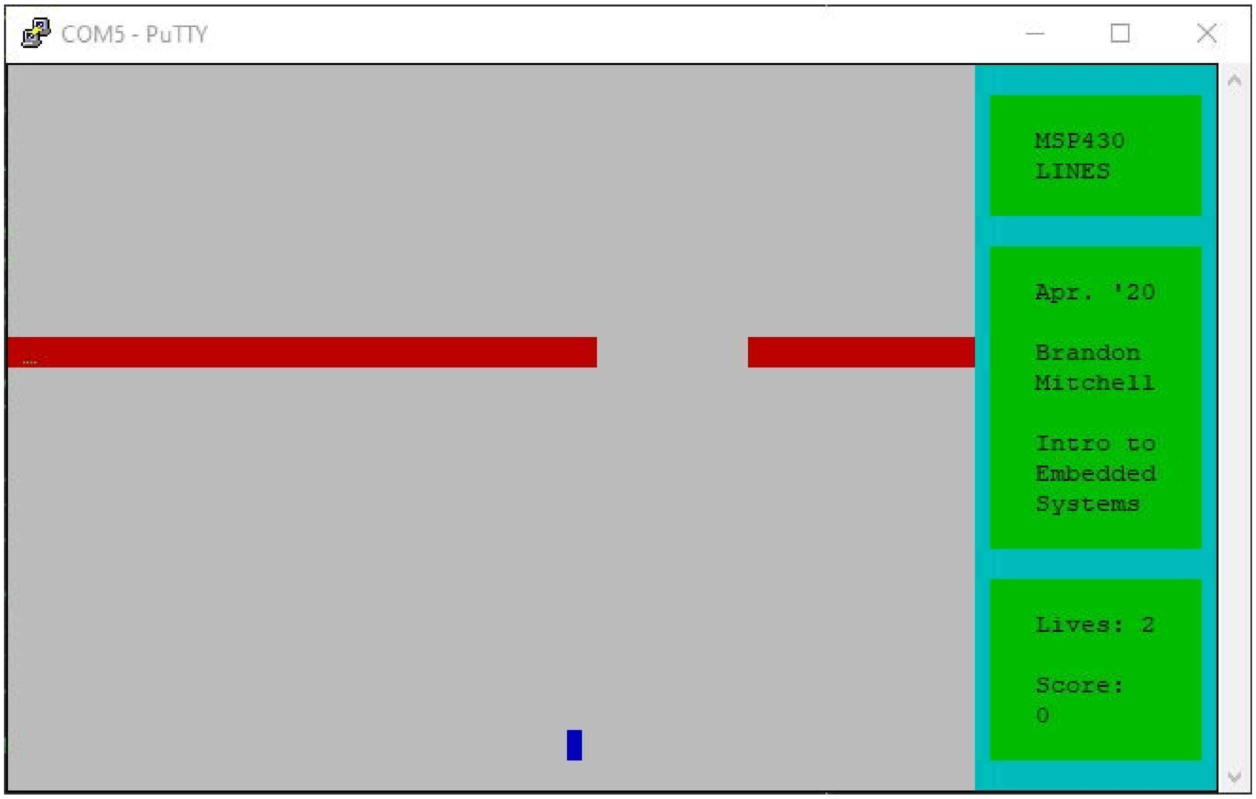


MSP430
LINES

Apr. '20

Brandon
Mitchell

Intro to
Embedded
Systems

Lives: 2

Score:
0

# MSP430 LINES

How to create a small game on the MSP430 microcontroller

By Brandon Mitchell

## Introduction:

Today, I am going to show you how to create a small and simple game on the MSP430 microcontroller. By the end of this tutorial, you should have learned more about using UART and the ADC with your microcontroller as well as some useful knowledge about escape sequences and working with terminals. Although this will focus on the game shown above, you should be able to use some of your knowledge to make many other types of games that will run in a terminal. Also, this is not a line by line tutorial. Instead, the large concepts needed for this game will be discussed.

## Supplies:

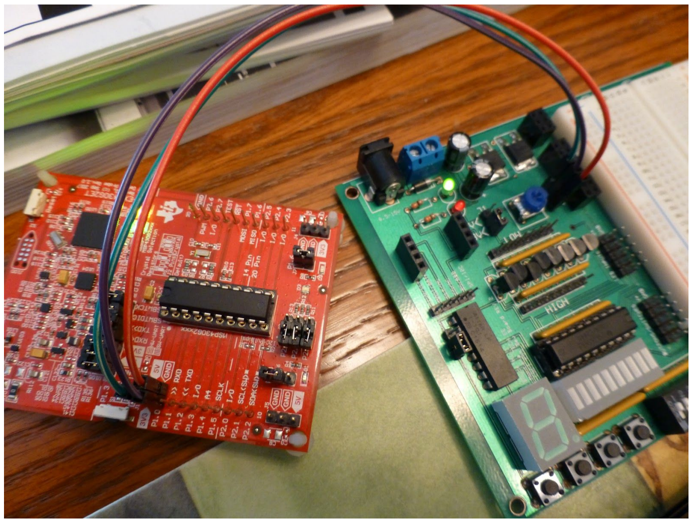
If you are a student of CSCI 255 Intro to Embedded Systems, you should have everything you need already for this project. You will need:

- One (1) MSP430 Launchpad
- One (1) potentiometer (most likely on your dev board)
- One (1) micro-USB power cable (needed for power and data transfer)
- Three (3) female-to-female pin connectors if you have a 255 dev board, different types might be needed for free / loose potentiometers

## Instructions:

### ADC

The first thing I would recommend doing is setting up the ADC to read the value from the potentiometer. This was, at least for me, the most complicated step as it was new; everything else I had done before. To the right is a photo of what your wires and hardware should look like. Connect ground to ground, 3.3V to 3.3V and then the middle output pin of the potentiometer should be connected to P1.0 on the launchpad. The launchpad's pin can change depending on what



pin is chosen in software. Just be sure to avoid P1.1 and P1.2 as those are used in UART.
In software, you are going to want to enable the ADC and set the channel it is going to read from. The code below should do the trick (note, underscores are not showing up):

```
    // ADC Setup
    ADC10CTL0 = SREF 1 + ADC10SHT 2 +   // ADC10ON, interrupt enabled
          ADC10ON + REFON;
    ADC10CTL1 = INCH 0;                  // input A1
    P1DIR     = P1OUT = 0x00;            // Input on P1.0, prevent noise
```

The first line enables the ADC, and you will likely not need to nor want to change those values. The second line of code (consider the first two lines one as it spilled onto two lines) sets the input channel for the ADC. This is what you may need to change. INCH_0 is P1.0, which is what we want. INCH_1 is PI.1, and so on. INCH_10 is the temperature reading, which you might find useful. The final line ensures P1.0 is an input and sets all other pins to be inputs and low to help prevent possible noise.

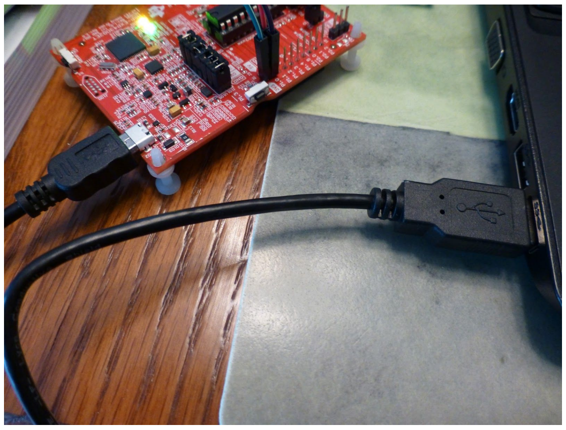
After all that has been set up, write some quick code to toggle a LED. You might need to change a pin value if you copied what was above, and also be sure to remove the jumper on the P1.0 LED. The code to read the ADC value is below.

```
    // Ensure ADC is ready to be read
    ADC10CTL0 |= ENC + ADC10SC;
    while (ADC10CTL1 & ADC10BUSY) {}
    unsigned int ADCVal = ADC10MEM;
```

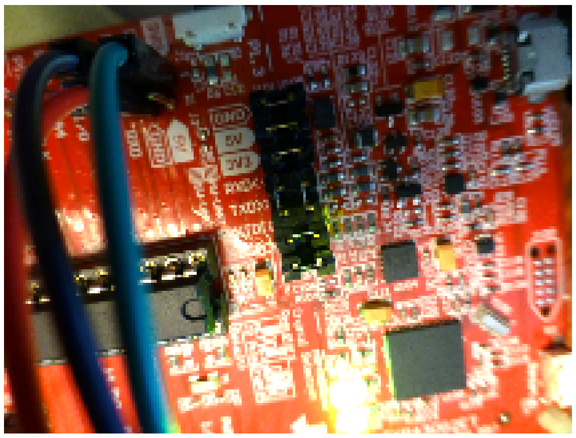
The first two lines deal with making sure the ADC is ready to be read from, and then we can read from it with the variable ADC10MEM. Use this to toggle a led when the ADC value is above a certain range so we can know everything is set up correctly. Keep in mind the ADC has a ten bit range, so the number will be in the range of zero to 1023.

## UART

Likely you will have done UART in one of the prior labs if you are a CSCI 255 student. Nonetheless, I will go over how to set everything up. The simplest way to connect it to your computer is with the micro-USB.



Also, be sure the RXD and TXD jumpers are crossed like in the photo to the right. Sorry about the image being a bit dark. Following that, we want to get the code set up. If you do not already have code for working with the UART, it would be best you practice with the example code before moving on. Try to figure out how to send strings of any length over UART as that will be important.



Once you are comfortable using UART and have experienced how to send characters back and forth, we can move on to the specific set up for this project.

```
    DCOCTL    =  0x00;                        // Select lowest DCOx, MODx settings
    BCSCTL1   =  CALBC1_16MHZ;                // Set DCO
    DCOCTL    =  CALDCO_16MHZ;
    P1SEL     =  P1SEL2 = 0x06;               // P1.1 = RXD, P1.2=TXD
    UCA0CTL1 |=  UCSSEL_2;                    // SMCLK
    UCA0BR0   =  0x88;                        // 16MHz 115200
    UCA0BR1   =  0x00;                        // 16MHz 115200
    UCA0MCTL  =  UCBRS2 + UCBRS0;             // Modulation UCBRSx = 5
    UCA0CTL1 &= ~UCSWRST;                     // Initialize USCI state machine
```

This should look very familiar if you have done UART before on the MSP430. One thing to note is that I never enable the RX interrupt as data will only be going one way, i.e., from the microcontroller to the computer. Also, I set the clock to be 16MHz and the baud rate to be 115,200. This is because the program will be doing a lot of I/O and string operations, so we want everything to be as quick as possible. It is easy to slow everything down if needed. Before moving on, ensure the code works by trying to send something back. Remember, the RX interrupt has been disabled, so simply hard-code in something to send back for now.

## Timer Interrupt

Now, this was more of a design choice, but I decided to have the line, which always moves at a constant rate, be regulated by a timer interrupt. I set it up like so:

```
    // Timer set-up, enable down below in main
    TACTL = TASSEL_2 + MC_2 + ID_3; // SMCLK, contmode
    CCR0  = 0x00;
```

Note the ID_3 causes the timer register CCR0 to only increment every eight clock cycles. You can choose a different value if you are so inclined, but the slower counting is appreciated with the faster clock rate. Also, I don't enable the timer right there. I set the pins and then draw the playfield. If I enabled the interrupt right there, the line would start moving before everything has been drawn! Your interrupt should look a little like the one below, with some code inside it, of course (again, note the underscores are not showing, there are two in front of "interrupt".

```
#pragma vector = TIMER0 A0 VECTOR
  interrupt void lineInterrupt()
```

Like everything, be sure to test it before moving on. Have it flash a light or maybe display characters to the terminal.

**Drawing Routine**

This part is very important, so it is best you get it right before doing anything else. You can start on the game before doing this, but you will have a lot of unnecessary, repeated, and complicated code to do your draw routines. To draw to the screen, we will be using what are called ANSI escape codes. These are certain sequences of characters that the terminal interprets and executes rather than print to the screen. There are quite a few, but the ones we are interested in are the escape sequences that allow us to move the cursor manually and also add color. Wikipedia has a great article on it if you are more interested. Important to note is that not all sequences are supported by all terminals. However, most support at least a few. I have used them in the Windows Console and even on the Nintendo 3DS game console in homebrew!

The two we will need are "ESC[r;cH" and "ESC[nm". If you look at an ASCII table, you will notice there is a character called "ESC" in the control characters section. To actually use it, you precede the hex value of the number with a "\". So, something like "\x1b[r;cH". Note, the octal value also works fine ("\033"), but I haven't used the decimal value, so try it out.

In the former escape code, the "r" is the row and the "c" is the column. It is important to realize the coordinates start at (1, 1) in the top left corner and that the code must be followed by "H". The following code will print out "Hello, World!" on row 10, column 25 followed by "Now we are here." on the first column of the fourth row. If either the row or column is excluded, the value of one is assumed. Also, when the cursor is moved, it stays in that position until another character is printed and moves it forward by one or when you manually move it.

```
      printf("\x1b[10;25HHello, World!");
      printf("\x1b[4;HNow we are here.");
```

Colors are done in a similar way. However, what colors you can use depend on your terminal. The eight basic colors below seem to work in every terminal. However, an RGB triplet can be specified. This latter works in the Windows Console, but your mileage will vary. The basic format is "\x1b[nm". Text colors are 30 to 37 and background colors are 40 to 47. Other values can do things like underline or bold, though most aren't supported. Also, "\x1b[0m" will turn off all attributes, so new text will be whatever the terminal default is, and the attributes stay on until overridden by another one or turned off with the above code.

```
      printf("\x1b[30mSample Text");  // Black text
      printf("\x1b[31mSample Text");  // Red text
      printf("\x1b[32mSample Text");  // Green text
      printf("\x1b[33mSample Text");  // Yellow text
      printf("\x1b[34mSample Text");  // Blue text
      printf("\x1b[35mSample Text");  // Magenta text
      printf("\x1b[36mSample Text");  // Cyan text
      printf("\x1b[37mSample Text");  // White text
      printf("\x1b[0mSample Text");   // Clear attributes
      printf("\x1b[34;47mSample Text");  // Blue text, white background
      printf("\x1b[38;2;115;226;35mSample Text");  // RGB example
```

I encourage you to look at the other escape sequences available and see which ones work or not. Also, 40 is black background, 41 is red background, and so on, and they can be combined to set the text and background color at the same time. I recommend picking some colors and combinations you like and doing a "#define" in the top of your program to make them easier to use and the program easier to read and change. Also, do a "#define" for many other variables that show up a lot, like the max size of the terminal, or the line size. This makes things more legible and makes it easier to change them later.

Now, we can actually start on the drawing routine that will be used for every single draw call. You may set it up a bit differently, but mine took a char pointer to the sentence to be printed, another char pointer to the color desired, and then an x and y position.

```
void dispToTerminal(const u8 * sentence, const u8 * color, u8 x, u8 y)
{
      u8 index, length, buffer[100];
      length = sprintf(buffer, "\x1b[%d;%dH%s%s", y, x, color, sentence);

      for (index = 0; index < length; index++)
      {
            while (!(IFG2 & UCA0TXIFG)) {}
            UCA0TXBUF = buffer[index];
      }
}
```

u8 simply means unsigned char. The important part is the highlighted line. Using sprintf(), we can insert numbers and strings into a char array using the format spec we give it. "%d;%d" will take the y and x and put them in the position escape code, and the "%s will place in the strings. Be sure the color comes before the string to be printed or else it would color the next string displayed that color. sprintf(), which is in the stdio.h header, returns the length of the new string, so that is used in the for loop to display everything over UART. From there, you can make pretty much anything now. One thing to keep in mind, though, is that you should try to draw as much as possible in one draw call to minimize the number of draw calls. For example, drawing the background one by one takes a couple of seconds for me, but drawing it in full lines is nearly instant! Also, change how and when you draw various objects if things are acting weird. I had a glitch that was fixed when I stopped drawing the line from left to right and started drawing from right to left.

**Game Time**

This part is really up to you. Even if you are planning on making the game above, how you go about setting everything up should be your call. However, I will show how the line is made as that part may be a bit complicated.

```
struct {u8 pos, count, activated, barrier[LINE_SIZE];}
      line = {1, 0, 0, {0}};

void lineInit()
{
      u8 lineStart = rand() % 55 + 1,
           lineEnd = rand() % 7 + 3 + lineStart, index;

      for (index = 0; index <= LINE_SIZE; index++)
      {
      line.barrier[index] = index >= lineStart && index <= lineEnd ? 0 : 1;
      }
      line.activated = 1;
}
```

I create a single, global struct that holds its position, a framecount used in drawing, a bool value used to reset the line, and a char array that is the size of the playfield. I calculate a random line start and line end, giving me a gap randomly sized and placed. Then, in the for loop, I fill the barrier with zeros for the gaps and ones for the solid spots. This is used when drawing the line so we can draw the gap easily and also for collision detection. I am using rand() here, but it isn't truly random, so I call srand() at the beginning. Typically, srand() is seeded with time(NULL), but that takes about all the RAM the microcontroller has, so I instead seed it with the current value from the ADC. Since it has over a thousand different values, the user is unlikely to get the same game each time. To use these random functions, you have to include stdlib.h.

As for drawing the line, I first draw a blank line in the previous position, making sure if the line is at the beginning or not. This is needed or else a trail will be left by the line. Remember, what you draw to the terminal stays there until overwritten. Then, I loop through the list and draw a red space at every index with a one. Finally, I increment the line position, check for rollover, and reset the line if needed.

```
void lineDraw()
{
        u8 index;

        dispToTerminal(BLANK, GRAY, 1, line.pos != 1 ? line.pos - 1 : YMAX);

        for (index = LINE SIZE; index > 0; index--)
        {
        if (line.barrier[index]) {dispToTerminal(" ", RED, index, line.pos);}
        }

        if (++line.pos > YMAX) {line.pos = 1; line.activated = 0;}
}
```

I am drawing right to left as drawing left to right would sometimes cause the escape sequence to be printed out and drawn over the side bar. Also, I set a flag activated in this function as if I reset the line in the function, a graphical glitch would happen with the player. I recommend trying a few different things to make sure everything looks nice.

Now, to move the line, I am going to use the previously set up interrupt. This will ensure the line moves at a constant rate. The code in the interrupt is.

```
        if (++line.count == LINE RATE)
        {
                if (!line.activated) {lineInit();}
                if (line.pos == YMAX - 1)
                {
                        line.barrier[player.pos - 1] ?
                              player.lives-- : (player.score += 50);
                        drawInfo();
                }
                lineDraw();
                if (line.pos == 1) {drawPlayer();}
                line.count = 0;
        }
```

I check if the counter has triggered, and then check if the line needs to be reset. Then, if the y position of the line is the same as the player, I check the x positions. This is where the ones and zeros in the line also come into play. I then redraw the side bar info to update it, draw the line, check if the clear line covered the player and redraw the player, and then reset the line count.

The player is very similar. A global struct is created since it will need to be accessed in the interrupt. It has a position, a previous position needed to clear the trail, a number of lives, a score, and a framecount. To draw the player, I draw a gray space at the previous position and then a blue space at the current. The order is important as the last drawn character will be on top. Also, I only draw the player when the player has moved to avoid unnecessary draw calls.

```
	while (player.lives)
	{
		ADC10CTL0 |= ENC + ADC10SC;
		while (ADC10CTL1 & ADC10BUSY) {}

		if (++player.count == MOVEMENT_SPEED)
		{
			player.pos = ADC10MEM * RATIO;
			if (player.ppos != player.pos) {drawPlayer();}

			player.ppos = player.pos;
			player.count = 0;
		}
	}
```

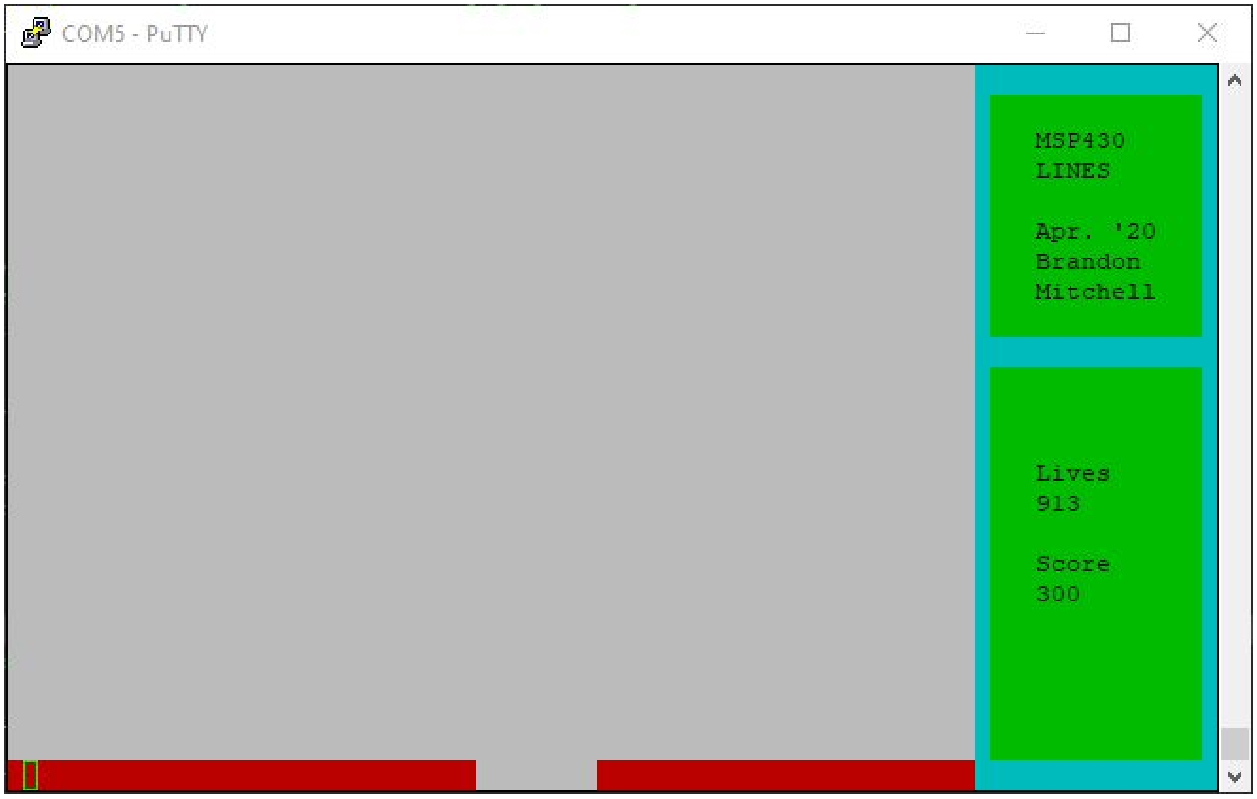
Like everything, how you go about setting this up is and should be up to you. I loop while the player is still alive, make sure the ADC is ready, and then check if it is time to update the player. I multiply the ADC value by a ratio. This converts the ADC value into a value in the range I want. The ratio is (64.0 / 1023.0), the max terminal position divided by the max ADC value. I then check if the player needs to be drawn as you want to minimize draw calls, and finally change the player's previous position to his current and reset the count.

One the game ends, I disable the timer interrupt so the line doesn't continue forever and then print a message to the player. I can reset the microcontroller to restart the game. Try setting a pin as a game restart so you don't erase scores in between play sessions. This way, you could possibly display high scores after each death. I encourage you to have fun and try out different things.

## Conclusion

That is pretty much it. I tried to cover the large portions, but I also want you to figure some things out on your own. I also encourage you to try out different things. In my game, I kept track of the player's score and displayed it on the side. Try figuring that out for yourself or do something else. Also, using what you now know, you should be able to make a lot of different games. I made a snake game on my Nintendo 3DS that used only text and escape characters. The potentiometer was used for this one, but try a few other input methods for other games. You should be able to reuse your drawing routines, and I would even recommend putting it in a header file with all the different colors. Try writing functions for drawing squares or even circles if you think you can do it. Use these drawing functions to make something really cool. I have even seen a first person shooter made to run on the command line!

I had a few weird things happen while doing this project. First, I had a few graphical glitches. If you have these, try changing the order you draw things or when you draw. I highly recommend only drawing what needs to be drawn. I.e., draw what has changed each frame. Also, the player bounces around. I believe the potentiometer or ADC generates some noise as the player will jerk about. I found checking if the ADC was ready to be read every loop instead of just when I needed to draw partially fixed the problem, but you still bounce, nontheless. Finally, some values rolled over weird. The player's score, an unsigned int with a max value of 65,535, for some reason became -13,000. The microcontroller reset itself after that, so I didn't get a picture. However the player's lives rolled over before I set an end condition and I got the value of 913, a value that doesn't fit in an unsigned char. Keep these in mind in case you have something weird happen. I was told that the micro controller acts funny when you use a lot of RAM, but I am only using 164 of the 512 bytes. It is a mystery, and this program is simple enough that I didn't do something stupid somewehre; there are more comments than code.



```
MSP430
LINES

Apr. '20
Brandon
Mitchell

Lives
913

Score
300
```