Prototype Control Board for an Automated Bar (Robotic Bar Monkey)

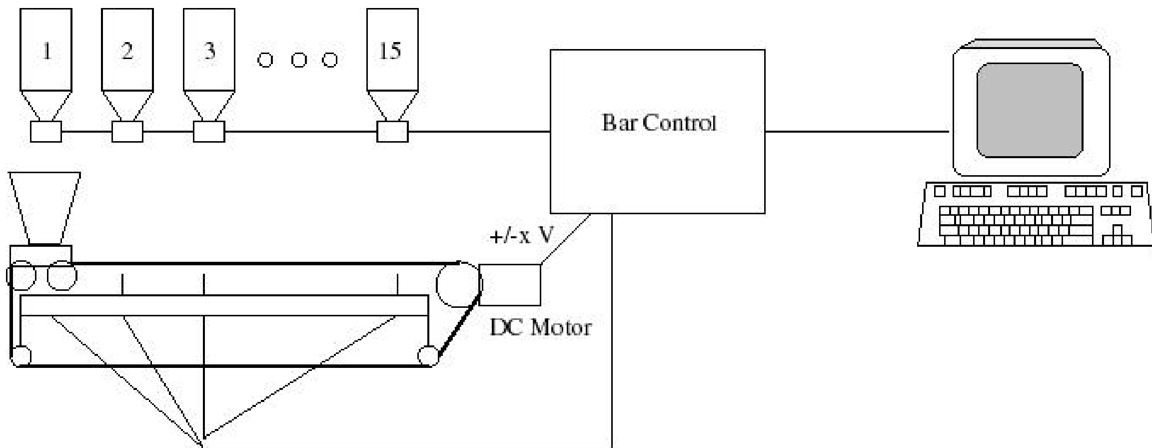
Final Project Report
December 9, 1999
E157

Andrew Cosand and Glenn Gebhart

Abstract:

We are interested in creating an automated beverage mixing system, affectionately termed a robotic bar monkey. Ideally, authorized* bar monkey users will request beverages, either locally or remotely, and the system will mix them from the available ingredients. The user will interact with the bar monkey via a custom interface on the host computer. Once the user selects a drink from the database of drinks, the host computer will determine how to make the drink, and send recipe steps to a microcontroller-based control system as 1-byte words containing some instruction bits and some parameter bits. The control system will then execute these instructions by sending out control signals to the appropriate valves and motors. Once the control system finishes executing the instruction, it will send a "done" signal to the host computer. When the host computer receives this done signal, it will either proceed with the next recipe step, or send an all done "recipe step" which should return the cup to the home position, and could initiate a diagnostic check or other system tasks.

Introduction



For our Micro Ps project, we are constructing a prototype of the bar monkey control system. The control system will be based on the Motorola 68HC11 microcontroller, and will involve a Xilinx Spartan FPGA for interface logic, as well as some other small components. The control system will receive 8 bit recipe steps from the host computer via an RS232 serial interface, utilizing the HC11's SCI communication capability with a Maxim MAX232A transceiver chip to convert between the HC11's 0,5 volt system and the ±10 volt RS232 system. Although future releases could take advantage of the 64 possible "recipe steps" available with 6 instruction bits and 2 parameter bits, (including steps such as stirring or shaking as well as system tasks), this version will utilize a 4/4 instruction/parameter division resulting in a reduced set of steps with more parameters. This basic set will be able to specify 15 different liquid components and a quantity for each one. In order to locate the virtual cup under the virtual valve, the virtual motor controller will be capable of lighting two lights, one to move the cup to a higher numbered valve, and one to move it to a lower numbered valve. When the system receives a pour instruction, it will compare the current reported position to the desired position, and toggle one of two outputs from the HC11 to light the up or down light. When the correct position indicator is activated, the motor light will go out and virtual pouring can commence. For adding liquid components, the four parameter bits will be multiplied by a smallest unit (such as a half shot), which adjustable using a potentiometer attached to the HC11's DAC. For rare instances where more than a single 15 unit pour is required, two recipe steps can be issued for the same ingredient. To open the virtual valve, the HC11 will send a number corresponding to that valve to the FPGA, which will convert it to a one-hot code. This will be output to an LED indicator, which can eventually be replaced with an opto-electric isolator capable of switching the AC power needed to actuate a real valve or other bar monkey sub-system. After the period of time required to pour the liquid has passed, the indicator will turn off, and the control system will notify the host computer that it has finished the recipe step by returning a done signal.

New Hardware

The robotic bar control system ultimately gets its information from a controlling PC. As such, it has to have some way to talk to it to get the information. In designing this project we had the choice of using either parallel or serial communication. We needed slow, two-way communication with a minimum of wiring. As such, serial seemed to use to be the best option. Having decided this we implemented a serial link which will communicate 8-bit serial data between an HC11EVB and any product implementing the RS232 serial communication protocol. The remainder of this section discusses the components of this serial link.

Setting up the HC11 Serial Hardware
The code we used to set up the HC11 hardware for serial communication is given below:

```
BAUD        EQU   $102B           * Baud rate control register
SCDR        EQU   $102F           * Serial communication data
                                  * register
SCSR        EQU   $102E           * Serial communication status
                                  *
SCCR1       EQU   $102C           * Serial communication control
                                  * register
SCCR2       EQU   $102D           * Serial communication control
                                  * register 2
BAUD_MASK   EQU   %00110101       * BAUD register mask: SCP1, SCP0,
                                  * R2, SR0. Sets baud rate to 300
                                  * when using an 8 MHz oscillator
                                  *
SCCR2_MASK  EQU   %00101100       * SCCR2 register mask: RIE, TE, RE.
                                  * Turns on interrupt generation on
                                  * receive,
*
            LDAB  #BAUD_MASK      * Load B with BAUD register mask.
            STAB  BAUD            * Store mask into BAUD register to
                                  * set baud
            CLR   SCCR1           * Clear SCCR1. Sets up for standard
                                  * serial communication.
            LDAB  #SCCR2_MASK     * Load B with SCCR2 register
            STAB  SCCR2           * Store mask into SCCR2 register to
                                  * enable transmitter, receiver, and
                                  * receiver interrupt.
            CLI                   * Turn on interrupts.
```

This code sets up the serial system to communicate at 300 baud using 8-bit characters, turns on the on-board serial transmitter and receiver, and enables the receiver interrupt. All of the configuration registers are described in detail in the HC11 reference manual and in Miller.

Receiving a Byte from the PC
You don’t generally know when the device at the other end is going to send data to you. The solution to this is to handle data reception using interrupts. The following code constructs the correct interrupt vector for the SCI interrupt. It is important to know that the vector for an HC11 operating in single chip mode can be different from the vector for an HC11 on an EVB running in expanded mode.

```
*
SERIAL_VECTOR     EQU   $00C4     * Vector for SCI interrupts
*
            ORG   SERIAL_VECTOR
            JMP   RDRF            * Jump to RDRF service routine
```

In this case, it sets the vector to the routine labeled RDRF show below:

```
SCDR        EQU   $102F           * Serial communication data
                                  * register
SCSR        EQU   $102E           * Serial communication status
                                  * register
```

```
RDRF    CLR   TEMP       * Store 0 into TEMP to signal main
                         * program to stop
        LDAA  SCSR       * Read SCSR to clear serial status
                         *
        LDAA  SCDR       * Load incoming input in
        STAA  INPUT      * Store input into INPUT
        RTI              * Return
```

This routine reads the serial status register and then reads the serial data register. This clears the serial status flags (necessary for resetting the serial interrupt) and load the received byte into port A. This byte is then stored in some memory location. You cannot use registers to return the result; when an interrupt is called on the HC11 it pushes the contents of the registers onto the stack. Conversely, when an RTI instruction is issued it pops the saved information from the stack back into the registers. Thus, if you try to pass information back using a register it will get overwritten.

Sending a Byte to the PC

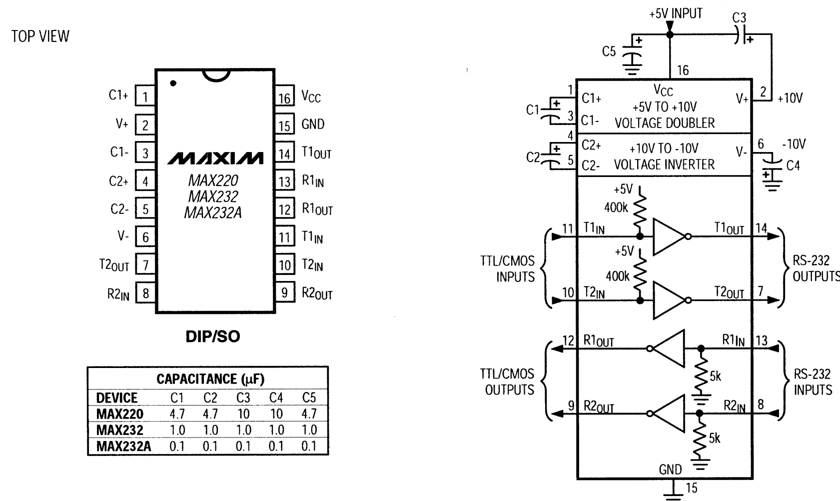
Sending a byte to the PC is easy. Just read the serial status register to make sure the status flags are cleared, and then write the desired byte to the serial data register.

Connecting the EVB to the PC

The RS232 standard used in all modern PCs specifies voltages from –5 to –15 for logical one and 5 to 15 for logical zero, as opposed to the HC11 which uses 5 volts for logical one and 0 volts for logical zero. Because the HC11's SCI interface is not RS232 compliant, a transceiver chip is required to convert the voltages. As shown in the schematics section, the Maxim MAX232A chip converts two 0,5 volt inputs to ±12 volt outputs, and two ±12 volt inputs to 0,5 volt outputs. This particular chip generates the required voltages using only a 5 volt input, but requires external capacitors to do so. Additionally, there are a number of control lines in the RS232 specification (see appendix D), but we found that using our system and a VT-220, we could transmit and receive at speeds of at least 9600 bps with only the Tx, Dx, and ground lines connected.

Schematics

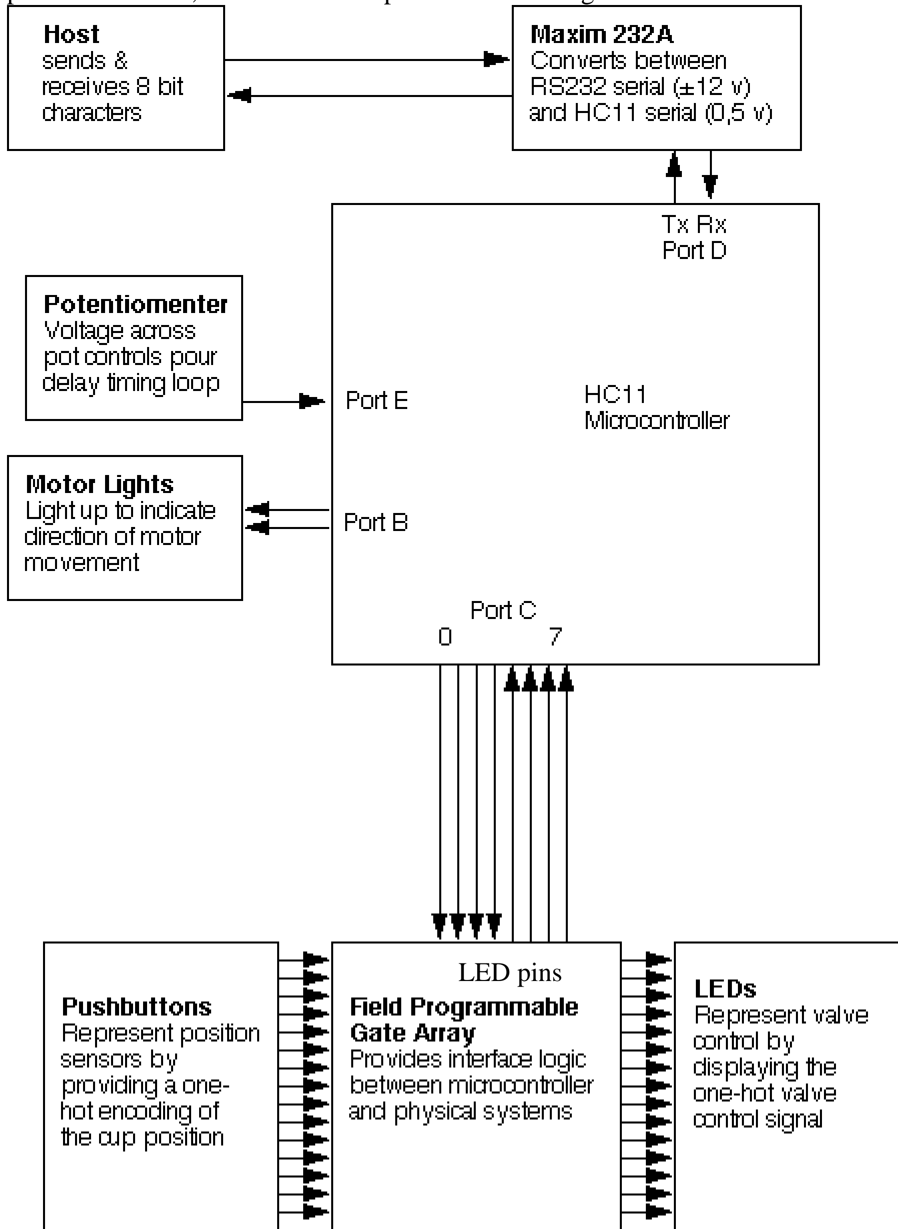
This is the schematic from the Maxim MAX232A data sheet (see appendix E), showing how to hook up the external capacitors and the serial inputs and outputs.



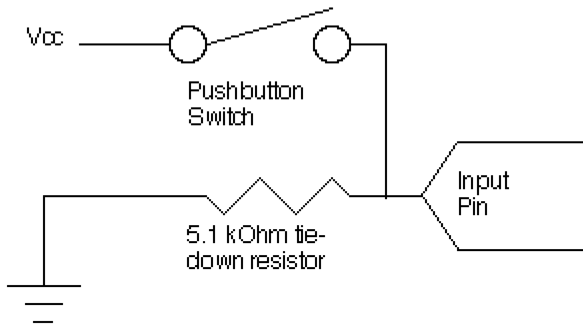
CAPACITANCE (µF)					
DEVICE	C1	C2	C3	C4	C5
MAX220	4.7	4.7	10	10	4.7
MAX232	1.0	1.0	1.0	1.0	1.0
MAX232A	0.1	0.1	0.1	0.1	0.1

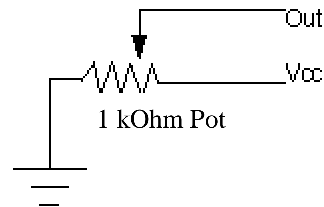
The following block diagram indicates the interconnection of the major components of out system. The host PC communicates with the HC11 through the Maxim MAX232A chip, which converts between HC11 and RS232 serial. The potentiometer for the timer control loop adjustment is connected to the HC11's digital to analog converter. The motor direction indicator lights are connected to two of the pins of port B. The connection between the EVB and FPGA uses a simple parallel interface, with 4 bits in each

direction. On the EVB side, this interface is connected to the wire wrap pins for port C, and on the FPGA side to the same pins as the on-board LEDs (for easy debugging). The FPGA receives 15 inputs from the position switches, and sends 15 outputs to the valve lights.



The following schematics are for the small circuits connected to the EVB and FPGA , including the valve indicator lights, input switches, and the pot used for adjusting the timing loop.





Microcontroller Design

One Motorolla HC11EVB is used in the construction of the bar control system. The HC11 is primarily responsible for the coordination of the control signals used to operate the physical apparatus. This was achieved using a small program written in HC11 assembly language which is discussed below. The code for the control program can be found in Appendix B.

Program Layout

1. Set up serial receive interrupt vector
2. Initialize HC11 serial hardware
    3. Set baud rate to 300
    4. Use default serial communication characteristics (8-bit chars, etc.)
    5. Turn on HC11 transmitter and receiver, and enable receive interrupts.
6. Initialize HC11 A/D hardware
    7. Turn on A/D hardware
    8. Set A/D to do continuous conversion of the signal on pin E7
9. Miscellaneous configuration
    10. Set direction of port C pins
    11. Set stack pointer. Stack is required by interrupt system
    12. Turn on interrupts
13. Wait until a character is received via serial communication with controlling PC. Service receive interrupt, reading input from serial data register and setting a flag which indicates that the program can proceed.
14. Calculate desired position based on input character and read current position of cup from port C. Write to port D to turn on the appropriate motor control line such that the cup moves towards desired position.
15. Wait, reading position from port C, until current position of cup is equal to desired position.

16. Write desired position to low 4 bits of port C, opening appropriate valve
17. Get timing loop "tweak" value from A/D data register 1 and calculate amount to pour based on input character.
18. Wait time specified by pour amount and "tweak" value
19. Write 0 to port C, closing the valve.
20. Send "done with step" signal to PC by writing to serial data register.
21. Branch to 13.

Input/Output
1. Port A: Not used
2. Port B: Pins B0 and B1 control the left and right motor lines.
3. Port C: Pins C7-C4 are configured as input pins. The FPGA passes them a 4-bit number which gives the current cup position. Pins C3-C0 are configured as output pins. They pass a 4-bit number to the FPGA which specifies which valve to turn on.
4. Port D: TX and RX pins used for serial communication with PC. These pins are tied to the appropriate terminals of the MAX232A transceiver.
5. Port E: The V_{RL} and V_{RH} pins are tied to pin 0 (ground) and pin 26 (V_{DD}) respectively. Pin E7 is connected to a 0-5 V potentiometer which supplies the "tweaking" signal for the timing loop.

General Comments
As currently constructed, the program given in Appendix B is small enough to fit in the onboard EEPROM of the HC11 with a little room to spare. However, there is plenty of opportunity for optimization of the program to decrease its size further to allow for additional features while remaining small enough to fit in onboard EEPROM.

Lessons for the Future
Serial I/O is non-trivial. In particular, when using an EVB rather than a straight HC11 chip it is important to set your interrupt vector correctly. The procedure which is given in Miller is appropriate only for a straight HC11 chip; check the EVB manual for how to set up interrupt vectors for the HC11EVB. Also, the easiest way to test serial I/O is to drag out a VT-220 and use it in lieu of a computer. Pressing a key automatically sends the character to the EVB, and anything sent from the EVB automatically shows up on the screen, making it unnecessary to deal with serial I/O on a PC.

FPGA Design

One Xilinx XCS10P84 FPGA is used in the construction of the bar control system. The FPGA is primarily responsible for encoding and decoding control signals obtained from the HC11 microcontroller via continuous combinational logic. It implements two basic hardware modules which are tied together in a single package by use of a simple top level module, all of which are discussed below. The Verilog code for these modules is given in Appendix C.

Top Level Module
The top-level module accepts two inputs, one 4-bit and one 15-bit. The 4-bit input is passed to the FPGA from pins C0-C3 of the HC11EVB, and specifies which one of 15 valves should be opened. The 15-bit input is passed to the FPGA from a series of 15 switches which are depressed by the cup-carrier, giving the current position of the cup. The top-level module passes the 4-bit input to the decode module and gets back a 15-bit decoding. These 15 bits control 15 valves, opening them when high and closing them when low. The top-level module passes the 15-bit input to the encode module and gets back a 4-bit encoding. These 4 bits are output to pins C4-C7 on the EVB, providing it with the current position of the cup.

The Decode Module
The decode module takes a 4-bit number and turns it into a 15-bit one-hot encoding. Inputting 0 causes the output to be zero, while inputting any other number *n* causes the $(n\text{-}1)^{th}$ bit of the output to go high.

The Encode Module
The decode module takes a 15-bit one-hot input and turns it into a 4-bit output. Inputting 0 or a signal which is not one-hot causes 0 to be output, while inputting a signal with bit *n* high results in the output of the number *n+1*.

Lessons for the Future
The decoder module is not designed in the most efficient manner for this application. Due to the physical construction of the mixing apparatus there is no chance that more than one bit will be high at any given instant. As such, this module could be redone using simple combination logic rather than a "case" statement, which would map to a significantly simpler circuit.

Results

Although we did not create a complete bar monkey as we had originally intended, we successfully completed a bar monkey control board prototype which met our revised specifications. The system receives characters from the VT-220 terminal we used for testing, lights a motor direction light until the correct location switch is depressed, and then turns on the valve light for the specified period of time. After that time, the light will turn off, and the system will return a done character (&).

The scope of the original project was reduced for a number of reasons. We first decided to switch to a simple indicator light scheme in order to avoid dealing with the 110 VAC required for the valves we had on hand. Once we eliminated the physical system, we also cut the PWM motor controller and pour duration multipliers, as these are of no use without the physical system. We ended up using the HC11 EVB and FPGA boards which we had on hand to create the system because of the difficulty of getting custom PCBs made.

References

[1] Miller, Glenn H. Microcomputer Engineering.
[2] HC11 Reference Manual
[3] ask.com's RS232 Serial Protocol Info Sheet http://www.ctips.com/rs232.html (in appendix D)

Parts List

Part	Source	Vendor Part #	Price
Maxim MAX232A serial transceiver	Scrap Bin (available from DigiKey)	MAX232ACPE-ND	$4.88
1 k-ohm potentiometer	Lab cabinet		
25 pin female D-sub connector	Stock Room		
10 pin Ribbon Cable	Stock Room		
Red and Green LEDs	Lab cabinet		
MC68HC11 EVB	Stock Room		
FPGA Board	Class		
Pushbutton switches (reset button type)	Stock Room		

Appendices

A. FPGA pin assignments.
B. ASM code.

C. Verilog code.
D. RS232 Serial Protocol Info Sheet (printed separately or at http://www.ctips.com/rs232.html)
E. Maxim Serial Transceiver Data Sheet (printed separately or at http://209.1.238.250/arpdf/1798.pdf)

Appendix A: FPGA Pin Assignments

FPGA Pinlock file gives the pinouts used for our prototype configuration.

```
#PINLOCK_BEGIN
#Thu Nov 18 13:59:27 1999

#these are the inputs for the position determining

NET "pos_in<0>"       LOC =
NET "pos_in<1>"       LOC =  "P8";
NET "pos_in<2>"       LOC =  "P10";
NET "pos_in<3>"       LOC =  "P14";
NET "pos_in<4>"       LOC =  "P18";
NET "pos_in<5>"       LOC =  "P19";
NET "pos_in<6>"       LOC =  "P24";
NET "pos_in<7>"       LOC =  "P26";
NET "pos_in<8>"       LOC =  "P27";
NET "pos_in<9>"       LOC =  "P28";
NET "pos_in<10>"      LOC =  "P29";
NET "pos_in<11>"      LOC =  "P38";
NET "pos_in<12>"      LOC =  "P39";
NET "pos_in<13>"      LOC =  "P40";
NET "pos_in<14>"      LOC =  "P44";

#this is the 4 bit parallel bus which sends the hex
#encoding of the position from the FPGA to the HC11

NET "pos_out<0>"      LOC =  "P66";
NET "pos_out<1>"      LOC =  "P65";
NET "pos_out<2>"      LOC =  "P62";
NET "pos_out<3>"      LOC =  "P61";

#this is the 4 bit parallel bus which receives the hex
#encoding of a valve number sent to the FPGA from the

NET "select_in<0>"    LOC =
NET "select_in<1>"    LOC =
NET "select_in<2>"    LOC =  "P68";
NET "select_in<3>"    LOC =  "P67";

#these are the outputs to the virtual valve control lights

NET "select_out<0>"   LOC =  "P50";
NET "select_out<1>"   LOC =  "P51";
NET "select_out<2>"   LOC =  "P56";
NET "select_out<3>"   LOC =  "P57";
```

```
NET "select_out<4>"  LOC =  "P58";
NET "select_out<5>"  LOC =  "P59";
NET "select_out<6>"  LOC =  "P60";
NET "select_out<7>"  LOC =  "P77";
NET "select_out<8>"  LOC =  "P78";
NET "select_out<9>"  LOC =  "P79";
NET "select_out<10>"  LOC = "P80";
NET "select_out<11>"  LOC =  "P81";
NET "select_out<12>"  LOC =  "P82";
NET "select_out<13>"  LOC =  "P83";
NET "select_out<14>"  LOC =  "P84";

#PINLOCK_END
```

Appendix B: HC11 Assembly Code

```
* Filname: final.asm
* Author: Glenn Gebhart  glenn@cs.hmc.edu>
* Date:
* Description: Assembly language program for controlling a robotic
* bar.
*
*************
* Constants *
*************
*
AMOUNT_MASK EQU   %00001111         * Masks off bits which carry
                                    * information about the amount of
                                    * liquid to
POSITION_MASK     EQU   %11110000   * Masks off bits which carry
                                    * information about where to
                                    * position the cup
*
PORTB       EQU   $1004             * Address of port B
PORTC       EQU   $1003             * Address of port C
DDRC        EQU   $1007             * Data direction register for port
                                    *
BAUD        EQU   $102B             * Baud rate controll
SCDR        EQU   $102F             * Serial communication data
                                    * register
SCSR        EQU   $102E             * Serial communication status
                                    * register
SCCR1       EQU   $102C             * Serial communication control
                                    * register
SCCR2       EQU   $102D             * Serial communication control
                                    * register
OPTION      EQU   $1039             * Hardware option
ADCTL       EQU   $1030             * A/D control register
ADR1        EQU   $1031             * A/D result register 1
*
SERIAL_VECTOR     EQU   $00C4       * Vector for SCI interrupts
*
```

```
MOTOR_OFF    EQU   0                  * Motor for positioning cup is off
MOTOR_LEFT   EQU   %00000001          * Motor moves cup left
MOTOR_RIGHT  EQU   %00000010          * Motor moves cup right
*
DDRC_MASK    EQU   %00001111          * Mask for DDRC, makes low 4 bits
                                      * of port C output bits, high 4
                                      * bits of port C input
INPUT_BITS   EQU   %11110000          * Masks off input
OUTPUT_BITS  EQU   %00001111          * Masks off output bits
*
CLOSE        EQU   %00000000          * Value to write to port C to turn
                                      * off pour
*
BAUD_MASK    EQU   %00110101          * BAUD register mask: SCP1, SCP0,
                                      * R2, SR0. Sets baud rate to 300
                                      * when using an 8 MHz oscillator
                                      * crystal
SCCR2_MASK   EQU   %00101100          * SCCR2 register mask: RIE, TE, RE.
                                      * Turns on interrupt generation on
                                      * receive, enables transmitter and
                                      * receiver.
*
OPTIONMASK   EQU   %10000000          * Mask for OPTION register: ADPI.
                                      * Turns on A/D
ADCTL_MASK   EQU   %00100111          * Mask for ADCTL register: SCAN,
                                      * CC, CB, CA. Continuous scan of
                                      * PE7, result dumped into
*
STEP_FINISHED      EQU   $A6          * Value to send to PC when a step
                                      * is
*
*************
* Variables
*************
*
             ORG   $0000
INPUT        FCB   0                  * Holds current input value
TEMP         FCB   0                  * Scratch space
CURRENT_POS  FCB   0                  * Current position of cup
DESIRED_POS  FCB   0                  * Desired position of cup
*
****************************
* Interrupt vector settings *
****************************
*
             ORG   SERIAL_VECTOR
             JMP   RDRF               * Jump to RDRF service routine
*
***********
* Program *
***********
```

```
*
            ORG    $d000
*
* Serial initialization
*

            LDAB   #BAUD_MASK        * Load B with BAUD register mask.
            STAB   BAUD              * Store mask into BAUD register to
                                     * set baud
            CLR    SCCR1             * Clear SCCR1. Sets up for standard
                                     * serial communication.
            LDAB   #SCCR2_MASK       * Load B with SCCR2 register
            STAB   SCCR2             * Store mask into SCCR2 register to
                                     * enable transmitter, receiver, and
                                     * receiver interrupt.
*
* A/D converter initialization
*
            LDAA   #OPTIONMASK       * Load A with mask for hardware
                                     * option
            STAA   OPTION            * Store in OPTION register to turn
                                     * on A/D
            LDAA   #ADCTL_MASK       * Load A with mask for A/D control
                                     * register
            STAA   ADCTL             * Store into ADCTL to start
                                     * continuous conversion of signal
                                     * on
*
* Misc
*
            CLRA                     * Zero out
            LDAB   #DDRC_MASK        * Load B with port C config
                                     * information
            STAB   DDRC              * Store into DDRC to program the
                                     * pin directions of port
            LDS    #$CFFF            * Initialize stack. Stack grows
                                     * from high to low memory, so the
                                     * stack is placed right before the
                                     * program code. Stack is needed by
                                     * interrupt
*
* Get next
*
            CLI                      * Turn on
GET_INPUT   LDAB   #1                * Load B with 1
            STAB   TEMP              * Store into temp
HERE        CMPB   TEMP              * Compare B and TEMP
            BEQ    HERE              * Wait here until value in TEMP is
                                     * modified by RDRF interrupt
                                     * service
*
```

```
* Turn on correct
*
            LDAA  INPUT              * Load A with input
            STAA  TEMP               * Store input value into TEMP
            BCLR  TEMP AMOUNT_MASK   * Clear all bits of input not
                                     * encoding position
            LDAA  PORTC              * Load A with current position of
                                     * cup
            STAA  CURRENT_POS        * Store current position into
                                     *
            BCLR  CURRENT_POS AMOUNT_MASK * Clear all bits of current
                                          * position not encoding
                                          * position
            LDAA  TEMP               * Load A with desired
            CMPA  CURRENT_POS        * Compare current position with
                                     * input
            BEQ   POUR               * If equal, we're already at the
                                     * right position, jump to
            BHI   MOVE_LEFT          * If the number is higher, move
                                     * left
            LDAB  #MOTOR_RIGHT       * Load move right mask into
            STAB  PORTB              * Store move right mask into port B
                                     * to turn on
            STAA  DESIRED_POS        * Store desired position
            BRA   WAIT               * Branch to positioning routine
MOVE_LEFT   LDAB  #MOTOR_LEFT        * Load move left mask into B
            STAB  PORTB              * Store move left mask into port B
                                     * to turn on
            STAA  DESIRED_POS        * Store desired
*
* waits for cup to hit correct position
*
WAIT        LDAA  PORTC              * Read from port C to get current
                                     * position of
            STAA  TEMP               * Store into
            BCLR  TEMP OUTPUT_BITS   * Clear bits which don't carry
                                     * input
            LDAA  TEMP               * Load back into
            CMPA  DESIRED_POS        * Compare current position of cup
                                     * with desired
            BNE   WAIT               * If not equal, cup isn't in the
                                     * right position yet, so go back
                                     * to top of the
*
* Turn off
*
            LDAB  #MOTOR_OFF         * Load B with MOTOR_OFF
            STAB  PORTB              * Store into port B, turning motor
                                     * off
*
* Open correct valve
```

```
*
POUR        LDAA  INPUT               * Load input value into A
            LSRA                      * Shift right 4 times to put valve
                                      * number in lower 4
            LSRA
            LSRA
            LSRA
            STAA  PORTC               * Store into port C, opening
                                      * correct
*
* Delay
*
            LDAA  ADR1                * Get "tweak" value from A/D of
                                      * attached
            STAA  TEMP                * Store value in TEMP
            CLRA                      * Zero out
            BCLR  INPUT POSITION_MASK     * Clear bits not dealing with
                                          * amount to
SLEEPTOP    CMPA  INPUT               * Compare A to amount to pour
            BEQ   SLEEPBOTTOM         * If equal we've poured enough
            CLRB                      * Zero out B
LOOP1TOP    CMPB  TEMP                * Compare to "tweak" value
            BEQ   LOOP1BOTTOM         * If equal jump to bottom of loop
            LDX   #0                  * Zero out X
LOOP2TOP    CPX   #1000               * Compare X to some medium sized
                                      * constant
            BEQ   LOOP2BOTTOM         * If equal jump to bottom of
            INX                       * Increment
            BRA   LOOP2TOP            * and jump to top of the
LOOP2BOTTOM INCB                      * Increment
            BRA   LOOP1TOP            * and jump to top of the loop
LOOP1BOTTOM INCA                      * Increment A
            BRA   SLEEPTOP            * and jump to top of the loop
*
* close spout
*
SLEEPBOTTOM LDAA  #CLOSE              * Load A with close mask
            STAA  PORTC               * Store into port C, turning off
                                      * valves
*
* send done with step signal
*
            LDAA  SCSR                * Load A with SCSR to clear serial
                                      * status
            LDAA  #STEP_FINISHED      * Load A with STEP_FINISHED
            STAA  SCDR                * Store into SCDR to send to
                                      * controlling
*
* Get another
*
            BRA   GET_INPUT
```

```
*
***********************
* RDRF service routine *
***********************
*
RDRF         CLR   TEMP              * Store 0 into TEMP to signal main
                                     * program to stop
             LDAA  SCSR              * Read SCSR to clear serial status
                                     * flags
             LDAA  SCDR              * Load incoming input in
             STAA  INPUT             * Store input into INPUT
             RTI                     * Return
             END
```

Appendix C: Verilog Code

```
// Filename: control.v
// Author: Glenn Gebhart glenn@cs.hmc.edu
// Date: 11/20/1999
// Description: Continuous combinational logic for robotic bar controll
// board
// select_in is a 4-bit number which selects which one of the 15 lines
// in select_out to set high. If select_in is 0, no lines in select_out
// are set high.
// pos_in is a 15-bit one-hot number giving the current position of the
// cup-holder pos_out is the 4-bit decoding of pos_in. 0 indicates that
// no lines of pos_in are hot.

`include "encode.v"
`include "decode.v"

module CONTROL (select_in, select_out, pos_in, pos_out) ;

input        [3:0]       select_in   ;
input        [14:0]      pos_in            ;

output       [3:0]       pos_out     ;
output       [14:0]      select_out  ;

ENCODE encode1(pos_in, pos_out);
DECODE decode1(select_in, select_out);

endmodule
```

```
// Filename: encode.v
// Author: Glenn Gebhart glenn@cs.hmc.edu
// Date: 11/20/1999
// Description: Encodes 15 1-hot to 4-bit binary .
// Returns 0 if no lines are hot or if more than one line is hot.
```

```
module ENCODE (din, dout) ;

input       [14:0]      din   ;

output      [3:0]       dout  ;

reg   [3:0] dout  ;

always @(din) begin
      case (din)
            15'b000000000000000:    dout = 0;
            15'b000000000000001:    dout = 1;
            15'b000000000000010:    dout = 2;
            15'b000000000000100:    dout = 3;
            15'b000000000001000:    dout = 4;
            15'b000000000010000:    dout = 5;
            15'b000000000100000:    dout = 6;
            15'b000000001000000:    dout = 7;
            15'b000000010000000:    dout = 8;
            15'b000000100000000:    dout = 9;
            15'b000001000000000:    dout = 10;
            15'b000010000000000:    dout = 11;
            15'b000100000000000:    dout = 12;
            15'b001000000000000:    dout = 13;
            15'b010000000000000:    dout = 14;
            15'b100000000000000:    dout = 15;
            default:          dout = 0;
      endcase
end

endmodule
```

```
// Filename: decode.v
// Author: Glenn Gebhart glenn@cs.hmc.edu
// Date: 11/20/1999
// Description: 4-bit binary to 15-bit 1-hot
// No output lines are set hot on input

module DECODE (din, dout) ;

input       [3:0]       din   ;

output      [14:0]      dout  ;

reg   [14:0]      dout  ;

always @(din) begin
      case (din)
            0 : dout = 15'b000000000000000;
            1 : dout = 15'b000000000000001;
```

```
            2 :  dout = 15'b000000000000010;
            3 :  dout = 15'b000000000000100;
            4 :  dout = 15'b000000000001000;
            5 :  dout = 15'b000000000010000;
            6 :  dout = 15'b000000000100000;
            7 :  dout = 15'b000000001000000;
            8 :  dout = 15'b000000010000000;
            9 :  dout = 15'b000000100000000;
            10:  dout = 15'b000001000000000;
            11:  dout = 15'b000010000000000;
            12:  dout = 15'b000100000000000;
            13:  dout = 15'b001000000000000;
            14:  dout = 15'b010000000000000;
            15:  dout = 15'b100000000000000;
        endcase
end

endmodule
```