Intro to Microcontrollers

Team Members:

Tatiana Jiselle Ensslin
Gengxiao Li

Table of Contents

Introduction

This lab report presents an introduction to the Freescale HCS12XS microcontroller, its hardware components, as well as the programming software capabilities it allows. The microcontroller contains several hardware components, such as the Analog to Digital Converter, a Timer, a PWM unit, as well as an SPI and SCI component. The Freescale HCS12XS also contains Flash Memory, RAM, and registers all for memory mapping. In this experiment, the HCS12XS contains 128KB of flash memory. Using a microcontroller allows a developer many perks, as for it contains several hardware components that allow for communication protocols to be administered and completed between different hardware. For this experiment, the HCS12XS is pinned to a safety power-board, which is connected to a BeagleBone board. This board allows for us to connect a 26-Pin Header to allow for port communication between the microcontroller and its programming software, CodeWarrior. This 26-Pin Header correlates to the ports for the microcontroller, which can be located in the datasheet for the Freescale HCS12XS.

As mentioned, throughout this lab we will explore the abilities of the HCS12XS. This experiment begins by identifying the components, as well as becoming familiar with the microcontroller and its parts/registers and datasheet. After careful inspection of the microcontroller and its inner pieces, attention is moved to the I/O Ports. This section begins by creating a small piece of software to test the functions of the I/O Ports by displaying an upward counter on the LED Display of the Altera board through the use of the 26-Pin Header. The next component explored on the microcontroller is the UART unit. Through the use of the 26-Pin Header, the Altera board and its software, one is able to configure the microcontroller to receive UART signals from the RealTerm software on the ports and transmit each character, so that the ASCII code for a received character will display on the LED display. After receiving through the UART is completed, data is then transmitted using the UART from the microcontroller to RealTerm. After exploring the UART component, attention is shifted to the SPI unit. In this section of the experiment, one is able to initiate communication to the Microchip 25LC1024 chip, a 1Mbit flash memory chip with data pre-stored, access that data with the ability to display it through RealTerm, and even search through the data to locate and display a secret word's address in the chips memory. The code used in both Altera (VHDL) and CodeWarrior (C++) in the experiment can be found in the appendix.

Methods

The Microcontroller

For the beginning of this experiment, start by looking up the datasheet for the Freescale HCS12XS microcontroller. Study and analyze Figure 1.1, a block diagram for the microcontroller, which details the entirety of the inner components. Identify the hardware units, which have been built previously in other experiments, as well as the hardware units that are unfamiliar. Take note to the fact that for this microcontroller, UART is known as SCI. Observe all of the hardware units and fully understand each of their abilities. Note the difference between the integrated circuit that is the microcontroller with the 64 pins seen in Figure 1.5, as well as Table 1.5, which display the several routing options for various peripherals. Take note of the 64 pins that are brought on your demonstration board and therefore the only pins that will be available while using this microcontroller. Next, analyze the figure of the memory map, which can be found in the datasheet. Discover the exact amount of memory specific to your chip.

Identify the start and end addresses for the program flash, user data flash, and RAM memory blocks. In the memory map, pinpoint the range of addresses that contain the hardware registers. Additionally, pinpoint the specific address from the following hardware registers: Port T register, Port T data direction register, Port M register, Port M data direction register, Port P register, and the Port P data direction register. Be sure to keep a detailed record of your findings throughout this experiment, including pictures and explanations of your results.

PWM

Next, we begin this section by testing the PWM hardware unit in order to learn about the integrated development environment (IDE) used to program the microcontroller. Start by writing no code. Create a new "C++" project in CodeWarrior using the project wizard. Be sure to select the proper HCS12XS chip, as well as the connection option TBDML. After the project is created, a small example code will be placed in the editor. Simply compile and run the code using the debugger. Look at the variables, single step options, breakpoints, and note the assembly language code that is running in the debugger. However, draw your attention to the memory map section. Use your address for Port P, discovered in the pervious section, to find Port P in the memory map. Read DDRP to see if Port P is set to input or output. Connect a wire from power to one of the Port P pins on the 26-Pin header. Refresh the memory map by left-clicking>refresh. Observe that it reads a 1. Now, connect a wire to ground and another Port P to make sure that it reads zero. Set one other PWM pin to pull-up and another to pull-down. Verify that the pull registers work through demonstration. After discovering the functionality of the memory map and Port P, set up the PWM unit so that it produces an 8-bit PWM signal. Use your birthday and birth month by multiplying the two numbers together and multiplying its result by 100. Set the PWM unit to run at that frequency. Be sure to demonstrate the duty cycle set to the birth month times 8 and the duty cycle set to the birth day times 3. Be sure that this entire step includes NO software.

I/O Ports

Next in the experiment, write a small piece of software to test the I/O Ports. Choose Port T as an output and write code to make Port T count. This requires setting the Port T data direction register to the proper number. Connect all eight pins of Port T to the FPGA board. Use the VHDL 7 segment decoder code to connect the 8 wires to the 2 LED display. Be sure to add a small delay in software to be sure that the counter can be legitimately read form the display, and is not running to fast to be seen. (Delays in code is poor design, however we will focus on the timer unit, which should be used for the display, at a later lab experiment). Be sure that you can demonstrate your work.

UART

For this section, leave the two 7 segment displays from the previous part connected to Port T. Use the UART (SCI) that connects to Port P. Run two wires from the UART Tx and Rx pins (Port P) over to the FPGA 40-Pin header and add VHDL code to route these pins to the FPGAs UART Tx and Rx pins. Now the HCS12 microcontroller will be connected to the desktop computers through UART. Open RealTerm and send characters to the HCS12 by typing in the RealTerm window. Configure the microcontroller to receiver UART signals on Port P and then transmit each character that is received to Port T. By sending the character to Port T, the ASCII code for the received character should be displayed on the FPGA's two 7-segment displays. Demonstrate that you can receive and display characters using the HCS12 microcontroller's UART. Be sure that you take note to the 6MHz clock when using the datasheet to properly program the microcontroller and its registers. Also be sure that you set no flag interrupts, or else

once a flag is triggered, the code below it will never be triggered and no data will be sent or received. Next, after successfully being able to demonstrate that you can receive a character through UART, focus your attention on transmitting characters. Using a counter as we did in the previous section with Port T, make the counter count the ASCII codes for the characters between the SPACE and small letter "z." Continue to use flags to ensure that the transmission is ready. Be sure to add a small delay so that the characters are sent to RealTerm with enough time so that can be read as they are received. Ensure that you are able to properly transmit the characters in demonstration.

SPI

Lastly, we will test the microcontroller's SPI unit by connecting it to an SPI based integrated circuit. In a previous experiment we used a SPI Digital to Analog converter, however, this time we will use a SPI based flash memory chip. This chip is the microchip 25LC1024 with 1M bit memory. Next, we will configure the HCS12 to initiate communication as the Master to the chip (slave). In order to do this, it is important to configure the SPI Master properly through the microcontrollers registers. Furthermore, look up the datasheet for the 25LC1024. Note that this section requires the read operation on the device, therefore it is important to look up Figure 2: Read Sequence. Note that you can read 1 byte at a time from the memory chip by sending the read instruction, followed by the address, and then preforming one more communication exchange in order to receive the data. Remember in SPI that a message must be sent first in order to receive data. This means that you must send "junk" data (0x00) over in order to receive the good data from the chip. You can read back more than 1 byte at a time if data is continuously being sent over. For reach transmission, you will receive the next byte in memory. The entire time that communication is occurring with SPI, you must be in control of the chip select (CS) pin. As soon as the chip select pin is reset to 0, the address that the chip is on is forgotten. When CS is set to 1, the address starts from the beginning again. This means that you will need to generate the CS signal within your code. To do this, simply use an unused pin (try not to use AD pins for they require a specific reset for its pins), and configure it for the general-purpose output using its registers. Set CS to 0 before communication is started and to 1 when communication is finished. The memory chip is preloaded with a particular set of data. Starting at address 0, the data bytes are programmed with the ASCII codes of the following message: "Hi HCSX12." This takes up the first 9 bytes in the chips memory. In order to read this message off of the chip, write a piece of code that will allow for SPI communication between the microcontroller and the chip. This code should read the first 9 bytes of data and send them sequentially to the UART, allowing it to be displayed via RealTerm for demonstration. It is okay if the message is sent repetitively in a loop.

Using the SPI protocols for the microcontroller, find the secret address of a preloaded message that will be assigned to you. Search the memory for the message and then display the start address on RealTerm. It is okay if the code searches over and over in a loop. Be able to demonstrate the code running and displaying the proper start address on RealTerm. The list of unknown messages are shown below:

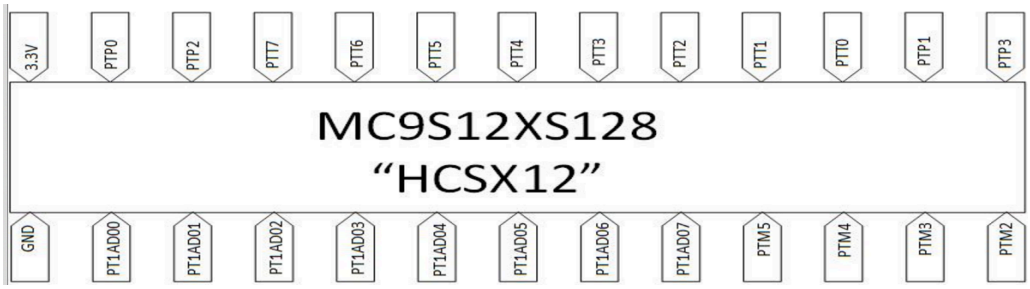
Message 1:Marshall Street
Message 2: Coach Boeheim
Message 3: Schine Student Center
Message 4: Otto The Orange
Message 5: Coach Shafer
Message 6: Qinru Qiu
Message 7: Coach McIntyre

Message 8: L.C. Smith
Message 9: Go Orange
Message 10: Coach Moore
Message 11: Syracuse University
Message 12: Carrier Dome
Message 13: Susan Older
Message 14: Prasanta Ghosh
Message 15: Carmello Anthony

Results

The Microcontroller

To become familiar with our microcontroller, we began this section by printing out the pin our chart for the HCS12XS.



We attached the programmer and the microcontroller to the platform, as well as the 26-Pin Header. We then looked up the following block diagram to study the inner components of the microcontroller.

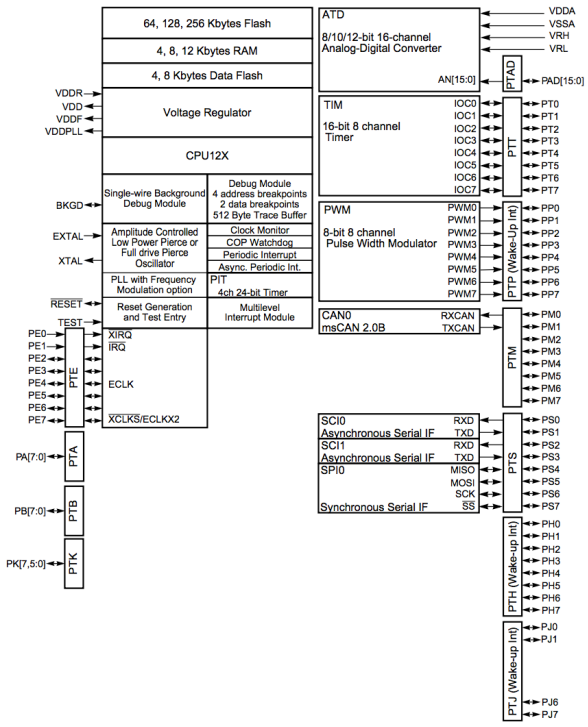


Figure 1-1. S12XS Family Block Diagram

There are 3 out of 4 hardware units which were used before in previous labs that are inside the block diagram. These are: the SCI(UART), SPI, and PWM units. A hardware unit which is in the block diagram which we have not used yet is the ATD (analog to digital converter). This converts voltage to a digital number. It also has a TIM (timer), which takes input capture and output compare. It also has a CANC, which tests for viotlated CAN protocols. This is a communication protocol. The voltage regulater regulates the voltage for the blocks. The microcontroller also has a data flash memory, RAM, and 128KB Flash memory. The microcontrollers we are using is from a 64pin package. There are 4 M Ports, 4 P Ports, 8 T Ports, and 8 AD Ports our of the 64pins on our board. Our controller also contains a memory map, which details carious memory units with addresses. This memory map can be seen below.

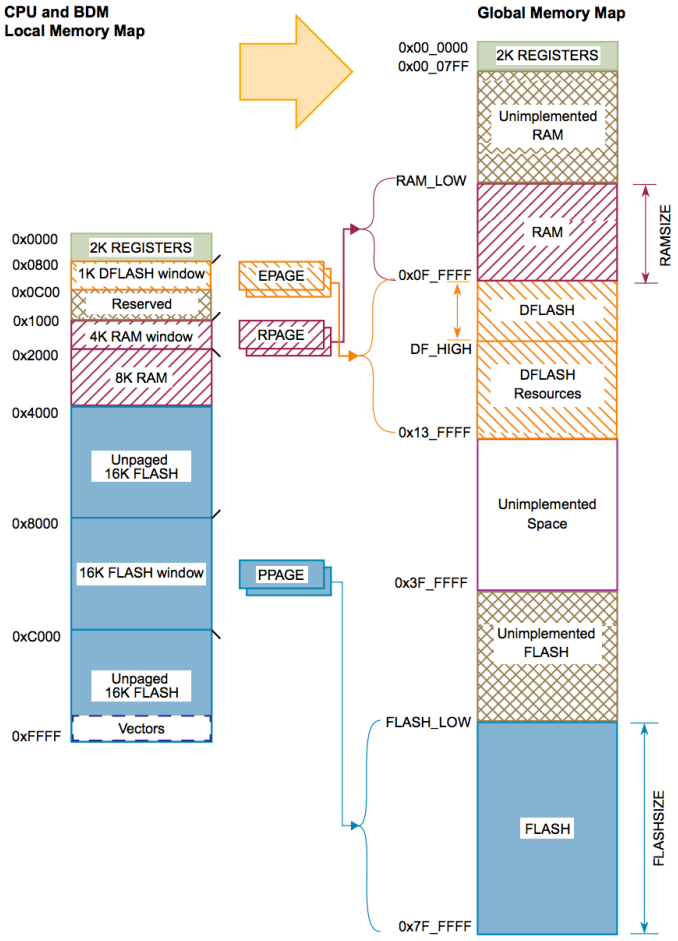


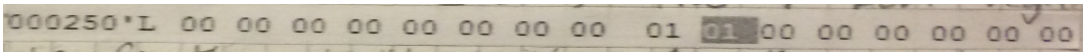
Figure 1-2. S12XS Family Global Memory Map

Our chip has 128KB of embedded flash memory with error correction coding, 8KB of RAM, and up to 4KB to 8KB of data from flash memory. The start and end addresses for the program flash is from 0x0800 to 0x0000. The program memory is from 0x4000 to 0x8000 and 0x0000 to 0xFFFF. The RAM goes from 0x2000 to 0x4000. All of the hardware units in our microcontroller can be communicated to through software which runs on the CPU12 central processing unit. The reange of addresses that contains the hardware registers is from locally 0x0000 to 0x0800, or globally 0x00_0000 to 0x00_07FF. The address for the Port T register is 0x0240 and 0x0241 and the Port T data direction register is 0x0242. The address for

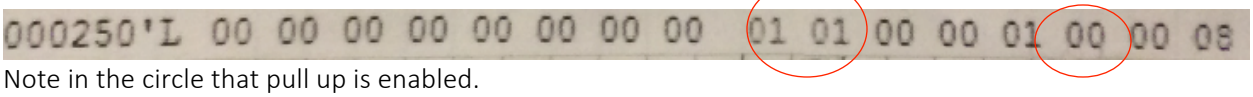
the Port M register is 0x0250 and 0x0251 and the Port M data direction register is 0x0252. The address for the Port P register is 0x0258 and 0x0259 and the Port P data direction register is 0x025A.

PWM

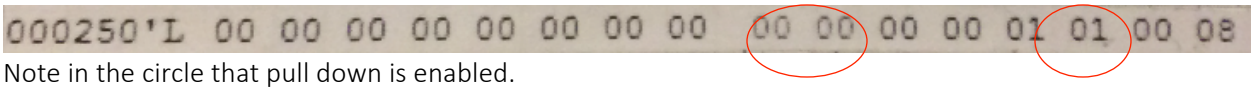
The IDE allows direct, real-time, access to all the registers. First, we start be creating a C++ project by selecting the HCS12X, the HCS12XS family and the MX9512XS128. Our connection is the TBDML. When we run the C++ file in the debugger, we find the DDRP address, which is set to 00. When we look at the data sheet, we see that means that the port is set to input. If it were 1, it would be set to output. Note that once we connect a P Port to power, the P Port registers as a 01:



Note that the enable for the pull register for P is at the address 0x25C. The polarity register for Port P is 0x025D. To enable the pull register, we set it to '1'. As for the polarity, when a '0' is enabled, a pull up device is selected, when a '1' is enabled, a pull down device is selected. See below:

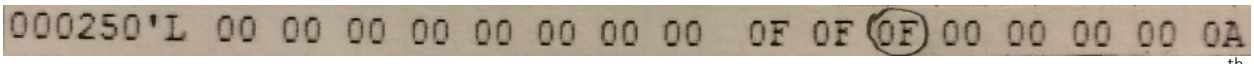


Note in the circle that pull up is enabled.



Note in the circle that pull down is enabled.

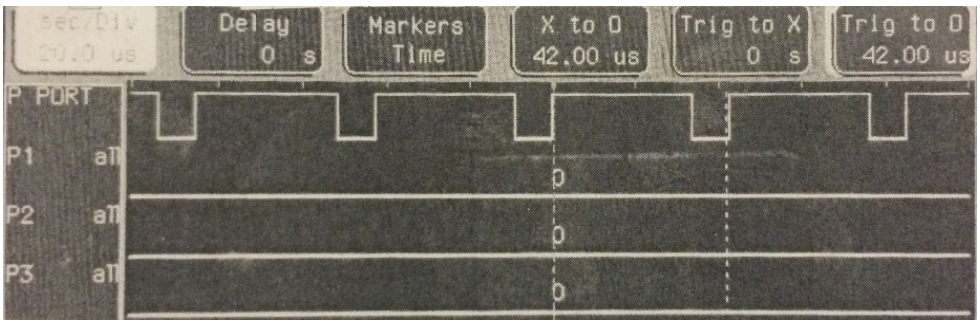
Next, we learn that we can set all P Ports at one time via hexadecimal in the memory of the eral time debugger. We set 0x025A to '0F,' representing "00001111" setting the Port P0-P3 all to 1 at the same time:



Now, we are going to set up the PWM unit so it produces an 8-bit PWM signal. We are using October 26th, (10*26) as our target signal x100, therefore 26,000. We will use the PWM byte to set our duty cycle to this number. We also have to focus on the clock (5 bits) in the memory to control the clock speed. We set the beginning of the PWM byte 0x0300 to 01 to enable one PWM. The polarity at 0x0301 is then set to 01. The clock starts at 6MHz, meaning that to get 20,000, we must divide by 16 to reach our number that is the closest: .375. This means we set 0x0302 to "000" or "00." 26,000 in binary is 16 bits. "1100101100100000." Now, to get as close as possible we need to scale the clock to '01.' You must set the duty cycle (in hex), which is at 0x031C. So, in our last lab, we built the circuit, which is currently in this experiment just set. In the end our PWM byte was set to 01 01 00 00 00 00 00 01 00 00 00 00 00 00 as seen below, with the duty cycle to 'CC' 204, or 'CA' or 202.

```
000300'L 01 01 00 00 00 00 00 5D  01 00 88 88 F2 FB CC CC
000310'L 00 00 00 00 FF FF FF FF  FF FF FF FF CA 25 FF FF
```

Note the duty cycle at 80% on Port P0.



I/O Ports

Now, we will set up a small piece of software to test the I/O Ports. We will choose Port T as an output and write code to make Port T count. We shall connect all eight pins of Port T to the FPGA board. We will then use the VHDL 7 segment decoder to connect the 8 wires to the 2 LED display. Note that although the display is counting, but may be too fast to see. Therefore, we will make a small software delay to make that counting visible. Port T is included with the variable name PTT_PTT*. Port T is page 85 in the datasheet. It is important to note that the direction for Port T must be enabled to output, and that we do so with the DDRT declaration by setting it to DDRT= 0xFF = "11111111."

```
void main(void) {
  /* put your own code here */   int counter=0; int a = 1;
  DDRT = 0xFF;
            while(a == 1){
               if (counter==20000) {
                PTT=PTT +1;
               counter = 0;
               }
               else
               counter = counter +1;
            }
```

Above was the code we used to create our counter through Code Warrior. Note the 3rd line where DDRT=0xFF. The counter slows down the display. The code for this section can be found in Appendix A.

UART

Leaving the two 7 segment displays from the previous part connected to Port T, we will now use the UART (SCI) that connects to Port P. We will do this by running two wires form the UART Tx and Rx pins (Port P) over to the FPGA 40 Pin Header. We will ass VHDL code to route these pins to the FPGAs UART Tx and Rx pins. We will then use the HCS12 microcontroller, which will be connected to the desktop through UART. We shall open RealTerm and use it to send characters to the HCS12 by typing in the RealTerm window. We will also configure the microcontroller to receive UART signals on Port P and then transmit each character that is received to Port T. The ASCII Code for the received character should then be displayed on the 7-segment display. In order to route our SPI connections, we use the module routing registers (MODRR) to set Port S to redirect to P. This is because SPI connects through Port S—yet we do not have a Port S on our controller. The MODRR is at address 0x0257, which we will set to '01,' which will set the TXD to Port P2 and the RXD to P0. We are picking a baud rate of 9600 in RealTerm. At SCICR2 0x0003, we set it to '111' in hex 0x07, which sets the SCI Register to receive. There are flags held in 0x0004/SCISRI, we will focus on RDRF and TDRE. RDRF = "00111111," or 0x3F. The SCIDRL register will receive a serial input from UART. Because our baud rate is 9600, we will set the two baud rate SCI registers to 0x27, or 39 in decimal. We calculated this by looking up the formula in the data sheet. Baud rate=bus rate/(16*BDL). So, $9600=6*10^6/(16x) = 39$. We used the SCI1 series of registers to code in Code Warrior.

```
void main(void) {
  /* put your own code here */
  DDRT = 0xFF;    // enables the T reg bits to output
  MODRR = 0x50; //reroutes the S port to P for SPI communication
  SCI1BDH = 0x00;    // upper half of the baud rate in hex
  SCI1BDL = 40;     // lower half of the baud rate in hex
  SCI1CR2_RE = 1; //turns on the receiving for the SPI
       while(1){
                while(!SCI1SR1_RDRF){}//SCI0SR1_RDRF is receving flag
                PTT = SCI1DRL;                // SCI receiving register


  }
}
```

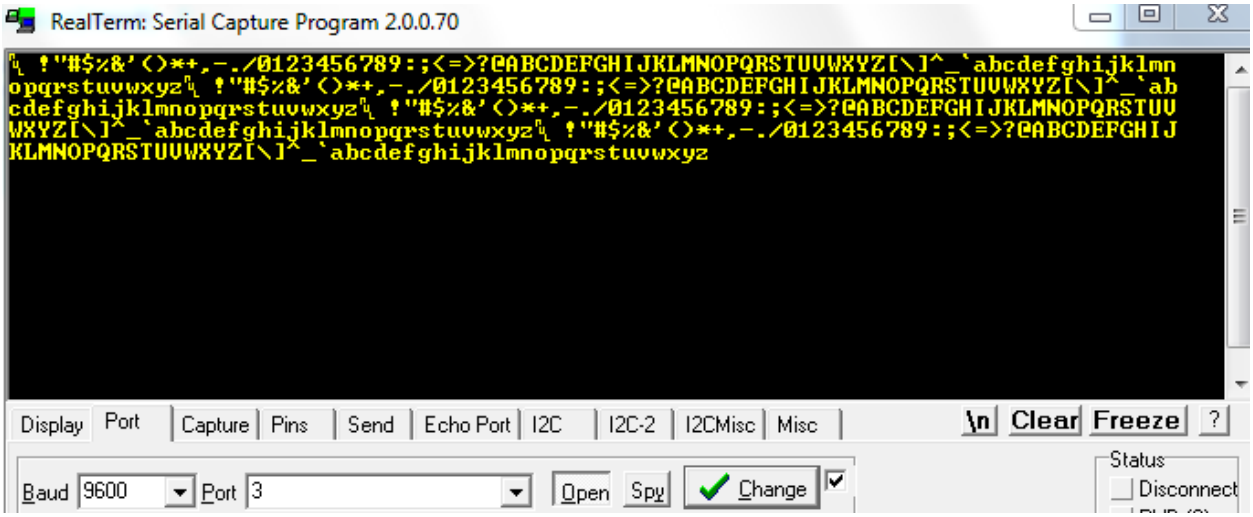
We were able to successfully receive serial input from RealTerm. We were able to receive the ASCII code for all the keys on the QWERTY keyboard. For example, we sent over G through the RealTerm and received a 67 on the segment display, its ASCII equivalent. Now that we have received properly through UART, we are going to transmit characters using UART. We will use counter just like our previous section for testing Port T. However, the time, the counter will count the ASCII codes for the characters between the SPACE and small letter 'z.' We will add a small software delay so the characters are sent slow enough to read them as they are received by the RealTerm program. The ASCII code for SPACE through 'z' will convert to RealTerm and display on the program. This means our counter will go from hex 20 to hex 7A. Below is the main function of our Code Warrior, and the proof of output form the transmitting being sent to RealTerm is also seen below. The code for this section, which details the registers used, can be found in Appendix B.

```
void main(void) {

  int counter=0x20; //ascii number this is "hex"
  int counterA=0;
  MODRR = 0x50; //reroutes the S port to P for SPI communication
  SCI1BDH = 0x00;   // upper half of the baud rate in hex
  SCI1BDL = 40;    // lower half of the baud rate in hex
  SCI1CR2_TE = 1; //turns on the transmitting for the SPI

       while(1){

           while (counter != 0x7B)
               if (counterA == 20000)   {
                while(!SCI1SR1_TDRE){}//SCI1SR1_TDRE is transmit flag
                  SCI1DRL = counter;   // SCI trasmitting port
                  counterA = 0;
                  counter++;
                } else
                counterA = counterA + 1;
  }
}
```

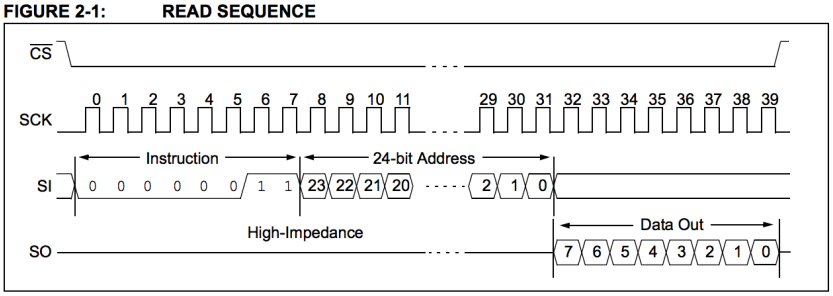


SPI

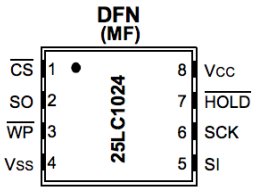
In order to test the microcontroller's SPI unit, we will connect it to an SPI based integrated circuit. In a previous lab, we used SPI D/A converter, but this time we will use a SPI based flash memory chip. We will configure the HCS12 as the SPI master and the 25LC1024 as the slave, allowing the microcontroller to initiate communication between both. In SPI, every communication event represents an exchange of data

between the master and slave devices. Because we are only preforming the read operation on the device, we only need “Figure 2: Read Sequence” from the 25LC1024 data sheet. We use this diagram to match up the outputs on the logic analyzer to make sure that the microcontroller and chip are communication properly. The logic analyzer should match this diagram perfectly.

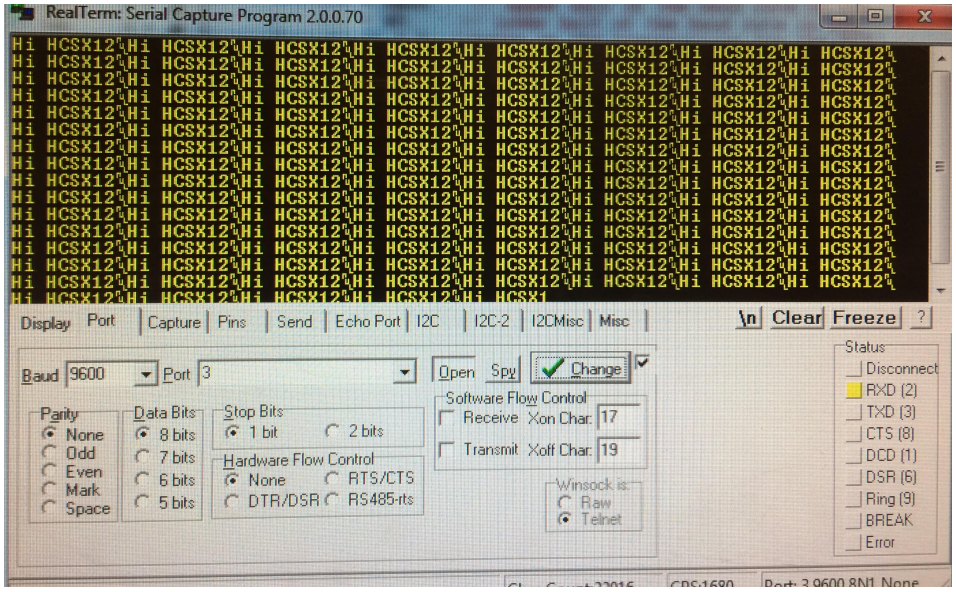
FIGURE 2-1: READ SEQUENCE



Note that you can read 1 byte at a time from the memory chip by sending the read instruction, followed by the address, then one more communication exchange. In SPI, data must be sent in order toe be received. This means we will have to send garbage data in order to receive data back form the memory. This also works by sending more than 1 byte at a time, for each transmission results in the next byte received in memory. It is important to keep in mind that the whole time one communicates with SPI, you must be in control of the chip select (CS chip) pin. It is important to note that when CS remains at ‘1’, the chip remembers what address is being accessed. As soon as CS returns to ‘0’, the address resets. To do this, you must set up the SPI master so that it does not generate the CS signal. It instead should be generated with software. To do this, we will choose an unused pin and configure it for general-purpose output. CS will be set to ‘0’ before the start of the communication sequence and then back to ‘1’ when its finished. The given memory chip has been pre-loaded with set data. At address 0, the data bytes are programmed with ASCII codes to make “Hi HCSX12.” Our goal is to write a piece of code that will read the first 9 bytes of data and send the sequentially to the UART. The message should appear in RealTerm. To set up the 25LC1024 with the microcontroller, we first must reroute the SPI to the M Ports for our chip. This requires settings MODRR to ‘1’. This means we can connect the pins to the microcontroller header. The picture below describes the pin out chart for the chip:



We will connect CS to pin PTIAD01, SCK to PTM5, S1 to PTM4, and S0 to PTM2. We will still use port P0 and P2 to receive and transmit. In order to set the baud rate for SPI connection, we must use the following formula: Baud Rate = bus clock/ baud rate divisor. To find the baud rate divisor = (SPDR + 1) * $2^{(SDR+1)}$. This means we do not have to modify the SPI baud rate because the clock inside the chip is faster than the microcontrollers. We then set the SPI enable bit to ‘1’, SPI0CR1_SPE = 1, and SPI0_CR1__CPHA to 1, saying to read in input on even cycles only. We set Port PT0 as the CS bit. After setting all the proper registers, we were able to receive “Hi HSCX12” on RealTerm. The code for this section, which details the registers used, can be found in Appendix C.



FINDING THE SECRET ADDRESS

For this section, we modified our code from the previous section to locate the address of "Coach Boeheim." To do this, we put the data from the slave into an array and then searched the array by comparing it to an array with our message in it. Our correct address was 0x12D2. Our result can be seen below:

Discussion

This lab has provided us with a good understanding of what a microcontroller is and what it is capable of. It contains several hardware units that have multiple capabilities for communication protocols that can be useful when programming a complex device/project. The microcontroller itself contains several programmable registers, which allow the microcontroller to preform different actions on its ports. This principle is apparent in throughout this lab. Through the PWM unit, it becomes easier to control the power supplied to electrical devices, for it is done entirely through the microcontroller's registers. After focusing on the PWM unit, we learned about I/O Ports on the microcontroller and how they can be used to program to another piece of hardware for display. It was in this section in which we learned that it was most important to familiarize oneself with the microcontrollers data sheet. There were a lot of mistakes and wasted time spent searching and identifying the different sections of data sheet. After we became familiar with the I/O Ports on the microcontroller, we shifted focus to the UART (SCI) hardware component in the microcontroller. It was interesting to see how what we created in a previous experiment could be so easily implemented and modifiable through the ease of programming the microcontroller. The most difficult part of this section was becoming familiar with the flags used to

transmit and receive data. If a flag interrupt was set (as which we did on accident), the data can never be sent or received because when the flag is triggered the code is halted. Through trial and error, we were eventually able to figure out the cause of our errors and use the debugging tool to correctly identify the proper registers and logic to successfully complete this section. Lastly, through the SPI unit, we were able to learn how to traverse through a slave's memory and retrieve a message/address. This section proves the power and capabilities that a microcontroller can truly have. In totality, one can really take advantage of a microcontroller, its hardware components, and what it has to offer. Through its use of moderately easy programmable registers, as well as ability to understand C code, the microcontroller has the ability to create some incredible experiments and is absurdly versatile.

Although we did encounter a few challenges while working through this lab, such as selecting incorrect registers to program, or picking an un-programmable port, our initial approach to implementing a microcontroller was overall fairly correct. Through use of the datasheet, header file in the Code Warrior, and general knowledge about the communication protocols, we found the use of these hardware components to be a practical and valuable approach towards fully implementing larger projects and experiments.

Appendix A – I/O Port

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */
void main(void) {
 /* put your own code here */   int counter=0; int a = 1;
 DDRT = 0xFF;
        while(a == 1){
          if (counter==20000) {
           PTT=PTT +1;
          counter = 0;
          }
          else
          counter = counter +1;
        }
============================================================
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity LAB5 is
            port        (
            InA         : in std_logic_vector(7 downto 0);

            outA        : out std_logic_vector(6 downto 0);
```

```
        outB        :out std_logic_vector(6 downto 0);
         clk : in std_logic;
        out1        : out std_logic;
        out2        :out std_logic
        );
end LAB5;
architecture behavioral of LAB5 is
        type state_T is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
        signal state : state_T;
        signal state1 : state_T;
        signal tOut1: std_logic_vector(3 downto 0);
        signal tOut2: std_logic_vector(3 downto 0);
        begin
                process(clk)
                        begin
                                if (rising_edge(clk)) then
                                                case state is

                                                when s0 =>
                                                state <= s1;

                                                when s1 =>
                                                tout1(0) <= InA(7);
                                                state <= s2;

                                                when s2 =>
                                                tout1(1) <= InA(6);
                                                state <= s3;

                                                when s3 =>
                                                tout1(2) <= InA(5);
                                                state <= s4;

                                                when s4 =>
                                                tout1(3) <= InA(4);
                                                state <= s5;
```

```
                                        when s5 =>
                                        tout2(0) <= InA(3);
                                        state <= s6;

                                        when s6 =>
                                        tout2(1) <= InA(2);
                                        state <= s7;

                                        when s7 =>
                                        tout2(2) <= InA(1);
                                        state <= s8;

                                        when s8 =>
                                        tout2(3) <= InA(0);
                                        state <= s9;

                                        when s9 =>
                                        state <= s0;
                                        end case;
                        end if;
end process;

with tOut1 select outA <=
"0000001" when "0000",
"1001111" when "0001",
"0010010" when "0010",
"0000110" when "0011",
"1001100" when "0100",
"0100100" when "0101",
"0100000" when "0110",
"0001111" when "0111",
"0000000" when "1000",
"0001100" when "1001",
"0000100" when "1010",
"1100000" when "1011",
```

```
                    "0110001" when "1100",
                    "1000010" when "1101",
                    "0110000" when "1110",
                    "0111000" when "1111";

                     with tOut2 select outB <=
                    "0000001" when "0000",
                    "1001111" when "0001",
                    "0010010" when "0010",
                    "0000110" when "0011",
                    "1001100" when "0100",
                    "0100100" when "0101",
                    "0100000" when "0110",
                    "0001111" when "0111",
                    "0000000" when "1000",
                    "0001100" when "1001",
                    "0000100" when "1010",
                    "1100000" when "1011",
                    "0110001" when "1100",
                    "1000010" when "1101",
                    "0110000" when "1110",
                    "0111000" when "1111";
end behavioral
```

Appendix B – UART

RECEIVING:

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */

    // receiving

void main(void) {
  /* put your own code here */
  DDRT = 0xFF;   // enables the T reg bits to output
  MODRR = 0x50; //reroutes the S port to P for SPI communication
  SCI1BDH = 0x00;   // upper half of the baud rate in hex
```

```
 SCI1BDL = 40;    // lower half of the baud rate in hex
 SCI1CR2_RE = 1; //turns on the receiving for the SPI
     while(1){
          while(!SCI1SR1_RDRF){}//SCI0SR1_RDRF is receving flag
          PTT = SCI1DRL;              // SCI receiving register
 }
}
```

```
==========================================================================
```

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity LAB5 is
          port          (
          InA           : in std_logic_vector(7 downto 0);
          InRX : in std_logic; --rx pin
          InMicro : out std_logic; --rx to micro
          clk : in std_logic;
          outA          : out std_logic_vector(6 downto 0);
          outB          : out std_logic_vector(6 downto 0)
          );
end LAB5;


architecture behavioral of LAB5 is
          type state_T is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
          signal state : state_T;
          signal state1 : state_T;
          signal tOut1: std_logic_vector(3 downto 0);
          signal tOut2: std_logic_vector(3 downto 0);
          begin

          InMicro <= InRX;

                    process(clk)
                              begin
```

```
if (rising_edge(clk)) then
                    case state is

                    when s0 =>
                    state <= s1;


                    when s1 =>
                    tout1(0) <= InA(7);
                    state <= s2;

                    when s2 =>
                    tout1(1) <= InA(6);
                    state <= s3;

                    when s3 =>
                    tout1(2) <= InA(5);
                    state <= s4;

                    when s4 =>
                    tout1(3) <= InA(4);
                    state <= s5;

                    when s5 =>
                    tout2(0) <= InA(3);
                    state <= s6;

                    when s6 =>
                    tout2(1) <= InA(2);
                    state <= s7;

                    when s7 =>
                    tout2(2) <= InA(1);
```

```
                                                    state <= s8;

                                                    when s8 =>
                                                    tout2(3) <= InA(0);
                                                    state <= s9;


                                                    when s9 =>
                                                    state <= s0;
                                                    end case;
                                        end if;
                end process;

                with tOut1 select outA <=
                "0000001" when "0000",
                "1001111" when "0001",
"0010010" when "0010",
                "0000110" when "0011",
                "1001100" when "0100",
                "0100100" when "0101",
                "0100000" when "0110",
                "0001111" when "0111",
                "0000000" when "1000",
                "0001100" when "1001",
                "0000100" when "1010",
                "1100000" when "1011",
                "0110001" when "1100",
                "1000010" when "1101",
                "0110000" when "1110",
                "0111000" when "1111";

                 with tOut2 select outB <=
                "0000001" when "0000",
```

```
                    "1001111" when "0001",
                "0010010" when "0010",
                    "0000110" when "0011",
                    "1001100" when "0100",
                    "0100100" when "0101",
                    "0100000" when "0110",
                    "0001111" when "0111",
                    "0000000" when "1000",
                    "0001100" when "1001",
                    "0000100" when "1010",
                    "1100000" when "1011",
                    "0110001" when "1100",
                    "1000010" when "1101",
                    "0110000" when "1110",
                    "0111000" when "1111";

end behavioral;
```

TRANSMITTING:

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */
void main(void) {
  int counter=0x20; //ascii number this is "hex"
  int counterA=0;
  MODRR = 0x50; //reroutes the S port to P for SPI communication
  SCI1BDH = 0x00;   // upper half of the baud rate in hex
  SCI1BDL = 40;    // lower half of the baud rate in hex
  SCI1CR2_TE = 1; //turns on the transmitting for the SPI

     while(1){

        while (counter != 0x7B)
           if (counterA == 20000)   {
            while(!SCI1SR1_TDRE){}//SCI1SR1_TDRE is transmit flag
```

```
            SCI1DRL = counter;   // SCI trasmitting port
            counterA = 0;
            counter++;
          } else
          counterA = counterA + 1;
 }
}
```

==========

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity LAB5 is
port (
InA : in std_logic;
clk : in std_logic;
out1 : out std_logic
);
end LAB5;

architecture behavioral of LAB5 is
type state_T is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
signal state : state_T;

begin
process(clk)
begin
if (rising_edge(clk)) then

case state is
when s0 =>
state <= s1;

when s1 =>
out1 <= InA;
state <= s2;
```

```
when s2 =>
out1 <= InA;
state <= s3;
when s3 =>
out1 <= InA;
state <= s4;
when s4 =>
out1 <= InA;
state <= s5;
when s5 =>
out1 <= InA;
state <= s6;
when s6 =>
out1 <= InA;
state <= s7;
when s7 =>
out1 <= InA;
state <= s8;
when s8 =>
out1 <= InA;
state <= s9;
when s9 =>
state <= s0;

end case;
end if;
end process;
end architecture;
```

Appendix C – SPI

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */
void main(void) {
  unsigned char dummy;
```

```
unsigned char good;
MODRR= 0x50; //reroutes to port M for Spi communication

SCI1BDH = 0x00;   // upper half of the baud rate in hex
SCI1BDL = 40;    // lower half of the baud rate in hex
SCI1CR2_TE = 1; //turns on the transmitting for the SCI

SPI0CR1_MSTR = 1; //set the SPI to MASTER
SPI0CR1_CPOL =0; // this is also probably wrong .. clock polarity
SPI0CR1_CPHA =0; //this is probably wrong .. phase
SPI0CR1_SSOE = 0; //keep SS off
SPI0CR1_SPE = 1;
DDRT_DDRT0 = 1;
    while(1){
        PTT_PTT0 =0; //CS bit
            while(!SPI0SR_SPTEF){}//transmit empty flag
            SPI0DRL=3;
            while(!SCI1SR1_TDRE){}//SCI1SR1_TDRE is transmit flag
            dummy = SPI0DRL;

            while(!SPI0SR_SPTEF){}
            SPI0DRL=0;
            while(!SCI1SR1_TDRE){}
            dummy = SPI0DRL;

            while(!SPI0SR_SPTEF){}
            SPI0DRL=0;
            while(!SCI1SR1_TDRE){}
            dummy = SPI0DRL;

            while(!SPI0SR_SPTEF){}
            SPI0DRL=0;
            while(!SCI1SR1_TDRE){}
            dummy = SPI0DRL;
```

```
while(!SPI0SR_SPTEF){}  //1
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}//2
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}//3
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}//4
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){} //5
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
```

```
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){} //6
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}   //7
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}   //8
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
  SCI1DRL = good;

while(!SPI0SR_SPTEF){}    //9 bytes
SPI0DRL=0x00;
while(!SCI1SR1_TDRE){}
good = SPI0DRL;

while(!SCI1SR1_TDRE){}
```

```
            SCI1DRL = good;
            ////////////////////////////////
           while(!SPI0SR_SPTEF){}    //9 bytes
           SPI0DRL=0x00;
           while(!SCI1SR1_TDRE){}
           good = SPI0DRL;
           while(!SCI1SR1_TDRE){}
            SCI1DRL = good;
         PTT_PTT0 =1;
         while(!SPI0SR_SPIF) {}//transfer not complete flag
 }
}
```

```
==========================================================================
```

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity LAB5 is
port (
InA : in std_logic;
clk : in std_logic;
out1 : out std_logic

);
end LAB5;

architecture behavioral of LAB5 is
type state_T is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
signal state : state_T;

begin
process(clk)
begin
if (rising_edge(clk)) then

case state is
```

```
when s0 =>
state <= s1;

when s1 =>
out1 <= InA;
state <= s2;
when s2 =>
out1 <= InA;
state <= s3;
when s3 =>
out1 <= InA;
state <= s4;
when s4 =>
out1 <= InA;
state <= s5;
when s5 =>
out1 <= InA;
state <= s6;
when s6 =>
out1 <= InA;
state <= s7;
when s7 =>
out1 <= InA;
state <= s8;
when s8 =>
out1 <= InA;
state <= s9;
when s9 =>
state <= s0;
end case;
end if;
end process;
end architecture;
```

ADDRESS MEMORY SEARCH:

```
#include <hidef.h>      /* common defines and macros */
```

```
#include "derivative.h"      /* derivative-specific definitions */
void main(void) {
  unsigned int counter=0; //ascii number this is "hex"
  unsigned char dummy;
  unsigned char input;
  MODRR= 0x50; //reroutes to port M for Spi communication
   int i = 0;
  SCI1BDH = 0x00;   // upper half of the baud rate in hex
  SCI1BDL = 40;    // lower half of the baud rate in hex
  SCI1CR2_TE = 1; //turns on the transmitting for the SCI
  SPI0CR1_MSTR = 1; //set the SPI to MASTER
  SPI0CR1_CPOL =0; // this is also probably wrong .. clock polarity
  SPI0CR1_CPHA =0; //this is probably wrong .. phase
  SPI0CR1_SSOE = 0; //keep SS off
  SPI0CR1_SPE = 1;
  DDRT_DDRT0 = 1;
  unsigned char address[13] = {'C','o','a','c','h',' ','B','o','e','h','e','i','m'};
  unsigned char array[131071];
            PTT_PTT0 =0; //CS bit
              while(!SPI0SR_SPTEF){}//transmit empty flag
              SPI0DRL=3;
              while(!SCI1SR1_TDRE){}//SCI1SR1_TDRE is transmit flag
              dummy = SPI0DRL;

              while(!SPI0SR_SPTEF){}
              SPI0DRL=0;
              while(!SCI1SR1_TDRE){}
              dummy = SPI0DRL;

              while(!SPI0SR_SPTEF){}
              SPI0DRL=0;
              while(!SCI1SR1_TDRE){}
              dummy = SPI0DRL;

              while(!SPI0SR_SPTEF){}
```

```
            SPI0DRL=0;
            while(!SCI1SR1_TDRE){}
            dummy = SPI0DRL;

          while (counter < 30000)  {
            while(!SPI0SR_SPTEF){}  //1
            SPI0DRL=0x00;

            while(!SCI1SR1_TDRE){}
            array[counter]= SPI0DRL;
            counter++;
          }
          for (int j=0; j<=30000 ; j++) {
              if ((array[j] == address[0])&& (array[j+1] == address[1])&&(array[j+2] == address[2])&&(array[j+3] == address[3])&&(array[j+4]
== address[4])&& (array[j+5] == address[5])&&(array[j+6] == address[6])) {
                  unsigned int unhappy = j;
                 unsigned int happy = j/255;
                 SCI1DRL = (happy);
                 while((SCI1SR1 & 0x80)==0);
                 SCI1DRL = unhappy;
                j++;
              }
          }
            PTT_PTT0 =1;
}
```

==========

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity LAB5 is
port (
InA : in std_logic;
clk : in std_logic;
out1 : out std_logic
);
end LAB5;
architecture behavioral of LAB5 is
type state_T is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
```

```
signal state : state_T;


begin
process(clk)
begin
if (rising_edge(clk)) then

case state is
when s0 =>
state <= s1;
when s1 =>
out1 <= InA;
state <= s2;
when s2 =>
out1 <= InA;
state <= s3;
when s3 =>
out1 <= InA;
state <= s4;
when s4 =>
out1 <= InA;
state <= s5;
when s5 =>
out1 <= InA;
state <= s6;
when s6 =>
out1 <= InA;
state <= s7;
when s7 =>
out1 <= InA;
state <= s8;
when s8 =>
out1 <= InA;
state <= s9;
when s9 =>
state <= s0;
```

```
end case;

end if;

end process;

end architecture;
```