Transmitting Serial Wirelessly

Callum Jones[1], Simon Coles[2], and Ryan Prekop[3]

[1]**s3601120@student.rmit.edu.au**
[2]**s3401569@student.rmit.edu.au**
[2]**s3544394@student.rmit.edu.au**

ABSTRACT

This project aims to introduce how to use the HC 11 RF module to transmit RS232 Serial wirelessly. In particular, the AVR ATMEGA32 will be used, on a prototyping board called OUSB. [1]

Keywords: ATMEGA32, Serial, RS232, RF, HC 11, OUSB

INTRODUCTION

The project our group has selected, is the RS232 Transmitter/Receiver project. It requires two OUSB boards, the first to act as a transmitter and the second a receiver. The transmitting board is required to scan a keypad, and if a key is pressed, send it over the serial port, to the second OUSB board, which will either display the binary version of the number on the LED's or display the number directly to an LCD screen (via serial). Some constraints must be defined first. The transmitter board will only send one key at a time, and will implement a delay long enough, such that a key is only pushed once. If multiple keys are pushed down, the first key scanned, will be sent over.

The OUSB boards are a microprocessor development board made for the ATMEGA32 (designed by PJ Radcliffe, please visit: https://pjradcliffe.wordpress.com/). The ATMEGA32 has four IO (input/output) ports available for use. Each port has 8 pins available to it. On the OUSB board, each port has a different function. PORTA are ADC pins (analogue to digital converters), of which a couple of the pins have an LDR and Potentiometer connected. PORTB has an LED connected to each pin. PORTC has an 8-dip switch connected to it, while PORTD is used by the board for communicating to a PC. [1]

The RS232 serial protocol is a way of sending data across a single wire (not including the ground connection), or data in both directions using two wires (not including the ground connection). The key to this protocol is both the receiver and transmitter must have a pre-set speed known as the baud rate. The baud rate is the speed of data transferring over the link. For example, in this project a baud of 9600 will be used, which also represents 9600 bits/second. To find out how long it takes one bit to transfer, simply invert the baud rate: 1/9600=0.104 ms. The RS232 protocol can be used in a couple of different configurations, but this project uses a baud of 9600, no parity bits, 8 data bits and 1 stop bit. While the serial port is idle (i.e., no data is transferring through), the port is high. To start sending serial, the transmitter must tell the receiver, using a start bit. A start bit is a change from high to low. The baud rate then determines how much time to wait before the sending the first bit. The following 7 bits are also sent over, with a delay of one baud rate after each bit. The final bit is the stop bit, which is simply a high. (See figure 1 below). [2]

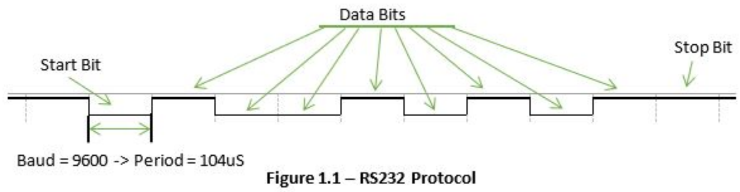

Figure 1. RS232 Protocol

The parallax LCD, part number: 27977, is a three-wire simplex device. That is, it has a power supply pin, ground pin and data pin. The simplex means the data pin is unidirectional, in the case of this LCD, it only receives data. The type of data it requires is RS232 serial data, much like rest of the project. Referring to the datasheet [3], the LCD will react according to the ASCII character it receives. For example, any letter, number or character received will be displayed on the LCD, but if ASCII 0x0C (represents form feed/new page in ASCII) is sent, the LCD will clear the screen and move the cursor back to position - (0,0).[3]

Once the serial connection is up and running, the next stage of the project is to use an RF module to transmit the serial data (ASCII codes) wirelessly. The module that will be used for this project is the HC 11. This module takes in serial data (with customisable settings, but default is baud of 9600, no parity, 8 data bits and 1 stop bit) and converts it into an analogue signal to be transmitted wirelessly at about 434MHz. The 2nd module receives this analogue signal and converts it back into serial. Each module can be used as a transmitter or receiver at one time (i.e. the modules are half duplex). All of this, and more can be found in the datasheet.[4]

BLOCK DIAGRAM

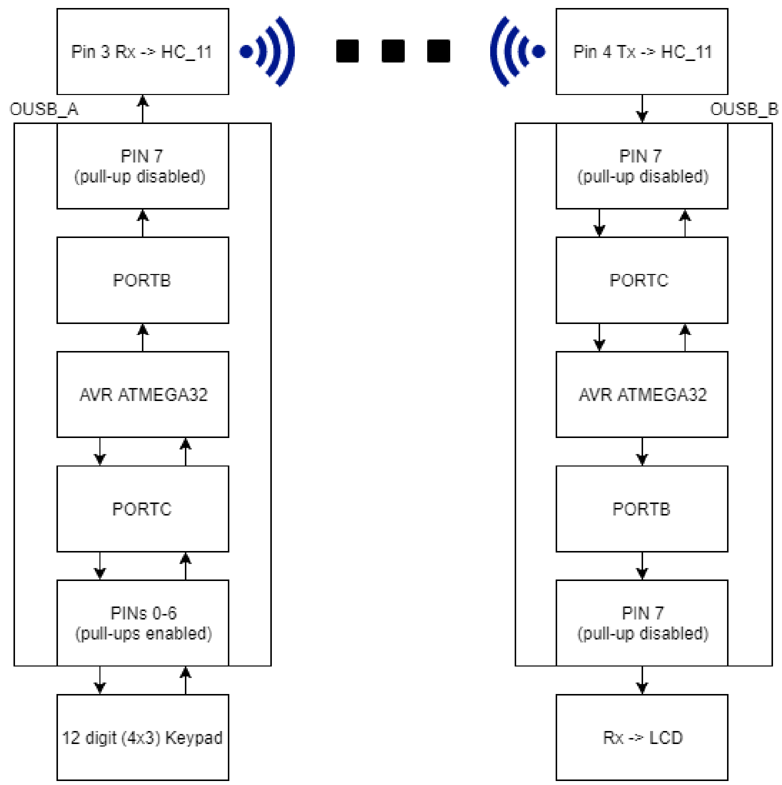

Figure 2. Block Diagram of the System

Figure 2 shows the block diagram of the hardware in the system. The diagram shows two OUSB boards, two HC 11 (Serial RF Modules), one 4x3 Keypad and an LCD. The arrows represent the flow of data.

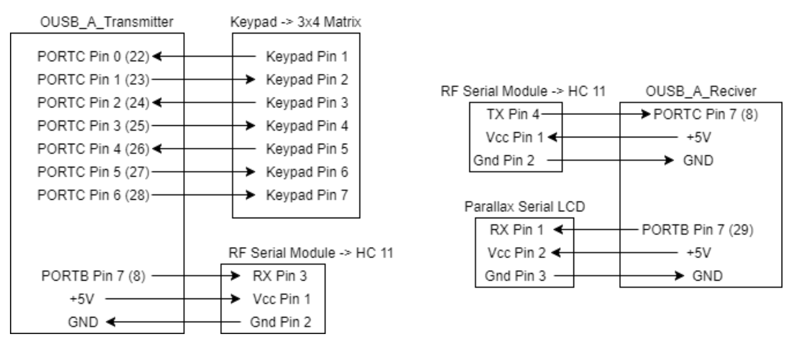

Figure 3. Wiring Diagram of the System

Figure 3 is the wiring diagram for the system. Again, the arrows show the flow of data (except for the power supplies). For this project, PORTB Pin 7 was chosen as the transmit pin, so RS232 serial can be sent to the RF module (HC 11). The keypad is most conveniently connected to PORTC pins 0-6. For the receiver board, the receiving pin (for RS232 serial) was chosen as PORTC pin 7 (but it could have been any from PORTB or PORTC). PORTB pin 7 was chosen to transmit RS232 serial to the LCD from the receiver board. The numbers in the brackets represent the actual pin numbers on the ATMEGA32.

FLOWCHARTS

Receiver Flowchart

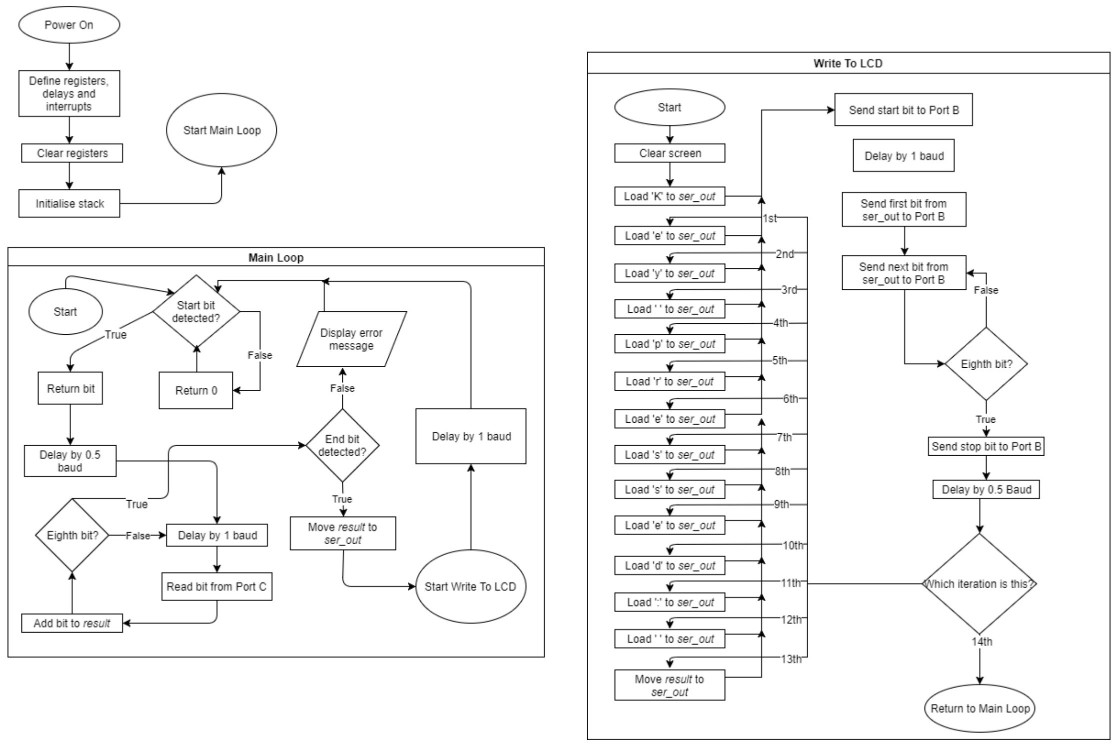

Figure 4. Flowchart of Receiver Software

Figure 4 shows a flowchart for the OUSB board which acts as the receiver, and displays the received result onto an LCD. The program writes “Key Pressed: ”, and then the value received, making it a little more user friendly.

Transmitter Flowchart: Part 1

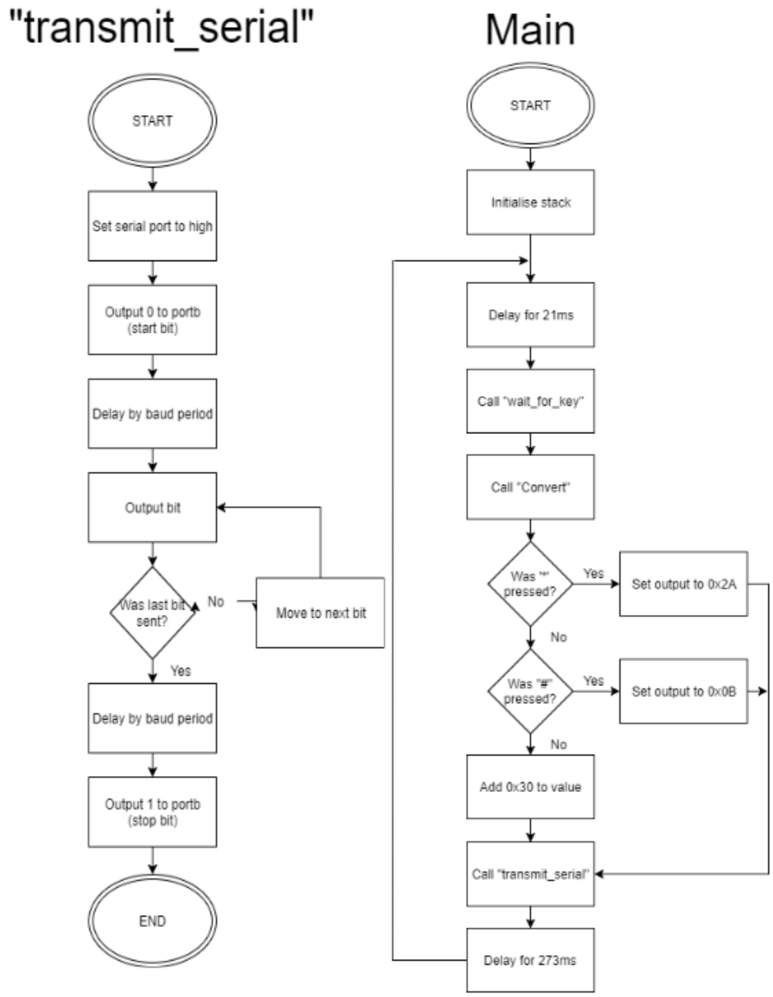

Figure 5. Main Loop and Transmit Serial Parts of Transmitter Code

Transmitter Flowchart: Part 2

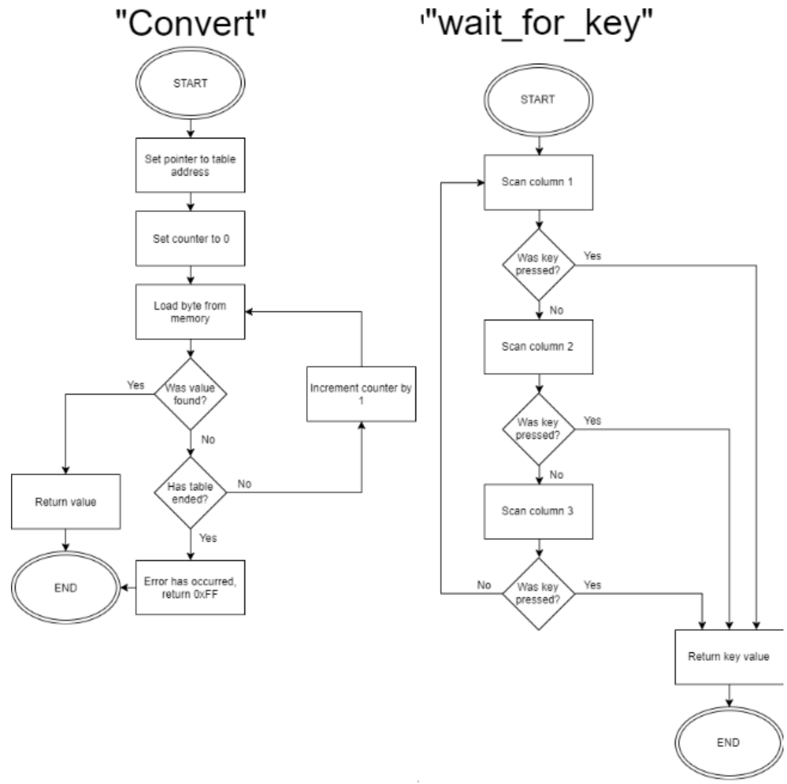

Figure 6. Convert and Keypad Scanning Parts of Transmitter Code

Figures 5 and 6 show the flowcharts for the OUSB that acts as the transmitter. It contains four main parts, the main loop, RS232 transmitter code, keypad scanning code, and a convert function (converts scan code into a decimal value of the key pressed).

TESTING

First Stage:

The first step was to understand and program the 4x3 keypad. The testing process for this was rather simple: using a single OUSB board, individually push each key, and it should display the correct number on the LED's on PORTB. Shown in figure 7 (below).

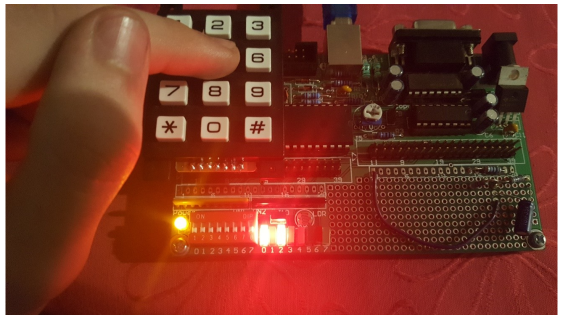

Figure 7. Testing the Keypad

At the same time, software was developed for transmitting serial (RS232 protocol) over PORTB pin 7. Rather than use the PC to decode the serial data, an Arduino Nano was set up as a receiver, which echoed results to the PC. The purpose of doing this was to prove, the programmed serial port, is working with external hardware, and because I know the is Nano a very reliable serial communication device (hence why our serial code wasn't between the two boards to begin with). A successful test would result in the PC showing letter 'C' appearing every 2-3 seconds or so. See figure 8 (below).

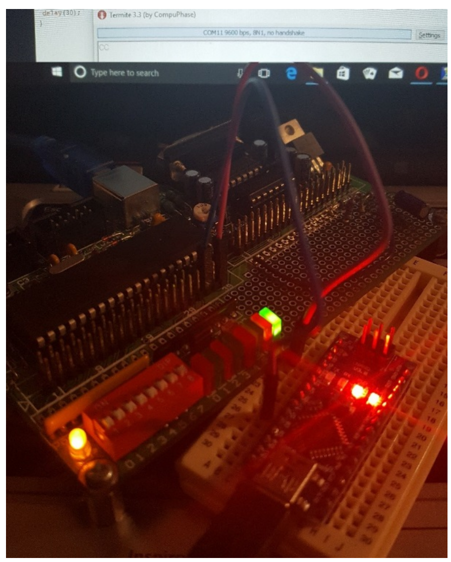

Figure 8. Testing the Serial Out Code

After the transmitting code was developed, some modifications were made to use PORTB pin 7 as a receiver for serial data (RS232 protocol). Much like before, the Arduino Nano was used as a transmitter, which was read by the OUSB PORTC pin 7 and displayed on PORTB LED's. A successful test would result in the incoming serial showing on PORTB. See figure 9 (below).

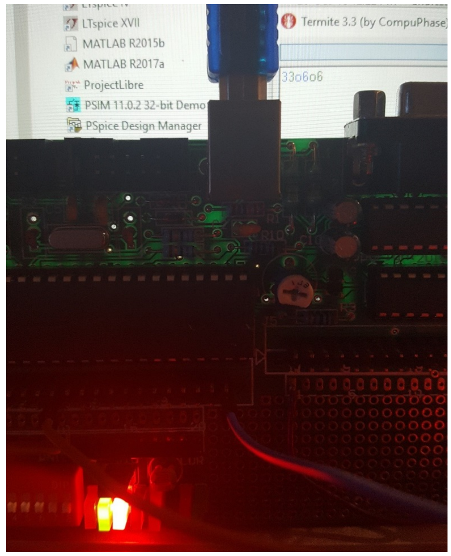

Figure 9. Testing the Serial In Code

The same code used for transmitting serial, was also used to test the LCD screen, that is, instead of wiring the Arduino Nano to the transmission pin, the LCD RX pin was connected to it instead. Just like before, a successful test, will result in the letter ‘C’ appearing every 2-3 seconds or so, on the LCD. See figure 10 (below).

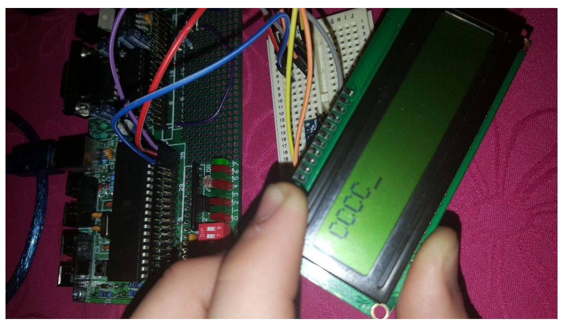

Figure 10. Testing the Serial Out to LCD Code

Second Stage:

As mentioned in the introduction, the base project consists of two OUSB boards. The first is acting as a transmitter, while the second is the receiver. As the block diagram suggests, the transmitter must use both the transmission code, as well as the keypad code. Once these two pieces of code have been joined together and are interfaced properly, then a successful test will result in the program waiting for a key to be pressed, and sending the ASCII equivalent of the key pressed over the serial port. In our case, the Arduino Nano was used to test this. The receiver, in the base project doesn’t need adjusted, and was already tested above. However, the next stage is to implement the serial receive from an external source, and serial transmit to an LCD screen. A successful test will result in the Serial data sent to the board, showing on the LCD screen. Just like before, an Arduino Nano was used to transmit data in ASCII to the OUSB board.

Third Stage

This is the final stage of the project, where the two pieces of code from the previous code will be uploaded to different OUSB boards, and tested. The test involves wiring up the receive pin from the receiver board to the transmit pin on the transmitter. A 2nd wire (ground) must also be connected between the boards for a reference. A successful test will demonstrate that when a key is pressed on the keypad, the corresponding number will show on PORTB of the 2nd OUSB board. Please see the video on GitHub (link provided in Appendix)

SPECIAL FEATURES AND CHALLENGES

The base project was just to construct a serial link between two OUSB boards, using 2-wires (single direction communication and ground), however, it was chosen to implement the bonus tasks after getting the base project to work. To see this in action, please visit the GitHub page for this project (link in the Appendix).

Once serial communication via the RS232 protocol has been established, a further challenge to upgrade the system for wireless capabilities can be attempted. Any difficulty surrounding this task was surmounted by careful selection of the HC-11 wireless module; these transceivers operate via serial connection and can quite literally work correctly in a plug-and-play fashion. Due to the nature of this problem, any issues that could arise lie in the development of the board prior to the addition of wireless functionality.

The HC-11 has a limitation of a maximum 200m in a direct line of sight (using the given antenna), making it effectively functional only for across room transmission.

The other bonus task is to implement the serial LCD module. After getting the transmit serial code to work between the OUSB boards, it was as simple as changing the port and pin to communicate to the LCD. If it receives an ASCII character that can be visualised, it will print it, otherwise it will form one of a few commands. For example, 0x0C is the line feed, which doesn't have a symbol or character, on the LCD, this clears the screen and moves the cursor back to starting position.

The third and final bonus task was to implement full duplex between the two OUSB boards, Figure 11 (below) shows a flowchart of the proposed software for it. However due to time constraints, and taking a wrong approach, this was not completed. As it turns out, the serial baud rate is faster, than the time it takes for the OUSB to save its state and move into the interrupt routine. There are two solutions which could fix this. First slowing the baud rate right down, so that there is plenty of time for the OUSB to save its state before entering the interrupt. However, the user will more than likely notice this delay. The second solution is to use an external interrupt to detect the start bit, then use the timer interrupt to periodically sample the incoming signal (but this still requires a slower baud rate).

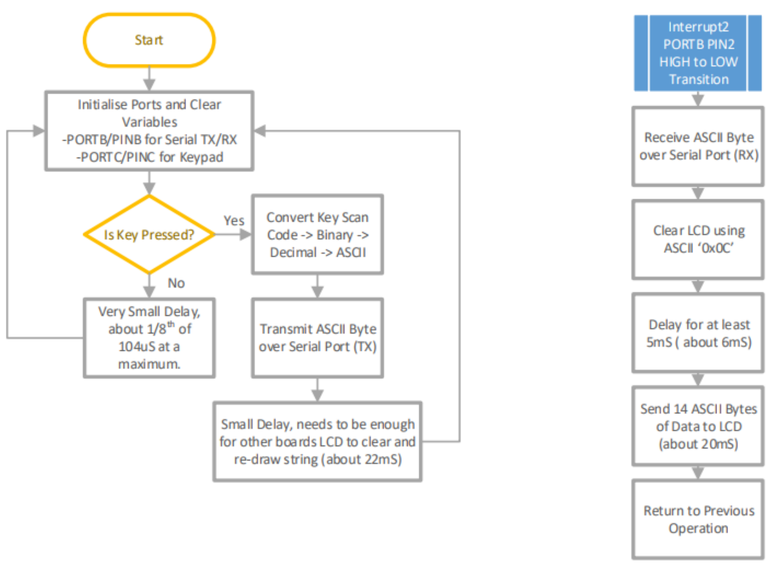

Figure 11. Proposed Full-Duplex Software Solution

INDIVIDUAL REFLECTION

Callum Jones - s3601120

Initial Evaluation:

My Mark: 4 – (difficult) I have decided to rate this project a four, because the Rs232 protocol is something we haven't learned about yet. Which means we will have to become reasonably familiar with the protocol before we can start coding it onto the OUSB boards. I am also expecting to have a little difficulty debugging the first few hardware tests. This is because the software simulator does not simulate everything, for example, the DDR registers do not have to be set up in the simulator, but do on hardware. The difficulty will increase again, when we attempt the bonus marks, adding an LCD to the 2nd board, and attempting a full duplex link between the two boards. I expect the most difficult part of this project will be the debugging of any bugs or problems in our software.

Final Evaluation:

My Mark: 3 – ("so-so") I honestly thought this project was going to be really challenging, but as it turns out, it wasn't that bad at all. The reason it has been rated 3, instead of 2, is because of the final bonus marks question, full duplex operation. Unfortunately, due to doing it the wrong way, and not finding out until the last minute, we have run out of time to complete the task. The reason this has not been rated more difficult, is because once we understood how serial worked, it was quite easy to implement in software. The reason for the difference between the initial evaluation and this evaluation is probably because the task at first looked extremely complicated. But once a block diagram of the system was drawn up, and Gantt chart created, the problem was approachable in steps at a time (and in some cases, some at the same time due to working in a team). The steps I should take solving a problem like this in the future are very similar to what I just mentioned. Look at the problem, work out what makes sense, and what doesn't. Then find out about the things that don't make sense (i.e. in our case serial made no sense), then draw a block diagram, to split the problem into many parts. This makes solving the problem easier, because each piece will be on a smaller scale. I personally found the most difficult/challenging part of this project to be debugging serial, because of how quick it is.

What would I do differently? (Technically):

What would I do differently: If was going to do this project differently, and get paid to do it, I would have chosen a much smaller microprocessor to work with. The ATMEGA32 is far too big for this project, even if extended upon. I also wouldn't choose the OUSB board as a platform for programming, mainly because the some of the technology is quite old now particularly the DB9 serial plug on PORTD. This sort of plug is not commonly found on PC's anymore, and wastes four pins from PORTD, which could be better utilised. As for the HC 11 modules, these are great for LOS (line of sight) operation or communicating within the same room, but are unable to transfer information from room to room or building to building. Perhaps there is a better option out there.

What would I do differently? (Professionally):

Professionally, the project should have been handled differently. That is the engineering design process should have been followed a little more closely. For example, during the initial phase, the design brief, the customer should be consulted for any clarifications (in this case we were the customer). Another example is the research stage, although we did research into serial, more should have been done into transmitting serial wirelessly and how it is currently implemented in a couple of different ways. Other than not documenting the report, according to the engineering design, process, the rest of the process was pretty much followed.

Possible Applications:

Using the HC 11, this project has applications that are limited to within the same room communication, much like Bluetooth. However, just about anything can be controlled using serial. For example, a friend purchased a 2nd hand till and couldn't open the draw, because it required serial communication. So we read the datasheet and found a baud of 9600 was sufficient and sending 'UUUUU' would open it (we used an Arduino UNO to do this, but a smaller AVR chip could have been used). The applications could range from controlling lights/lighting in a room, to controlling heavy machinery (provided the connection was reliable, HC 11 isn't great for much more than within room communication).

Simon Coles - s3401569

Initial Evaluation:

My Mark: 3 – (so-so) I've given this project a 3 on the basis that any code we attempt to write for our solution ought to be predominantly something we've covered before, and that generally suggests a simple task. That said, problems are guaranteed to arise, and it'll take time to learn what is causing these problems and how to fix them, bumping the difficulty of the task up a significant amount.

Final Evaluation:

My Mark: 2 – (simple) The serial communication project turned out to be substantially simpler than originally expected. The primary difficulty came from a lack of understanding about how serial worked, but after a brief amount of research, the rest of the project devolved to an almost "plug and play" effort. As I had originally expected, the majority of the code written was predominantly something that had been covered already, so that made the task comparatively easy.

What would I do differently? (Technically):

The biggest change I'd make would be to swap out the wireless modules we used for something a little more powerful; the effective range was only really across a single room with a direct line of sight. Changing from those could open up more opportunity to expand functionality of the project as its current state doesn't hold much practical value and only serves as a declaration of topical knowledge.

What would I do differently? (Professionally):

Were I to attempt this project a second time in a professional environment, I feel that a more even distribution of the workload and closely following the previously determined Gantt chart would benefit the group.

Ryan Prekop - s354439

Initial Evaluation:

My Mark: 4 – (difficult) After the previous lab, it seems that programming the OUSB board with AVR will be a lot more troublesome than it looks. So far, we have seen that there are many potential issues that cannot be seen in the software, while only showing in the hardware as "it just doesn't work", which leaves things rather difficult to diagnose without help. Unless we find some good documentation, I expect that we will have a lot of similar issues with the RS232 protocol.

Final Evaluation:

Your mark (1 to 5): 4 While the problems we had were very different to what I expected, the difficulty was much the same. Thanks to previous tasks, the input device, a keypad using pull-down resistor, was something that we knew how to configure and was quickly dealt with as a result. The problems arose when we ventured into unknown territory; the RS232 protocol - setting up the start/stop signals and rotating the stack to accept the bytes correctly were both quite troublesome tasks. Thankfully to make up for it, the wireless was straightforward and the LCD worked in a similar manner to the RS232.

What would I do differently? (Technically):

If possible, I would use a higher-level language; AVR is unnecessarily difficult and time consuming for hardware as powerful as the OUSB board. If AVR was required, then I would spend more time familiarising myself with the instruction set; namely a more elegant way to handle delays, and how to handle interrupts

What would I do differently? (Professionally):

I would put more effort into the Gantt chart, putting more time aside for the more difficult tasks instead of an even split. We were unable to complete full duplex, but it might have been possible had we dedicated more time to it.

REFERENCES

[1] P. J. Radcliffe, "Open-USB-IO Reference", OUSB datasheet. [Revised Aug. 2010]. Available: *https : //pjradcliffe.files.wordpress.com/2009/04/open − usb − io_reference_1v083.pdf*

[2] JIMB0, "Serial Communication", sparkfun.com, para. 3-6. [Online]. Available: *https : //learn.sparkfun.com/tut communication/rules − of − serial*

[3] Parallax, "Parallax Serial LCD", 27977 datasheet. [Revised Mar. 2013]. Available: *https : //www.parallax.com/sites/default/files/downloads/27979 − Parallax − Serial − LCDs − Product − Guide − v3.1.pdf*

[4] SEEED, "434MHz Wireless Serial Port Module", HC-11 datasheet. Available: *http : //wiki.seeedstudio.com/images/a/a8/HC11_User*$_M$*anual.pdf*

[5] ATMEL, "8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash", ATMEGA32 datasheet, 2011. Available: *http : //www.atmel.com/images/doc2503.pdf*

[6] 4x3 Keypad Supplied by Lecturer, part available for purchase here: *https : //www.jaycar.com.au/12 − key − numeric − keypad/p/SP0770*

[7] R. Eklind, "Language definitions and styles for listings in LaTeX.", (2014), GitHub repository, *https : //github.com/mewspring/latex*

APPENDIX A: CODE LINK

Code Link:

https://github.com/callumjones17/Atmega32_Serial_Over_RF_Using_HC_11

APPENDIX B: CODE:

Receiver Code (rx.asm):

```
;**********************************************************************
;***    RS232 Reciver Code with Attached Parallax Serial LCD Screen  ****
;**********************************************************************
;This program is intened for use with the ATMEGA32,
;where PORTC pin 7 is the RX pin, and PORTB pin 7 will be used as a TX
;pin, to serial out to the parallax LCD screen. This particular screen
;uses ascii characters to both display and control it.

;RS232 Settings RX and TX:
;Baud - 9600
;Stop Bit - 1 or 2 (Won't affect the program)
;Data Bits - 8
;No Parity Bits.

;2256 - Introduction into Embedded Systems - 2017
;Callum Jones
;RMIT

.include "C:\2256\VMLAB\include\m32def.inc"

.def  temp  =r16         ;Define registers 19-21 for use as varibles
.def result =r17
.def delay_reg =r18
.def temp2 =r19
.def ser_out = r20
.def ser_out_temp = r21
.def debug = r22                ;For slow debugging only.

.equ DelayBaud = 205                    ;Define delay counter values.
.equ DelayBaudHalf = 10

reset:
   rjmp start
   reti      ; Addr $01
   reti      ; Addr $02
   reti      ; Addr $03
   reti      ; Addr $04
   reti      ; Addr $05
   reti      ; Addr $06        Use 'rjmp myVector'
   reti      ; Addr $07        to define a interrupt vector
   reti      ; Addr $08
   reti      ; Addr $09
   reti      ; Addr $0A
   reti      ; Addr $0B        This is just an example
   reti      ; Addr $0C        Not all MCUs have the same
   reti      ; Addr $0D        number of interrupt vectors
   reti      ; Addr $0E
   reti      ; Addr $0F
   reti      ; Addr $10

start:
   nop
   nop

        clr temp                        ;Clear the registers with a 0 so they can be safely used.
        clr temp2
        clr result
        clr delay_reg
        clr debug
```

```

        ldi temp, low(RAMEND);Setup the stack point to the very bottom of RAM.
        out SPL, temp          ;When an item is pushed on the stack, it moves up the RAM (ie the address is decremented),
        ldi temp, high(RAMEND) ;so RAMEND is the perfect spot - completely out of the way of our code.
        out SPH, temp

        ldi temp, 0x80          ;Set up portb pin 7 as an output (TX pin to LCD).
        out DDRB, temp       ;The Data dirction pin needs to be set to 1 (0 for input).
        ldi temp, 0x80       ;For transmitting serial, during the unused state, the link needs to be high.
        out PORTB, temp      ;Set it high.

        ldi temp, 0x00       ;Set up portc as inputs.
        out DDRC, temp       ;Send 0's to DDRC, 0 for input, 1 for output.
        ldi temp, 0x00       ;Disable the internal pullup resistors.
        out PORTC, temp      ;This is done by writing 0 (1 for enable) to portc when its an input.

        call Delay           ;Use a delay, give the LCD some setup time.

        ldi ser_out, 0x11        ; Turn back light on. According the data sheet 0x11 turns on the backlight.
        call transmit_serial  ;Use serial to send 0x11 to the LCD.
        call delay_baud       ;Wait for a bit, give the LCD some time to write.



;------------------
;forever:
;       This is the main loop. All it does is call the RX subroutine.
;       There are no delays in this loop, because the RX pin (pin 7 of portc)
;       needs to be sampled as much quicker than the baud rate for 9600. This
;       is so the start bit is detected instead of a 0 in the data. Start bit is
;       represented by a change from high to low.
;------------------
forever:
   nop
   nop
   nop
   nop
        call rx                                         ;This sub routine checks for a low start bit and returns value if true, 0 otherwise
rjmp forever

;------------------
;rx:
;       This subroutine checks for the start bit (high to low change on the RX pin).
;       When no data is passing, the RX pin will be high. If the change is detected,
;       go to read_serial routine, otherwise go back to the main loop.
;------------------
rx:
        in temp, PINC         ;Start bit?
        andi temp, 0x80
        ldi temp2, 0x80
        cp temp, temp2
        brne read_serial
        ldi temp, 0x00
        ret


;------------------
;read_serial:
;       Start bit has already been detected, now the next 8 baud rates must be read.
;       The baud rate for this example is 9600, so the period between the bits sent
;       is 1/9600 = 104.2uS. So after the start bit, after 104uS the first data bit
;       will be sent, and after another 104uS, the 2nd data and so on. Please see
```

```
;       github for an image called serial.png for a better view. To read this serial,
;       the processor needs to sample every baud period, however the processor should
;       also delay for half a period, so that it's not sampling on the transition,
;       between data bits.
;------------------
read_serial:
        call delay_baud_half    ;Shifts our sampler, so the sampler is clear of transitions.

        ldi temp2, 0x08                 ;Number of data bits to detect.
read_bit:
        call delay_baud         ;Delay by baud_period
        in temp, PINC           ;Read from portc, into temp
        andi temp, 0x80         ;AND result with 10000000, so only pin 7 remains.
        rol temp                ;Rotate bit 7 from temp into  carry.   (ie -> 1000 0000 -> C = 1, temp = 0000 00000)  Rotating will move all bits to the left
        ror result              ;Rotate bit from carry into bit 7 of result (ie -> 0000 0000 C = 1 -> result = 1000 0000) Rotating will move all bits to the right
        dec temp2               ;Decrement our data bit counter
        brne read_bit           ;If no more bits, then continue on, otherwise go back and read the next bit.

        call delay_baud         ;Delay by baud period. After the data bits, we should expect a stop bit (HIGH).
        in temp, PINC           ;Read portc into temp
        andi temp, 0x80         ;Only interested in pin7, so and with 1000 0000
        ldi temp2, 0x80         ;Load temp2 with comparison
        cp temp, temp2          ;Compare temp2 with temp, if they are equal, then RX is high and stop bit has been recived.
        brne error_code         ;Otherwise there is an error, better handle that.

        mov ser_out, result             ;Move the Resulting ASCII character into ser_out register
        ;call transmit_serial    ;Transmit contents of ser_out to LCD
        call write_to_lcd
        call Delay_baud    ;Delay, give the LCD some time to write.
        ret
error_code:
        ldi result, 0x55        ;I have decided to use 0x55 as the error. On the OUSB board, this turns on all the RED LED's
        ret



;----------------------
;delay_baud:
;       Delays the sampler by the baud period, 104uS.
;  Using the instruction set manual for avr, the following was determined:
;       103uS = (205 * 6) + 5 * 83.33nS, baud rate is about 104uS
;  Where 205 is the number stored in DelayBaud and 6 (2 for brne) is the number of cycles in the loop.
;  The extra 5 are for the return and ldi at the start.
;----------------------
delay_baud:
        ldi delay_reg, DelayBaud      ;Load the counter value from program memory into delay_reg, 1 cycle
db_l:
        dec delay_reg                 ;decrement the counter delay reg - 1 cycle
        nop                           ;nop - 1 cycle
        nop
        nop
        brne db_l                     ; if not equal to zero, redo the loop again, otherwise return (2 cycles).
        ret

;----------------------
```

```
;delay_baud half:
;       Delays the sampler by the baud period, 104uS.
;  Using the instruction set manual for avr, the following was determined:
;       5.4uS = (10 * 6) + 5 * 83.33nS, baud rate is about 104uS
;  Where 10 is the number stored in DelayBaudHalf and 6 (2 for brne) is the number of cycles in the loop.
;  The extra 5 are for the return and ldi at the start.
;       It was mentioned earlier this value should be about half the baud_rate, but testing and experimentation,
;       resulted in the highest accuracy at 5.4uS. This is probably due to operations between reading the start bit,
;  and first data bit.
;----------------------
delay_baud_half:
        ldi delay_reg, DelayBaudHalf      ;Load the counter value from program memory into delay_reg, 1 cycle
db_h_l:
        dec delay_reg                    ;decrement the counter delay reg - 1 cycle
        nop                              ;nop - 1 cycle
        nop
        nop
        brne db_l                        ; if not equal to zero, redo the loop again, otherwise return (2 cycles).
        ret


;--------------------
;Delay:
;       Calls a lot of 21mS delay routines
;       80 Delay_more's = 80*21mS = 1.68s
;--------------------
Delay:
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
```

```
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        ret




;--------------------
;delay_more:
;       Uses inner and outer loop to delay
;       processor by 21mS. Each loop decrements counters,
;       from 255 down to zero.
;       Based on this:
;       4 cycles times 256 * 256 * 83.3 *10e-9 = about 21mS
;--------------------
Delay_more:
         PUSH R16                       ; save R16 and 17 as we're going to use them
         PUSH R17       ; as loop counters
         PUSH R0        ; we'll also use R0 as a zero value for compare
         CLR R0
         CLR R16        ; init inner counter
         CLR R17        ; and outer counter
L1:      DEC R16         ; counts down from 0 to FF to 0
                        CPSE R16, R0    ; equal to zero?
                        RJMP L1                  ; if not, do it again
                        CLR R16                  ; reinit inner counter
L2:      DEC R17
         CPSE R17, R0    ; is it zero yet?
         RJMP L1                        ; back to inner counter
;
```

```
        POP R0          ; done, clean up and return
        POP R17
        POP R16
        RET

;--------------------
;transmit serial:
;       Assumes a value for sending has been placed into ser_reg.
;       This functions transmits a serial over portb pin 7 use rotation functions.
;       The order of sending is LSB (least significant bit) first, so ser_reg is
;  rotated right, and the bit to send is moved into carry. rotating ser reg temp
;       right as well, will move the contents of carry into bit 7 of the 2nd register.
;       this bit is then written to portb. This happens every baud period 104uS after the start bit.
;       The start bit needs to be a high changing to a low, and then a stop bit is just set high.
;       See serial.png on github.
;-------------------
transmit_serial:
   clr ser_out_temp             ;Clear a temporary varible for use.

        ldi temp, 0xff                  ;Make sure serial port is already high.
        out portb, temp
        call delay_baud

        call delay_baud                         ;Start bit requires a high to low change.
        ldi temp, 0x00          ;Load 0 into temp.
        out portb, temp         ;Will set pin 7 low.

        push temp2                      ;just incase its used elsewhere, push the contents onto the stack.
        ldi temp2, 0x08         ;Load the number of data bits into the register.

write_bit:
        call delay_baud                         ;Delay by baud period.
   ror ser_out              ;rotate right, so that bit 0 (sent first) is moved into carry register.
   ror ser_out_temp         ;rotate right, so the carry contents, is moved into bit 7 of ser_out temp.
        out portb, ser_out_temp ;output to portb.
        dec temp2               ;decrement the data bit counter
        brne write_bit          ;If data bit counter is zero, then continue, otherwise go back and write another bit.

        pop temp2               ;Pull the value of temp2 from the stack and back into temp2.

        call delay_baud                 ;Delay baud before send stop bit.
        ldi temp, 0xff          ;Stop bit requires a high
        out portb, temp         ;Output high on pin 7.
        call delay_baud
        call delay_baud                 ;Technically, this is an extra stop bit. But a single stop bit reciever will also work.
        ret


;************************
;Setup LCD:
; -This function clrs the lcd and writes: 'Key Pressed: ' + key to the LCD.
;************************
write_to_lcd:
        ldi ser_out, 0x0C  ;Clears the screen and moves the cursor to 0,0
        call transmit_serial
        call delay_more

        ldi ser_out, 0x4b ;K
        call transmit_serial
        call delay_baud_half
```

```
        ldi ser_out, 0x65 ;e
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x79 ;y
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x20 ;
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x50 ;P
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x72 ;r
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x65 ;e
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x73 ;s
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x73 ;s
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x65 ;e
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x64 ;d
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x3a ;:
        call transmit_serial
        call delay_baud_half

        ldi ser_out, 0x20 ;
        call transmit_serial
        call delay_baud_half

        mov ser_out, result ; Result
        call transmit_serial
        call delay_baud_half

        ret
```

Transmitter Code:

Transmitter Code - Definitions (defs.asm)

```
.def  temp   = r16
.def delay_reg = r18
.DEF temp2 = r19
.def SER_OUT = r20
.def SER_OUT_TEMP = r21
.DEF array_count = R22

.equ delayBaud = 205
```

Transmitter Code - Main Loop (tx_kepd.asm)

```
;*********************************
;***    RS232 Transmitter Code  ****
;*********************************
;This program is intened for use with the ATMEGA32,
;where pin 8 of portb is used for transmitting serial,
;over a wired or wireless link. Portc is used in this
;program for the keypad (4x3 -> 12 key)

;RS232 Settings:
;Baud - 9600
;Stop Bit - 1 or 2 (Won't affect the program, unless your sending serial really quickly, in which case stop bits is 1)
;Data Bits - 8
;No Parity Bits.

;2256 - Introduction into Embedded Systems - 2017
;Callum Jones
;RMIT


.include "C:\2256\VMLAB\include\m32def.inc"

;The program must start at this file, org 0x00 is used to specify that this code
;will be placed at 0x00 in program memory. Also see bottom of the file for other includes.
;Note, that they are included, after this code is defined.
.org 0x00

reset:
   rjmp start
   reti      ; Addr $01
   reti      ; Addr $02
   reti      ; Addr $03
   reti      ; Addr $04
   reti      ; Addr $05
   reti      ; Addr $06        Use 'rjmp myVector'
   reti      ; Addr $07        to define a interrupt vector
   reti      ; Addr $08
   reti      ; Addr $09
   reti      ; Addr $0A
   reti      ; Addr $0B        This is just an example
   reti      ; Addr $0C        Not all MCUs have the same
   reti      ; Addr $0D        number of interrupt vectors
   reti      ; Addr $0E
   reti      ; Addr $0F
   reti      ; Addr $10

start:
   nop       ; Initialize here ports, stack pointer,
   nop       ; cleanup RAM, etc.
   nop       ;
   nop

        ldi temp, low(RAMEND)
        out SPL, temp
```

```
        ldi temp, high(RAMEND)
        out SPH, temp

        ldi temp, 0x80
        out DDRB, temp
        out PORTB, temp

        ldi     TEMP, $15     ;See the keypad project folder on github for more information
    out   DDRC, TEMP    ; set up keypad ports
    ldi  TEMP, $FF     ; Enable Pullup resistors
    out  PORTC, temp


forever:
    nop
    nop
    call Delay                          ;Give everything some time to recover
    call wait_for_key    ;Wait for a key, stored in temp
    call Convert         ;Convert scan code in temp, to a binary equivalent of the key pressed.

    push temp2                          ;Push onto stack, not really necessary, oh well.
        ldi temp2, 0x0A      ;Ok so I will be using the ascii character set to send serial, therefore the converted number must be converted to ascii before sending.
        cp temp, temp2       ;Check to see if 10 was pressed ('*').
        brne check_2         ;If not check if has was pressed.
        ldi temp, 0x2A       ;If so, move the ascii code for '*' (0x2A) into temp.
        mov ser_out, temp    ;Move temp into the serial out register, so it can be sent over a serial link
        jmp continue_main_loop
check_2:
    ldi temp2, 0x0B
    cp temp, temp2        ;Was '#' pressed (11 (0x0B))?
    brne is_number        ;If not, it must be a number.
    ldi temp, 0x23        ;If so, this is the ascii for '#'.
    mov ser_out, temp     ;Move ascii code into serial out register and continue main loop.
    jmp continue_main_loop
is_number:
        ldi temp2, 0x30        ;Ok if the key pressed is a number, then simply just add 0x30 to the number, which gives the ascii code.
        add temp, temp2        ;This is because in ascii the numbers 0-9 map to 0x30-0x39
        mov ser_out, temp      ;Move it into serial out, so it can be sent over serial link.
continue_main_loop:
        pop temp2              ;Pop off the stack.
    call transmit_serial   ;Transmit contents of ser out register over the link.
    call DelayBig          ;Delay for quite some time, so that Reciver has time to write to the LCD.
    nop        ; behaviour here
rjmp forever

;These are the other files, which contain key sub routines, just helps clean up the code.
.include "defs.asm"
.include "TX.asm"
.include "keypad.asm"
.include "delays.asm"
```

Transmitter Code - Transmit Code (tx.asm)

```
;--------------------
;transmit serial:
;       Assumes a value for sending has been placed into ser_reg.
;       This functions transmits a serial over portb pin 7 use rotation functions.
;       The order of sending is LSB (least significant bit) first, so ser_reg is
;  rotated right, and the bit to send is moved into carry. rotating ser reg temp
;       right as well, will move the contents of carry into bit 7 of the 2nd register.
;       this bit is then written to portb. This happens every baud period 104uS after the start bit.
;       The start bit needs to be a high changing to a low, and then a stop bit is just set high.
;       See serial.png on github.
;-------------------
transmit_serial:
   clr ser_out_temp            ;Clear a temporary varible for use.

        ldi temp, 0xff                  ;Make sure serial port is already high.
        out portb, temp
        call delay_baud

        call delay_baud                         ;Start bit requires a high to low change.
        ldi temp, 0x00          ;Load 0 into temp.
        out portb, temp         ;Will set pin 7 low.

        push temp2                      ;just incase its used elsewhere, push the contents onto the stack.
        ldi temp2, 0x08         ;Load the number of data bits into the register.

write_bit:
        call delay_baud                         ;Delay by baud period.
   ror ser_out             ;rotate right, so that bit 0 (sent first) is moved into carry register.
   ror ser_out_temp        ;rotate right, so the carry contents, is moved into bit 7 of ser_out temp.
        out portb, ser_out_temp ;output to portb.
        dec temp2               ;decrement the data bit counter
        brne write_bit          ;If data bit counter is zero, then continue, otherwise go back and write another bit.

        pop temp2               ;Pull the value of temp2 from the stack and back into temp2.

        call delay_baud                 ;Delay baud before send stop bit.
        ldi temp, 0xff          ;Stop bit requires a high
        out portb, temp         ;Output high on pin 7.
        call delay_baud
        call delay_baud                 ;Technically, this is an extra stop bit. But a single stop bit reciever will also work.
        ret
```

Transmitter Code - Keypad Code (kepad.asm)

```
;****************
;wait_for_key:
;       This function continuously loops around, scanning
;       individual rows, looking for a key. When a key is
;       detected, the loop is broken.
;****************
wait_for_key:
;Column 1 ;2
        ldi temp, 0xfb          ;First Column. Recall that pin 2 is the first column -> 1111 1011
        ldi temp2, 0xfb      ;A register to compare to.
        out portc, temp      ;Output required for column 1.
        call delay           ;Let output settle and capture key press.
        in temp, pinc        ;When reading an input, pinc must be used, not portc.
        cp temp, temp2       ;If the input, is the same as what was sent out...
        brne key_pressed     ;Continue, otherwise a key has been pressed, so move to key pressed.

;Column 2
        ldi temp, 0xfe
        ldi temp2, 0xfe
        out portc, temp
        call delay
        in temp, pinc
        cp temp, temp2
        brne key_pressed

;Column 3
        ldi temp, 0xef
        ldi temp2, 0xef
        out portc, temp
        call delay
        in temp, pinc
        cp temp, temp2
        brne key_pressed
        rjmp wait_for_key
key_pressed:                            ;If a key is pressed, return to main loop and continue.
        ret

;********************
;convert:
;       This takes the scan stored in temp, from the wait key press sub
;  routine, and converts it into the correct number in binary/hex.
;       eg, '1' should be 0000 0001, not 0111 1001 (scan code).
;       Basically the table at the end of this file contains all the scan
;       codes in order. This routine iterates through the table in order
;       until it finds the matching scan code. When it does, the array counter,
;       which is incremented every loop, is taken as the output.
;********************
Convert:
   ldi   ZH, high(Tble<<1)    ;Set the Z pointer to the table address. The address is shifted to the right, so
   ldi   ZL, low(Tble<<1)     ;that we can access each byte from program memory individual, instead of 2 bytes at a time.
   clr array_count                      ; start array_counter at zero

convert_loop:
   andi   TEMP, $7F
        LPM temp2, Z                    ; Now load a byte from table in memory pointed to by Z (r31:r30)
        INC ZL
   cp temp2 , Temp      ;Compare value from prgram memory (table) to scan code.
   brne continue_loop   ;Not equal, continue going through table.
        MOV Temp, array_count ;Otherwise, move array counter into the temp register, which is the output.
   ret
```

```

continue_loop:
        INC array_count         ;Increment the array counter
        cpi ZL, low(Tble<<1)+12 ;Check to see if its the end of the table (table is 12 bytes long) (0-9 + * + #) = 10+2=12 bytes
        brne convert_loop       ;If not the end of the loop, go back and check the next byte in the table.
        ldi Temp, 0xFF    ;If it is the end of the table, then an error has occured, a wrong scan code has been obtained.
        ret


;Table containing the scan codes. The index of the scan code, represents the number on the
;keypad. Eg, the number 2 is 0x7C -> 0111 1100 (scan code from reading the pins).
Tble:
.db $76, $79, $7C, $6D, $3B, $3E, $2F, $5B, $5E, $4F, $73, $67
```

Transmitter Code - Delay Code (delays.asm)

```
;----------------------
;delay_baud:
;       Delays the sampler by the baud period, 104uS.
;  Using the instruction set manual for avr, the following was determined:
;       103uS = (205 * 6) + 5 * 83.33nS, baud rate is about 104uS
;  Where 205 is the number stored in DelayBaud and 6 (2 for brne) is the number of cycles in the loop.
;  The extra 5 are for the return and ldi at the start.
;----------------------
delay_baud:
        ldi delay_reg, DelayBaud      ;Load the counter value from program memory into delay_reg, 1 cycle
db_l:
        dec delay_reg                 ;decrement the counter delay reg - 1 cycle
        nop                           ;nop - 1 cycle
        nop
        nop
        brne db_l                     ; if not equal to zero, redo the loop again, otherwise return (2 cycles).
        ret



;--------------------
;DelayBig:
;       Calls a lot of 21mS delay routines
;       13 Delay_more's = 13*21mS = 273mS
;--------------------
DelayBig:
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        call Delay_more
        ret





;--------------------
;delay_more:
;       Uses inner and outer loop to delay
;       processor by 21mS. Each loop decrements counters,
;       from 255 down to zero.
;       Based on this:
;       4 cycles times 256 * 256 * 83.3 *10e-9 = about 21mS
;--------------------
Delay_more:
         PUSH R16
         PUSH R17
         PUSH R0
         CLR R0
         CLR R16
         CLR R17
L1:      DEC R16
                        CPSE R16, R0
                        RJMP L1
                        CLR R16
L2:      DEC R17
```

```
          CPSE R17, R0
          RJMP L1
;
          POP R0
          POP R17
          POP R16
          RET



Delay:    call delay_more
          RET
```