Scrolling Marquee Display

Final Project Report
December 9, 2000
E155

Dan Smith and Katherine Wade

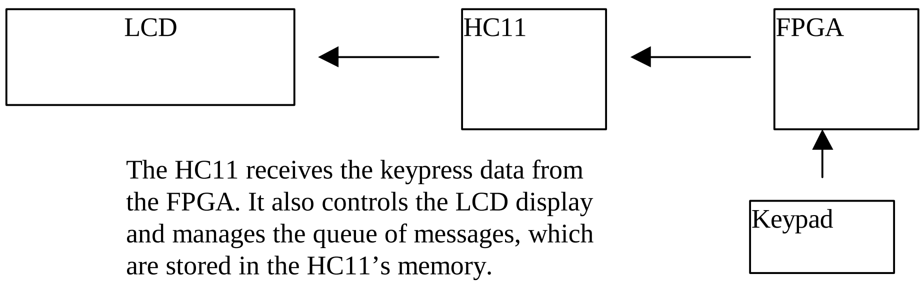
Abstract:

A Scrolling Marquee Display is a visually appealing way to display more information than can be fit upon the single screen. This project will create a scrolling Marquee Display using a 2X16 LCD that will display up to ten messages that scroll across the screen. The messages will be input to the FPGA, "video game high score"**-** style, using a 16 button keypad. The FPGA will debounce the keypad entries and pass them to the HC11, which will control the LCD functions. Users will be able to create, delete, and edit the 16-character long messages, which will then be scrolled across the LCD screen for their viewing enjoyment.

Introduction

We would like to make a nifty scrolling marquee that displays messages. However, most commercial scrolling marquees use lots of LEDs, which is difficult to control using the small-scale equipment available in the Microprocessors laboratory. Instead, we are using an LCD display with built-in character display capabilities to serve the same purpose. The messages will still be scrolled across the display as in a regular marquee. The LCD device we are using can be easily controlled with an HC11.

Overview:

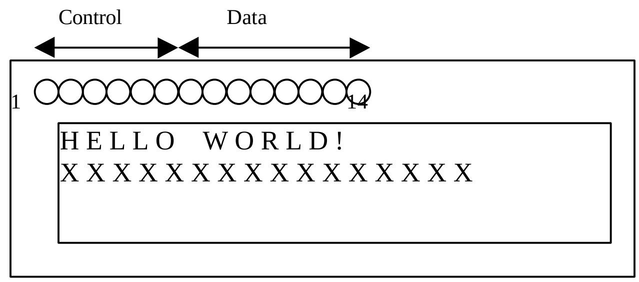


The FPGA Polls and debounces the keypad entry, and sends it via the serial port to the HC11

The HC11 receives the keypress data from the FPGA. It also controls the LCD display and manages the queue of messages, which are stored in the HC11's memory.

Optrex DMC 16249 LCD

We received the Optrex DMC-16249 2x16 LCD display. Here is what it looks like:



On the LED display, 14 pins are used to display text. The first six of these, Pins 1 – 6, are the control bits used to power the LCD and enable reading and writing. The other 8 pins, 7-14, are used to send data to the LCD.

The HC11 will control the LCD directly. A parallel port (Port B) will be used to send the 8-bit data to pins 7-14, and additional pins from the HC11 will be wired directly to the appropriate control bits.

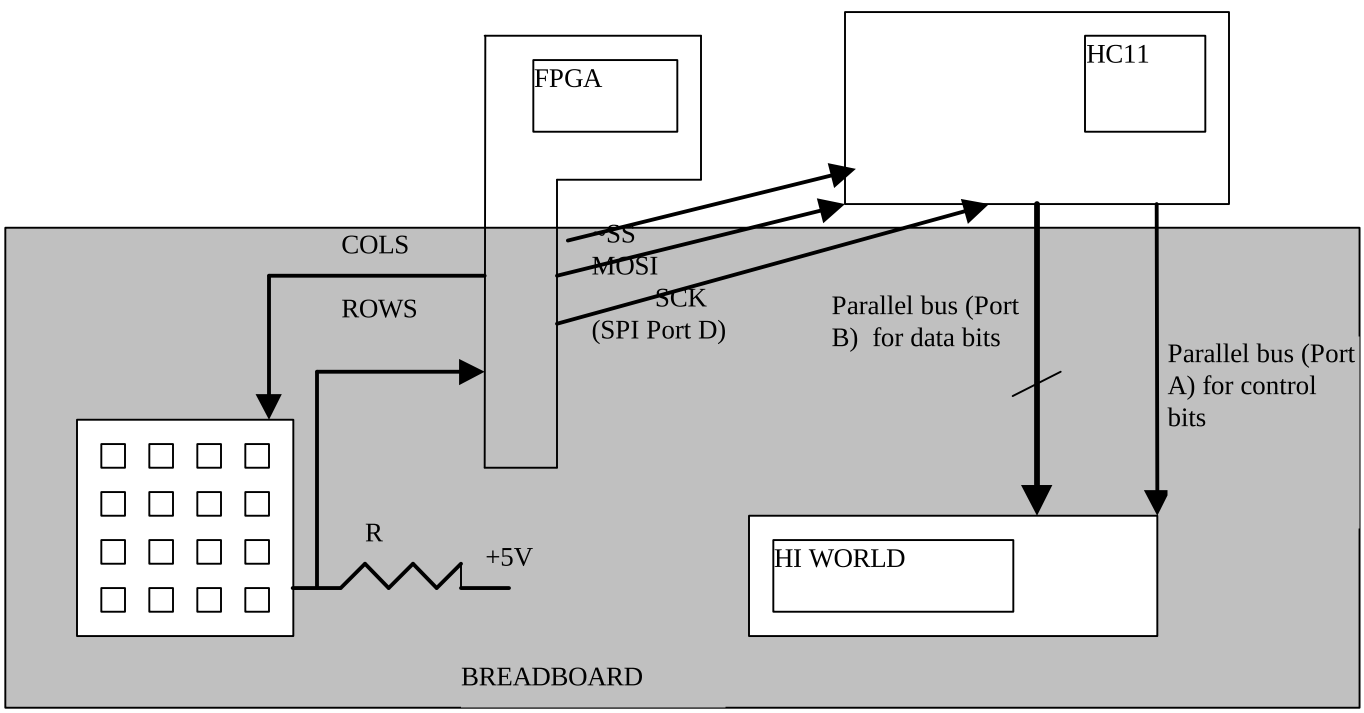
The pin holes on the LCD were filled with jumper pins and soldered to the LCD board. This way we can put the LCD on our breadboard for ease of wiring. If the jumper pins are not desired for future use, they can be unsoldered and removed.

Schematics

The FPGA will communicate with the HC11 via the SPI. To do this, we only need three wires connecting the two devices. One of the wires will send the MOSI (master out, slave in) data from the FPGA to the HC11, and the other wire will send the clock signal that will coordinate the receiving of each bit of the serial information (SCK), and the third will be slave select (~SS) that tells the HC11 when data is being sent. MOSI, ~SS, and SCK will be output pins on the FPGA and can be easily wired to the HC11EVB's input pins. In this configuration, the FPGA acts as a master, and the HC11 acts as the slave.

The 16-button keypad will be placed on the solderless breadboard and wired to the FPGA as was done in lab 4. The row pins will be inputs to the FPGA, and the column pins will be outputs so that the FPGA can poll the keypad. Polling consists of each column being alternately pulled low while the other columns are pulled high. The row pins which are input to the FPGA are attached to fairly high resistors, so that their values are by default high if no current is passed through them. If a key is pressed, the row corresponding to that keypress will be pulled low. The FPGA logic will look at the combination of row and column inputs and determine which key was pressed.

Overall Breadboard Schematic:



Microcontroller Design

The hc11 is responsible for controlling the LCD display based on the key presses on the keypad. It receives messages serially from keypad (with the keypad as master, the hc11 as slave) using the SPI interface. These messages are received as a binary encoding of the value that was pressed on the keypad. It sends messages out to the LCD display via parallel ports A and B.

The hc11 stores a queue of messages that are edited using the keypad. We decided to represent this queue with a two dimensional array. Since the LCD display shows 16 characters on screen, we decided to make each messages exactly 16 bytes. We also decided to fix the size of the queue to be 32 messages long, or 320 bytes. Empty messages in the queue are represented by all NULLs.

C000	Hello World
C010	0000000000000000
…	…
C200	0000000000000000

0…………………………………………………………………16

Message Queue

We broke the operation of the hc11 into three different modes. These modes are normal mode, message control mode, and message entry mode. In normal mode, the messages are displayed on the screen until a key is pressed, at which point the program enters message control.

```
normalMode()
{
      loop forever
      {
            scrollMessages()
            if(checkInput())
                  MessageControl()
      }
}
```

Message control mode allows the user to edit, delete, and create new messages

```
MessageControl()
{
      Loop until done is pressed
      {
            displayMessage(selected message)
            input = getInput()
            switch(input)
                  up: increment selected message
                  down: decrement selected message
                  new: new()
```

```
                        edit: edit(selected message)
                        remove: remove(selected message)
            }
}
```

The remove() function just converts the selected message to NULLs. The new() function finds an empty slot in the array, converts all the characters to spaces, and then calls edit() on that position.

```
new()
{
      find empty message
      convert to spaces
      edit(empty message)
}
```

The edit() function is the third mode of operation: message entry. The edit function gets keystrokes from the user and modifies the message. It keeps a cursor position on the message and allows the user to change the current character, or move left or right.

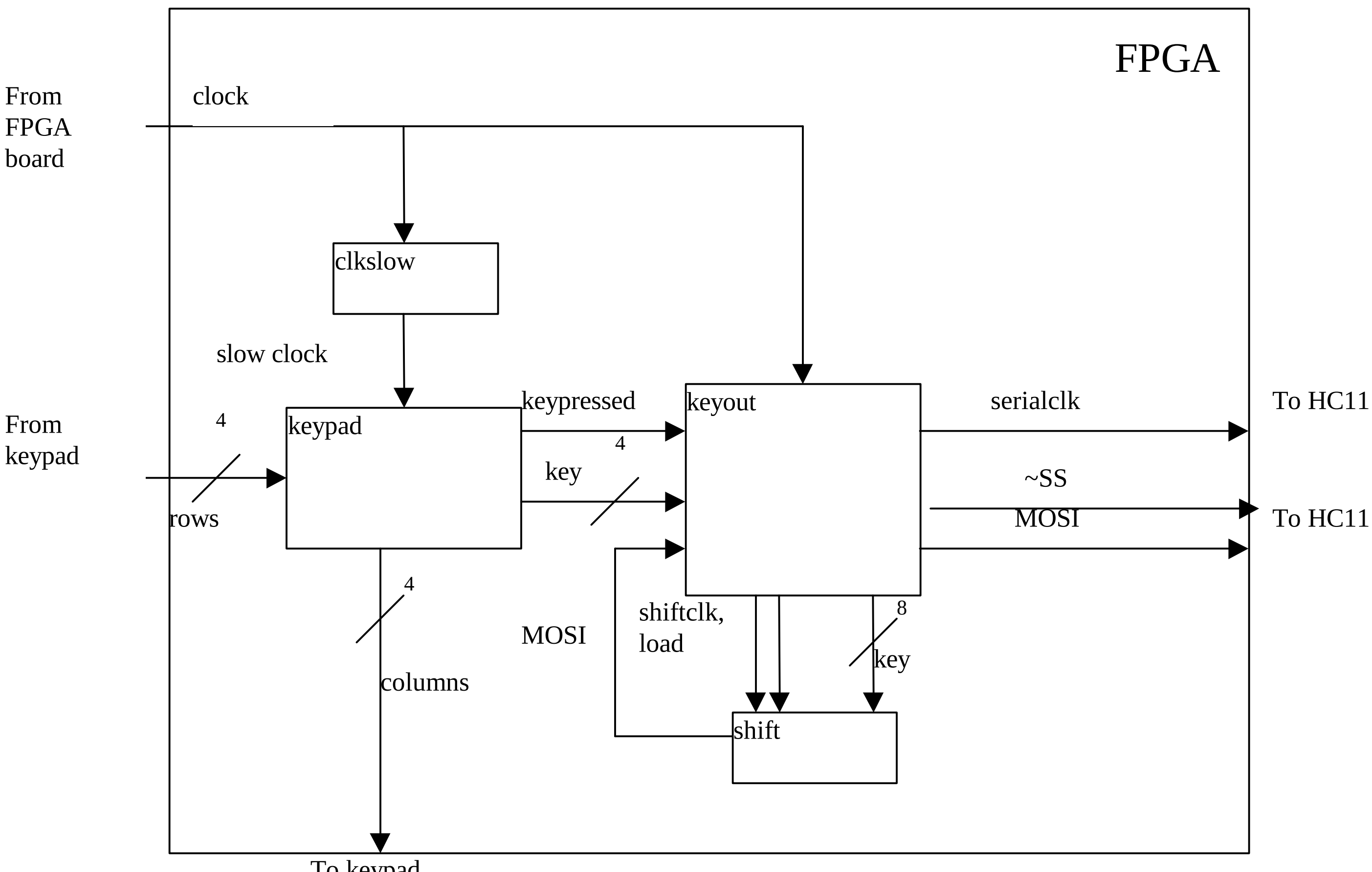
```
edit()
{
      loop until done is pressed
      {
            displayMessagewithCursor(cursor position)
            input = getInput()
            switch(input)
            up: increment selected message[cursor position]
            down: decrement selected message[cursor position]
            left: increment cursor positions
            right: decrement cursor position
      }
}
```

FPGA Design

Describe the function of the hardware in your FPGA, including inputs, outputs, and major hardware modules. Describe the key logic, using datapath or FSM diagrams as needed. This section should give the reader enough information to understand the Verilog code and/or FPGA schematics in the appendix.

So far we have coded modules in Verilog to take care of the polling, debouncing, and SPI communication. The code for these modules can be found in Appendix A.

FPGA Logic Diagram:



Explanation of modules:

The *main.v* module is the controlling module that calls all the other ones. It takes in the global inputs of *clk*, *reset*, and *rows*, and outputs the global outputs *cols*, *MOSI*, *~SS*, and *serialclk*.

The *clkdiv.v* module takes the FPGA's internal clock and slows it down by a factor of 2^11. This clock is then used in the keypad module to debounce the signal by sampling it at greater than 5 ms. This slowed-down clock is outputted to the *keypad.v* module as slowclk.

The *keypad.v* module outputs the cols data, pulling each column alternately low. It also takes in the *rows* data, and decodes the *rows* and *cols* data together to determine if a key was pressed (*keypressed*) and what it was (*key*). Each keypress is sampled and debounced using *slowclk*.

The *shift.v* module is a 8-bit shift register which takes in a *clk*, *load,* and in*data*, and outputs MOSI. When *load* is enabled, the in*data* is loaded directly into the shift register in parallel. The *clk* then shifts the data through the registers. The MSB is outputted as *outdata*.

The *keyout.v* module takes in *clk*, *reset*, *keypressed*, *key*, and outputs *MOSI*, *~SS*, and *serialclk*. It stores key in *shift.v* and then generates clocks for the serial output and the shift register. The shift register clock is the opposite of the serial output clock, delayed by one clock cycle. The *serialclk*, *~SS,* and the *MOSI* are outputted to the FPGA in SPI format.

Results

We were able to complete our project as proposed. The marquee could store up to 32 messages, with each message being 16 characters long. The marquee could scroll the messages, and supported the add, delete, edit, and new message operations described above. The edit message mode supported the character 'a-z', "A-Z', and some special characters like ' ', "#', '!', etc. Scrolling through all these characters to find the one you wanted was a little cumbersome. It was suggested to us that instead of using one keypad and entering in messages video game style, we could instead use two keypads and enter in the letters "A-Z' directly (26 buttons for 'A-Z', 6 control buttons).

One of the difficult parts of the project was trying to get the FPGA to communicate serially with the HC11 using the FPGA as a master. The Verilog code to send data using this technique ended up being more difficult than we thought. However, we were able to get it working.

We were unable to write the code in C as we hoped, because we did not get the C compiler working in time. However, after we had written the code in assembly, we did get the C compiler to work. We were unable to use the compiler for our project because we had already written the assembly code, however, we hope that students will be able to use what we've learned about the compiler for future projects.

References

Check the following links for up-to-date information about LCDs.

[1] Fil's FAQ Link-In Corner: LCD Program and Pinout FAQ, `http://www.repairfaq.org/filipg/LINK/F_LCD_progr.html`
[2] Optrex DMC-16249 Documentation, `http://www.optrex.com/pdfs/Dmcman_full.pdf`

Parts List

List all of the components you used other than standard resistors, capacitors, and parts available in the MicroP's lab.

Part	**Source**	**Vendor Part #**	**Price**
Optrex DMC 16249 2x 16 LCD	MicroPs lab		
16-button keypad	MicroPs lab		

Appendix A: Pinouts

Serial Interface:

	FPGA	HC11
MOSI	45	PD3
SCK	46	PD4
~SS	47	PD5

LCD Interface:

LCD	HC11	OTHER
Vss		GND
Vcc		Vcc
Vee		GND
RS	PA4	
R/W	PA5	
E	PA3	
DB0	PB0	
DB1	PB1	
DB2	PB2	
DB3	PB3	
DB4	PB4	
DB5	PB5	
DB6	PB6	
DB7	PB7	

Keyboard Interface:

Keyboard	FPGA
Row0	P38
Row1	P39
Row2	P40
Row3	P44
Col0	P18
Col1	P19
Col2	P20
Col3	P23

Appendix B: Report on GCC Compiler

Report on gcc

We tried using a C compiler for the HC11 instead of writing in assembly. We were unable to get it to work before writing our assembly code, however after we were done we realized that we were not point the compiler at a valid location for the stack. After giving it a good location for the stack, the compiler worked.

The compiler is a port of gcc, a free compiler written by the gnu project. It does not yet support C++, but the C compiler works. It also comes with a simulator, a debugger, and an assembler. The simulator works fine. The debugger is supposed to work with both the simulator and buffalo, but we only got it working with the simulator. The assembler might be useful for linking in assembly functions with C code but it does not use the same syntax as as11 and we couldn't get it working.

The compiler is availible on the web from http://home.worldnet.fr/~stcarrez/m68hc11_port.html. I compiled it under linux, but the author has binaries for windows availible from his webpage. You need to get both the binutils and gcc for a basic setup, but I'd recommend the debugger. You might also want the newlib library, which has some useful functions. Follow his instruct for installing the stuff, for linux it was very straightforward.

In order to get the code to put the code, data, and stack in the right locations, you need to put a file called memory.x in the directory which you are compiling in. The format of memory.x is shown below (I copied this from one of his examples, but modified it to use locations which work with buffalo).

To compile code into .s19 so you can download it to the hc11, you do the following:

? compile the .c code into an .o file
  m68hc11-elf-gcc -mshort -g -Wall -0s file_name.c
? link the .o file(s) into a .elf file
  m68hc11-elf-gcc --mshort -Wl,-m,m68hc11elfb file_name.elf file_name(s).o
? translate the .elf file into .s19
  m68hc11-elf-objcopy --output-target=srec --only-section=.text
  --only-section=.rodata --only-section=.vectors file_name.elf file_name.s19

If you want to run the simulator or the debugger, you just need the .elf file. The -Wl,-m,m68hc11elfb option is telling the linker to use the memory.x file you specified, instead of just putting the code into default locations which won't work the hc11's we have, so if you don't do this it'll simulate fine but won't work on the board. the --only-section parts of the objcopy don't seem to matter, but the author used them in his examples, so...

We didn't test the compiler very thoroughly, but it seemed to work. We did notice that it seemed to be using unsigned comparisons (<, >), so be careful with comparisons.

memory.x file:

```
/* Fixed definition of the available memory banks.
   See generic emulation script for a user defined configuration.  */
MEMORY
{
  page0 (rwx) : ORIGIN = 0x0, LENGTH = 30
  text  (rx)  : ORIGIN = 0xC200, LENGTH = 0x1E00
  data        : ORIGIN = 0xC000, LENGTH = 0x200
}
/* Setup the stack on the top of the data memory bank.  */
PROVIDE (_stack = 0xC200 - 1);
```

Sample Makefile:

```
#Makefile
# Dan Smith
# written to compile .elf and .s19 files from c code

#programs
BASE = m6811-elf-
CC = $(BASE)gcc
OBJCOPY = $(BASE)objcopy

#compiler options
CFLAGS = -mshort -g -Wall -Os
LFLAGS = -mshort -Wl,-m,m68hc11elfb
OFLAGS = --output-target=srec --only-section=.text --only-
section=.rodata --only-section=.vectors

#change this to build something else
TARGET = greatest.elf
STARGET = greatest.s19
OBJECTS = greatest.o

all: $(TARGET)

$(TARGET): $(OBJECTS) memory.x
	$(CC) $(LFLAGS) -o $(TARGET) $(OBJECTS)
	$(OBJCOPY) $(OFLAGS) $(TARGET) $(STARGET)

%.o: %.c
	$(CC) $(CFLAGS) -c $<

clean:
	rm *.o
	rm *.elf
	rm *.s19
```

Appendix C: Verilog Files

```
module main (clk, reset, rows, cols, mosi, serialclk, slaveselect) ;

input clk;              //FPGA internal clock
input reset;            //FPGA internal reset (big red button)

input [3:0] rows;
output [3:0] cols;

output mosi;            //the keypress bits for serial output
output serialclk;       //the clock used to control the serial transfer
output slaveselect; //goes low during the transmission, high otherwise

wire slowclk;           //the slowed-down FPGA clock
wire[3:0] key;          //the key that was pressed

clkdiv  myclkdiv(clk, reset, slowclk);              //slow down the clock
keypad  mykeypad(slowclk, reset, rows, cols, key, keypressed);      //poll the keypad
keyout  mykeyout(clk, reset, key, keypressed, mosi, serialclk, slaveselect);
        //output the pressed key serially

endmodule

///////////////////////////////////////
module clkdiv(clk, reset, slowclk);

input clk;              //FPGA internal clock
input reset;            //FPGA internal reset (big red button)
output slowclk;         //the slowed-down FPGA clock

reg[11:0] count;

//This synthesizes to an asynchronously resettable counter.
//The reset line is tied to the global set/reset line of the FPGA
always@(posedge clk or posedge reset)
        if (reset) count = 0;
        else count = count +1;

assign slowclk = count[11];

endmodule

///////////////////////////////////////////////////////
module keypad(slowclk, reset, rows, cols, key, keypressed);

input slowclk;          //slowed-down clock
input reset;            //FPGA internal reset (big red button)

input [3:0] rows;
output [3:0] cols;

output [3:0] key;       //contains the binary value of the pressed key
output keypressed;     //a key was pressed

reg keypressed;         //for FSM
reg [3:0] cols;
reg [3:0] key;

//scanning FSM
always @(posedge slowclk or posedge reset)
        if (reset) begin
                keypressed <=0;
                cols <= 4'b0111;
        end else if (&rows) begin
                //no key pressed on this column, so keep scanning
                keypressed <= 0;
                cols <= {cols[0], cols [3:1]}; //shift cols right
        end else if (~keypressed) begin
                keypressed <= 1;
        end
        //otherwise wait until all keys are released before continuing

//keypad conversion
always@(rows or cols)
```

```
	case ({rows, cols})
		8'b0111_0111: key <= 'hC;
		8'b1011_0111: key <= 'hD;
		8'b1101_0111: key <= 'hE;
		8'b1110_0111: key <= 'hF;
		8'b0111_1011: key <= 'h3;
		8'b1011_1011: key <= 'h6;
		8'b1101_1011: key <= 'h9;
		8'b1110_1011: key <= 'hB;
		8'b0111_1101: key <= 'h2;
		8'b1011_1101: key <= 'h5;
		8'b1101_1101: key <= 'h8;
		8'b1110_1101: key <= 'h0;
		8'b0111_1110: key <= 'h1;
		8'b1011_1110: key <= 'h4;
		8'b1101_1110: key <= 'h7;
		8'b1110_1110: key <= 'hA;
		default: key <= 'h0;
	endcase

endmodule

/////////////////////////////////////////////
module shift (clk, reset, load, indata, outdata) ;

//this is a basic 8-bit shift register

input clk, reset ;
input load;
input [7:0] indata ;
output outdata ;

reg [7:0] data;

always @(posedge clk or posedge reset)
begin
  if(reset == 1)
	data <= 8'b1010_1010;
  else if (load == 1)
			data  <= indata; //if loading, immediately load everything in
		else
			begin				//otherwise, shift everything through
			data[7] <= data[6];
			data[6] <= data[5];
			data[5] <= data[4];
			data[4] <= data[3];
			data[3] <= data[2];
			data[2] <= data[1];
			data[1] <= data[0];
			data[0] <= 1;
			end
end

	assign outdata = data[7];	//the 7th bit is the MOSI out.

endmodule

////////////////////////////////////////////////////////////////////////
module keyout (clk, reset, key, keypressed, mosi, serialclk, slaveselect) ;

input clk, reset;		//serial clock
input [3:0] key;		//the key that was pressed
input keypressed;

output mosi;		//the keypress bits for serial output
output serialclk;	//clock that controls the serial transfer
output slaveselect; //low during transmission, high otherwise.

reg [7:0] shiftreg;	//shift register that holds the serial output
reg [3:0] bitcounter;

reg	shiftclk;
reg	serialclk;
reg	previouslynotpressed;
reg		load;
```

```
always @(posedge clk or posedge reset)               //at every clock tick
        if(reset)
                begin
                        serialclk <= 0;
                        shiftclk <= 0;
                        load <= 1;
                        bitcounter <= 4'h8;
                        previouslynotpressed <= 1;
                end
        else if(keypressed)
                begin
                        if(previouslynotpressed)
                                begin
                                        load  <= 1;
                                        shiftclk <= 1;   //the shifting clock is high
                                        serialclk <= 0;  //the serial clock outputted to the
                                                         //HC11 is low
                                        bitcounter <= 0; //we initialize the bit we start
                                                         //transmitting to 0
                                        previouslynotpressed <= 0;
                                end
                        else
                                begin
                                        load <= 0;
                                        if(bitcounter < 4'h8)  //if we haven't yet
                                                               //transmitted bit 8,
                                                begin
                                                        shiftclk <= ~shiftclk;
                                                        //toggle shift clock
                                                        serialclk <= ~serialclk;
                                                        //toggle serial clock
                                                        if(shiftclk)
                                                                bitcounter <= bitcounter + 1;
                                                end
                                        else
                                        begin
                                                //otherwise, we're on bit 8.
                                                //end. transmission.
                                                        shiftclk <= 0;
                                                //shift clock stays low
                                                        serialclk <= 0;
                                                //the serial clock goes low (we aren't
                                                //transmitting everything)
                                                end
                                end
                end
        else
                begin
                        previouslynotpressed <= 1;
                        shiftclk <= 0;
                        serialclk <= 0;
                        bitcounter <= 4'h8;
                        load <= 1;
                end


assign slaveselect = ~(keypressed & (bitcounter < 4'h8) | serialclk);
//slave select goes low when we transmit data

shift outRegister(shiftclk, reset, load, {4'b1111, key}, mosi);
//shift out the data with 1’s in the bits we don’t use.

endmodule
```

Appendix D: Assembly Code

queue.asm

```
************************************************************
* Dan Smith
* 11/28/00
* e155 final project
* main event loop
* description - this program maintains a queue of messages
* it allows editing of the messages. It displays the messages
* to an lcd display and receives the messages from an FPGA. This
* file contains the message control aspects.

************************************************************
*constants

*locations, sizes
QUEUE_LOC    EQU $C000
QUEUE_END    EQU $C200
MESSAGE_SIZE EQU $10
NG_MESS_SIZE EQU $fff0
NUM_MESSAGES EQU $1f
STACK_LOC    EQU $C380
PROGRAM_LOC  EQU $C400
SCRATCH_LOC  EQU $C390  ;used for scratch space in memory

*keys
UP          EQU $FD
DOWN        EQU $F6
LEFT        EQU $F3
RIGHT       EQU $F9
NEXT        EQU $F5
PREV        EQU $F2
DONE        EQU $F8
DELETE            EQU $F7
ADD         EQU $F4
EDIT        EQU $F1

*chactacters
SPACE       EQU ' '
NULL        EQU #0
FIRST_CHAR    EQU ' '
LAST_CHAR    EQU 'z'

************************************************************
* initial queue

      org QUEUE_LOC
      fcc "Marquee         "
      fcc "Display         "
      fcc "Katherine       "
      fcc "            &Dan"
      fcc "E155 is fun&EZ!!"
PREDEF:     fcc "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"


************************************************************
* initialization code
* clear the queue, initialize the i/o devices
      org  PROGRAM_LOC
```

```
      lds  #STACK_LOC
      jsr  INITSPI            ;initialize the serial interface
      jsr  INITDR       ;initialize the display
      jsr  CLEAR        ;clear the display
      jsr  CLEAR_QUEUE  ;clear the queue
      ldx  #QUEUE_LOC         ;load the X register with a pointer to
queue

*************************************************************
*main loop
* scroll through messages, on a keypress go the the next message
MAIN:
      pshx              ;push x onto the queue for subroutine calls
      jsr  DISPLAY_MESSAGE
      jsr  WAITASEC
      pulx
      inx               ;shift the display by 1 (scrolling)
      clra
      cmpa 0,X          ;check to see if the next value is null
      bne  CONT1        ;if the next value is not null go on
      ldx  #QUEUE_LOC         ;go back to the beginning of the queue
CONT1:      pshx
      jsr  TESTDATA           ;after this, A will be 0 if no input
      pulx
      cmpa #$0
      beq  MAIN         ;if there is no input, display again
      pshx
      jsr  MESSAGE_CONTROL    ;otherwise
      pulx
      bra  MAIN

******************************
*main loop helper functions

*clear the queue
CLEAR_QUEUE:
      ldy  #PREDEF            ;start the end of the predefined messages
      ldab #NULL        ;write null to the queue
LOOP4:  stab 0,Y
      iny
      cpy #QUEUE_END
      bne LOOP4

      rts


*************************************************************
*message control loop
*create, delete, edit messages
*sequence
* display the current message
* read input (wait for it)
* switch(input)
*  create a new message
*  delete current message
*  edit the current message
*  go to the next message
*  go to the previous message
*  return to main loop
```

```
MESSAGE_CONTROL:
      ldx  #QUEUE_LOC         ;start at the beginning of the queue
LOOP1:      pshx
      jsr  DISPLAY_MESSAGE    ;display the message
      jsr  GETDATA            ;wait for input
      pulx

      cmpa #NEXT        ;go to the next message
      bne  CASEA1
      jsr  NEXT_MESSAGE
      bra  LOOP1

CASEA1:     cmpa #PREV        ;go to the previous message
      bne  CASEA2
      jsr  PREV_MESSAGE
      bra  LOOP1

CASEA2:     cmpa #DELETE            ;delete a message
      bne  CASEA3
      jsr  DELETE_MESSAGE
      bra  LOOP1

CASEA3:     cmpa #EDIT        ;edit a message
      bne  CASEA4
      jsr  EDIT_MESSAGE
      bra  LOOP1

CASEA4:     cmpa #ADD         ;add a message
      bne  CASEA5
      jsr  NEW_MESSAGE
      bra  LOOP1

CASEA5: cmpa #DONE            ;return to main loop
      bne  CASEA6
      rts
      bra  LOOP1

CASEA6:     bra  LOOP1        ;default case


******************************
*message control helper functions

*go to the next message
* X = message
NEXT_MESSAGE:
      ldab #MESSAGE_SIZE
      abx
      clra              ;check to see if we have reached the end
      cmpa 0,X
      bne  CONT2
      ldx  #QUEUE_LOC         ;if we have, go to the beginning
CONT2:      rts

*go to the previous message
* X = message
PREV_MESSAGE:
      cpx  #QUEUE_LOC         ;check to see if we have reached the
beginning
      bne  CONT3
```

```
      rts                 ;if we have, don't go anywhere
CONT3:
      xgdx                ;go back a message
      addd  #NG_MESS_SIZE      ;
      xgdx                ;
      rts


*message creation
NEW_MESSAGE:
      jsr  FIND_TAIL           ;find the end of the queue
      cmpa #$0            ;see if the queue is full
      bne  CONT4          ;if the queue is not full, branch
      pshx                ;display an error
      jsr  ERROR
      pulx
      rts

CONT4:      pshx               ;set message might corrupt X
      ldaa #SPACE
      jsr  SET_MESSAGE  ;set the message to spaces
      pulx
      jsr  EDIT_MESSAGE ;edit the newly created message
      rts


*message deletion
* X = message
DELETE_MESSAGE:
      pshx
      jsr  MOVE_TAIL           ;move the last nonempty message to
*                         ;the deleted locations
      pulx
      jsr FIND_TAIL            ;find the tail
      cpx #QUEUE_LOC           ;if the queue is empty return
      beq CONT12          ;

      xgdx                ;
      addd #NG_MESS_SIZE       ;find the last nonempty message
      xgdx                ;
      ldaa #NULL
      pshx
      jsr  SET_MESSAGE  ;set the last message to NULL
      pulx
CONT12:     rts


*find the tail of the queue (the first empty message
*if the queue is full, store 0 in accumulator A
*otherwise store 1
*tail is stored in X
*return in A
FIND_TAIL:
      ldx  #QUEUE_LOC          ;start a message below the queue
      xgdx
      addd #NG_MESS_SIZE
      xgdx
      ldaa #NULL               ;we want to check for NULL
*                              ;at each message
LOOP2:      ldab #MESSAGE_SIZE            ;increment the pointer
```

```
      abx
*     note, we want to check to see if we are at the last message
*     the reason is because we cant to leave the last message all NULLs
      cpx  #QUEUE_END
      bne  CONT5
      ldaa #0                       ;if the queue is full return 0
      rts

CONT5:      cmpa 0,X
      bne  LOOP2              ;if this isn't the end of the queue, go *
                        ;on

      ldaa #1                       ;return 1
      rts


*move last message in queue to a location pointed to by the X index
register
*used for delete.
* X = message
MOVE_TAIL:
      pshx
      jsr  FIND_TAIL
      puly              ;getting tricky, we want Y to have the old
*                       ;location and X to have the tail of the list
      cpx  #QUEUE_LOC         ;check to see if tail = head (empty queue)
      beq  CONT6        ;if they're equal just return

      xgdx              ;
      addd #NG_MESS_SIZE      ;
      xgdx              ;go to the last nonempty message
      sty  SCRATCH_LOC  ;compare y to x
      cpx  SCRATCH_LOC  ;
      beq  CONT6        ;if they're equal, just return

      ldab  #MESSAGE_SIZE     ;initialize counter
LOOP6:  ldaa  0,X       ;get a byte from the tail
      staa  0,Y         ;put it in the location
      inx               ;
      iny               ;go to the next byte
      decb              ;decrement counter
      bne LOOP6         ;loop

CONT6:      rts

************************************************************
*message editing
*this function assumes the X index register is already pointing
*at the message in memory. It just diplays the cursor and changes
*the characters
* sequence
* display message
* display cursor
* get input
* switch (input)
*  move cursor left
*  move cursor right
*  increment character
*  decrement character
*  return to message control
```

```
* X = message

EDIT_MESSAGE:
      stx  SCRATCH_LOC  ;
      ldy  SCRATCH_LOC  ;copy X to Y
      pshy
      pshx
      jsr  DISPLAY_MESSAGE    ;display the message
      jsr  HOME
      jsr  CUR_ON       ;turn on the cursor
      pulx
      puly

LOOP7:      pshy
      pshx
      jsr  GETDATA            ;wait for input
      pulx
      puly

      cmpa #UP          ;go up a character
      bne  CASEB1
      jsr  UP_CHAR
      bra  LOOP7

CASEB1:     cmpa #DOWN        ;go down a character
      bne  CASEB2
      jsr  DOWN_CHAR
      bra  LOOP7

CASEB2:     cmpa #LEFT        ;go left
      bne  CASEB3
      jsr  PREV_CHAR
      bra  LOOP7

CASEB3:     cmpa #RIGHT       ;go right
      bne  CASEB4
      jsr  NEXT_CHAR
      bra  LOOP7

CASEB4: cmpa #DONE            ;return to message control loop
      bne  CASEB5
      pshx
      jsr CUR_OFF
      pulx
      rts
      bra  LOOP7

CASEB5:     bra  LOOP7        ;default case

      rts

******************************
*edit message helper functions

*rotate character pointed at by Y up
*Y = char
UP_CHAR:
      inc  0,Y          ;go up a character
      ldaa 0,Y          ;
      deca              ;
```

```
      cmpa  #LAST_CHAR  ;check to see if we're at the last character
      bne  CONT8
      ldaa #FIRST_CHAR  ;
      staa 0,Y          ; jump to the first character

CONT8:      ldaa  0,Y         ; write the character to the LCD
      pshx
      pshy
      jsr WRITED
      jsr CUR_LEFT
      puly
      pulx
      rts


*rotate character down
*Y = char
DOWN_CHAR
      dec  0,Y          ;go down a character
      ldaa 0,Y          ;
      inca              ;
      cmpa  #FIRST_CHAR ;check to see if we're before the first char
      bne  CONT9
      ldaa #LAST_CHAR         ;
      staa 0,Y          ;jump to the last character

CONT9:      ldaa 0,Y          ;write the character to the LCD
      pshx
      pshy
      jsr WRITED
      jsr CUR_LEFT
      puly
      pulx
      rts

*go to the next character (in memory and display)
*Y = char
* X = start of message
NEXT_CHAR:
      pshx              ;save for later
      ldab #MESSAGE_SIZE
      decb
      abx               ;X now holds the end of the message
      stx  SCRATCH_LOC  ;
      pulx
      cpy  SCRATCH_LOC  ;check to see if Y is the end of the message
      bne  CONT10
      stx  SCRATCH_LOC
      ldy  SCRATCH_LOC  ;go the the beginning of the message
      pshx
      pshy
      jsr HOME          ;move the cursor home
      puly
      pulx
      rts

CONT10:     iny               ;increment the pointer
      pshx
      pshy
      jsr CUR_RIGHT           ;move the cursor left
```

```
      puly
      pulx
      rts


*go to the previous character (in memory and display)
*Y = char
*X = start of message
PREV_CHAR:
      stx  SCRATCH_LOC  ;
      cpy  SCRATCH_LOC  ;check to see if Y is the start of the message
      bne  CONT11
      rts               ;just stay at beginning if at beginning, no
wrap*                         ;around

CONT11:     dey               ;decrement the pointer
      pshx
      pshy
      jsr CUR_LEFT            ;move the cursor right
      puly
      pulx
      rts

************************************************************
*generic helper functions

*display a message
* X = pointer to message
DISPLAY_MESSAGE:
      ldab #MESSAGE_SIZE      ;initialize counter
      incb
      pshx
      pshb
      jsr  HOME         ;go to the beginning of the display
      pulb
      pulx
LOOP3:
      ldaa  0,X
      pshx
      pshb
      jsr  WRITED
      pulb
      pulx
      inx               ;increment the pointer
      decb              ;decrement the counter
      bne  LOOP3        ;loop until counter=0
      rts

*set a message to the value accumulator A
*used to clear a message or initizialize it to some character
* A = value to set
* X = pointer to message
SET_MESSAGE:
      ldab #MESSAGE_SIZE            ;b is a counter
LOOP5:  staa 0,X
      inx
      decb
      bne LOOP5

      rts
```

```
*display an error message

ERRM  FCC "ERROR                          "      ;the acual error message
ERROR:      ldx #ERRM                     ;
      jsr DISPLAY_MESSAGE                 ;display the string above
      jsr WAITASEC                        ; delay for 1 second
      jsr WAITASEC
      jsr WAITASEC
      ldx QUEUE_LOC                       ;reinitialize X
      rts

*wait one-third of a second
WAITASEC:   ldab #10    ; 10 overflows
DELAY1:           ldaa #10000000    ; clear the TOF to start the delay
            staa $1025  ; store in TFLG2
SPIN1:            tst  $1025  ; do 10 overflows for approx. 1/3 sec
            bpl  SPIN1  ; is flag 0? branch on bit 7 is clear
            decb        ; decrement counter
            bne  DELAY1 ; if we haven't counted to 0 yet, delay again
            rts
```

interface.asm

```
*LCD Assembly Subroutines
*Katherine Wade
*11/30/00
******************************
*this code is based upon Jason Fong and Ferndando Mattos' code from last
year.
***********************

PORTA EQU $1000   *LCD Control Register
PORTB EQU $1004   *LCD Data Register

DDRD  EQU $1009   *SPI Configuration Register
SPCR  EQU $1028   *SPI Control Register
SPSR  EQU $1029   *SPI Status Register
SPDR  EQU $102A   *SPI Data Register

ZERO  EQU $0000   *for comparison purposes
DELAY EQU $0002   *holds the amount to wait

************************************
*this org is commented out so that this code will be put following
*the queue code in memory by the assembler
*        ORG $c000

*************************************************************
*initialize the serial port as a slave
INITSPI: ldaa #%00000100
         staa DDRD
         ldaa #%01001100
         staa SPCR
         clra
         rts

****************************************************
INITDR: ldaa #$38       //initializes the LCD driver
        jsr  WRITEC
        ldaa #$38
        jsr  WRITEC
        ldaa #$38
        jsr  WRITEC
        ldaa #$06
        jsr  WRITEC
        ldaa #$0C
        jsr  WRITEC
        rts

********************************
*port A
*bit 3 = enable
*bit 4 = register select (0 for control)
*bit 5 = R/W (0 for writing data)
WRITEC: ldab PORTA      //writes to the LCD control, control data in
acc.A
        andb #%11000111
        stab PORTA
        staa PORTB
        ldab PORTA
```

```
        andb #%11001111
        orab #%00001000
        stab PORTA
        ldab PORTA
        andb #%11000111
        stab PORTA
        ldab PORTA
        andb #%11100111
        orab #%00100000
        stab PORTA
        ldaa #10        //wait 10 ms
        staa DELAY
        jsr WAIT
        rts

**********************************
*port A
*bit 3 = enable
*bit 4 = register select (1 for data)
*bit 5 = R/W (0 for writing data)
WRITED: ldab PORTA      //write char data to LCD, char data in acc. A
        andb #%11010111
        orab #%00010000
        stab PORTA
        staa PORTB
        ldab PORTA
        andb #%11011111
        orab #%00011000
        stab PORTA
        ldab PORTA
        andb #%11010111
        orab #%00010000
        stab PORTA
        ldab PORTA
        andb #%11110111
        orab #%00110000
        stab PORTA
        ldaa #2         //wait 2ms
        staa DELAY
        jsr  WAIT
        rts

*************************************
WAIT1:  ldy #40 //waits for 1 ms
LOOPW1: dey
        cpy  #ZERO
        bne  LOOPW1
        ldy #40
LOOPW2: dey
        cpy #ZERO
        bne LOOPW2
        ldy #40
LOOPW3: dey
        cpy #ZERO
        bne LOOPW3
        rts

WAIT:   ldaa DELAY      //wait for variable amount of seconds
LOOPW:  cmpa #ZERO
        beq  RETURN
```

```
        jsr  WAIT1
        deca
        jmp  LOOPW
RETURN:
        RTS

SWI


*********************************
CLEAR:  ldaa #$01        //clears the LCD
        jsr WRITEC
        rts

**********************************
CUR_ON: ldaa #$0D       //turns cursor on
        jsr WRITEC
        rts

********************************
CUR_OFF: ldaa #$0C     //turns cursor off
         jsr WRITEC
         rts

************************************
CUR_LEFT: ldaa #$10    //moves cursor left
          jsr WRITEC
          rts

********************************
CUR_RIGHT: ldaa #$14    //move cursor right
           jsr WRITEC
           rts

***************************************
HOME:   ldaa #$02       //move cursor home
        jsr WRITEC
        rts

***********************************************
*wait for data from the serial port
CHECKDATA:      ldaa SPSR
                anda #%10000000
                cmpa #$80
                bne CHECKDATA
                rts

***********************************************
*check for data from the serial port
*regA = 0 if no data
*regA = 0x80 if data
TESTDATA:
            ldaa SPSR
            anda #%10000000
            rts

***************************************
*wait for data from the serial port
*put in it register A
```

```
GETDATA:        jsr CHECKDATA
                ldaa SPDR
                rts
```