many interesting of aspects of engineering that is both beneficial to know and practical in embedded systems. It is the purpose of this lab to introduce you to a new piece of peripheral called an accelerometer (which is a device that measure acceleration) that came with your RX development kit and to introduce you to a few useful numerical equations. It should be noted that the mathematical background provided has been simplified to focus on the embedded implementation aspect rather than the theoretical aspect and it is highly recommended that you either independently study or take a few classes on these subjects as they are useful tools to have in your metaphoric engineer's tool box.

Before beginning the lab a few fundamental concepts should be reviewed

You might recall from physics class the classical gravitational acceleration equation is

$$g = -\frac{GM}{r^2}$$

Where g is the barycentric gravitational acceleration at a point in space, G is the gravitational constant of the universe, M is the mass of the attracting object, and r is the distance between the two objects.

You might also remember that the gravitational acceleration for objects on earth is around

$$a(t) = 9.81\frac{m}{s^2} = 1g$$

Hopefully also remember, that a relationship between acceleration a(t), velocity v(t), and position u(t) exist

$$a(t) = \frac{dv(t)}{dt} = \frac{du^2(t)}{dt^2(t)}$$

$$v(t) = \frac{du(t)}{dt}$$

But to make things easier to read and write let's take the differential equations that were in Leibniz's notation (that's the d/dt notation) and rewrite them in Lagrange's notation (prime notation)

$$a(t) = v'(t) = u''(t)$$

$$v(t) = u'(t)$$

At this point you might be wondering why we are covering the acceleration due to gravity and the relationship between acceleration, velocity, and position. The conceptual answer is, we are trying to use the 3-axis accelerometer on the RX development board to determine the gravitational acceleration in a particular direction (X,Y,Z) and use this acceleration information to move a virtual ball around a surface of a cone.

In order to accomplish this task, we will need to get gravitational acceleration information from the 3-axis accelerometer and solve 6 ordinary differential equations (ODE) - initial value problems (IVP) in order to find the velocity of the ball and the position of the ball on the surface of the cone.

While this might sound complicated, the truth of the matter is, you have most likely preformed this task before using a numerical method that simplified the ODE-IVP into an algebraic equation and dint even realize it.

Without further ado let's introduce the Euler's Method of solving an ODE-IVP

$$f(i+1) = f(i) + \Delta_t f'(i)$$

To understand how this equation works, you must first understand that when we work with an embedded system that we are no longer utilizing a function that is continuous in time which is typically represented by the variable t, but rather a discreetly sampled version of the same function that consist of n number of samples that is evaluated for at particular sample called i.

The Euler's Method fundamentally states that the future value of the function f is equal to the current value of the function f plus the amount of time that passes between each sample times the current value of the functions derivate.

Application of this method for our given problem results in the formulation of 6 equations that describe how velocity and position change as a function of acceleration.

The velocity equations for the X,Y,Z axis are

$$v_x(i+1) = v_x(i) + \nabla_t a_x(i)$$

$$v_y(i+1) = v_y(i) + \nabla_t a_y(i)$$

$$v_z(i+1) = v_z(i) + \nabla_t a_z(i)$$

While the X,Y,Z position equations are

$$x(i+1) = x(i) + \nabla_t v_x(i)$$

$$y(i+1) = y(i) + \nabla_t v_y(i)$$

$$z(i+1) = z(i) + \nabla_t v_z(i)$$

It is important to not overwrite your current function value with your future function value until you have solved all of the equations. Also it is important to size your step size delta t correctly. A time step that is too small will make the simulation respond very slowly to changes in the accelerometer, while a large time step will make the simulation overact to accelerometer movements.

Note: it is also important to check your (X,Y,Z) position and ensure it is within the boundary of your lcd. A simple if than statement can be used to accomplish this task, however, forcing a position in either the (X,Y,Z) direction to be a particular value should only be done if you recalculate the current velocity and future velocity for that direction.

For example if the future value of position x is less than the LCD screen we can force the x position to be zero to prevent the ball from rolling beyond the screen but we should do the following to prevent the velocity from increasing which after a long period of time could result in unresponsive behavior.

First we set the future value to the boundary

$$x(i+1) = 0$$

Next we recalculate the current velocity since it's a function of the future position

$$v_x(i) = \frac{x(i+1) - x(i)}{\nabla_t}$$

Then we recalculate the future velocity since the current velocity changed

$$v_x(i+1) = v_x(i) + \nabla_t a_x(i)$$

All of this would have to be done inside the if past boundary then statement

It should be pointed out that the Euler's Method is not the most accurate method for solving an ODE-PVT problem and other methods do exist, which implies that if you wish to implement another method you are free to-do so (trapezoidal, Rk2,Rk3,Rk4,Rk5) are all valid ways to solve the numerical ODE. Euler is the simplest and is recommended.

In order to make our ball roll around the surface of a cone, we need to formulate the equation of a cone and implement it as our z boundary condition.

If you didn't already know the equation of a cone is defined as

$$(x - x_c)^2 + (y - y_c)^2 = [(z - z_c)r]^2$$

In which x,y,z is the coordinates of a point in 3d space xc yc zc is the coordinates for the center of the cone and r is the radius of the cone.

Since we want to use this equation as a boundary condition for the z axis, we can solve for z in terms of our balls current x and y location which results in

$$z = \frac{\sqrt{(x - x_c)^2 + (y - y_c)^2}}{r} + z_c$$

Because we want to scale this equation to match the LCD screen the following configuration is recommended but not required

$$z = \frac{\sqrt{(x - 48)^2 + (y - 31.5)^2}}{3} - 10$$

At this point you should have enough information to begin the lab.

Laboratory Assignment:

1. Do the prelab activity described above.
2. Read over the mathematical equations provided
3. Once you have your new project with custom LCD graphics support ready per instructions from lab 2, preform a quick test to ensure that you are able to write custom characters to the LCD screen along with regular text.
4. Next you will need to figure out how to communicate with the accelerometer. Hints have been provided in the prelab activity

   Note that accelerometer.c uses

   R_CMT_Create(2, PDL_CMT_PERIOD, 100E-3, CB_CMT_Accelerometer, 3)

   To call a function that reads the accelerometer values at a 100ms interval. The first parameter 2 defines what timers should be used, and there are a total of 4 timers on the RX board. This means that you can have 4 different functions being called at different periodic intervals for example

   R_CMT_Create(0, PDL_CMT_PERIOD, 100E-3, Function_A, 3)
   R_CMT_Create(1, PDL_CMT_PERIOD, 200E-3, Function_B, 3)
   R_CMT_Create(2, PDL_CMT_PERIOD, 300E-3, Function_C, 3)
   R_CMT_Create(3, PDL_CMT_PERIOD, 400E-3, Function_D, 3)

   Results in function A being called every 100ms, function B being called every 200ms, function C being called every 300ms, and function D being called every 400ms. See the RX API for more info

Once you have figured out how to communicate with the accelerometer, ensure that you are able to read the accelerometer data from main.c

You should strongly consider using the volatile keyword in your accelerometer global declaration and use the float storage type to avoid register errors later. For example I defied the following global variables in accelerometer.c

```
volatile float adjusted_X, adjusted_Y, adjusted_Z;
```

to hold my accelerometer data.

Also to prevent HEW related casting bugs you might want to consider modifying CB_CMT_Accelerometer (which you will have to gut some of the functionality you do not need) to have the following

(Code snippet)

```
adjusted_X = (float)(Accel_X - Accel_X_Zero);
adjusted_Y = (float)(Accel_Y - Accel_Y_Zero);
adjusted_Z = (float)(Accel_Z);
```

Note that the Accel_Y_Zero term was removed because we want to observe the effects of gravity in the z direction on our ball and that the casting term (float) was used to avoid nasty short to float data errors.

5. At this point you need to add the screen buffer character provided on the embedded website to font_bitmap.c and add a function to lcd.c called LCD_set_pixel that will toggle a pixel on in the empty screen character as being on. Pseudo code was provided in the prelab.

   Note you will need to extern in the character in lcd.c by

   ```
   extern uint8_t Screen[];
   ```

6. Next you will need to write a function that will clear the screen buffer as an pseudo example

   ```
   Function Clear_Screen()
       define lp=0
       for  lp from 2 to 2+96*8  increment by 1
           Screen[lp]=0;
       End for
   End Function
   ```

7. Next you will need to write a function in lcd.c that will draw a circle on the screen buffer whose center is at (x,y) with a radius r. I recommend that you look into the Midpoint circle algorithm; in particular the raster circle algorithm works well. Note be sure you cite any sources used to accomplish this, (NOTE: other students outside of your group cannot be used as a reference)

8. In main.c you will need to implement a function that is called every 100ms using R_CMT_Create that will not interfere with the accelerometer.c R_CMT_Create function that will toggle a draw flag similar to the draw flag you used in the lab2 project.

9. In the main loop you should write a function that will draw the contents of the screen buffer to the LCD similar to the following pseudo code

```
While Loop // the main loop
        If draw==1
                Set_Font_Bitmap()
                ClearLCD()
                Clear_Screen()

                // Code that draws on the screen goes here

                Set_LCD_Pos(0,0)
                Set_LCD_Char( custom character number)
                draw=0
        End if
End While
```

10. At this point test your circle algorithm, pixel drawing algorithm and ensure that it is correctly displaying on the LCD screen.

11. At this point you are ready to implement the Euler methods discussed above, first you will need to define variables to hold your ball's current position and velocity and future position and velocity.

    Note because there are a sizeable number of floating point variables required, it is recommend that you make these variables global, as stack corruption can occur if too many floats are defined locally. There is nothing worse than having your code correct but the debugger reports the wrong answer because of a complier quark. For your sanity, I recommend that you define all of your floats globally!

    I recommend naming the current position x y z , the current velocity as vx vy vz , the future position x_n y_n z_n , and the future velocity as vx_n vy_n vz_n

12. Next you will need to define your simulation step size as a variable (delta_t)

    I recommend a step size of 250E-3

13. Next you need to define 3 floating point variables that will translate the value from the accelerometer into a useable acceleration.

    The best way to figure this out is to use the debugger to observe the accelerometer 3 axis values at their maximum exposer to gravity (IE flip the accelerometer so its axis is facing the ground in 6 different directions for the X,Y,Z axis and record those values)

    You can then take the distance between each axis values and divide 9.81 by this number to find your acceleration per accelerometer unit.

    Typically the accelerometer will report that 1g is between -40 to 40 on an top and bottom of the axis

    The conversion coefficients I used were

    ```
    float gx=.3111;
    float gy=.297;
    float gz=.297;
    ```

    Yours will be different, but these will give you a baseline.

    You could also take an average of these numbers and use a single number for all conversions.

14. Create variables that will hold your current acceleration
    Note: I recommend ax,ay,az

15. Inside the draw function, since we only want to update our position at a rate of 100ms to avoid having a fast moving ball calculate ax ay az by multiplying your measured accelerometer value by your conversion coefficient(s).

    Note you will need to multiply ax and az by a negative 1.0 to make the simulation forces respond as expected since the board's accelerometer is mounted differently than our simulation equation expect.

    Note always use 1.0 not 1 to let the complier know that the number is not an integer to avoid casting errors.

16. Calculate the future velocity (vx_n vy_n vz_n) and position (x_n y_n z_n) using Euler's method, note we are assuming all initial conditions are zero.

    To provide one Euler example vx_n is found by

    ```
    vx_n=vx+ delta_t *ax;
    ```

17. Check x_n, y_n, and ensure it is inside the lcd boundary, if it's not then change the value and update velocity and future velocity.
    X is between 0 to 96
    Y is between 0 to 63

    note you will need to recalculate velocity and future velocity if you force the position to a fixed value as discussed earlier.

18. Using the x_n, y_n values calculate the lower z boundary using the equation of a cone. Also assume that the upper z boundary is defined by a plain located at Z boundary upper =10.

    Note the lower z boundary is both a function of X and Y and is in the shape of a cone along the negative axis

    note you will need to recalculate velocity and future velocity if you force the position to a fixed value as discussed earlier.

19. You can now safely set the future position equal to the current position.

    or in other words

```
vx=vx_n;
vy=vy_n;
vz=vz_n;
x=x_n;
y=y_n;
z=z_n;
```

20. Using the z value calculate above change the radius of the circle you will draw to a smaller size when z becomes more negative and larger when z nears the upper boundary.

21. Use your screen draw function you created earlier to draw a circle at the calculate x,y value with a radius you defined based upon the z value calculated.

22. Test your program and watch the ball roll around the screen.

    I recommend that you try changing the time step and observe how it changes the speed the ball moves around the screen.

23. Next add some code that will cause the green LEDs to flash in a wig-wag pattern similar to the pattern created in LAB 2 when the ball hits the top Z boundary and causes all the red LEDs to blink in a wig-wag pattern when the ball hits the Right X boundary otherwise the LEDS are off.

24. At this point I would like you to add your own unique spin on the simulator, maybe add addition equations to force the ball to gravitate to the bottom of the cone, sketch the cone surface, add code to fill in the circle, or make one of the walls rubber so the ball bounces off the wall.

25. Bring the new board to the lab TA and demonstrate the new code (without the HEW application running). When the TA checks your board, he will also take your lab report. You will not need to include a printout or soft copy all of the code – just "snippets".

Lab Report:

1	LCD is rendered using a screen buffer character	
2	LCD is rendered at a proper refresh rate	
3	Circle is drawn on the LCD	
4	Circle moves with changes in accelerometer, in a manner that follows the laws of physics	
5	Circle changes sizes the further into the z direction it goes	
6	Circle follows the surface of a cone	
7	LEDS blink correctly when the ball hits the appropriate boundary	
8	Group added their own unique spin to the project	

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 . . . .