Nicole Yu and Joshua Vasquez
December 10, 2012
E155 MicroProcessors

E155 Final Project: The Claw Crane Motion System

Introduction

For quite some time, we've been dazzled by classic arcade claw cranes; they're a fantastic digital system. Of course, most former contestants probably recall almost *never* being able to claim a prize from the machine--dubbing it rigged. With this unfairness in mind, we've decided to construct a similar replica to these claw cranes that duplicate the fundamental features of the motion system. However, unlike most other claw cranes, we hope to offer our users more enjoyment by giving them more control of the claw. Specifically, users can move the claw in two horizontal dimensions (x and y axes), vertical dimension (z-axis), and grabbing control. In most classic arcade games, the user can only control the x and y axes, but, with these design changes, we hope to grant users a much more relaxed experience manipulating objects on the floor.

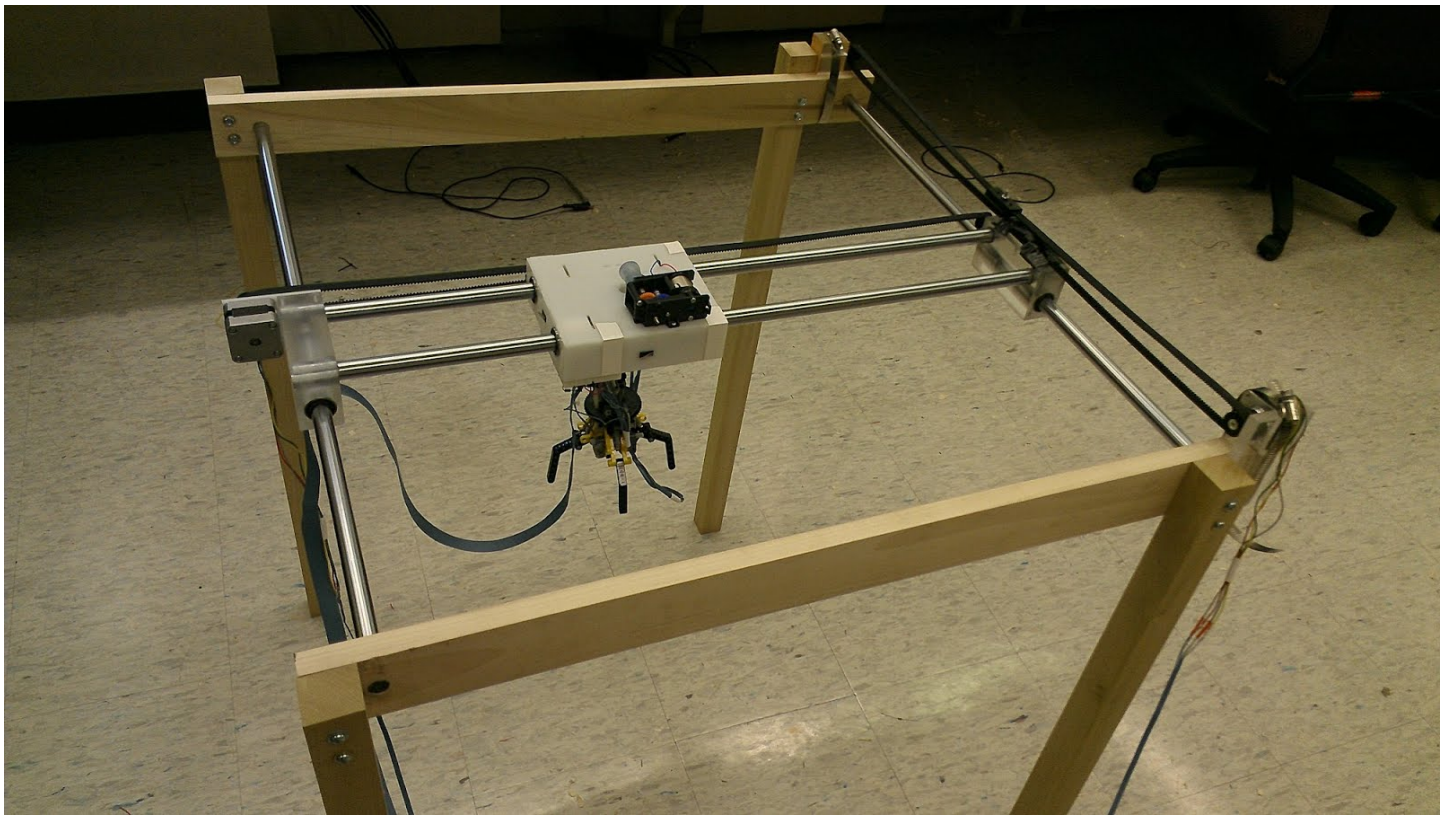

Figure 1: Completed arcade game system

Electronic Hardware Components

The electronics needed to drive this project is a combination of motors, analog sensors, and digital sensors. With this combination, we can both move the claw from our user-input *and* prevent the claw system from damaging itself with the analog and digital input. Specifically, our project implements:

- two stepper motors
- two brushed DC motors
- one analog Hall effect sensor
- five small push-button switches
- one joystick (consisting of 4 push-button switches)
- four user-input push-buttons (to drive *up*, *down*, *open*, *close* capabilities)

Both the sensing and driving of the system is distributed between the FPGA and the PIC32 microcontroller.

The overall high-level schematic is as follows:

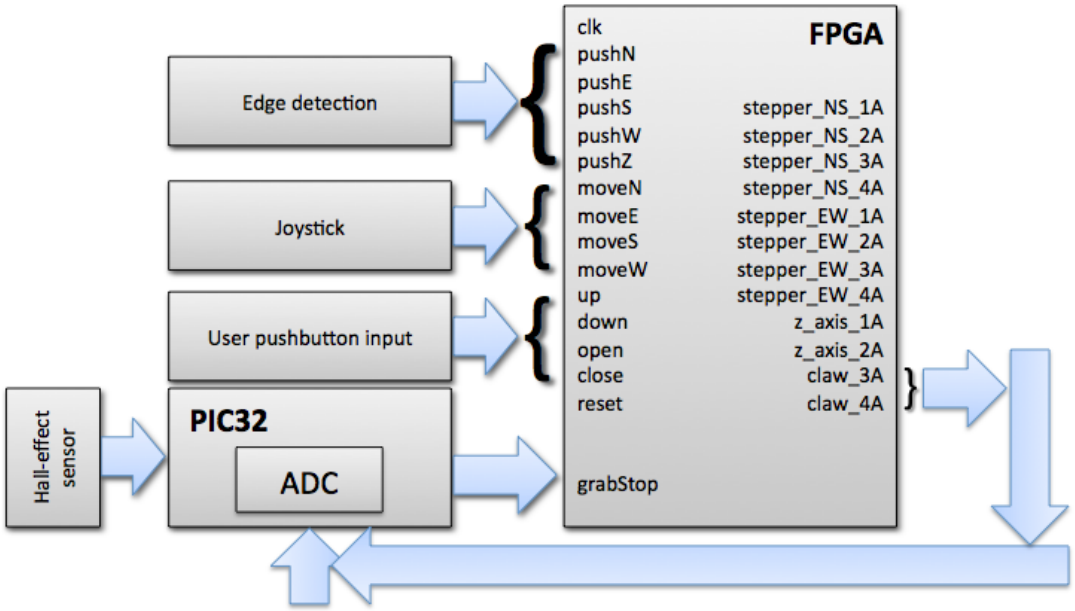

Figure 2: Functional block diagram

FPGA Tasks

As revealed in the diagram above (Fig. 2), the FPGA drives the majority of the system, controlling both stepper motors, both DC motors and accepting all switches as input.

PIC32 Tasks

Despite the FPGA's hefty role, however, the PIC32 plays a critical role in the system by detecting whether or not the claw grabber is *stalled*. It can then alert the FPGA (driving the claw motor) whether or not the system has been stalled in the form of binary digital input (*jammed* or *~jammed*).

Synthesized Hardware Implementation (FPGA) and Software Algorithm (PIC32)

Below, we've detailed the theory behind our circuitry and how we specifically implemented the hardware on both the FPGA and the PIC32.

FPGA Component

Switch Inputs

The switches are placed in five strategic locations that define the edges of the axis of the system. When any one of these switches is depressed, the gantry can no longer continue forward in that direction, though it can still move in any of the remaining directions. The switches are placed in these locations:

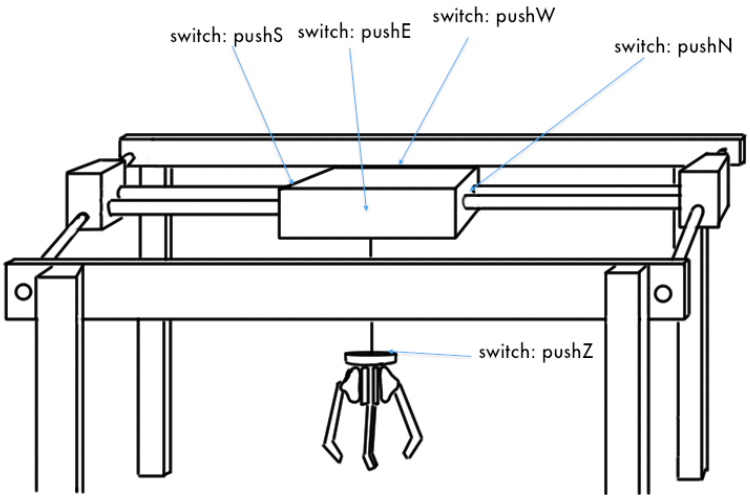

Figure 3: Diagram of arcade claw game setup

The switches were circuited as follows where a pull-down resistor holds the output as default low. When the switch is activated, the output turns high.

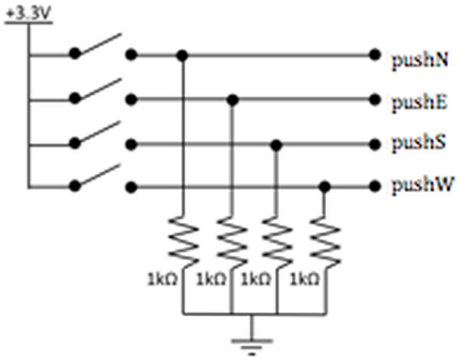

Figure 4: Schematic of switch circuitry.

This circuitry was implemented for the joystick and user-input push-buttons (*up*, *down*, *open*, *close*).

Finite State Machine

The finite state machine for this system is defined by six states:

- MOVE: The user moves freely in the x-y plane using the joystick.
- DOWN: The user moves freely in the z-axis using the *up* and *down* push-buttons.
- GRAB: The user grabs the object using the *open* and *close* push-buttons.
- RETRACT: The system auto-retracts the claw up the z-axis to the x-y plane.
- ALIGN: The system auto-aligns the claw to (0, 0) in the x-y plane.
- RELEASE: The system opens the claw to drop the prize at (0, 0).

In different states, certain movement is restricted to simulate the classic arcade game. For example, in the DOWN state the user cannot move in the x-y plane despite joystick input. Though the movement is restricted in certain states, it still allows for the user to have more control than the classic arcade game. Normally, the user moves the claw in the x-y plane, then the system automatically attempts to retrieve the object. In our version of the game, the user can move both directions the z-axis in the DOWN state, and in the GRAB state the user can control the open and close of the claw.

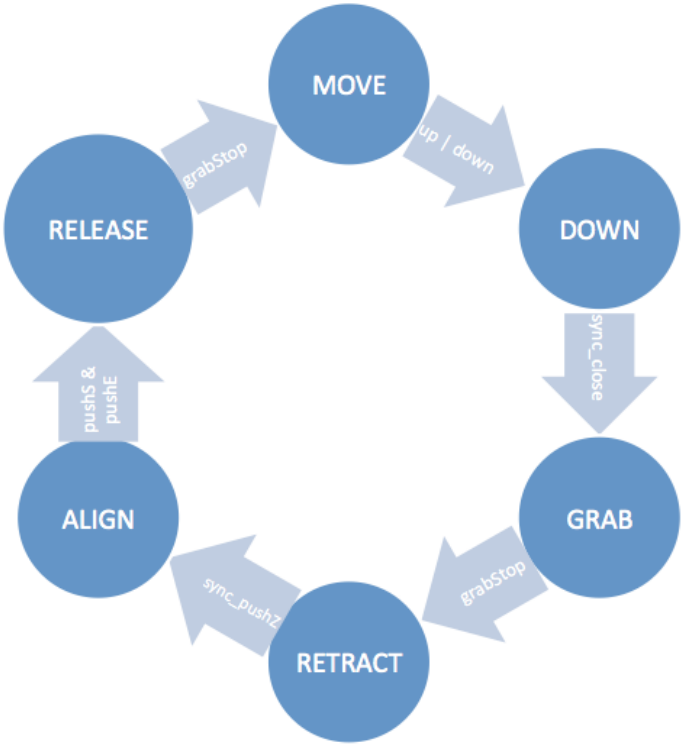

Figure 5: Finite state machine sequence and logic iteration

Based on the current state and input logic, the FPGA uses combinational logic to determine the next state. The state transitions to the next state at the 40 [MHz] clock edge. Again, using the current state and input logic, the FPGA allows or restricts certain movements such as those described previously.

Motor Control Theory - DC Motor

Both the two stepper motors and two brushed DC motors can be controlled through a configuration of H-bridges. In general, a motor cannot directly be plugged into either chip and driven directly from the chip I/O pins. Rather, because it consumes far more current than the I/O pins can source and sink, motors must be driven with an H-bridge configuration. One stepper motor must be driven with two H-bridges while one DC motor can be driven with a single H-bridge.

An H-bridge is simply a specific arrangement of four transistors configured to act like switches as shown below:

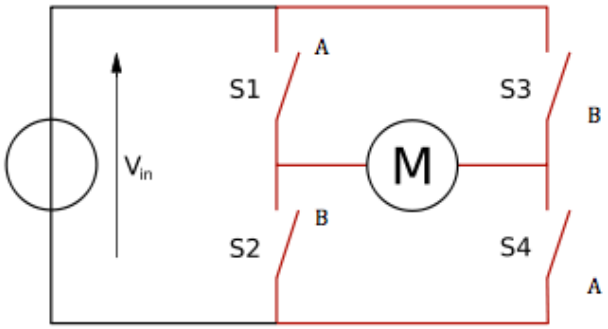

Figure 6: Single H-bridge controlling a brushed DC motor

To drive a DC brushed motor in one direction, we would *open* switches S2 and S3 and we would *close* switches S1 and S4. The full truth table is as follows:

S1	S2	S3	S4	OUTPUT
CLOSED	OPEN	OPEN	CLOSED	FORWARD*
OPEN	CLOSED	CLOSED	OPEN	REVERSE*
OPEN	OPEN	OPEN	OPEN	COAST
CLOSED	OPEN	CLOSED	OPEN	BRAKE

*Note that the choice of FORWARD and REVERSE is aribitrary.

Table 1: Truth table for output states based on transistor states.

Other configurations of S1 through S4 are unnecessary, and some can lead to short-circuits .

Fortunately this entire arrangement of H-bridges is nicely packed into an *H-Bridge IC chip*. A ready-to-go H-bridge has two control inputs, which greatly simplifies the above logic. Each control input pin simultaneously drives a pair of transistors. Referring to Fig. 6, input A drives the pair S1 and S4 and input B drives the pair S2 and S3. In our case, we implemented the SN75441, which contains two H-bridges per chip. This enables us to drive the DC motor forward, reverse, or off by setting (A, B) as combinations of highs and lows. The dual h-bridge schematic for both brushed dc motors is as follows:

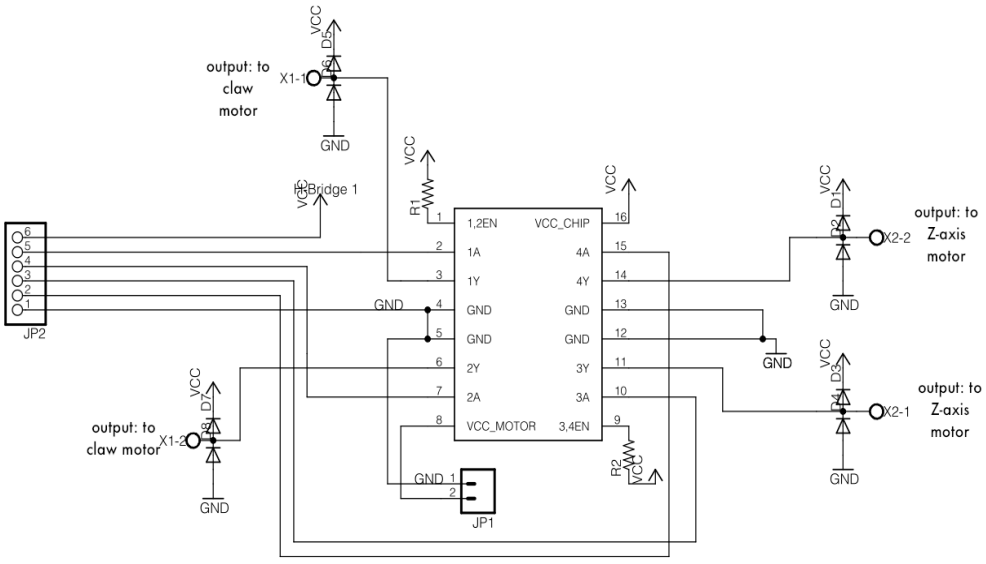

Figure 7: H-bridge schematic for DC motor control

Motor Control Theory - Stepper Motor

Controlling a Bipolar Stepper Motor requires two H-bridges and slightly different logic, however. In this case, each H-bridge controls one of the two coils inside of the bipolar stepper motor.

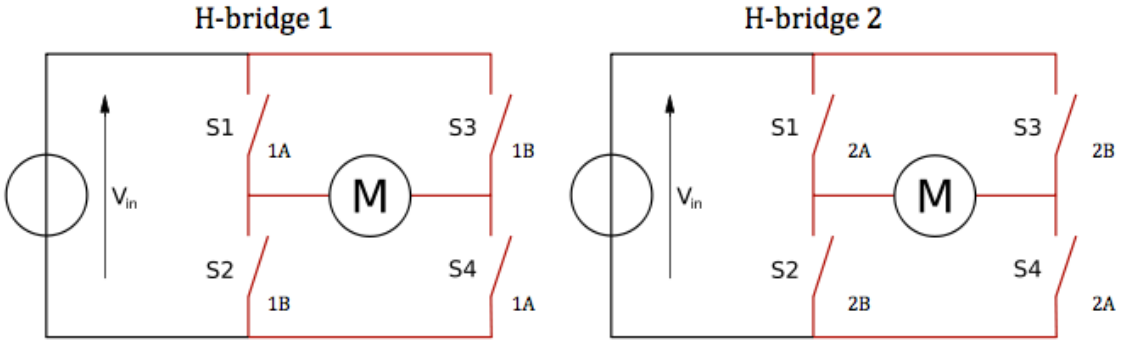

Figure 8: Double H-bridge controlling a stepper motor

Furthermore, to drive a stepper motor, switches can't simply be turned ON and OFF, as with a brushed DC motor before. Rather, the switches must be pulsed to drive the H-bridge through a specific pattern, or control sequence, which translates into continuous movement on the motor shaft. To drive the stepper motors in either direction is a simple matter of reversing the pattern that is being sent to the enable pins of the H-bridge.

The control sequence for driving the stepper motor is listed below. To simplify the diagram, we've listed the control sequence in terms of the pins (A, B) on the H-bridges (1, 2), but recall that *each pin on an H-bridge drives two switches to direct current flow.*

Step 1	Step 2	Step 3	Step 4
1A: 1 1B: 0 2A: 0 2B: 1	1A: 1 1B: 0 2A: 1 2B: 0	1A: 0 1B: 1 2A: 1 2B: 0	1A: 0 1B: 1 2A: 0 2B: 1

* To reverse motor direction, simply reverse the sequence.

Table 2: Four-phase sequence to drive a stepper motor one direction.*

In order to utilize the maximum torque rating of the stepper motor, we implemented the full step drive sequence. If we look at this drive sequence as a waveform, the control for a stepper motor is a set of four cascading square waves at the same frequency and offset by half its period:

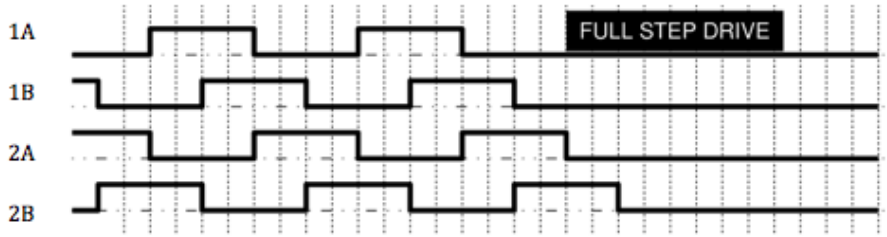

Figure 9: Waveform of the full step drive phase sequence. Each phase corresponds to an input pin in Fig. 8, and the square wave oscillates switches high and low.

Changing the frequency of these square waves alters the speed that the stepper motor drives. To achieve cascading square waves, we executed logic on slower clock cycles defined by the `counter` variable that increments at every 40 [MHz] clock edge. The schematic for both stepper motors is as follows:

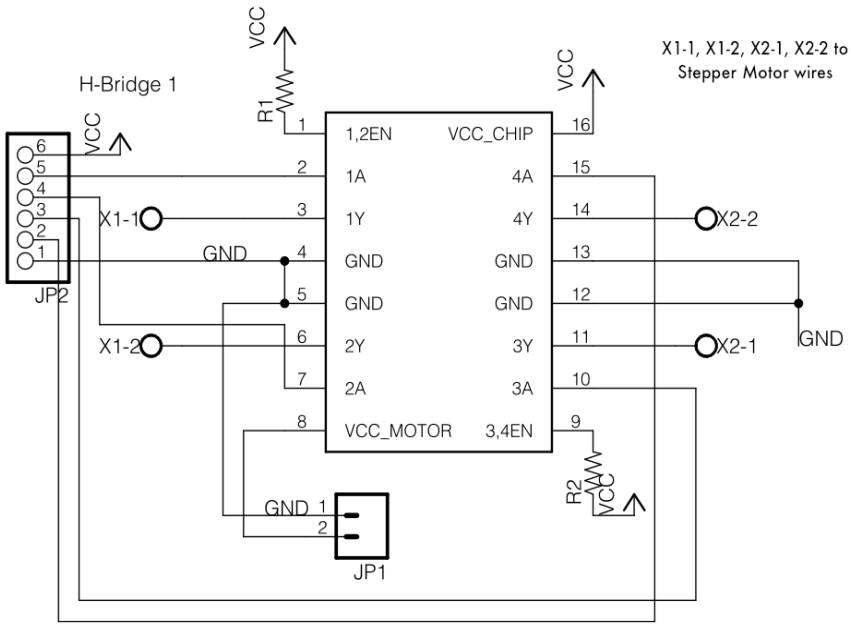

Figure 10: H-bridge schematic for stepper motor control

Edge Detection

To protect against driving the gantry and claw continuously into the wall of the housing, the claw crane features several switches that will deactivate specific motors if pressed. There are five of these switches located on the edges of the moving part of the gantry. Two switches lay opposite each other on opposing sides of the platform and inhibit movement in the "north and south" directions. Two more switches on the other opposing sides of the platform ("east and west") are placed to avoid driving beyond the dimensions of the housing.

PIC32 Component

The PIC32's task is to monitor the claw and output a digital high signal if the the claw is jammed while the FPGA is trying to drive it. To fulfill this task, we took advantage of two digital inputs, one analog input, and one digital output. To gauge whether the claw is jammed, we attached a Hall effect sensor to the sensor of the motor shaft to detect the shafts rotations.

Hall-Effect Sensor Behavior

Hall effect sensors respond to magnetic field intensity. In our case, we have a linear Hall effect sensor, the SS495a, which responds by linearly varying its output voltage based on the magnetic field intensity. We were able to swing the voltage output of the sensor within a range detectable by the PIC32(about 2.0 to 3.3 [V]) when we held it between a 1 [cm] range with a common rare earth magnet.

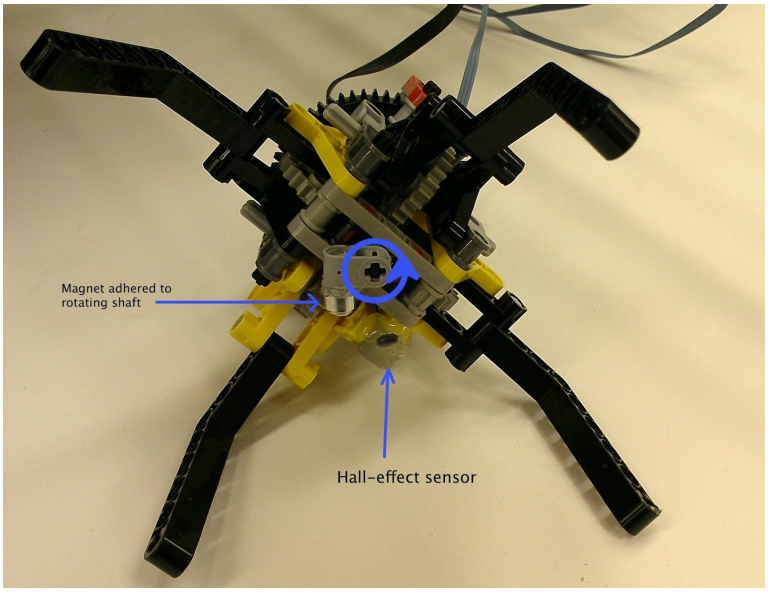

Figure 11: Diagram of the claw with the magnet and Hall effect sensor configuration.

The setup is as follows (refer to Fig. 11). To detect the rotations of the shaft, we attached the rare earth magnet to the shaft and attached the Hall effect sensor near the shaft as pictured below in the image to the right. As the motor shaft rotates, the Hall effect sensor outputs a pulse that the PIC32 samples with an analog input.

Stall-Detection Algorithm

To detect whether or not the motor has jammed, the PIC32 has three inputs: the analog hall-effect sensor, and two digital inputs from the FPGA. The two digital inputs are the signal lines that drive the claw motor to open or close. Thus, the PIC32 can determine when the claw motor is on *and* when the shaft is not rotating. The pseudo-code is depicted below:

```
while (1)
{
  if ( noSpinDetected() && motorOn() )
  {
    output high signal to FPGA;
  }
  else
  {
    output low signal to FPGA;
  }

  delay 250 ms before next sampling;
}
```

Thus, the PIC32 sets a single output pin high or low to the FPGA to indicate whether the claw is or is not jammed, respectively.

The `noSpinDetected()` algorithm is slightly more complicated. The goal is to accurately return `True` when the shaft is not rotating. Essentially, we take a moving average of Hall effect samples for a small amount of time while the shaft could be moving (about .5 seconds). During this time, we compare the moving average to the current sample from the sensor. If the current sample is significantly different from the moving average, then we assume that the shaft is rotating and return `False`. Otherwise, we make several more comparisons and ultimately return `True` to indicate that the shaft is stalled. The pseudo-code is depicted below:

```
arrayOfMagnetVals[100];

noSpinDetected()
{
// take 100 initial samples and store to arrayOfMagnetVals[];

  magnetValAverage = averageMagnetVals();

  for (j=0;j<50<j++)
  {
        currentSample = sampleHallEffectSensor;

        if( (difference between current sample and magnetValAverage) > 10%)
        {
                voltage spike is detected; return 0;
        }
        else
        {
          insert currentSample into arrayOfMagnetVals and shift out oldest sample;
          delay 10 ms before taking the next sample;
        }
  }
  /* if we made it this far, no voltage spike was detected on the sensor, so
  we assume that the Hall effect sensor is stalled. */

  return true to indicate that the shaft is not spinning;
}
```

Thus, the overall algorithm accurately outputs a high signal to the FPGA when the grabber claw is jammed, and a low signal when it is not jammed. To allow enough time for the motor to begin spinning, we delay 250 [ms] before reevaluating whether or not the claw is jammed.

Conclusions and Future Design Notes

Overall, the claw finite-state machine works as expected with three minor mechanical kinks in our current implementation. First, we adjusted our z-axis motor's gearbox to a slightly faster gear ratio; in the process, we discovered that one of our gears has a slight kink in it which stalls the motor unless we physically rotate the gear past the kink. Second, the parallel wood beams holding together the gantry are slightly offset in angle. While this seems negligible, this offset prevents the East-West axis from completely reaching the boundary of the outer dimension for West without having the motor spin in place. To work around this issue, we propose machining the gantry outer beams with more precise stock and instrumentations. Finally, because of this offset, we physically hold one corner of the claw crane in place to appropriately hold the East-West pulley at the proper length such that the motor can deliver sufficient torque to drive the gantry.

Despite these hardware limitations, the system behaves as expected, allowing a user to position the claw above an object, lower the claw, grasp it, and release input such that the claw auto-retracts to the corner of the system and then releases the claw.

In future implementations of such a project, we recommend an alternative H-bridge than the SN75441, which we implemented in this project. For this particular dual H-bridge, snubber-diodes are not built-in; thus, we must add these four diodes per H-bridge ourselves.

Citations and Additional Resources

Figure Number (if applicable)	*Title*	*Description*	*Citation*
	Lego Claw Design	We created a slightly-modified version of this inspirational Lego Claw Grabber	http://www.rjmcnamara.com/wp-content/instructions/QuadTrackCrane/CraneGrab/CraneGrab.html
6, 8	H-bridge diagram	To discuss motor control theory	http://upload.wikimedia.org/wikipedia/commons/thumb/d/d4/H_bridge.svg/500px-H_bridge.svg.png
9	Stepper motor phase sequence waveform	Illustrate cascading square waves to drive stepper motor at full torque rating	http://upload.wikimedia.org/wikipedia/commons/8/85/Drive.png
	Bipolar stepper motor theory	This document provides details on driving Bipolar Stepper Motors	http://homepage.cs.uiowa.edu/~jones/step/an907a.pdf
	SS495a Hall-Effect Sensor	datasheet	http://html.alldatasheet.com/html-pdf/124002/HONEYWELL/SS495A/485/1/SS495A.html
	Adafruit stepper motor	Technical info for using the stepper motor we bought	http://adafruit.com/products/324

Table 3: Citations and references

Appendix A: Bill of Materials

	Description	Link	Cost/Item	Quantity	Total Cost
Stepper Motors	1.8 degree per step, 12V power supply, 350mA	https://www.adafruit.com/products/324	$14.00	2.	$28.00
Joystick	8-directions (can be changed to 4-directions)	https://www.adafruit.com/products/480	$14.95	1.	$14.95
Linear bearings (4-pack)	Two per rod per platform	http://www.vxb.com/page/bearings/PROD/16mmLinearMotionSystems/kit994	$19.95	2.	$39.90
Linear motion rods	16mm Shaft 60" Hardened Rod Linear Motion	http://www.vxb.com/page/bearings/PROD/kit1008	$39.95	2.	$79.90
Pulley	Interfaces motors w/ belt: A 6Z23M017DF0905	https://sdp-si.com/ss/PDF/80502109.pdf	$5.19	6.	$31.14
Belt	Provides linear movement - 3mm pitch (HTD): A 6R23MC090	https://sdp-si.com/ss/PDF/80502008.pdf	$9.26	3.	$27.78
Clamp	Interfaces platform w/ belt: A 6M53M090	https://sdp-si.com/ss/PDF/80502010.pdf	$5.34	2.	$10.68
Belt (second order)	Provides linear movement - 3mm pitch (HTD): A 6R23MC090	https://sdp-si.com/ss/PDF/80502008.pdf	$9.26	2.	$18.52
Clamp (second order)	Interfaces platform w/ belt: A 6M53M090	https://sdp-si.com/ss/PDF/80502010.pdf	$5.34	1.	$5.34
SHIPPING - Linear motion	Shipping cost for VXB products		$16.83	1.	$16.83
TAX - Linear motion	Tax on VXB products		$9.28	1.	$9.28
HANDLING - Pulley system	Handling cost for SDP-SI products		$3.95	1.	$3.95
SHIPPING - Adafruit (Joystick)	Shipping cost for Joystick		$10.42	1.	$10.42
SHIPPING - Pulley system	Shipping cost for SDP-SI products		$10.03	1	$10.03
SHIPPING/HANDLING - SDP (second order)	Shipping/handling cost for SDP-SI products (second order)		$13.98	1	$13.98
Hall Effect sensor	Transducer for magnetic field strength --> voltage	Pre-owned	$0.00	1	$0.00
DC motors	Z-axis motion and claw grabbing	Pre-owned	$0.00	2	$0.00
H-bridges, resistors, capacitors	Circuiting components	Stockroom	$0.00	0	$0.00
				Items Cost	256.21
				Tax/Shipping Cost	64.49
				Total Cost	**320.7**

Appendix B: Source code for FPGA

```
1   // Names: Joshua Vasquez and Nicole Yu
2   // Email: joshua_vasquez@hmc.edu and nyu@hmc.edu
3   // Code to control an arcade grabber machine
4
5   // arcade_grabber: Top-level module
6   // Input pins: clk, reset
7   //           pushN, pushE, pushS, pushW, pushZ, grabStop
8   //           moveN, moveE, moveS, moveW,
9   //           open, close, up, down
10  // Output pins: stepper_NS_1A, stepper_NS_2A, stepper_NS_3A, stepper_NS_4A,
11  //           stepper_EW_1A, stepper_EW_2A, stepper_EW_3A, stepper_EW_4A,
12  //           z_axis_1A, z_axis_2A, claw_3A, claw_4A
13
14  module arcade_grabber (  input logic clk, reset,
15                           input logic pushN, pushE, pushS, pushW, pushZ, grabStop, // NESWZ
    push buttons, grabStop
16                           input logic moveN, moveE, moveS, moveW, // NESW joystick switch
17                           input logic open, close, up, down, // buttons to control z-axis and
    claw
18                           output logic stepper_NS_1A, stepper_NS_2A, stepper_NS_3A,
    stepper_NS_4A, // NS stepper motor
19                           output logic stepper_EW_1A, stepper_EW_2A, stepper_EW_3A,
    stepper_EW_4A, // EW stepper motor
20                           output logic z_axis_1A, z_axis_2A, claw_3A, claw_4A  ); // output
    to z-axis and claw DC motors
21
22      // Clock counter
23      logic [31:0] counter;
24
25      // FSM variables
26      logic [5:0] state, nextstate;
27
28      // Stepper motor variables
29      logic NS_on, NS_clockwise, EW_on, EW_clockwise;
30      logic z_axis_on, z_axis_clockwise, claw_on, claw_clockwise;
31
32      // Synchronizer variables
33      logic flop_pushZ, sync_pushZ, flop_close, sync_close;
34
35
36
37      // Synchronizer logic
38      flopr pushZ_one(counter[20], reset, pushZ, flop_pushZ);
39      flopr pushZ_two(counter[20], reset, flop_pushZ, sync_pushZ);
40      flopr close_one(clk, reset, close, flop_close);
41      flopr close_two(clk, reset, flop_close, sync_close);
42
43
44
45      // FSM States
46      parameter MOVE = 6'b000001; // User positions on x and y-axes
47      parameter DOWN = 6'b000010; // User positions on z-axis
48      parameter GRAB = 6'b000100; // User closes claw on object
49      parameter RETRACT = 6'b001000; // Auto-retracts claw up z-axis to x-y plane
50      parameter ALIGN = 6'b010000; // Auto-positions x and y-axes to (0, 0)
51      parameter RELEASE = 6'b100000; // Drops prize!
52
53
54
55      // Next state logic
56      always_comb
57         case(state)
```

```
58            // MOVE state unless up or down button is pressed to DOWN state
59            MOVE: if ( up | down )
60                     nextstate = DOWN;
61                  else
62                     nextstate = MOVE;
63
64            // DOWN state unless close button is pressed to GRAB state
65            DOWN: if ( sync_close )
66                     nextstate = GRAB;
67                  else
68                     nextstate = DOWN;
69
70            // GRAB state until grabStop is detected
71            GRAB: if ( grabStop )
72                     nextstate = RETRACT;
73                  else
74                     nextstate = GRAB;
75
76            // RETRACT state until claw is in x-y plane to ALIGN state
77            RETRACT: if ( sync_pushZ )
78                        nextstate = ALIGN;
79                     else
80                        nextstate = RETRACT;
81
82            // ALIGN state until south and west bumper are both pressed
83            ALIGN:   if ( pushS & pushE )
84                        nextstate = RELEASE;
85                     else
86                        nextstate = ALIGN;
87
88            // RELEASE state until grabStop is detected
89            RELEASE: if ( grabStop )
90                        nextstate = MOVE;
91                     else
92                        nextstate = RELEASE;
93
94            // default state is MOVE
95            default: nextstate = MOVE;
96         endcase
97
98
99
100      // Synchronous logic: get nextstate and increment counter
101      always_ff @ (posedge clk)
102      begin
103         state <= nextstate;
104         counter <= (counter + 31'b1);
105      end
106
107
108
109      // Stepper motor logic
110
111      // Steppers are on if in MOVE or ALIGN state and joystick
112      // movement does not correspond to push button
113      assign NS_on = ( state[0] & ( (!pushN & moveN) | (!pushS & moveS) ) ) | ( state[4] & !
     pushS );
114      assign EW_on = ( state[0] & ( (!pushE & moveE) | (!pushW & moveW) ) ) | ( state[4] & !
     pushW );
115
116      // CW and CCW direction depends on joystick movement
117      always_comb
```

```
118         if (state[4])
119            NS_clockwise = 1'b0; // move south during ALIGN
120         else
121            NS_clockwise = moveN;
122
123      always_comb
124         if (state[4])
125            EW_clockwise = 1'b0; // move east during ALIGN
126         else
127            EW_clockwise = moveW;
128
129      stepper_motor stepper_NS( NS_on, NS_clockwise, counter, stepper_NS_1A, stepper_NS_2A,
     stepper_NS_3A, stepper_NS_4A);
130      stepper_motor stepper_EW( EW_on, EW_clockwise, counter, stepper_EW_1A, stepper_EW_2A,
     stepper_EW_3A, stepper_EW_4A);
131
132
133
134      // DC motor logic
135
136      // Z-axis motor is on if in DOWN or RETRACT state and
137      // movement does not correspond to push button
138      assign z_axis_on = ( state[1] & ( (!sync_pushZ & up) | down ) ) | ( state[3] & !
     sync_pushZ );
139
140      always_comb
141         if (state[3])
142            z_axis_clockwise = 1'b0; // move up during RETRACT
143         else
144            z_axis_clockwise = down;
145
146      // Claw motor is on if in GRAB state
147      assign claw_on = (~grabStop) & ( ( sync_close | open ) | state[5] );
148
149      always_comb
150         if (state[5])
151            claw_clockwise = 1'b0; // open during RELEASE
152         else
153            claw_clockwise = sync_close;
154
155      DC_motor z_axis_motor( z_axis_on, z_axis_clockwise, z_axis_1A, z_axis_2A );
156      DC_motor grab_motor( claw_on, claw_clockwise, claw_3A, claw_4A );
157
158   endmodule
159
160
161
162   // stepper_motor
163   // Control module for stepper motors given on/off and direction bits
164   // Outputs cascading square waves
165   module stepper_motor(   input logic on, clockwise, // On/off and direction logic bits
166                           input logic [31:0] counter,
167                           output logic A1, A2, A3, A4    ); // Stepper motor pins
168
169      // Variables for on state
170      logic on_A1, on_A2, on_A3, on_A4;
171
172      // XORing two adjacent bits of counter creates a square wave of the same
173      // frequency of the higher bit but offset by half that square wave's period
174      assign on_A1 = ~counter[19];
175      assign on_A2 = (counter[18] ^ counter[19]);
176      assign on_A3 = ~on_A1;
```

```
177      assign on_A4 = ~on_A2;
178
179      // If on is high, select the on variables. If not, set all the outputs to low.
180      // If counter clockwise, invert the offset waves.
181      assign { A1, A2, A3, A4 } = on ? ( clockwise ? { on_A1, ~on_A2, on_A3, ~on_A4 } : { on_A1
     , on_A2, on_A3, on_A4 } ) : 4'b0000;
182
183   endmodule
184
185
186
187   // DC_motor
188   // Control module for DC motors given on/off and direction logic bits
189   module DC_motor(  input logic on, clockwise, // On/off and direction logic bits
190                     output logic A1, A2     ); // DC motor pins
191
192      // Variables for on state
193      logic [1:0] on_A;
194
195      assign on_A = 2'b10;
196
197      // If on is high, select the on variable. If not, set all the outputs to low.
198      // If counter clockwise, invert the on variable.
199      assign { A1, A2 } = on ? ( clockwise ? on_A : ~on_A ) : 2'b00;
200
201   endmodule
202
203
204
205   // flopr
206   // Flip flop module that passes d to q on clk or reset posedge
207   module flopr (  input  logic clk, reset,
208                   input  logic d,
209                   output logic q  );
210
211     always_ff @(posedge clk, posedge reset)
212       if (reset) q <= 0;
213       else       q <= d;
214
215   endmodule
216
```

Appendix C: Source code for PIC32

H:\E155\FinalProject\PIC_code\Claw_output\pic_uart.c

```
/*
Authors Nicole Yu and Joshua Vasquez
December 9, 2012

*/

// NOTE: pic should be set at 40 [MHz] --> Pb_Clk is Sys_Clk/1
// Otherwise: U2BRG will be configured incorrectly.

// NOTE: to compile at the -std=c99 argument to the command line options

// include files:
#include <stdint.h>
#include <stdio.h>
#include <P32xxxx.h>
#include "GenericTypeDefs.h"
#include <math.h>



// Constants:
#define TRMT_MASK    0x0100
#define URXDA_MASK    0x00000001
#define UART_DISABLE_MASK    0x8000
#define INT32_MAXCHARS 10    // maximum number of chars in an int32_t datatype
#define FALSE 0
#define TRUE 1
#define ASCIIOFFSET 48

#define MAGPIN    4 // hall-effect sensor pin
#define NUMSAMPLES 100 // how many initial hall-effect sensor samples we want.
#define THRESHOLD 0.05  // deviation from average value.

#define MOTOR_L_PIN 1
#define MOTOR_R_PIN 3


/*****************************************************************************
function prototypes:
*****************************************************************************/
// Serial Functionality
void    serialInit(void);
void    serialWriteByte(char valToWrite);
void    serialWrite( char* charArray, uint8_t howManyChars);
uint8_t serialAvailable(void);// True if at least one char is available to be read
char    serialReadByte(void);
void    serialReset(void);

int16_t analogRead( uint8_t pin);
void initTimers(void);
void delay_ms( uint8_t howManyMs);
int16_t averageMagValues(void);
uint8_t noSpinDetect(void);


/*****************************************************************************
global variables:  (alas!)
*****************************************************************************/
int16_t magVals[NUMSAMPLES];



/*****************************************************************************
 Main Loop
*****************************************************************************/
int main(void)
{
    serialInit();
    printf("I'm ready for action! \r ");

    TRISF &= ~(1);  // setup PORTF 0 as an output.
```

```
    TRISF |= 1 << MOTOR_L_PIN;  // setup PORTF 1 as an input.
    TRISF |= 1 << MOTOR_R_PIN;  // setup PORTF 2 as an input.

    uint8_t motorLSig;
    uint8_t motorRSig;


    while (TRUE)
    {
        // sample both motor wires to see if FPGA is driving motor:
        motorLSig = 0x01 & (PORTF >> MOTOR_L_PIN);
        motorRSig = 0x01 & (PORTF >> MOTOR_R_PIN);

        // if motors are being driven by the shaft is jammed:
        if (noSpinDetect() && (motorLSig || motorRSig) )
        {
            printf("Jam Detected! \r");
            PORTF |= (1 << 0);  // turn on PORTF bit 0.
        }
        else
        {
            printf("NOPE \r");
            PORTF &= ~(1 << 0); // turn off PORTF bit 0.
        }

        // delay slightly before next reading to give motor time to spin up
        delay_ms(250);
    }

    return 0;   // never reached
}



/*****************************************************************************
Delay Functions
*****************************************************************************/

/*****************************************************************************
void initTimers(void)
    sets appropriate values to configure one of the PIC's hardware timers.
*****************************************************************************/
void initTimers(void)
{
    T1CON = 0b1001000000110000;
}


/*****************************************************************************
void delay_ms( uint8_t howManyMs)
    inputs: how many milliseconds (max 250)
   outputs: none, but the program does nothing for the selected amount of time
*****************************************************************************/
void delay_ms( uint8_t howManyMs)   //max 250 ms!
{
    uint32_t duration;

    initTimers();

    TMR1 = 0;   // reset timer 1
    duration = 156 * howManyMs;     // conversion: (39062 [ticks] )/(1000 [ms])
    while (TMR1 < duration);    // do nothing;
}


/*****************************************************************************
analogRead Functions
*****************************************************************************/


/*****************************************************************************
void initadc(int pin)
```

```
*****************************************************************************/
void initadc(int pin)
{
    AD1CHS |= (pin << 16);  //  AD1CHSbits.CHOSA = channel;
    AD1PCFGCLR = 1 << pin;
    AD1CON1 |= 1 << 15; //AD1CON1bits.ON = 1;
    AD1CON1 |= 1 << 1;  //AD1CON1bits.SAMP = 1;
    AD1CON1 &= ~(0x00000001);   //AD1CON1bits.DONE = 0;
}

/*****************************************************************************
int readadc(void)
*****************************************************************************/
int readadc(void)
{
    AD1CON1 &= ~(0x00000001 << 1);      //AD1CON1bits.SAMP = 0;
    while ( !(AD1CON1 & 0x00000001) );  //(!AD1CON1bits.DONE);
    AD1CON1 |= 1 << 1;      //AD1CON1bits.SAMP = 1;
    AD1CON1 &= ~(0x00000001 << 0);  //AD1CON1bits.DONE = 0;
    return ADC1BUF0;
}


/*****************************************************************************
int16_t analogRead( uint8_t pin)
    inputs: which analog pin (0 through 15) on port RB
    output: the 10-bit ADC value.
*****************************************************************************/
int16_t analogRead( uint8_t pin)
{
    initadc(pin);
    return readadc();
}


/*****************************************************************************
uint8_t noSpinDetect(void)
    inputs: none
    output: a boolean whether or not the shaft is moving
*****************************************************************************/
uint8_t noSpinDetect(void)
{
    uint8_t i;          // index
    uint8_t j;          // index
    uint8_t k;          // index
    int16_t avgMagVal;  // average of 100 hall-effect sensor samples.

    for (i=0;i<NUMSAMPLES;i++)  // take 100 samples:
    {
        magVals[i] = analogRead(MAGPIN);
        delay_ms(1);
    }

    // average samples:
    avgMagVal = averageMagValues();


    /*  Now detect a spike in readings between now and the next 500 [ms].
        (A "spike" is simply a  value that deviates significantly from
        the average.)  If no spike is found, then the motor is stalled.
    */
    int16_t currMagVal;
    float percentDiff;

    for (j=0;j<50;j++)
    {
        // take a measurement:
        currMagVal = analogRead(MAGPIN);
        // calculate the current percent difference:
        percentDiff = ((float)fabs(currMagVal - avgMagVal) / (float)avgMagVal);

        if ( percentDiff > THRESHOLD )
```

```
        {
            return FALSE;   // Spike detected, so motor is not stalled.
        }
        else
        {
            // update avgMagVal array to include currentMagVal:
            // left-shift magVals
            for (k=0;k<(NUMSAMPLES-1);k++)
            {
                magVals[k] = magVals[k+1];
            }
            // insert latest reading.
            magVals[NUMSAMPLES - 1] = currMagVal;

            // update new avgMagVal:
            avgMagVal = averageMagValues();
            // delay between sample times:
            delay_ms(5);
        }
    }
    return TRUE;    // a jam is detected if we didn't see any spikes!
}



/*****************************************************************************
int16_t averageMagValues(void)
    inputs: none
    output: the average of the hall-effect sensor values stored in the global
            array.
*****************************************************************************/
int16_t averageMagValues(void)
{
    uint8_t i;
    int32_t averageValue = 0;

    for (i=0;i<NUMSAMPLES;i++)
    {
        averageValue += magVals[i];
    }
    averageValue /= NUMSAMPLES;

    return (int16_t)averageValue;
}



/*****************************************************************************
SERIAL FUNCTIONS
*****************************************************************************/

/*****************************************************************************
void serialInit(void)
    inputs: none
    outputs: none
Description: initializes parameters to communicate via the serial line at
            115 Kbaud.
*****************************************************************************/
void serialInit(void)
{
    U2BRG = 21; //259; // set baud-rate-generator value for 9600. (Ref manual 21-13)

    // By default, UART is configured for 8-bit data, one stop bit, no parity.
    // no action needed.

    U2MODE = 0x8000; // enable UART <15>;
    U2STA = 0x1400; //  enable TX pin <12>; enable RX pin <10>

}
```