IR Rover

E155 Microprocessor Design
Fall 2002

By Stephen Friedman
and Micah Garside-White

Abstract

Our goal was to create a light-weight lego rover that could be remotely controlled using the built in IR technology of Lego Mindstorms RCX. The controller, which consists of a 4 X 4 keypad, an FPGA, and a HC11 microcontroller, transmits user input signals to the rover, takes in a user defined sequence of directions and durations, and plays the program back to the robot. The FPGA is responsible for retrieving and debouncing the signals from the key-pad, sending the signal to the microcontroller, and triggering an interrupt on the microcontroller. The microcontroller is responsible for checking that the signal is valid, driving the LED User Interface, storing a program, outputting signals to the IR controller either directly or playing back a previously input program. An RCX was used as a transmitter to the RCX on the rover, and the RCX on the rover was responsible for controlling the DC motors that made the rover move.

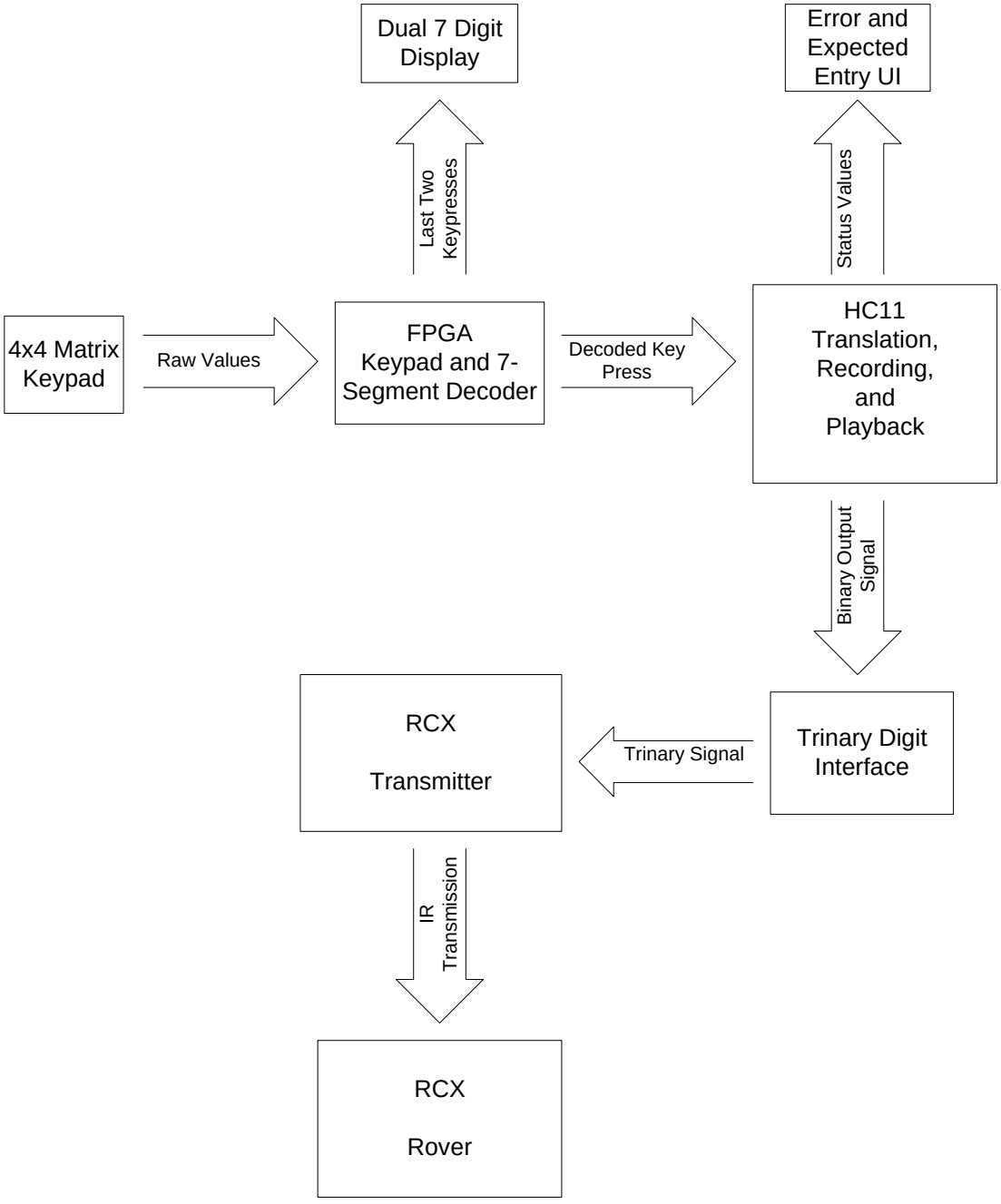
Introduction

The goal of this project is to create a smart controller that uses IR to direct a slave Lego rover. The controller is responsible for the capture and interpretation of user commands. The user interface consists of a 4 X 4 matrix key-pad used for the input of signals, a red error LED to report when a keypress was ignored, two green state LEDs that report what type of input the controller is excepting, and a 7 segment display that shows the 2 most recent keypresses. The key-pad has the directional keys, start programming, start playback, stop, and numerical values all clearly designated on the keys themselves.

The FPGA is responsible for the capturing of the user input from the keypad, the transmission of the keypress to the HC11 microcontroller, triggering the keypress interpreter on the microcontroller, and driving the 7 segment display for the user interface. The FPGA actively polled the keypad, and debounced the signal it received from the keypad by using an appropriate signal. The FPGA stored the keypress along with the other most recent and displayed both results to the 7 segment display. The most recent keypress was also sent to the HC11, along with a pulse that fell and rose which triggered the IC1 interrupt on the HC11.

The HC11 is responsible for the validation of all signals it recieves from the FPGA, driving the LED user interface, storing programs, running stored programs, transmission of directional signals to the Lego rover. The keypress handler was an interrupt routine that was triggered by the IC1 interrupt. The routine would keep the state current, so the microcontroller knew whether to record incoming signals, transmit incoming signals, or playback a stored signal. The microcontroller tracked what type of input it was expecting and with that information drove the state LEDs. Lastly the microcontroller checked the validity of the signal and used that information to drive the error LED.

System Block Diagram



New Hardware

For this project, we interfaced with the Lego RCX Brick[1] with embedded microcontroller. One RCX was programmed to run the rover bot, while the other was used as an IR communications interface with the RCX on the rover. The RCX has 3 input pads available. Connection was made to these input pads by melting wires and tin foil onto a 2x6 Lego plate in such a way that would allow the wires to make contact with the contacts for the input pads on the RCX. Using the Robotics Invention System 2.0 software, a program was designed and loaded onto the RCX that would allow input from these 3 pads. The first input pad was configured in the software as a touch sensor. The other two were configured as temperature sensors. In touch mode, a high or infinite resistance between the two pad connections is interpreted as unpressed and a low resistance is interpreted as pressed. When configured as a temperature sensor, the RCX interprets the resistance between the two pad connections as a temperature of 158.0 degrees Fahrenheit for little resistance, and –4 degrees Fahrenheit for infinite resistance. More detailed information can be found on the **MindStorms RCX Sensor Input Page** and the **Lego Mindstorms Internals** page**.**[2],[3]

To communicate with the HC11, the sensor configured for touch was used as a synchronization signal and the other two inputs were used as trinary input digits. This was accomplished by changing the resistance between the sensor connections using transistors. For the circuit we used, the corresponding values were –4 degrees for a High trinary value, approximately 107 degrees for a Mid trinary value, and 158 degrees for a Low trinary value. Thus, using two inputs we have the values 0 through 8 that can be decoded by the RCX. The data value was put out using 4 bits from the HC11 and encoded in trinary by the interface circuit. The HC11 then notified the RCX that data was available by pulsing the synchronization bit low for approximately a quarter of a second. This was long enough to trigger the click event watcher on the touch sensor in the RCX software, at which point it would grab the data from the other two sensors and send the appropriately numbered IR message to the Rover Bot.

Rover Bot

The Rover Bot was designed using a dual motor drive system. The primary drive motor controls the robot's forward and reverse motion. Through a subtraction differential system, the secondary drive motor controls the turn rate by controlling the differential between the right and left drive wheels. This allows for straight line drive even when the characteristic torques or speeds of the two motors are different.

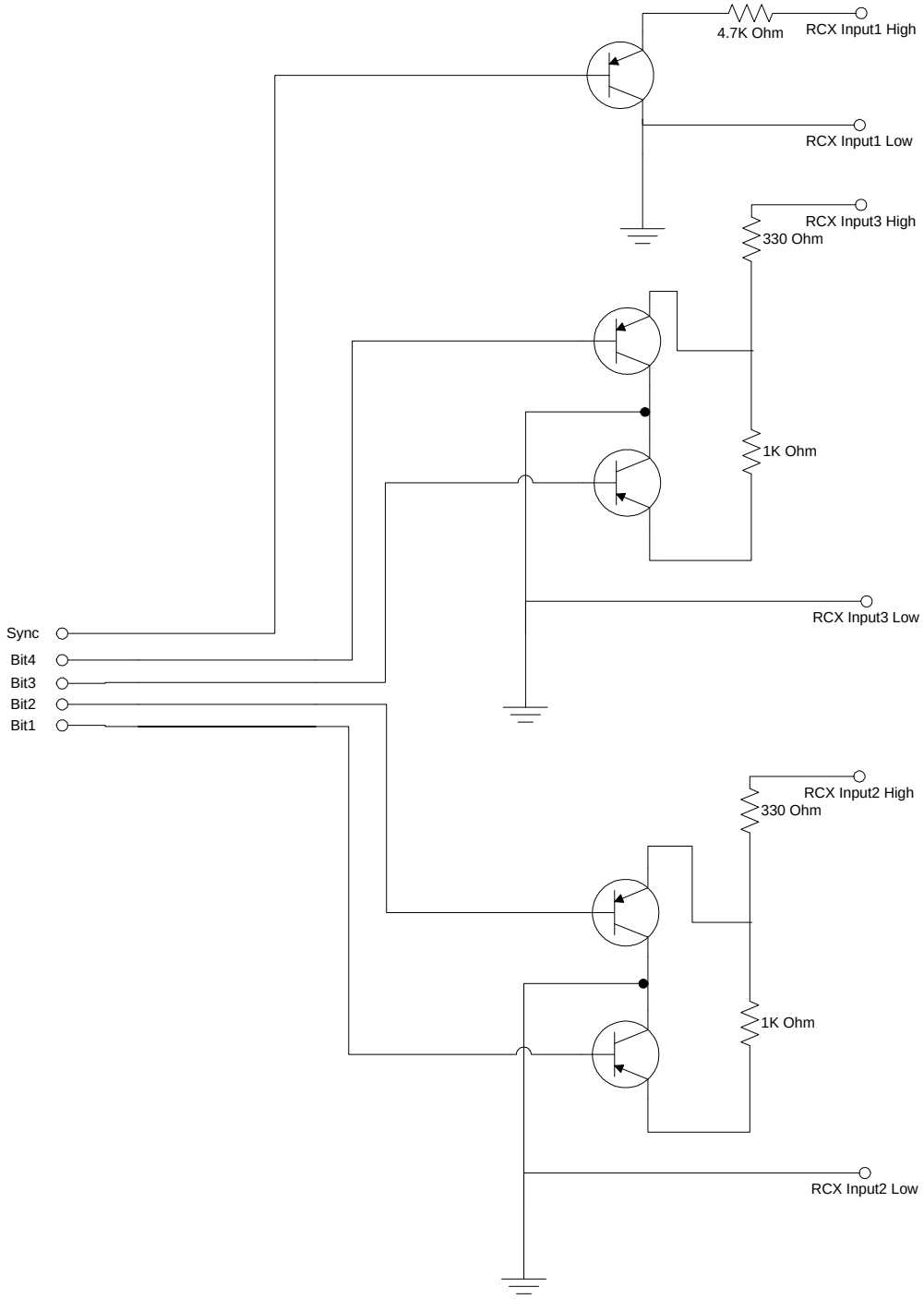
The onboard control software for the Rover Bot was kept simple in support of the lightweight rover paradigm. It was written using the Lego Robotics Invention System 2.0 because of its ease of use. The Rover accepts its input in the form of IR messages numbered 0-255. There is no data payload associated with the message. The Rover interprets messages in the following way:

1. Forward
2. Reverse
3. Turn counter-clockwise
4. Turn clockwise
5. Stop motors

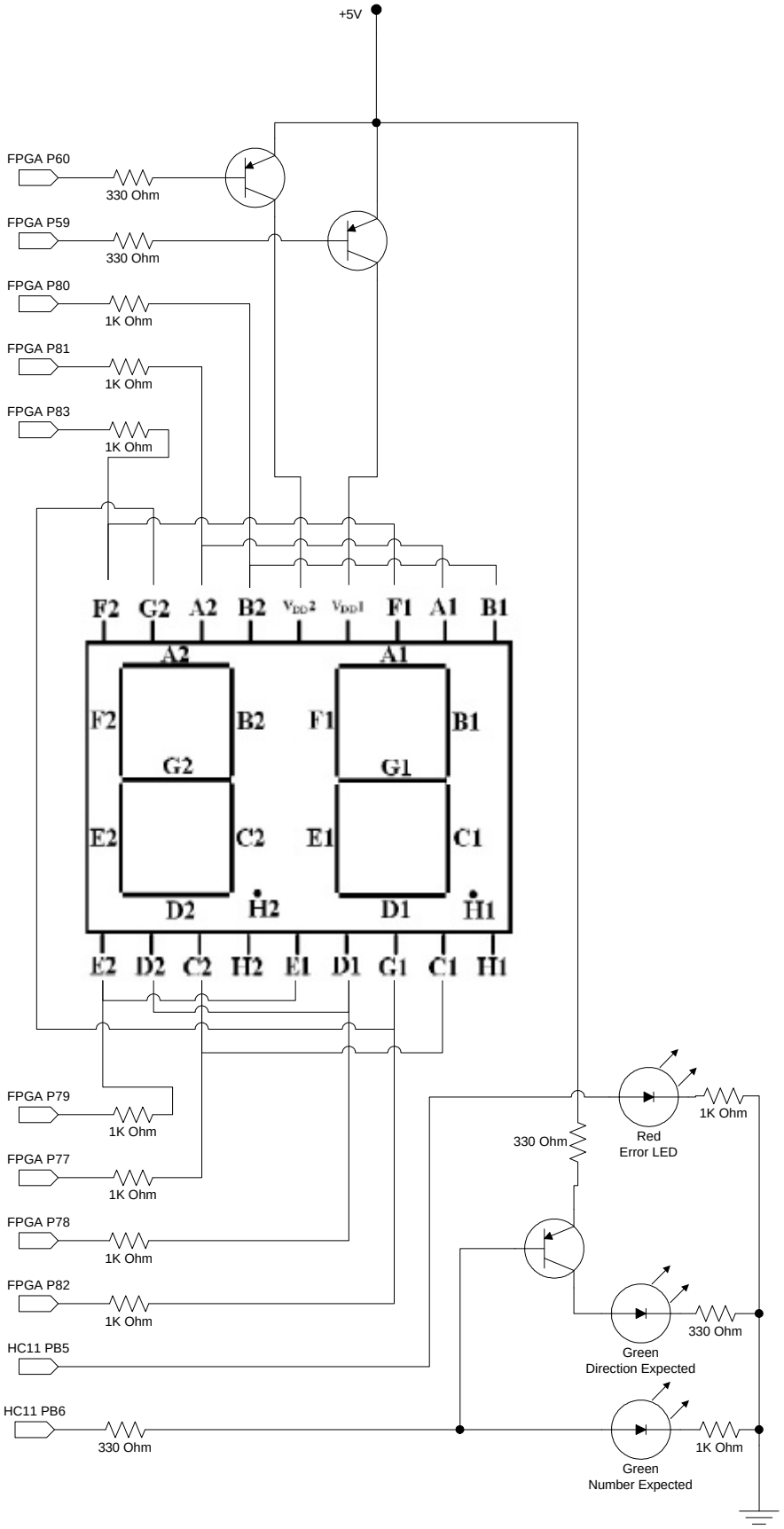
When the Rover receives one of these messages, it sets the motor speeds and directions to appropriately execute that action, and continues to execute it until the next message is received. The software can be easily extended to accept more IR signals and execute more complicated behaviors in the future.

Schematics

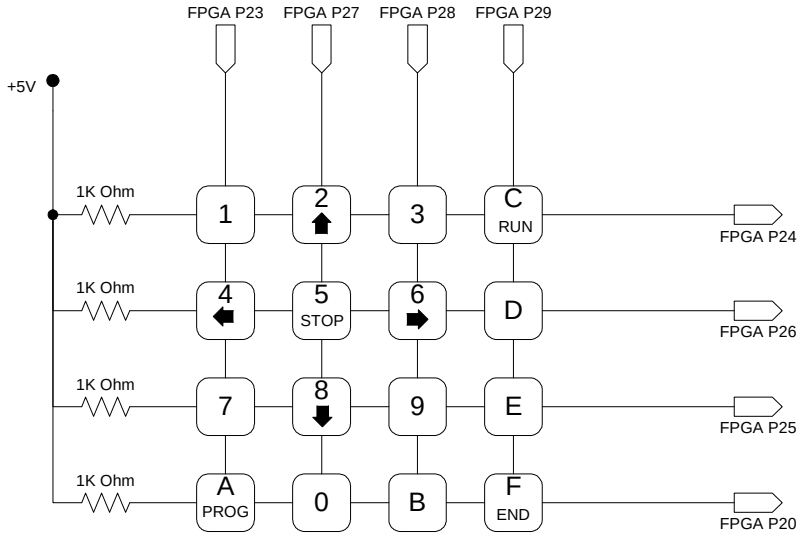
Connection between HC11 & RCX



User Interface Schematic



Connection between FPGA & Keypad



Microcontroller Design

The HC11 is the controller for the system as a whole, and as such it contained the vast majority of the intelligence. It has two main modules that worked cooperatively to provide the signal translation, error and state reporting, singal transmission, and storage and playback capability. The observer pattern is the model for the interactions between the two components. The first component is responsible for setting the state, interpreting the signals, transmitting user signals, and driving the error LED. The second component uses the state to drive the playback functionality and it drives the expected state LEDs.

The first module is an interrupt response triggered by the Input Capture 1 interrupt, on Port A bit 2. When the HC11 sees the signal on Port A bit 2 go from low to high it triggers the IC1 interrupt jump table, which executes the code starting at memory location $D100. First the routine determined whether the keypress, which was brough in over the 4 least significant bits of Port E, would effect the current state of the software (i.e. whether the keypress would trigger the initiation of play-back mode or end programming mode). If there was any change in state the STATE memory location was updated with the new state and the routine would finnish. If the keypress had no effect on state it was then checked to be a valid keypress for the state the controller was in. For the numerical accepting state any keypress was valid, but for the direcrtional state only 2, 4,5,6, & 8 where valid keypresses.

Once the input was validated one of two subroutines would be called to either execute the programming or transmission functionality of the controller. If the controller was in the transmit state the directional input would be translated into a signal that the RCX controller could understand and then sent to the 4 least significant bits of Port B. Then the 5 LSB of Port B was toggeled down, the controller busy waited for approximately ¼ of a second, and the bit was brought back to high.

If the controller was in programming state the microcontroller would once again determine which of two subroutines the input should be passed to, but this time using the expected input, INPUT, memory location. If the controller was expecting a direction the direction would be translated to the RCX code and stored in the first 4 bits of the memory table that is pointed to by the NEXT memory location. If the controller was expecting a numerical input the controller would store that number in the upper 4 bits of the table and incriment the NEXT memory location. If any input was found to be problimatic the error LED was activated via PORT B bit 5 and the input was dropped.

The main loop of the program would output the expecting state of the controller to the 2 STATE LEDs using Port B bit 6, then it would check to see if the controller was in playback state. If it wasn't then the main program would loop. If it was in play-back state then the program would loop through the Table and play the signal in the current entry then busy wait for the time, in seconds, noted in that same entry. While busy waiting the program would check to see if the controller had ceased being in the play-back state. If any entry had a 0 for the time of delay that signal was not output.

For the switching of states the controller could only switch into play-back from transmission and vice-versa. The same was true for the programming state. If the user tried to change state in any other way an error was raised. Whenever the programming state ended a null entry ($0) was written into the table to signify where the table ended; on initiation the program put a null entry at the beginning of the tabel ($D300). Whevenever the null entry was reached in play-back the NEXT memory location was set to the beginning of the table and the main loop would start again.

The key algorithm was to use interrupts to implement an observer pattern based on the input of the user and the use of dedicated memory locations to tell the main loop what state it is in. The rest of the program was straight forward signal error checking, table storage and retrieval of information, and busy waiting.

FPGA Design

The FPGA was primarily configured to read the matrix keypad and relay key press information on to the HC11. A Polling FSM of 3 states reads the keypad. In the PollKeypad state, a second 4-state Cycling FSM is running which cycles one bit low on each of the column inputs. While the Polling FSM is in the PollKeypad state, the Cycling FSM successively transitions through its states. Otherwise, the polling FSM is held in its current state. This allows the decoder to read both the output and the input and map it to the sixteen corresponding values on the keypad. When a press is detected, the cycling FSM is stopped and a sync signal is generated for one cycle. Then the FSM enters the Hold state, where the input to the decoder will be held the same until the user releases the key. At that time, the PollKeypad state is re-entered, and the Cycling FSM is restarted.

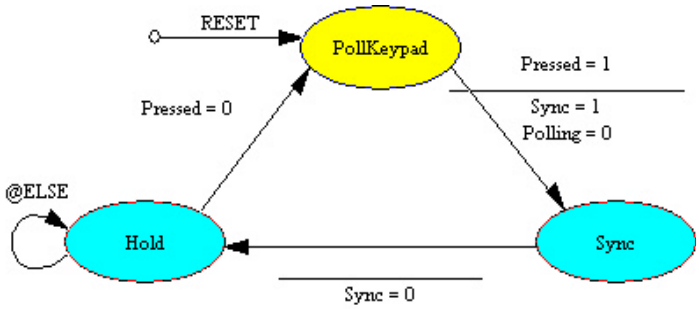


Figure 1 Polling FSM

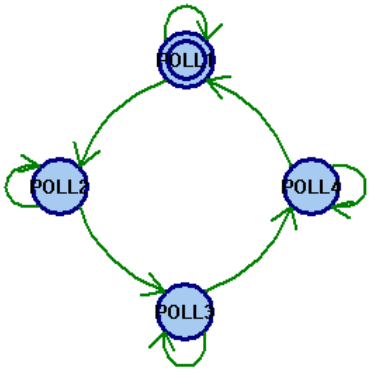


Figure 2 Cycling FSM

There is also a multiplexed dual 7-segment display driver on the FPGA that is used to display the user's last two key presses. This simply takes the 4-bit input representing the key press from a storage register, and decodes it into a seven-segment display. It also uses a clock divider to obtain a frequency around 60 Hz for switching between displaying the left and right digits, to provide the illusion of two simultaneous displays.

In addition, there is logic to provide an interrupt pulse to the HC11 when a new key press is available and a 4-bit bus to allow access to that key press. The rest of the logic on the FPGA is set up to clean the signal from the keypad, run the FSM at a reasonable clock rate, and store the last two key presses. See the Appendix for the block diagram of the FPGA.

Results

The results of our project was a fully functional rover that could respond to commands issued by our smart controller. The controller itself could reliably and consistantly transmit signals, store programs, and transmits the program it currently had stored in memory. Our final project did not use the SCI or SPI, as was origionally planed, nor the IR module, nor the independent propulsion of the rear wheels. Programming and playback of a stored program was added, as was a custom protocol for communicating with the RCX (using trinary digits and an A/D converter).

The two most difficult parts of our design where our failed attempts at using a non-RCX based IR transmitter and the building of the differential that was used to steer the rover. Origionally we planned on sending a 2400 Baud RS-232 protocol signal to a 238 KHz singal modulator to an IR transmitter. We were going to use the SCI port on the HC11 to generate the RS-232 signal at 2400 Baud. The modulator and transmitter were constructed out of a 555 timer circuit and an infered LED. This should have functioned correctly, but unfortunately the RCX IR protocal uses bit balancing rather than a true RS-232 protocol. Thus we needed to have one of the controll signals inverted and this couldn't be adjusted on the HC11. Rather than try to hack together hardware to invert the first controll signal on the FPGA we opted to create an interface to another RCX and have the new RCX communicate with the one on the rover.

The origional plan was to have a single motor drive a single wheel, and thus steering would be accomplished by the differnce in speed of the two motors. The origional model created a right hand drift, because the 2 motors had different output characteristics. To correct for this a single motor was used to drive the forward and reverse directions through a differential gear box and a second motor was used to spin the differential thus causing one wheel to slow and the rover to turn. The syncing of the gears proved to be a wicked problem.

References

[1] Official Lego Mindstorms Site, http://mindstorms.lego.com/ .

[2] Gasperi, Michael. **MindStorms RCX Sensor Input Page** http://www.plazaearth.com/usr/gasperi/lego.htm

[3] Lego Mindstorms Internals, http://www.crynwr.com/lego-robotics/ .

[4] HC11, M68HC11 Reference Manual. ©Motorola Inc 1991.

[5] HC11, MC68HC11A8 Technical Data. ©Motorola Inc 1991.

[6] M68HC11EVB Evaluation Board User's Manual. ©Motorola Inc September 1986.

Part List

Part	Source	Vendor Part #	Price
Lego RIS 2.0	http://shop.lego.com/	Item#3804	$199.99
Lego RCX1.0	Lego Imagination Center, Downtown Disney, Aneheim	Item#9709	$69.99

Appendices

A) Assembly Code:

```
*******************************************************
*       Authors:  Micah Garside-White & Stephen Friedman
*       Purpose:  Create a controller for a rover robot that will
*                 transmit signals, and program and replay user input
*       Date:  12 – 10 – 02
*******************************************************
*       Ports of interest
*
*       IC1 = PA2
*
*       Interrupt Vectors of interest
*
*       Timer Input Capture 1 = $00E8 - $00EA
******************************************************* 11
*******************************************************
*       The Constanstants
*
*       NEXT - A poitner to the next place in the memory table that a record should
*            be written to
*
*       TABLE - Designates the memory location where the user programed sequence of
*               commands will be kept.
*
*       TMSK1 - Timer Interrupt Mask Register 1
*
*       TCTL1 - Timer Control Register 1
*
*       TCTL2 - Timer Control Register 2
*
*       TFLG1 - Timer Interrupt Flag Register 1
*
*       PORTE - Input Port E
*
*       PORTB - Output Port B
*
*       TRANS - The memory location of the subroutine used
*               to transmit signals to the lego SmartBrick
*
*       RECORD - The memory location of the subroutine used
*                to record a user defined program for the rover
*
*       SIG - The number used in STATE to identify the transmission
*           state, where directional commands are passed directly through
*           to the rover.
```

```
*
*       REC - The number used in STATE to identify the record satet,
*           where the controller inputs directional commands and durations
*           into a table in memory.
*
*       PLAY - The number used in STATE to identify the play-back state,
*            where the controller will continually loop through the table
*           playing back the directional commands for the durations recorded.
*
*       INPUT - A memory location that stores a number to identify whether the
*               controller should expect a number or direction for its next input.
*               The only time this should change is when the controller is in the
*               recording state.
*
*       NUM - The number used in INPUT which tells the controller it should expect a
*           a numberical input from the key-pad.
*
*       DIR - The number used in INPUT which tells the controller it should expect a
*           directional input from the key-pad.
*
*       STATE - A memory location that stores a number to identify
*               which state the controller is currently in.
*
*       BBAD - A number representing that an improper directional keypress has been
entered
*
*       BFORWARD - The signal that will send the lego rover forward
*
*       BCCLOCK - The signal that will turn the rover counter clock wise
*
*       BSTOP - The signal that will stop the rover
*
*       BCLOCK - The signal that will turn the rover clockwise
*
*       BREVERSE - The signal that will make the rover go backwards
*****************************************************
NEXT        EQU   $0
TABLE       EQU   $D300
TMSK1       EQU   $1022
TCTL1       EQU   $1020
TCTL2       EQU   $1021
TFLG1       EQU   $1023
PORTE       EQU   $100A
PORTB       EQU   $1004
SIG         EQU   $1
REC         EQU   $2
PLAY        EQU   $3
INPUT       EQU   $20
NUM         EQU   $1
```

```
DIR             EQU   $0
STATE           EQU   $10
BBAD            EQU   $0
BFORWARD        EQU   $6
BCCLOCK         EQU   $9
BSTOP           EQU   $B
BCLOCK          EQU   $A
BREVERSE        EQU   $7
*****************************************************
*       Modify the interrupt vector table.
*
*       Use my code at $D100 for the IC1 interrupt response.
*****************************************************
        ORG $00E8
        JMP $D100
*****************************************************
*       The main section of code starts at $D000.
*****************************************************
        ORG $D000
*****************************************************
*       INITIAL SETUP
*
*       Iniate the value in $0 - $2 (the NEXT variable) to the
*       beginning of the TABLE used to store user programs ($D300).
*
*       Set the first value in the TABLE to null ($0) to signify that the
*       table is empty.
*
*       Put the controller in the transmit state.
*
*       Set the expected input to a direction.
*
*       Set the LegoBrick sync bit on PORTB to high, the idle state.
*****************************************************
        LDY #TABLE
        LDAA #$0
        STAA 0,Y      * Initiate the table
        STY NEXT      * Initiate the table pointer
        LDAA #SIG
        STAA STATE* Initiate transmit state
        LDAA #DIR
        STAA INPUT * Set expected input to directional
        LDAA #$10

        STAA PORTB          * Initiate the sync bit on port B
*****************************************************
*       Set up the interrupts
*
*       TMSK1 - 00000100
```

```
*       OC1I = 0, Output compare interrupt 1 is disabled
*       OC2I = 0, Output compare interrupt 2 is disabled
*       OC3I = 0, Output compare interrupt 3 is disabled
*       OC4I = 0, Output compare interrupt 4 is disabled
*       OC5I = 0, Output compare interrupt 5 is disabled
*       IC1I = 1, Input compare interrupt 1 is enabled
*       IC2I = 0, Input compare interrupt 2 is disabled
*       IC3I = 0, INput compare interrupt 3 is desabled
*
*       TCTL1 - 00000100
*       OM2,OL2 = 00, OC2 doesn't effect the pin
*       OM3,OL3 = 00, OC3 doesn't effect the pin
*       OM4,OL4 = 01, Toggle the OC4 pin n Successful compare
*       OM5,OL5 = 01, OC5 doesn't effect the pin
*
*       TCTL2 - 00010000
*       0, unused
*       0, unused
*       EDG1B,EDG1A = 01, IC1 - capture rising edges only
*       EDG2B,EDG2A = 00, IC2 - capture disabled
*       EDG3B,EDG3A = 00, IC3 - capture disabled
************************************************************
        LDAA #$04    * Configure the first Timer Interupt Mask Register.
        STAA TMSK1
        LDAA #$04    * Configure the first Timer Control Register.
        STAA TCTL1
        LDAA #$10    * Configure the second Timer Control Register.
        STAA TCTL2
************************************************************
*       Enable Interrupts
************************************************************
        CLI
************************************************************
*       THE MAIN LOOP
*
*       The main program loop impliments the controllers play-back
*       functionality.  It busy waits till the STATE variable designates
*       that the controller is in play-back mode.  Then it will grab the
*       section of the TABLE designated by NEXT.
*
*       First it checks if the entry is null ($0) designating that the end of the
*       TABLE has been reached.  If it is null then the NEXT pointer is reset
*       to the begining of the table and it continues looping through the TABLE.
*
*       Then it checks if the duration of the input signal is null ($0).  If it is
*       that signal is skipped and the main loop looks at the next signal that is
*       in the table.
*
*       Lastly the loop will put the recorded signal out to PORTB and then it
```

```
*       will busy wait for the number of seconds defined by the user in Record mode
*       before starting the loop over again.
*************************************************************
STAY:       LDAA INPUT
        LDY #PORTB          * Load Y to perform BSET or BCLR
        CMPA #NUM* Check if we are expecting NUM
        BNE EDIR     * If not, we are expecting DIR
ENUM:  BSET  0,Y #$40     * Expecting NUM, set expecting bit on port B
        BRA EFIN
EDIR:  BCLR  0,Y #$40       * Expecting DIR, clear expecting bit
EFIN:  LDAA STATE           * Check if we should be in playback state
        CMPA #PLAY
        BEQ GO2             * If we are move out of busy waiting
        BRA STAY    * Loop untill it playback state is entered
*****************************************************************
PlayBack state
GO2:   LDX NEXT    * Get the pointer to the next entry
        LDAA 0,X     * Get the next entry
        CMPA #$0     * Check if the entry is null
        BNE GO3             * If not then continue outputting the program
        LDX #TABLE* Reset the pointer to the begining of the table
        STX NEXT
        BRA STAY    * Start the playback from the beginning
*****************************************************************
TABLE has a valid entry
GO3:   LDAA 0,X
        ANDA #$0F   * Load the outgoing signal

        LDAB #$F0    * Get the control bits for the outgoing signal
        ANDB PORTB
        ABA          * Compose the complete outgoing signal
        LDAB 0,X
        ANDB #$F0   * Get the duration of the signal

        LSRB         * Shift the number 4 times to the right
        LSRB         * to put the 1 byte number in the lower byte
        LSRB         * so it can be used to determine the number of seconds
        LSRB         * to busy wait
        CMPB #$0
        BNE SKIPME* If the time is zero then don't send the signal
        LDX NEXT    * Look at the next entry in the table
        INX
        STX NEXT
        BRA STAY    * Restart the playback loop.
************************************************************** send
the signal
SKIPME:     STAA PORTB          * Send the signal
        ANDA #%11101111  * Set the sync low
        STAA PORTB      * Send the sync signal
```

```
        LDX #$4FFF  * Number of times to repeat routine
wloop3:         ABA             * wait for a small time to simulate a keypress
        DEX             * this activates an event on the lego brick which will
        BNE wloop3  * cause the signal to register and be sent to the rover
        LDAA PORTB
        ORAA #%00010000  * Set the sync bit high
        STAA PORTB
************************************************************** Look at
the next entry
        LDX NEXT    * Look at the next part of the table
        INX
        STX NEXT
************************************************************** Busy wait for
delay time
check: CMPB #$0
        BEQ DONE
        LDAA STATE          * This checks to see if we
        CMPA #PLAY          * were stopped in the middle of playing a signal
        BNE DONE    * so that we don't wait for a very long signal to end
        PSHA            * Push all registers info onto the stack
        PSHB
        PSHX
        LDX #$EFF0  * Number of times to repeat routine
wloop4:         ABA             * These ABAs are useless and only provide a set amount of
        ABA             * delay, allowing the entire loop to waste just about a second
        ABA             * The defined delay is then called to loop and busy wait
        ABA             * for an intiger time period between 1 and 15 s
        ABA
        ABA
        ABA
        ABA
        ABA
        ABA
        ABA
        ABA
        ABA
        DEX
        BNE wloop4  * end of busy wait loop
        PULX            * Restore the registers information
        PULB
        PULA
        DECB
        BRA check     * For every second loop make sure the state hasn't changed
DONE:           JMP STAY
************************************************************
* XLATE - Directional input to IR compatible direction signal translation
* parameter - register A holds signal to be translated
* return - the translated signal is available in register A upon completion
* Old value of register A is clobbered
* Returns null if the value isn't a direction.
```

```
*
* Example, you can read a signal in from port E and check if it was valid
*       LDAA #$0F    * Load the mask for the key-pad signal
*       ANDA PORTE       * Retrieve the signal
*       JSR XLATE
*       CMPA #$0
*       BEQ ERROR
*   *do whatever you want with the signal
************************************************************
XLATE:      PSHB                * Save our registers information
        PSHX
        LDX #XLTAB              * Load the base of the table
        TAB
        ABX             * Add the signal value as an index into table
        LDAA 0,X        * Load the translated value
        PULX            * Restore the registers
        PULB
        RTS
XLTAB:      FCB BBAD            * The XLTAB stores a translation mapping for
        FCB BBAD        * keypresses when we are expecting a direction.
        FCB BFORWARD            * The idea is to use the current key press to
        FCB BBAD        * detrmine what the output signal should be.
        FCB BCCLOCK
        FCB BSTOP
        FCB BCLOCK
        FCB BBAD
        FCB BREVERSE
        FCB BBAD
        FCB BBAD
        FCB BBAD
        FCB BBAD
        FCB BBAD
        FCB BBAD
        FCB BBAD
************************************************************
*       Main Iterrupt response to Input Capture Interrupt:
*
*       A signal has been detected from the key-pad and it needs
*       to be interpreted and the proper action taken.  This
*       routine is responsible for passing control to either a
*       sub-routine that will transmit the signal out or a
*       sub-routine that will record the users input in a table.
*       A single bit supplied by a Finite state machine will
*       dictate which sub-routine to run.
************************************************************
        ORG $D100
************************************************************
        LDY #$1000   * Clear the IC flag
        BSET TFLG1,y $FB
```

```
        LDAA PORTB          * Clear the error LED
        ANDA #%11011111
        STAA PORTB
CONT:       LDAA PORTE
        ANDA #$0F
        LDAB STATE          * Check if we are in Recording state
        CMPB #REC
        BNE HERE
        LDAB INPUT      * If the controller is expectin a number input then no
        CMPB #NUM * error checking or state change is desired, else we
        BNE HERE    * determine the new state based on the keypress and check

        JMP RECORD          * for a correctly formed directional input.
************************************************** input A
HERE:CMPA #$A    * Check if the start recording button was pressed
        BNE AHEAD1
        LDAA STATE          * Check if currently in recording state
        CMPA #REC
        BNE ON1
        JMP ERROR  * If so raise error and ignore the input
ON1:  CMPA #PLAY          * Check if currently in playback state
        BNE ON2
        JMP ERROR  * If so raise error and ignore input
ON2:  LDAA #REC  * Switch the state from transmit to record
        STAA STATE
        LDX #TABLE* Reset the table pointer
        STX NEXT
        LDAA #DIR   * Reset the excpecting input variable
        STAA INPUT
        JMP END             * Done
************************************************** input C
AHEAD1:     CMPA #$C    * Check if the start playback button was pressed
        BNE AHEAD2
        LDAA STATE          * Check if currently in playback state
        CMPA #PLAY
        BNE ON3
        JMP ERROR  * If so then raise error and ignore input
ON3:  CMPA #REC  * Check if currently in recording state
        BNE ON4
        JMP ERROR  * If so raise error and ignore input
ON4:  LDAA #PLAY          * Switch to playback state and ignore input
        STAA STATE
        LDX #TABLE* Reset the table pointer
        STX NEXT
        LDAA #DIR   * Reset the excpecting input variable
        STAA INPUT
        JMP END
************************************************** input F
AHEAD2:     CMPA #$F    * Check if the stop button was pressed
```

```
        BNE AHEAD3
        LDAA STATE          * Check if in recording state
        CMPA #REC
        BNE ON5
        LDAA #$0     * If so change state to transmit, make last entry null, and end

        LDY NEXT
        STAA 0,Y     * null last entry in the table
        STAA $15
        LDX #TABLE* Reset the table pointer
        STX NEXT
        LDAA #SIG    * Change state to transmit
        STAA STATE
        LDAA #DIR    * Reset the excpecting input variable
        STAA INPUT
        JMP END
ON5:    CMPA #PLAY          * Check if in playback state
        BNE ON6
        LDAA #SIG    * If so change state to transmit
        STAA STATE
        LDX #TABLE* Reset the table pointer
        STX NEXT
        LDAA #DIR    * Reset the excpecting input variable
        STAA INPUT
        LDAA #BSTOP
        STAA PORTB          * Transmit stop signal
        ANDA #%11101111
        STAA PORTB
*************************************************************************
Delay loop
        LDX #$4FFF  * Number of times to repeat routine
wloop:  ABA             * Waste time
        DEX
        BNE wloop
*************************************************************************
        LDAA PORTB
        ORAA #%00010000
        STAA PORTB
        JMP END
ON6:    JMP ERROR  * Raise error and ignore input, because we are in trans. state

AHEAD3:     LDAB STATE          * Get the state
        CMPB #SIG
        BEQ TRANS  * Transmit state
        CMPB #REC
        BEQ RECORD          * Record State
*************************************************************************
Errors come here
ERROR:      LDAA PORTB
```

```
        ORAA #%00100000
        STAA PORTB
*       BSET PORTB $40    * Invalid command input, output error and ignore command
*****************************************************************
Finish the interrupt
END:  RTI             * The sub-routine has finished
*****************************************************************
        Transfer state
TRANS:      LDAA #$0F   * Load the mask for the key-pad signal
        ANDA PORTE        * Retrieve the signal

        JSR XLATE   * Use XLATE to translate the keypress
        CMPA #$0    * Check if a valid trasnlation is created.
        BNE AHD1    * If it is transmit it
        JMP ERROR   * Else raise an error and ignore it.
AHD1:LDAB #$F0    * Load the mask for the output control signals
        ANDB PORTB        * Retrieve the output control signals
        ABA           * Compose the output signal
        STAA PORTB        * Send the signal to the SmartBrick
        ANDA #%11101111
        STAA PORTB
*****************************************************************
Delay loop
        LDX #$4FFF  * Number of times to repeat routine
wloop: ABA            * Waste time
        DEX
        BNE wloop
*****************************************************************
        LDAA PORTB        * Reset the sync bit high
        ORAA #%00010000
        STAA PORTB
        JMP END           * Finnish up
*****************************************************************
RECORD
*****************************************************************
Directional input
RECORD:     LDAA INPUT* Determine the type of input that is expected
        CMPA #DIR
        BNE MOVE1 * A number is expected, no error checking to be done
*****************************************************************
        LDAA #$0F   * Load the mask for the key-pad signal
        ANDA PORTE        * Retrieve the signal

        JSR XLATE   * Use the XLATE function to translate the directional input
        CMPA #$0
        BNE AHD2    * If the translation creates a valid signal record it
        JMP ERROR   * Else raise and error and ignore the input
*****************************************************
AHD2:LDX NEXT    * Get the location of that is to be rcorded to.
```

```
        STAA 0,X      * Store the direction
        LDAA #NUM * Change the excpected input to number.
        STAA INPUT
        JMP END
************************************************** Number input
MOVE1:      LDX NEXT    * Get the location of the next record
        LDAA PORTE        * Get the number input from the user
        LSLA          * Shift the # left by 4
        LSLA
        LSLA
        LSLA
        LDAB 0,X      * Load the direction from memory
        ABA           * Combine the number and direction
        STAA 0,X      * Store the completed part of the table
        INX           * Point at the next entry in the table.
        STX NEXT
        LDAA #DIR   * Change the excpected input to direction.
        STAA INPUT
        JMP END             * Finnish up
************************************************************
*       Port B
*       MSB -
*       6 - Expected input
*       5 - Error bit
*       4 - the sync signal for the Lego SmartBrick
*       3,2,1,0 - the bits carying the signal to the SmartBrick
*
*       Port E
*       MSB -
*       6 -
*       5 -
*       4 -
*       3,2,1,0 - the signal from the FPGA (number or direction)
************************************************************
*********************************************************************
*       translate signals based on expected input
*
*       signal          command             translation
*       2               forward             6
*       8               reverse         7
*       4               counter clk     9
*       6               clk             10
*       5               stop            11
*********************************************************************
```

B) Verilog

freqdivide.v:

```
/*********************************************************\
|     freqdivide.v - Frequency division module             |
|  Provide frequency division through clock counting.      |
| Copyright(C) 2002 Stephen Friedman                       |
\*********************************************************/

module freqdivide(clk, reset, divf);
    input clk;
    input reset;
       output divf;

    /* The clk input will provide the pulses to count.
          A counter will count these pulses, and the difv
             output will be the MSB of the counter, effectively
             dividing the input by 2^(#of bits in counter).
             Reset is provided mostly for useful simulation,
             the counter doesn't really ever have to be in a
             known state.
       */

       /* We need to save these values, so reg them */
   reg [13:0] count;

       /* Directly connect the output to the MSB of the
          counter. */
      assign divf = count[13];

      /* Create a seqential counter that counts the
         clock pulses by incrementing the counter
      on each clock edge. On reset simply set the counter
            to 0 */
      always@(posedge clk or posedge reset)
        if(reset) count <= 0;
        else count <= count+1;
endmodule
```

poll_freq.v:

```
module poll_freq(clk, reset, divf);
    input clk;
    input reset;
       output divf;

    /* The clk input will provide the pulses to count.
          A counter will count these pulses, and the difv
             output will be the MSB of the counter, effectively
             dividing the input by 2^(#of bits in counter).
             Reset is provided mostly for useful simulation,
             the counter doesn't really ever have to be in a
             known state.
       */

       /* We need to save these values, so reg them */
   reg [12:0] count;

       /* Directly connect the output to the MSB of the
          counter. */
      assign divf = count[12];

      /* Create a seqential counter that counts the
         clock pulses by incrementing the counter
      on each clock edge. On reset simply set the counter
```

```
            to 0 */
       always@(posedge clk or posedge reset)
         if(reset) count <= 0;
         else count <= count+1;
endmodule
```

sevenseg.v:

```
module sevenseg(s,seg);
    input [3:0] s;
    output [6:0] seg;
       wire [6:0] seginv;

            /* seginv[6] to seginv[0] represent outputs to control
               7-segment LED's A to G respectively
          seg[6]
                 --------
                 |      |
            seg[1]|      | seg[5]
                 |seg[0]|
                 --------
                 |      |
            seg[2]|      | seg[4]
                 |      |
                 --------
          seg[3]


                 The truth table for the outputs is implemented
              in sum of product form to allow the synthesis
                 tools to optimize the design.
            */
            assign seginv[6] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] &  s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] &  s[0]|
                                    ~s[3] &  s[2] &  s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] &  s[0];
            assign seginv[5] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] & ~s[2] & ~s[1] &  s[0]|
                                    ~s[3] & ~s[2] &  s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] &  s[0]|
                                    ~s[3] &  s[2] & ~s[1] & ~s[0]|
                                    ~s[3] &  s[2] &  s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|
                                     s[3] &  s[2] & ~s[1] &  s[0];
            assign seginv[4] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] & ~s[2] & ~s[1] &  s[0]|
                                    ~s[3] & ~s[2] &  s[1] &  s[0]|
                                    ~s[3] &  s[2] & ~s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] &  s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] &  s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] &  s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] &  s[0];
```

```
            assign seginv[3] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] &  s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] &  s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] & ~s[0]|
                                     s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] &  s[2] &  s[1] & ~s[0];
            assign seginv[2] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|
                                     s[3] & ~s[2] &  s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] & ~s[0]|
                                     s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] &  s[2] &  s[1] &  s[0];
            assign seginv[1] =      ~s[3] & ~s[2] & ~s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] & ~s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|

                                     s[3] & ~s[2] &  s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] &  s[0];
            assign seginv[0] =      ~s[3] & ~s[2] &  s[1] & ~s[0]|
                                    ~s[3] & ~s[2] &  s[1] &  s[0]|
                                    ~s[3] &  s[2] & ~s[1] & ~s[0]|
                                    ~s[3] &  s[2] &  s[1] & ~s[0]|
                                    ~s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] & ~s[1] & ~s[0]|
                                     s[3] & ~s[2] &  s[1] & ~s[0]|
                                     s[3] & ~s[2] & ~s[1] &  s[0]|
                                     s[3] & ~s[2] &  s[1] &  s[0]|
                                     s[3] &  s[2] & ~s[1] &  s[0]|
                                     s[3] &  s[2] &  s[1] & ~s[0]|
                                     s[3] &  s[2] &  s[1] &  s[0];

        assign seg[0] = ~seginv[0];
        assign seg[1] = ~seginv[1];
        assign seg[2] = ~seginv[2];
        assign seg[3] = ~seginv[3];
        assign seg[4] = ~seginv[4];
        assign seg[5] = ~seginv[5];
        assign seg[6] = ~seginv[6];

endmodule
```

muxdisplay.v:

```
/*********************************************************\
|     muxdisplay.v - Sum and display multiplexing module.   |
|  Use multiplexing to display two numbers on two 7-segment |
|  displays using only 1 decoder, and display their sum in  |
|  binary on the FPGA board's on board LED's.               |
| Copyright(C) 2002 Stephen Friedman                        |
\*********************************************************/
module muxdisplay(s1, s2, reset, clk, seg, enLled, enRled);
```

```
    input [3:0] s1;
    input [3:0] s2;
    input reset;
    input clk;
       output [6:0] seg;
    output enLled;
       output enRled;

    wire [3:0] sdisp;
       wire divf;



      /* Use a four bit mux with the select input tied
         to the signal that controls which display is
            on, so that the corresponding number will be
            read for that display. */
      mux_4 inmux(s1, s2, divf, sdisp);

      /* Instantiate a 7-segment decoder that is attatched to
         the 4b mux to display the numbers.*/
      sevenseg digit(sdisp, seg);

      /* Divide our clock frequency to slow it
         down so to a usable frequency for multiplexing
            the display. */
   freqdivide fdiv(clk, reset, divf);


      /* Use continuous logic to make the divided
         clock signal multiplex the two displays. */
      assign enLled = divf;
   assign enRled = ~divf;

endmodule
```

keypoll_fsm.v:

```
/*********************************************************\
|keypoll_fsm.v - FSM to poll a 4x4 matrix keypad, output    |
|  the keycode, and output a sync signal on key reception.  |
| Copyright(C) 2002 Stephen Friedman                        |
\*********************************************************/

module keypoll_fsm(clk,reset,krow,kcol,scanned_num,sync);
    input clk;
    input reset;

       /* Rows and columns of the 4x4 matrix keypad */
    input [3:0] krow;
    output [3:0] kcol;

       /* Output of scanned key press */
    output [3:0] scanned_num;

       /* Sync to signal when keypress is ready to be read */
    output sync;

      /* I/O patterns for reading specific matrix lines */
      parameter POLL1 = 4'b1110;
      parameter POLL2 = 4'b1101;
      parameter POLL3 = 4'b1011;
      parameter POLL4 = 4'b0111;

      /* Output register for driving matrix columns */
      reg [3:0] kcol;

      /* One bit flag to keep track of whether or not
         we are currently polling the keypad */
      reg polling;

      /* Sync signal that is raised for one clock cycle
         when a new keypress is detected and decoded */
      reg sync;

      /* Wire to let us know if a key has been pressed */
      wire pressed;

      /* This decodes the row and column pattern to a
         4-bit binary number */
      keydecode keypad1(krow, kcol, scanned_num);

      /* Detects when any row is pulled low by a keypress */
      assign pressed = ~(&krow);

      /* FSM with async reset driven by clk */
      always@(posedge clk or posedge reset)
      /* If reset, change to the waiting state and reset
         the polling pattern. */
      if(reset)
            begin
             sync <= 0;
             polling <= 1;
             kcol <= POLL1;
            end
       else
            /* If a keypress is detected and we are polling,
                  there is a new keypress ready on the decoder
                  output, so raise sync and stop polling */
            if (pressed)
               if(polling)
```

```
                        begin
                              sync <= 1;
                              polling <= 0;
                        end
                  /* We aren't polling and a key is still pressed
                     so we stay in the waiting state, and ensure
                        that our sync is low because the decoder output
                        is now considered old */
                  else // pressed and ~polling
                   begin
                        sync <= 0;
                        polling <= 0;
                    end
            /* We are in the polling state and nothing is currently
               detected on the keypad, so remain polling and cycle
                  to the next output pattern to check the next column.*/
            else
              if(polling) //~pressed and polling
                  begin
                    sync <= 0;
                     polling <= 1;
                     case(kcol)
                        POLL1: kcol <= POLL2;
                        POLL2: kcol <= POLL3;
                        POLL3: kcol <= POLL4;
                        POLL4: kcol <= POLL1;
                        default: kcol <= POLL1;
                      endcase
                  end
            /* The key press was released, so we
                  can return to the polling state from
                  the waiting state. */
            else // ~pressed and ~polling
                  begin
                        sync <= 0;
                        polling <= 1;
                  end
endmodule
```

keydecode.v:

```
/*********************************************************\
|keydecode.v - Maps the keypad matrix locations using one   |
|  hot encoding to their binary counterparts.               |
| Copyright(C) 2002 Stephen Friedman                        |
\*********************************************************/


module keydecode(krow,kcol,y);
    input [3:0] krow;  /* Row inputs of matrix */
    input [3:0] kcol;  /* Column outputs of matrix */
    output [3:0] y;       /* Binary output */

       reg [3:0] y;
            /* One-hot encoding of matrix locations */
            parameter POLL1 = 4'b1110;
            parameter POLL2 = 4'b1101;
            parameter POLL3 = 4'b1011;
            parameter POLL4 = 4'b0111;

            /* Binary output equivalents */
            parameter ZERO    = 4'b0000;
            parameter ONE           = 4'b0001;
            parameter TWO           = 4'b0010;
            parameter THREE   = 4'b0011;
            parameter FOUR    = 4'b0100;
```

```
            parameter FIVE    = 4'b0101;
            parameter SIX           = 4'b0110;
            parameter SEVEN   = 4'b0111;
            parameter EIGHT   = 4'b1000;
            parameter NINE    = 4'b1001;
            parameter HEXA    = 4'b1010;
            parameter HEXB          = 4'b1011;
            parameter HEXC          = 4'b1100;
            parameter HEXD          = 4'b1101;
            parameter HEXE    = 4'b1110;
            parameter HEXF    = 4'b1111;

      /*Map matrix locations to binary values according to following
diagram:

                  col
            row   1 2 3 4
                  4  1 2 3 C
                  3  4 5 6 D
                  2  7 8 9 E
                  1  A 0 B F
       */
      always@(krow or kcol)
            case({krow,kcol})
                  {POLL1,POLL1}: y <= HEXA;
                  {POLL1,POLL2}: y <= ZERO;
                  {POLL1,POLL3}: y <= HEXB;
                  {POLL1,POLL4}: y <= HEXF;
                  {POLL2,POLL1}: y <= SEVEN;
                  {POLL2,POLL2}: y <= EIGHT;
                  {POLL2,POLL3}: y <= NINE;
                  {POLL2,POLL4}: y <= HEXE;
                  {POLL3,POLL1}: y <= FOUR;
                  {POLL3,POLL2}: y <= FIVE;
                  {POLL3,POLL3}: y <= SIX;
                  {POLL3,POLL4}: y <= HEXD;
                  {POLL4,POLL1}: y <= ONE;
                  {POLL4,POLL2}: y <= TWO;
                  {POLL4,POLL3}: y <= THREE;
                  {POLL4,POLL4}: y <= HEXC;
                  default: y <= ZERO;
            endcase
endmodule
```

C) Lego RCX code

Microps Rover.lsc:

```
/*
/0LOGBOOK
0

*/
program test {

        #include <RCX2.h>
        #include <RCX2MLT.h>
        #include <RCX2Sounds.h>
        #include <RCX2Def.h>
        var msg = 0
        event range_messageEventRange when message is 1..6


        main {
                ext InterfaceType “kFreestyle”
                rcx_ClearTimers
                bbs_GlobalReset([A B C])
                try {
                        power [ A  ] 8
                        power [ C  ] 4
                        direction [ A B C  ] [ ]
                } retry on fail
                try {
                        forever {
                                repeat {
                                        float [ A B C  ]
                                        if msg = 10{
                                                direction [ A  ] [ ]
                                                on [ A  ]
                                                off [ C  ]
                                        }
                                        else
                                        {
                                        }
                                        if msg = 20{
                                                direction [  ] [ A ]
                                                on [ A  ]
                                                off [ C  ]
                                        }
                                        else
                                        {
                                        }
```

```
                                        if msg = 30{
                                                direction [ A  ] [ C ]
                                                on [ A C  ]
                                        }
                                        else
                                        {
                                        }
                                        if msg = 40{
                                                direction [ C  ] [ A ]
                                                on [ A C  ]
                                        }
                                        else
                                        {
                                        }
                                        if msg = 50{
                                                off [ A B C  ]
                                        }
                                        else
                                        {
                                        }
                                } until range_messageEventRange
                                msg = (message*10)
                                clear message
                                display msg:1
                        }
                } retry on fail
        }


}
```

mp controller.lsc:

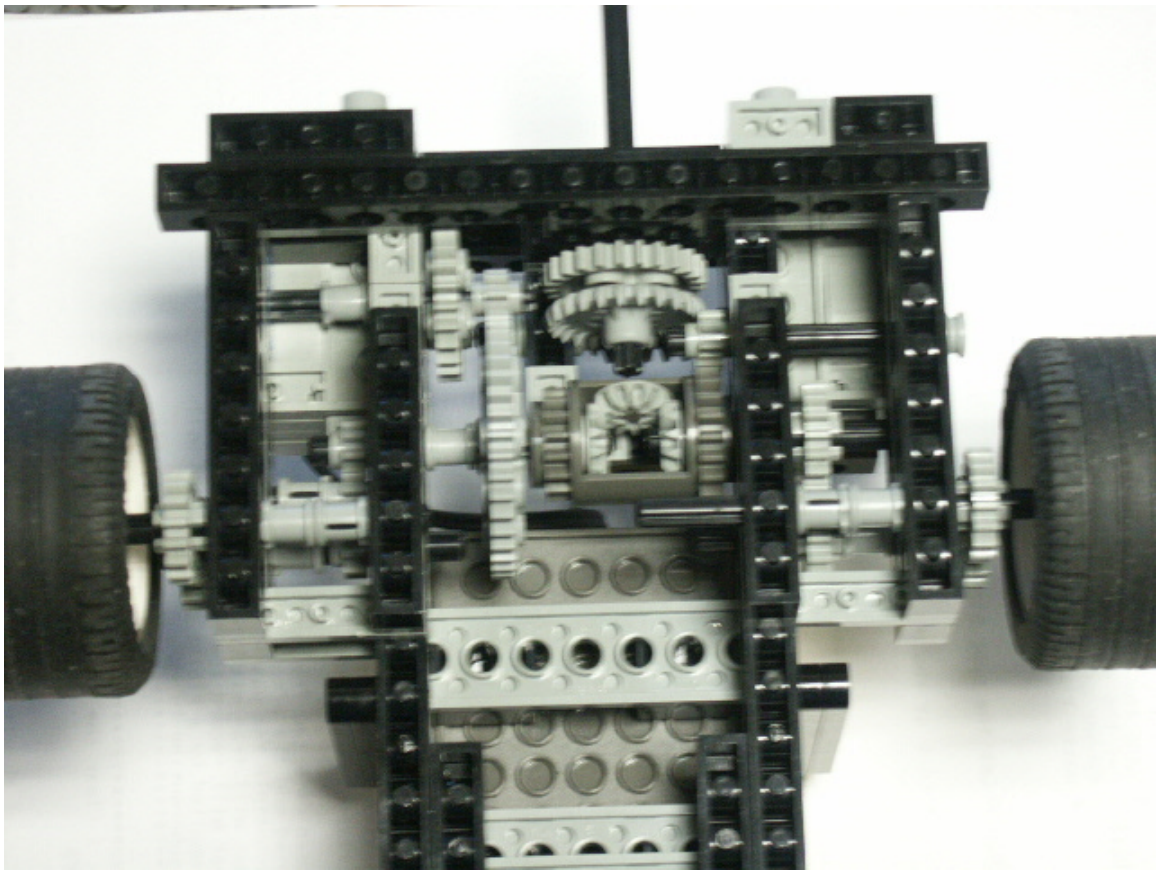
```
program test {

        #include <RCX2.h>
        #include <RCX2MLT.h>
        #include <RCX2Sounds.h>
        #include <RCX2Def.h>
        var bit1 = 0
        var bet2 = 0
        var bit2 = 0
        sensor temperature2 on 2
        temperature2 is temperature as fahrenheit
        event tRange_temperature2EventRange when temperature2 is -40..450

        event tRange_temperature2EventRange0 when temperature2 is 500..1300
```

```
event tRange_temperature2EventRange1 when temperature2 is 1330..1580

sensor temperature3 on 3
temperature3 is temperature as fahrenheit
event tRange_temperature3EventRange when temperature3 is -40..490

event tRange_temperature3EventRange0 when temperature3 is 500..1300

event tRange_temperature3EventRange1 when temperature3 is 1350..1580

sensor touch1 on 1
touch1 is switch as boolean
event tClick_touch1EventClick when touch1.click



main {
	ext InterfaceType "kFreestyle"
	rcx_ClearTimers
	bbs_GlobalReset([A B C])
	trigger tClick_touch1EventClick
	start TemperatureWatcher0
	start TemperatureWatcher1
	start TemperatureWatcher2
	start TemperatureWatcher3
	start TemperatureWatcher4
	start TemperatureWatcher5
	rcx_Priority( 8)
	trigger tRange_temperature2EventRange
	trigger tRange_temperature2EventRange0
	trigger tRange_temperature2EventRange1
	trigger tRange_temperature3EventRange
	trigger tRange_temperature3EventRange0
	trigger tRange_temperature3EventRange1
	try {
		forever {
			wait until tClick_touch1EventClick
			counter1 = bit1
			counter1 += bit2
			send (counter1/10)
		}
	} retry on fail
}


watcher TemperatureWatcher0 monitor tRange_temperature2EventRange
{
	rcx_Priority( 5 )
```

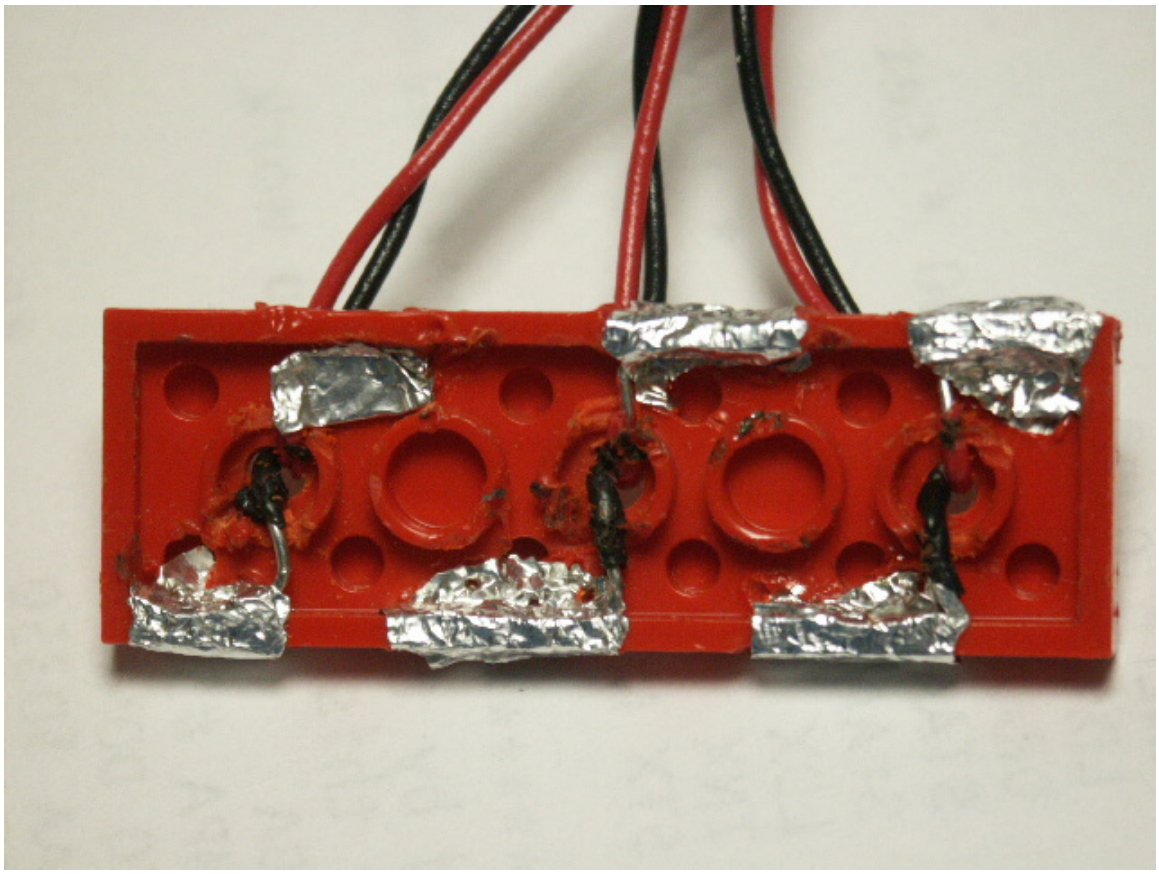
```
		try {
			bit1 = 20
		} restart on fail
	} restart on event
	watcher TemperatureWatcher1 monitor tRange_temperature2EventRange0
	{
		rcx_Priority( 5 )
		try {
			bit1 = 10
		} restart on fail
	} restart on event
	watcher TemperatureWatcher2 monitor tRange_temperature2EventRange1
	{
		rcx_Priority( 5 )
		try {
			bit1 = 0
		} restart on fail
	} restart on event
	watcher TemperatureWatcher3 monitor tRange_temperature3EventRange
	{
		rcx_Priority( 5 )
		try {
			bit2 = 60
		} restart on fail
	} restart on event
	watcher TemperatureWatcher4 monitor tRange_temperature3EventRange0
	{
		rcx_Priority( 5 )
		try {
			bit2 = 30
		} restart on fail
	} restart on event
	watcher TemperatureWatcher5 monitor tRange_temperature3EventRange1
	{
		rcx_Priority( 5 )
		try {
			bit2 = 0
		} restart on fail
	} restart on event
}
```

D) Lego Implementation Photos

Picture of the Gearing on the Rover



Picture of the Custom Connection piece used on the RCX



Picture of the connection betweent he RCX controller and the Rover