burger stache

Design Overview

Game Strategy:
Our strategy for the game was to score all the balls in goal #3 (which is a box), since it is worth the most points. This eliminated the need to differentiate between yellow and black balls. Since there is a 10 second delay to reset the dispenser tower after 5 balls have been dispensed, our goal was to dump a payload into the goal and return in 10 seconds to maximize our efficiency. Our robot moved so fast that we actually had to introduce an extra time delay so that the round-trip took over 10 seconds.

Software Design:
To structure the code, we employed the state machine template for the overall game and then used a nested state machine for tape sensing. A high level outline of the overall game can be found below and is broken into five game segments for easier reading:

INITIAL ORIENTATION + APPROACH

```
Wait for electronic flash
Scan left until you see what could be a beacon
    stop the motors and run the fake beacon check, make sure it's really a beacon
    if it is a beacon, go to next stage
    if not, rotate a bit to make sure we don't sense the same fake beacon again and then continue checking
Move forward a bit without tape sensing to get off the green square
Move forward until we hit tape
Flip the mirror flag high based on whether we hit L or R tape sensor first
Go forward until the first black tape line is passed
Rotate a bit so we're aimed at the ball dispenser
Approach tape, follow tape until T is hit
```

GET BALLS FROM DISPENSER

```
Move forward until you hit the front bumper OR if bumper is already hit, create a bump event
Sit still until a ball is collected
Move backwards
(repeat 4 more times to get 5 balls)
```

GO TO GOAL
Turn 90 degrees to face the green line of tape that leads to goal 3
Go forward a little bit until we're off the tape, then enable tape sensing
TURBO BOOST for a second or so!
Go forward until tape is hit
Activate tape sensing, follow to the T
Back up a bit to better align with goal 3
Turn to face the goal
Go forward until front bumper is hit

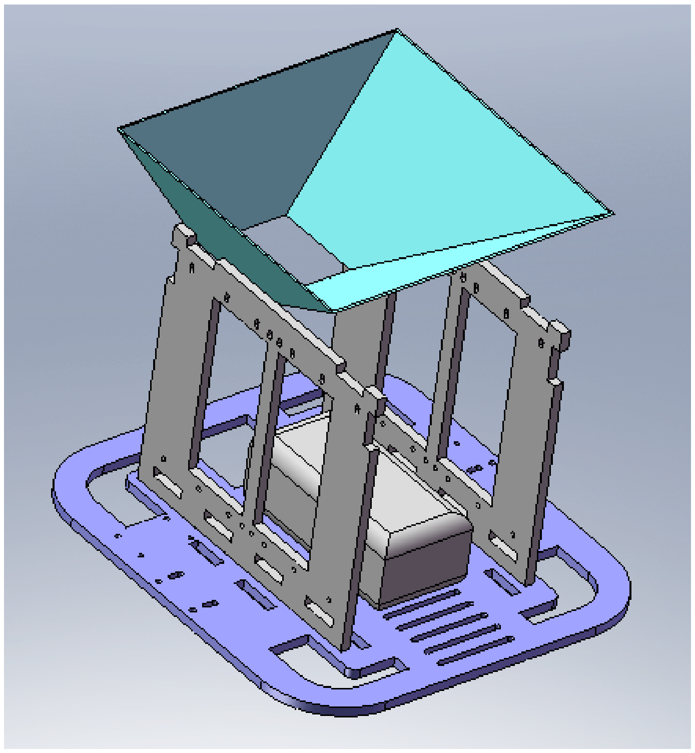
SCORE POINTS!
Activate the servo to dispense the balls

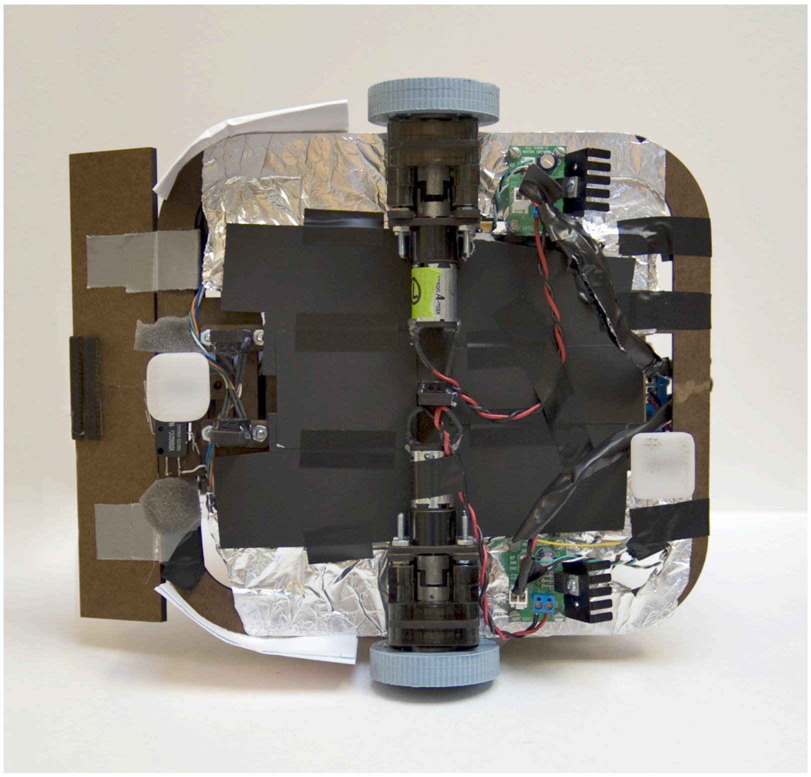
GO BACK TO THE DISPENSER
Back up a bit
Turn to face the dispenser
Go forward so we pass the green tape
TURBO BOOST for a second or so
Then activate tape sensing
Follow the black tape to the T
Cycle through "get balls from dispenser" code, do this until either the game timer runs out or we've deposited 20 balls

(for a full code listing, including headers, please see the *end* of the document…)

Hardware Design

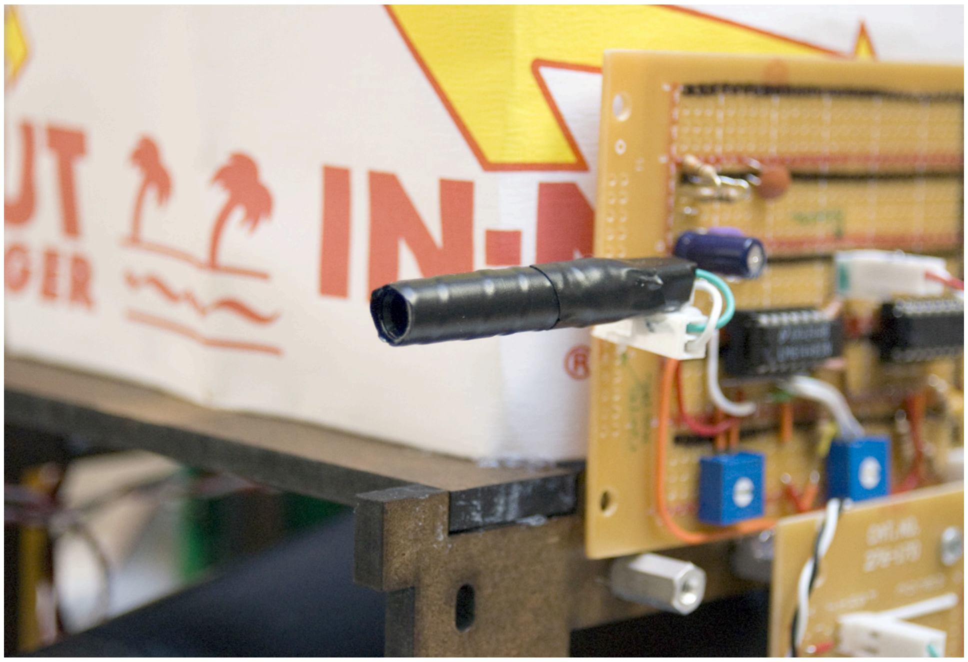
Our hardware design was focused on simple construction and stable, consistent performance. The basic robot chassis is a single 10"-square board, with two vertical walls coming up from the base. These pieces were modeled in SolidWorks and cut from 1/4" Masonite on the LaserCAMM. The goal of the design was to make it as flexible and modifiable as possible. Every mounting hole for any component (drive train, side wall, tape sensor, circuit board, etc…) was designed with slots so the components could be moved around and shifted as necessary.



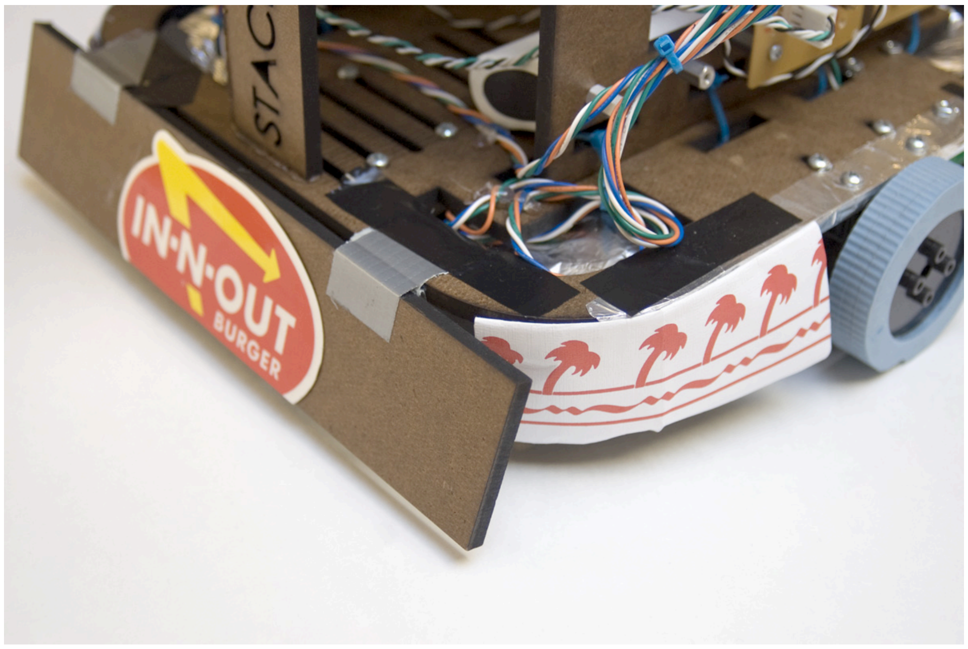


Drive train:
The drive train of the robot consists of two wheels mounted in the middle of the robot, and a sliding support in front and back. The wheels are made of custom-cast rubber. Motors, generously donated by Maxon Motors, are coupled to the wheels through spider couplings. The whole assembly is mounted on the transverse center line of the robot so it can pivot about its center, simplifying maneuvers. The decision was made to forgo the use of shaft encoders for simplicity. Sliding supports were chosen for the front and back because they are significantly lighter than casters.

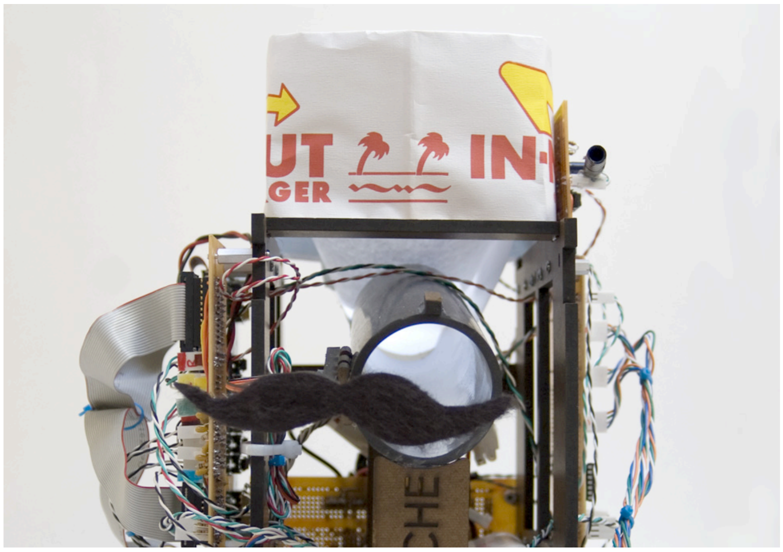
Sensors:
The robot has three IR sensors used to detect lines of tape placed on the field. One sensor is mounted in the exact center between the motors so that it will remain in place during pivoting maneuvers. The other two sensors are mounted on the front, spaced slightly wider than the lines of tape. The line following strategy is a simple bang-bang design (there is no proportional sensing of tape), but for the assigned game it is adequate. The robot has a phototransistor that is used a beacon sensor. It is housed inside a short section of a pen body wrapped in electrical tape, and mounted directly to a molex connector on the corresponding circuit board.



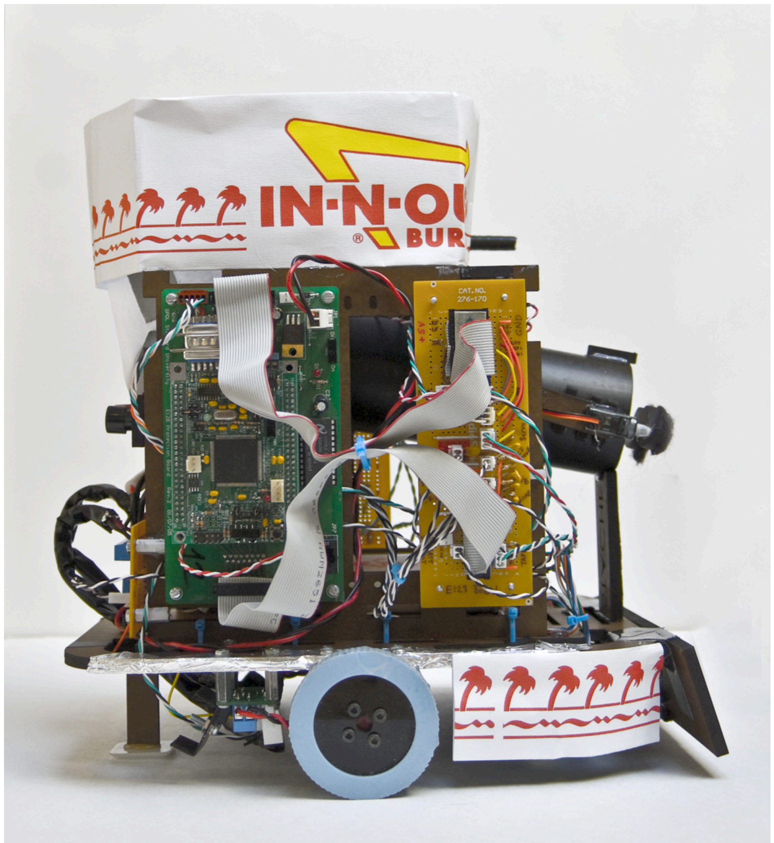
Bumper:
A limit switch is mounted under the front of the robot between the tape sensors. A wide piece of masonite hung from the front contacts the limit switch whenever a wall is encountered. Pieces of foam are used to damp the movement of the bumper to avoid bouncing.



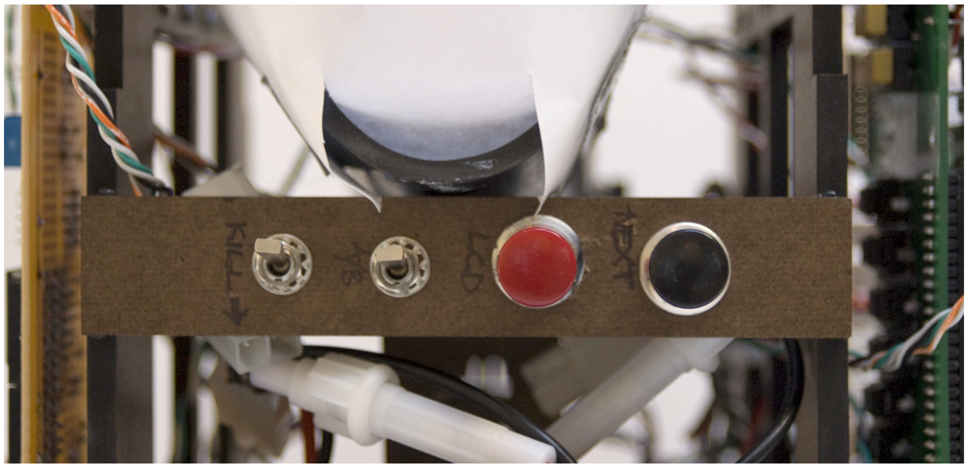
Ball System:
Our strategy for the game is to collect 5 balls at a time from the dispenser, deposit them in goal #3, and repeat. Since the ball dispenser drops the balls from above, we designed a funnel (made of a paper hat from In-n-Out Burger) that directs the balls to the top of a slanted plastic tube. The 5 balls line up in the tube and are released all at once into the goal. The stopper on the front of the tube is a moustache-shaped piece of Masonite mounted to a 180º servo.



Board Mounting:
The vertical walls of the robot serve to support the funnel for the ball system, but more importantly, they support all the circuit boards! The two walls are identical, but mounting holes are spaced to allow for 3 of our perf-boards on one side and 1 perf-board and the E128 on the other side. Stand-offs are used to provide necessary separation.



Handy Buttons:
The robot has two push-buttons and two toggle switches mounted on the back. One toggle switch is the software emergency stop, and one button is a “next state” button that lets us, for example, start the robot when people have stolen or otherwise mis-placed the flashers. The other push button was intended to update the LCD screen, and the other toggle is unassigned, but neither one is actually used.

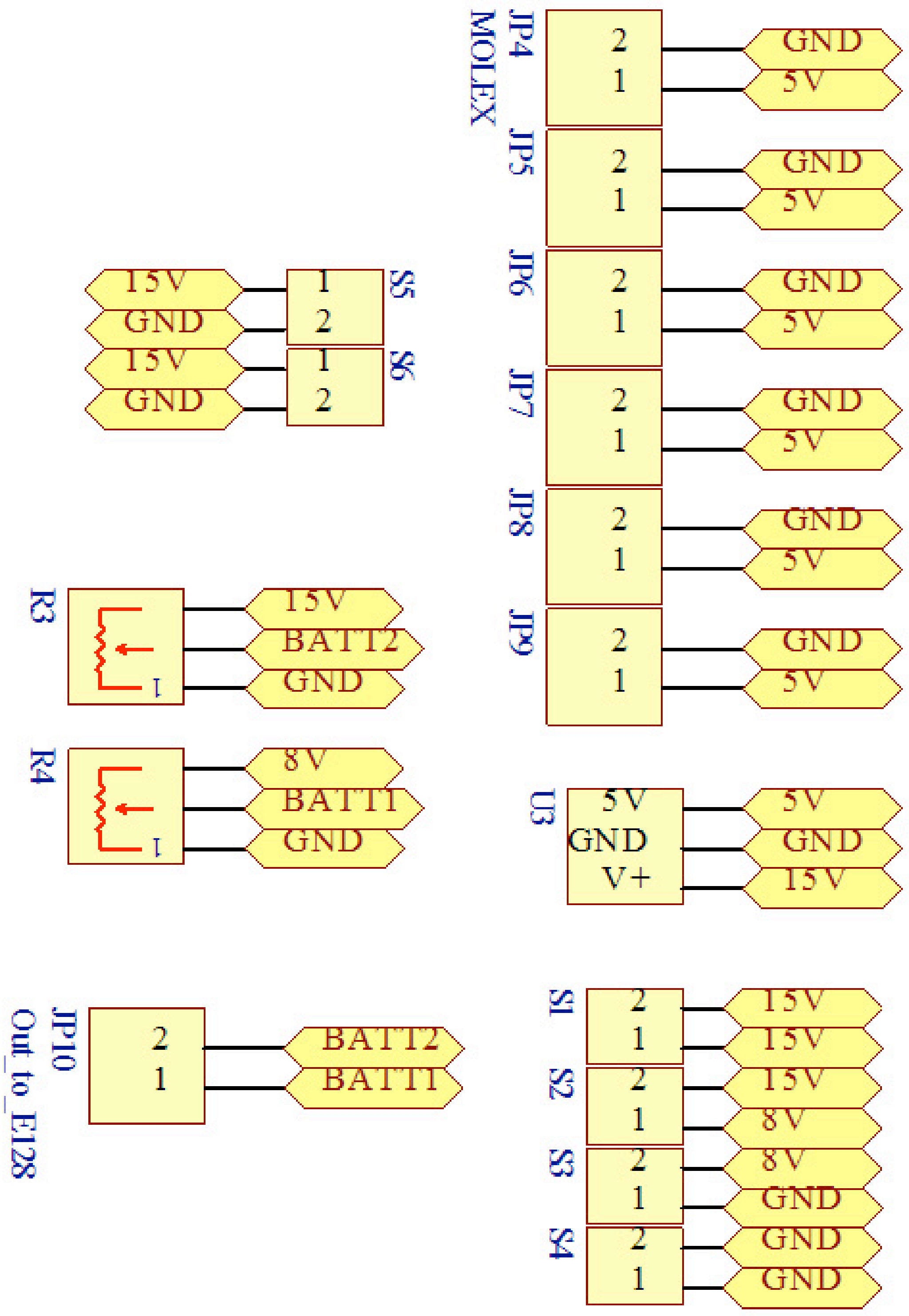


Electrical Design

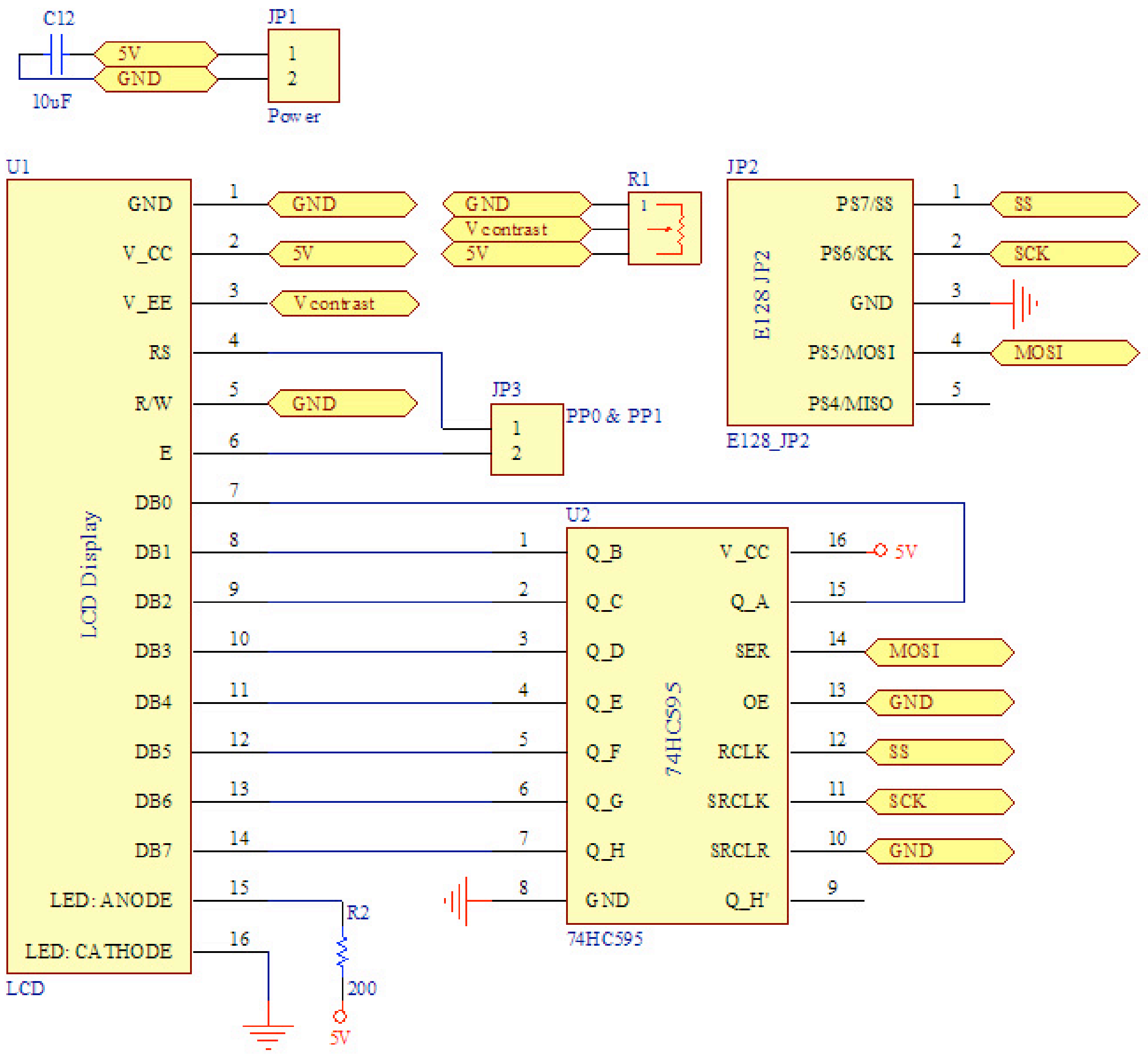
We wanted our electronics to be simple, accessible, and easy to debug. We planned to modularize the circuitry by putting different functions on separate boards. Each board receives +5V/GND from the power board, and molex connections are used to transmit signals from board to board. Thus, we have 6 boards in total:

- Power board
- LCD board
- Beacon board
- Tape sensor board
- E128 board
- Connector board

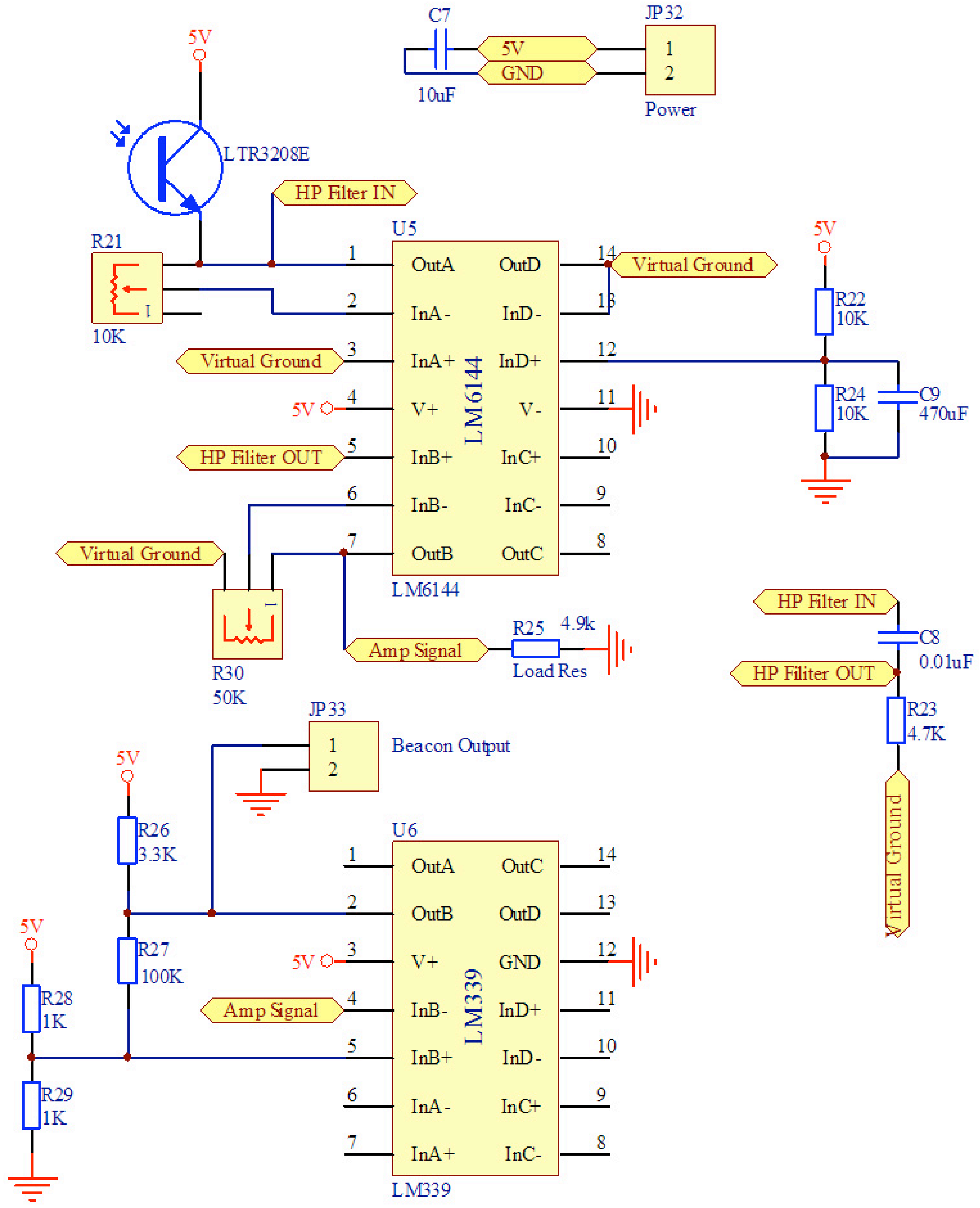
Power Board:
The power board has screw terminals for connecting the batteries and supplying drive-current to the motors. These are physically laid out next to each other to minimize conductive noise on the ground rail. A LM7805 regulator (with heat sink!) provides a steady 5V supply. An array of two-pin molex headers provides 5V/GND connections for all the other boards. The entire robot is powered by a pair of 7.2V (nominal) rechargeable Ni-Cad battery packs, wired in series for a total of about 15V. We placed 100k pots across the batteries and calibrated them to provide an analog voltage proportional to the voltage of the “lower” battery, and the sum total of both batteries. This is read by the E128 so that we can always know the current state of charge of the batteries.



LCD Board:
Our original design called on having an LCD screen that would print out a range of game state information to facilitate debugging, and add a splash of pizzazz. The LCD board has a 5-pin molex connector that receives SPI data from the E128, as well as a 2-pin molex connector for additional control lines. A shift register on the board takes the received serial data and outputs it in parallel for the LCD. Finally, a 2-pin molex provides 5V/GND from the power board. Unfortunately, printing to the LCD requires extensive use of blocking code, and it was prohibitively slow to print updates to the screen continuously, so we decided to not include the LCD on the final robot.

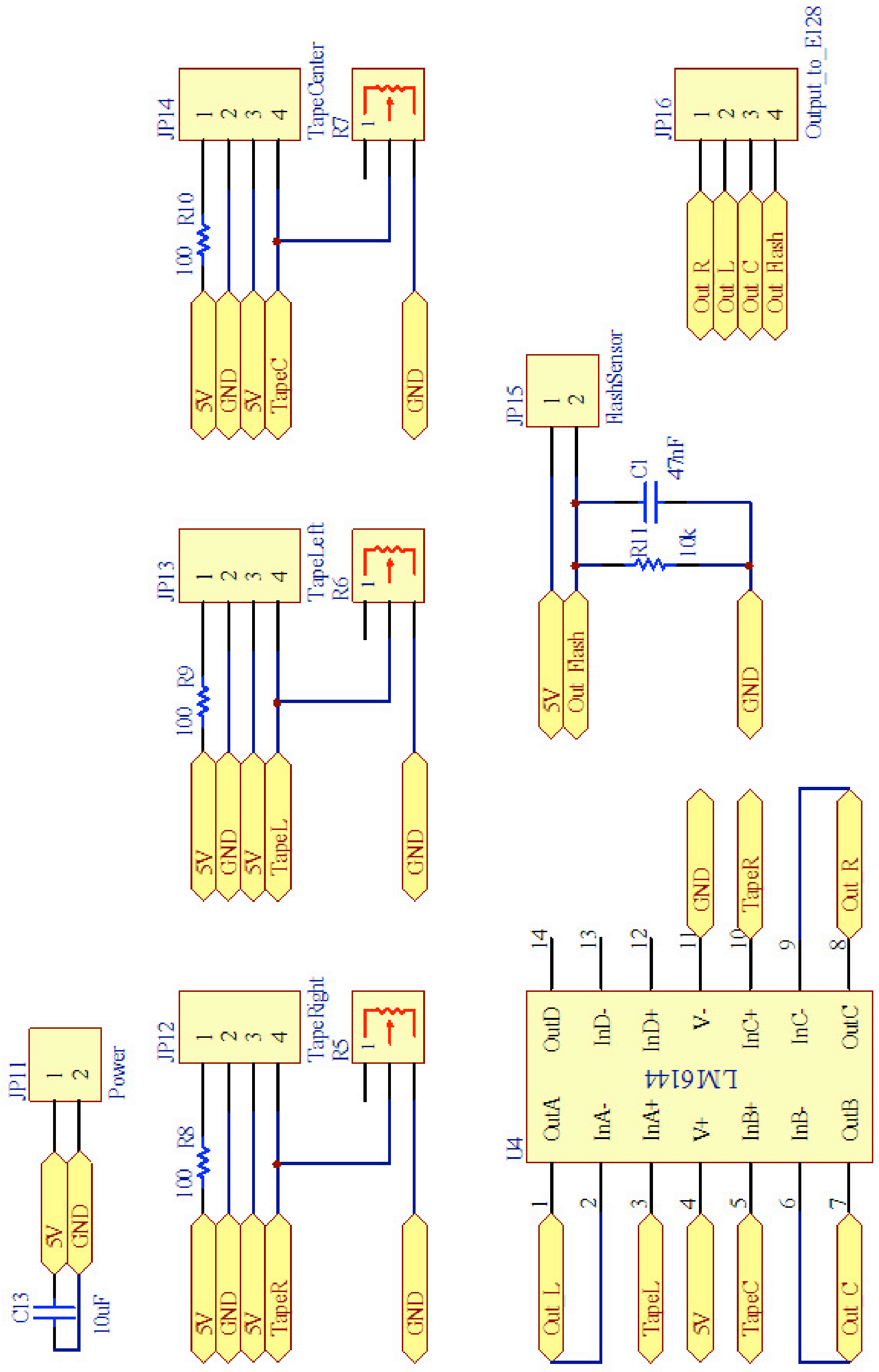


Beacon Board:
The beacon sensor was one of the last pieces designed for this project. The basic design uses a photo-transistor to receive incoming beacon pulses. The signal is passed through a transresistive amplifier, a high-pass filter (-3dB at 340 Hz), a non-inverting amplifier (gain 7.0), and a Schmidt trigger. The output is connected to a 2-pin molex for sending to the E128. As usual, a 2-pin molex is included for power connections.



Tape Sensor Board:

This board has the circuitry for all 3 tape sensors, as well as the flash sensor. There is a 4-pin molex for each tape sensor. Each molex provides +5V/GND for the IR LED, with 200 ohms of series resistance, and +5V/signal for the phototransistor. The signal is connected to a potentiometer so that all three tape sensors can be tuned to the same sensitivity, which simplifies the software. The output from each tape sensor is buffered using a quad op-amp chip, and connected to a 4-pin molex for sending to the E128. The fourth pin of this molex is for the flash sensor signal. The flash sensor is simply a phototransistor connected to a high-pass filter. In software, the ambient light-level is automatically calibrated each time the robot is booted. The flash level is set very slightly above this so that it is very sensitive to flashes received from anywhere.



E128 board:
This is the protection board that the SPDL provides for the E128. It has a 20-pin header, a 24-pin header, and a 5 pin molex for SPI. It receives 15V, unregulated, directly from the power board.

E128 PIN TABLE

JP6 24 Pin **RIBBON CABLE -->**

Use	Name	Pin Number	Pin Number	Name	Use
	NC	1	24	NC	
RMOTOR EF	PU7	2	23	GND	
LMOTOR EF	PU6	3	22	PU5	LMOTOR IN2
	PT0	4	21	PU4	RMOTOR IN2
IR BEACON RISE	PT1	5	20	PU3	
IR BEACON FALL	PT2	6	19	PU2	SERVO POS (PWM)
	PT3	7	18	PU1	LMOTOR IN1 (PWM)
	PT4	8	17	PU0	RMOTOR IN1 (PWM)
	PT5	9	16	PE7	
	PT6	10	15	PT7	
	PE0	11	14	PE1	
	NC	12	13	NC	

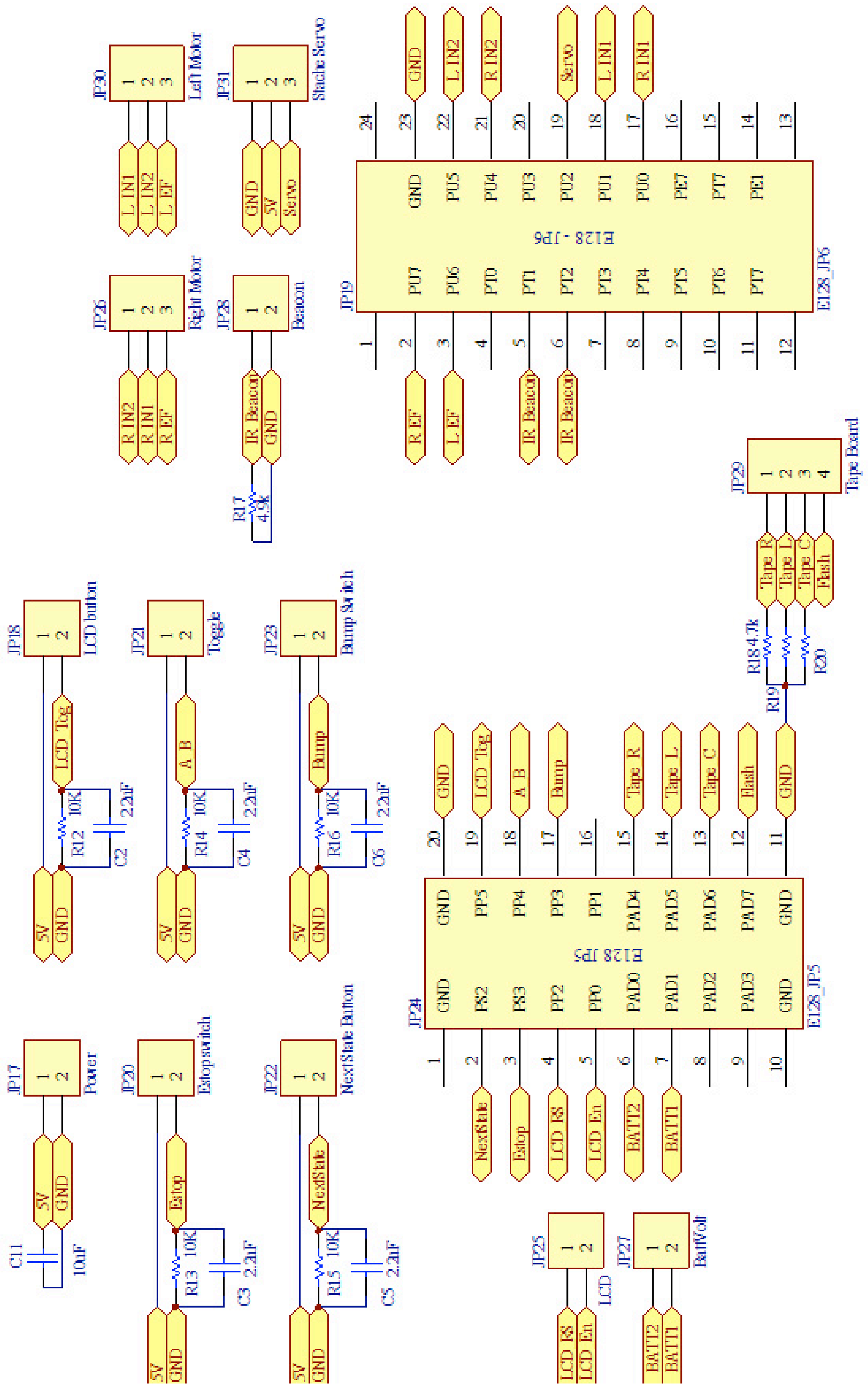
<-- RIBBON CABLE **JP5 20 Pin**

Use	Name	Pin Number	Pin Number	Name	Use
	GND	1	20	GND	
NEXT STATE BUTTON	PS2	2	19	PP5	LCD TOGGLE
E-STOP BUTTON	PS3	3	18	PP4	UNASSIGNED
LCD RS	PP2	4	17	PP3	BUMP SENSOR
LCD EN	PP0	5	16	PP1	
BATT2 VOLTAGE	PAD0	6	15	PAD4	RTAPE SENSOR
BATT1 VOLTAGE	PAD1	7	14	PAD5	LTAPE SENSOR
	PAD2	8	13	PAD6	CTAPE SENSOR
	PAD3	9	12	PAD7	FLASH SENSOR
	GND	10	11	GND	

<-- KEY **JP2 5 Pin**

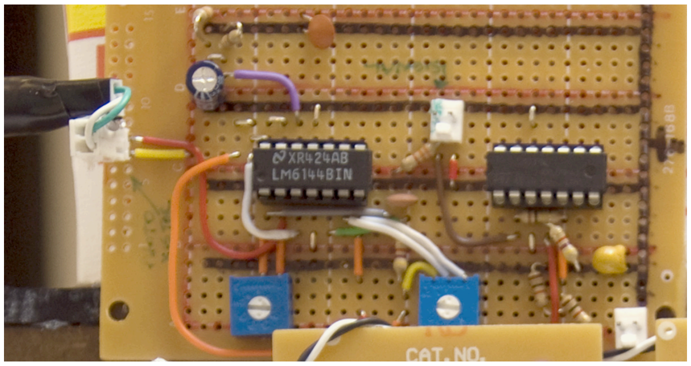
Use	Name	Pin Number
SHIFT REGISTER	PS7/SS	1
SHIFT REGISTER	PS6/SCK	2
SHIFT REGISTER	GND	3
SHIFT REGISTER	PS5/MOSI	4
SHIFT REGISTER	PS4/MISO	5

Connector Board:
All input and output connections for the E128 and routed through this board. Ribbon cables connect the E128 to this board. Outputs from this board include a 3-pin molex for each motor driver, a 3-pin molex for the servo, and a 2-pin molex for the LCD control lines. All the signals from the other boards are received on this board, including tape sensing, flash sensing, beacon sensing, and the battery voltage indicator. The tape sensor and beacon sensor inputs to the E128 have a 4.7k pull-down resistor to load the opamps that are transmitting the signals, to minimize the effects of capacitive noise on the signal lines. Finally, the connector board has five 2-pin molex connectors that are connected to the four "handy buttons" (e-stop, LCD update, next state, unassigned) and the bumper switch.

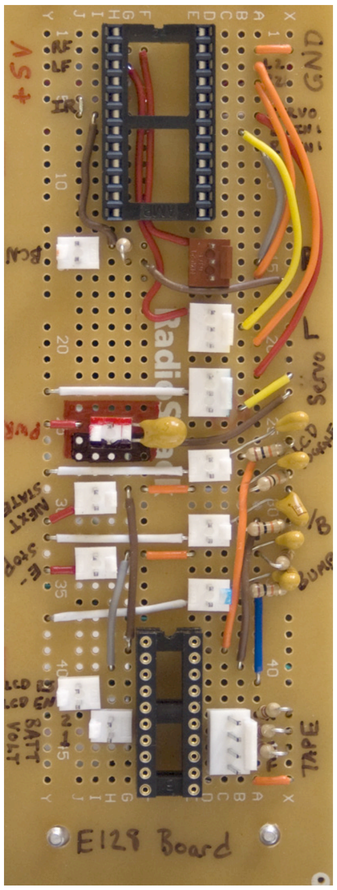


Board Photos

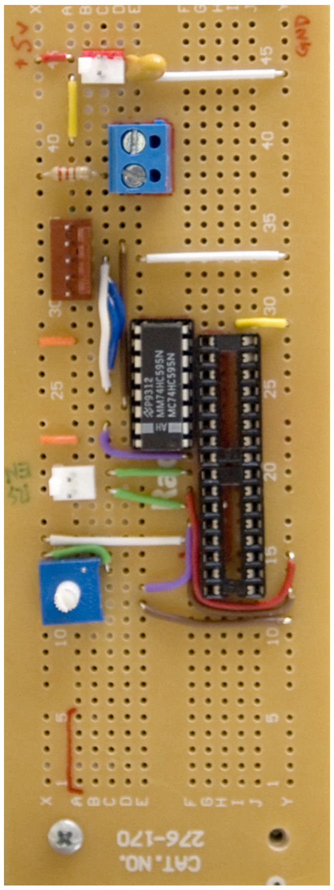
Beacon:



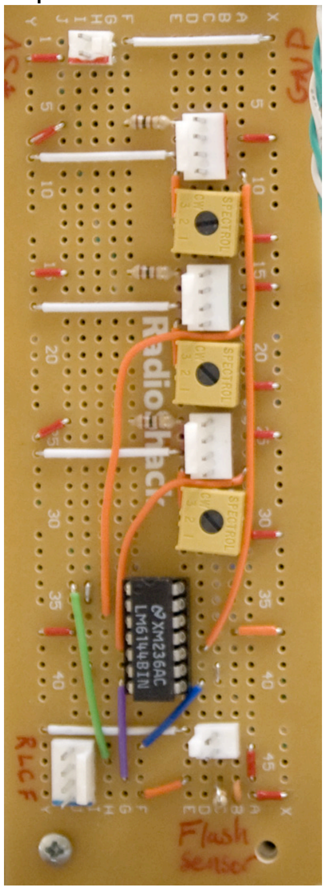
Header:



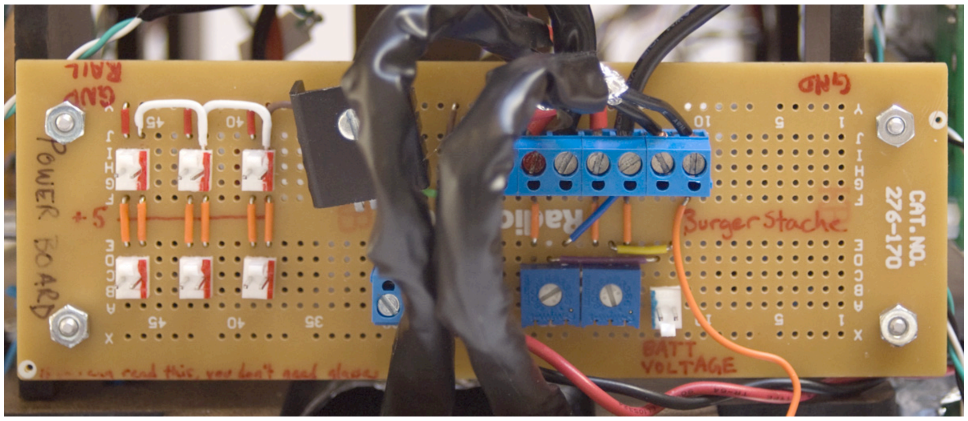
LCD:



Tape:



Power:



Shielding considerations:
We took steps to avoid noise in our system. Conductive noise was minimized by putting the high-current connections as close to the battery source as possible. Inductive noise was avoided by twisting all connector cables to minimize flux area. Capacitive noise was reduced by creating a grounded shield of aluminum foil on the bottom of the robot, as well as shielding the high-current wires running from the power board to the motor drivers. We found that our most noise-sensitive system was the beacon sensor, and the shielding did wonders to improve our signal.

Selected component values and calculations:

Tape Sensor Board:

Pots for tuning the IR tape sensor outputs to the same value:		
Right Tape Pot:	26.0	k
Left Tape Pot:	33.3	k
Center Tape Pot:	36.2	k
Current flowing in IR LED:		
Series resistor:	100	ohms
Drop across LED:	1.7	V
Current: (Vcc – 1.7)/R	33	mA

Beacon Board:

Signal Amplification:		
Transresistive Feedback:	2.15	k
Beacon Amplifier Rf:	40.8	k
Beacon Amplifier Ri:	6.82	k
Beacon Amp Gain (1+Rf/Ri):	7.0	
High-pass filter:		
HP resistor:	4.7	k
HP capacitor:	0.1	uF
HP corner frequency:	340	Hz
Beacon frequency:	1.25	kHz

→ Filter eliminates low-frequency noise from other light sources

Power Board:

Battery voltage dividers (100k pot):		
Battery Voltage 1:	59k / 100k	
Battery Voltage 2:	33k / 100k	
Quiescent current draw from voltage regulator:		
(Approximately 3*IR_LED_CURRENT + odds and ends)		
Actual steady current draw:	125	mA
Maximum 7805 rating:	1000	mA

→ within maximums

Motor Drivers (TLE5206-2):

Rated continuous current:	5	A
Rated peak current:	6	A
Max Stall Current:		
Motor coil resistance:	1.8	ohms
Max supply voltage:	~16	V
Max stall current (V/R):	8.9	A

Average operating stall current (60% duty cycle):

→ Over continuous limit, but we never stalled. Performance was excellent.

Connector Board:

Filter for debouncing button presses:

Resistance:	10	k
Capacitance:	2.2	uF
Discharging time constant:	22	mSec

→ Ultimately we did this in code anyway

Code Listing

balldispense.h

```
#ifndef BALLDISPENSE
#define BALLDISPENSE

//FUNCTION PROTOTYPES
void InitServoPWM(void);
void Open_Ball_Dispenser(void);
void Close_Ball_Dispenser(void);

#endif BALLDISPENSE
```

balldispense.c

```
//----------- balldispense.c -------------//
//-- code courtesy of BurgerStache --//
//-------------------////-----------//

//Standard Libraries
#include "headers.h"

//Initializes the PWM subystem on the E128
void InitServoPWM(void)
{
	//Initialize the clock
	PWMSCLB = POSTSCALER_B; //scale the A clock by / (2*75)
	PWMPRCLK |= 0x40;		//use clock A with M/16 scalar
	//Initialize PWM for servo
	PWME |= BIT2HI;		//enable PWM on bit 2
	MODRR |= BIT2HI;	//map T2 to PWM
	PWMCLK |= BIT2HI;	//use SA (scaled clock)
	PWMPOL |= BIT2HI;	//select the PWM polarity. 1 = output initially high
	PWMCAE |= BIT2HI;	//center align the PWM signal
	PWMPER2 = SERVO_PWM_PERIOD;
	PWMDTY2 = SERVO_CLOSED_DUTY;  //contains the count of the total number of cycles on either clock
A or SA that will constitute the active period for PWM channel 0
}

void Open_Ball_Dispenser(void)
{PWMDTY2 = SERVO_OPEN_DUTY;}

void Close_Ball_Dispenser(void)
{ PWMDTY2 = SERVO_CLOSED_DUTY;}

//------------------Test Routine----------------------//
#ifdef BALLDISPENSE_TEST

void main(void)
{
	char i;
	InitAll();
	//Cycle through .5 ms pulse to 2.5 ms pulse lengths
	while(TRUE)
	{
		/*
		for(i=2; i<=13; i++)
		{
			PWMDTY2 = i;
			printf("DUTY CYCLE: %d \r\n", i);
			Wait(1500);
		}*/
		printf("OPEN\r\n");
		Open_Ball_Dispenser();
		Wait(1500);
		printf("CLOSED\r\n");
		Close_Ball_Dispenser();
		Wait(2000);
	}
}

#endif
```

beacon.h

```
#ifndef BEACON
#define BEACON

#include "headers.h"

// Public Function Prototypes
void InitBEACON(void);
char CheckForSpecificBeacon(char beaconDuty);
int CheckForBeacon(void);

//Private function prototypes

#endif
```

beacon.c

```
#include "headers.h"

/*---------------------------- Module Variables ---------------------------*/
static unsigned int uPeriod;
static unsigned int uPulseWidth;
static unsigned int uLastRise;
static unsigned int uLastFall;
static int DutyCycle;
static unsigned char counter = 0;
static unsigned int DutyHistory[DUTY_HISTORY_LENGTH]=0;
static char DutyHistoryIndex = 0;
static char RisingFlag = 0;


/*------------------------------ Module Code ------------------------------*/
void InitBEACON(void)
{
   TIM0_TSCR1 = _S12_TEN; /* turn the timer system on */
   TIM0_TSCR2 = _S12_PR2; /* set pre-scale to /16 = 1.5MHz timer clk */

   //Set up OC4 to time the control loop.
   TIM0_TIOS = _S12_IOS4; /* set cap/comp 4 to output compare rest are inputs */
   TIM0_TCTL1 = TIM0_TCTL1 & ~(_S12_OL4 | _S12_OM4); /* no pin connected for OC4 */
   TIM0_TC4 = TIM0_TCNT + PERIOD_T; /* schedule first rise */
   TIM0_TFLG1 = _S12_C4F; /* clear OC4 flag */
   TIM0_TIE |= _S12_C4I; /* enable OC4 interrupt */

   //Set up IC5 to capture rising edge.
   TIM0_TCTL3 |= ( _S12_EDG5A );
   TIM0_TFLG1 = _S12_C5F; /* clear IC5 flag */
   TIM0_TIE |= _S12_C5I; /* enable IC5 interrupt */

   //Set up IC6 to capture falling edge
   TIM0_TCTL3 |= ( _S12_EDG6B );
   TIM0_TFLG1 = _S12_C6F; /* clear IC5 flag */
   TIM0_TIE |= _S12_C6I; /* enable IC5 interrupt */

   //Turn on them interrupts
   EnableInterrupts;
}

//returns true if the 30% beacon is found
char CheckForSpecificBeacon(char beaconDuty)
{
        DutyCycle = CheckForBeacon();

        if((DutyCycle < (beaconDuty + DUTY_TOLERANCE)) && (DutyCycle > (beaconDuty - DUTY_TOLERANCE)))
return TRUE;
        else return FALSE;
}

//Falling edges
void interrupt _Vec_tim0ch6 SignalFallTimer (void)
{
   uPulseWidth = TIM0_TC6 - uLastRise; /*Calculate high time*/
   uLastFall = TIM0_TC6;
   TIM0_TFLG1 = _S12_C6F; /* clear IC3 flag */
}

//Rising edges
```

```
   uPeriod = TIM0_TC5 - uLastRise; /*Calculate period*/
   uLastRise = TIM0_TC5;
   TIM0_TFLG1 = _S12_C5F; /* clear IC3 flag */
   RisingFlag = 1;
}

//Timer for testing
void interrupt _Vec_tim0ch4 ControlTrigger (void)
{
   PTT ^= 128;
   TIM0_TC4 += PERIOD_T; /* program next compare */
   TIM0_TFLG1 = _S12_C4F; /* clear OC4 flag */
   //EnableInterrupts;

   //counter += 1;
}

//Looks for the nearest whole number of detected duty cycle
//returns the duty cycle
int CheckForBeacon(void)
{
        int i;

        //Check to make sure we aren't looking at a noise pulse
        if(uPulseWidth < MIN_PULSE_WIDTH)
                DutyCycle = 0;
        else if(uPulseWidth > MAX_PULSE_WIDTH)
                DutyCycle = 0;
        else if(RisingFlag == 0)
                DutyCycle = 0;
        else
                DutyCycle = (uPulseWidth*10)/(uPeriod/10);
                RisingFlag = 0;

        //Increment duty history index to place the duty cycle
        DutyHistoryIndex++;
        if(DutyHistoryIndex >= DUTY_HISTORY_LENGTH)
                DutyHistoryIndex=0;

        //Place current duty cycle into appropriate position in history array
        DutyHistory[DutyHistoryIndex] = DutyCycle;

        //Check for a stable duty cycle. If it isn't stable, set it to zero (but not in history)
        for(i=0; i<DUTY_HISTORY_LENGTH; i++)
        {
                if((DutyHistory[i] < DutyCycle - DUTY_TOLERANCE) || (DutyHistory[i] > DutyCycle +
DUTY_TOLERANCE))
                {
                        DutyCycle=0;
                }
        }

        //printf("      CheckForBEACON = %d\r\n", DutyCycle);

        return DutyCycle;
}

#ifdef BEACON_TEST
void main(void){
        InitAll();

        printf("Beacon testing\r\n");
        Turn(LEFT, DUTY_BEACON_SEEK);
        while(TRUE){
                printf("CheckForBEACON = %d\r\n", CheckForBeacon());
                printf("CheckForThirtyBEACON = %d\r\n", CheckForSpecificBeacon(30));
                printf("CheckForFiftyBEACON = %d\r\n", CheckForSpecificBeacon(50));
                printf("CheckForSeventyBEACON = %d\r\n\r\n", CheckForSpecificBeacon(70));
                Wait(250);
        }
}
#endif
```

defines.h

```
#ifndef DEFINES
#define DEFINES

//Test defines
#define REAL_DEAL //UNCOMMENT TO RUN OUR NON-TEST MAIN FUNCTION
//#define SPI_SHIFT_REGISTER_TEST
//#define LCD_TEST
//#define DRIVING_TEST
//#define PWM_TEST
//#define DEBUG_RUG_TEST
//#define BALLDISPENSE_TEST
//#define SENSORS_TEST
//#define BEACON_TEST
//#define LINE_FOLLOW_TEST
//#define CALIBRATION_MOVE_TEST
#define SIMULATE_EVENTS //use if you want to simulate events with keyboard presses

//RUN WITH EACH BATTERY AT 8.1V

//Convenience
#define TRUE 1
#define FALSE 0
#define SUCCESS 0
#define FAILURE 1

//main
#define LINE_1 0x80
#define LINE_2 0xC0
#define LINE_3 0x94
#define LINE_4 0xD4

//flash
#define FLASH_TOLERANCE 20 //and so it is.

//tape
#define TAPE_THRESH_HIGH 800 //and so it is.
#define TAPE_THRESH_LOW 0
#define RIGHT_TAPE 4
#define LEFT_TAPE 5
#define CENTER_TAPE 6

//timer definitions (can use timers 0 through 8) ??NOT SURE FOR THE E128 WHICH ARE AVAILABLE??
#define TIMER_WAIT            0 //timer for the blocking wait function
#define TIMER_STATE           1 //timer for states of a specified duration
#define TIMER_GAME            2 //timer to keep track of game end

//timer lengths
#define TIME_DUMP                             2750 //1250 is ok, but we're too fast!
#define TIME_GAME                             1200 //in tenths of seconds (not ms)
#define TIME_COLLECT                  1500 //time to sit in front of the dispenser (with grace period)
#define ANGLE_BEACON_OVERSHOOT   5 //ANTI-overboost to get centered on the beacon
#define ANGLE_PASS_FAKE_BEACON   10 //also very small for now
#define DIST_BACKUP                            3 //old=4 back up 4 inches from the ball dispenser
#define DIST_OFFTAPE_CLEARANCE   2 //how far to go when going off the tape before switching to TAPESM
#define DIST_BOOST                             21 //fuck yes
#define DIST_CLEAR_SQUARE              8 //dist off of initial position (green square)
#define DIST_BACK_OFF_GREEN_T  2 //back off before turning to goal

//tape following movement duty cycles
//BE WARNED: duties less than 30 are pretty useless
#define DUTY_APP              40 //old=30 both wheels move forward with this duy as we approach
#define DUTY_PIVOT            40 //old=30 the wheels move in opposite directions with this duty when we
pivot
#define DUTY_BOP_OUTER 40 //old=30 the duty of the outer wheel during bopping
#define DUTY_BOP_INNER 0 //old=0 the duty of the inner wheel during bopping
#define DUTY_GO                       40 //old=30 both wheels move forward with this max duty when we're
going
#define DUTY_BEACON_SEEK 40 //old=30 duty to use during beacon seeking
#define DUTY_DEFAULT_TURN 50 //old=30 duty to use during beacon seeking
#define DUTY_DEFAULT_MOVE 60 //old=40 duty to use during moving while NOT tape following
#define DUTY_FIRST_APP   50 //off of beacon sensing, moving past first line
#define DUTY_GENTLEBUMP 40 //old=30
#define DUTY_BOOST            100 //YES! YES!

//game stages
//waiting for the electronic flash and getting oriented
#define ST_START              10 //waiting for electronic flash
#define ST_BEACON             11 //scan L toward the beacon, stop when we see a blip
```

```
#define ST_BEACON_3           14 //move past the blip w/o looking for beacons so we don't get stuck.
risk missing a beacon.
#define ST_CLEAR_SQUARE 15 //move forward a bit to get away from the green square

//Hello, dispenser. Nice to meet you! travelling toward the dispenser for the first time
#define ST_MIRROR_CHECK                       20 //go forward until L or R is hit to determine what side
we're on
#define ST_MEET_DISPENSER             21 //going forward until the first black line is passed
#define ST_MEET_DISPENSER_1           22 //correction factor
#define ST_MEET_DISPENSER_2           23 //approaching tape from the right until T is hit (nested tape
SM)
//collecting balls from the dispenser
#define ST_COLLECT_BALLS              30 //go forward till we hit our bumper
#define ST_COLLECT_BALLS_1            31 //sitting still till we get the first ball (timer)
#define ST_COLLECT_BALLS_2            32 //back up for a short amount of time (timer)
//now that we're full of balls, go to goal three
#define ST_GOTO_GOAL          40 //turning 90 degrees left
#define ST_GOTO_GOAL_1                41 //GO FORWARD A LITTLE BIT UNTIL WE'RE OFF THE TAPE, THEN ENABLE
TAPE SENSING
#define ST_GOAL_BOOST         98
#define ST_GOTO_GOAL_2                42 //GOING FWD UNTIL TAPE IS HIT (approach from left)
#define ST_GOTO_GOAL_3                43 //not used
#define ST_GOTO_GOAL_4                44 //backing up a bit
#define ST_GOTO_GOAL_5                45 //turning R 90 degrees
#define ST_GOTO_GOAL_6                46 //going forward until bumper is pressed
//release them all
#define ST_DUMP               50 //dumping, plain and simple (timer)
//go back to the dispenser
#define ST_REVISIT_DISPENSER  60 //backing up a bit
#define ST_REVISIT_DISPENSER_1        61 //turning around 180
#define ST_REVISIT_DISPENSER_2        62 //going forward until the green line is passed
#define ST_RETURN_BOOST                       99
#define ST_REVISIT_DISPENSER_3        63 //approaching tape from the left side until T is hit
//ending sequence
#define ST_END 70 //do some sort of LED light show

//tape states
#define ST_APPR                       110
#define ST_APPL                       111
#define ST_PIVOTR             112
#define ST_PIVOTL             113
#define ST_BOPR                       114
#define ST_BOPL                       115
#define ST_GO                 116

//general events
#define EV_NO_EVENT           301
#define EV_ENTRY              302
#define EV_EXIT                       303
#define EV_ERROR              304

//game events
#define EV_ESTOP                      201
#define EV_FLASH                      202
#define EV_BEACON                     203      //found a verified beacon
#define EV_FAKE_BEACON        204      //verification resulted in a wrong beacon
#define EV_BEACON_BLIP        205      //found what might be a beacon
#define EV_BUMP                               206
#define EV_GAME_OVER          207
#define EV_STATE_TIMEUP               208
#define EV_NEXT                               209
#define EV_ALREADY_MIRRORED 210

//tape events
#define EV_RLC        0x07
#define EV_RL         0x03
#define EV_RC         0x05
#define EV_LC         0x06
#define EV_R          0x01
#define EV_L          0x02
#define EV_C          0x04
#define EV_NOTAPE   0
#define EV_ATT                108

//helpers
#define AD_PIN_ASSIGN  "AAAAAAAA" //All analog inputs
#define BATT_V1               77 //0 to 1023 = 0 to 5v. 896=7.76v This value scales AD in to tenths of a
volt
#define BATT_AD1              896
#define BATT_V2               161 //908=16.1v
#define BATT_AD2              908
```

```
#define R_MOTOR       1 //use to ID the right motor
#define L_MOTOR       0 //use to ID the left motor
#define BOTH_MOTORS 2 //makes both motors do their thing
#define FORWARD       1 //motor pushes the robot forward
#define BACKWARD      0 //motor pushes the robot backward
#define RIGHT         1
#define LEFT          0

//dc motor PWM
#define PRESCALER 2            //24Mhz clock / 2 = 12 MHz
#define POSTSCALER 3    //12 MHz / (3*2) = 2000 kHz
#define MS (24000/(PRESCALER*POSTSCALER*2)) // =1000 defines the number of ticks in a microsecond
#define MOTOR_PWM_PERIOD 100 //(MS/10) //MS/10 = 20kHz
#define DEFAULT_MOTOR_DUTY (MOTOR_PWM_PERIOD) //default duty cycle = 100%

//ball dispense
#define PRESCALER_B 32        //24Mhz clock / 16 = 1500 Khz
#define POSTSCALER_B 75               //24Mhz / 16 / (2*75) = 10 Khz
#define MS_B (24000/(PRESCALER_B*POSTSCALER_B*2)) // = 10 defines the number of ticks in a ms
#define SERVO_PWM_PERIOD 100 //(MS_B/100) = 100Hz
#define SERVO_OPEN_DUTY 13
#define SERVO_CLOSED_DUTY 3    //used to be 2

//beacon
#define MS_T 1500
#define PERIOD_T (1*MS_T)
#define PERIOD_THRESH_HIGH 1100
#define PERIOD_THRESH_LOW 900
#define DUTY_THRESH_HIGH 80
#define DUTY_THRESH_LOW 20
#define MIN_PULSE_WIDTH 200
#define MAX_PULSE_WIDTH 1000

//beacon decision making
#define DUTY_HISTORY_LENGTH                  1 //make this lower to increase our beacon-finding
sensitivity
#define DUTY_TOLERANCE                                     10
#define BEACON_VERIFY_CYCLES                 20 //check twenty times to verify we're at our beacon
#define BEACON_VERIFY_ERRORS_ALLOWED  5 //how many times can we blow it
#define BEACON_TO_FIND                                     50 //which beacon do we want to find?

//Debug
#define NUM_LCD_SCREENS               2
#define NUM_VARS_ON_FIRST_SCREEN      4
#define NUM_VARS_ON_SECOND_SCREEN     2
#define MAX_DEBUG_VARS                10 //MAKE SURE THIS IS BIGGER THAN THE NUMBER OF DEBUG VARS
#define DEBUG_GAME_TIME               0
#define DEBUG_EVENT          1
#define DEBUG_GAME_STATE     2
#define DEBUG_TAPE_STATE     3
#define DEBUG_BATTV1         4
#define DEBUG_BATTV2         5

////////////////////////////////////
//       LCD ARRAY ADDRESSES      //
////////////////////////////////////
#define LINE_1_POS_1 0x80
#define LINE_1_POS_2 0x81
#define LINE_1_POS_3 0x82
#define LINE_1_POS_4 0x83
#define LINE_1_POS_5 0x84
#define LINE_1_POS_6 0x85
#define LINE_1_POS_7 0x86
#define LINE_1_POS_8 0x87
#define LINE_1_POS_9 0x88
#define LINE_1_POS_10 0x89
#define LINE_1_POS_11 0x8A
#define LINE_1_POS_12 0x8B
#define LINE_1_POS_13 0x8C
#define LINE_1_POS_14 0x8D
#define LINE_1_POS_15 0x8E
#define LINE_1_POS_16 0x8F
#define LINE_1_POS_17 0x90
#define LINE_1_POS_18 0x91
#define LINE_1_POS_19 0x92
#define LINE_1_POS_20 0x93

#define LINE_2_POS_1 0xC0
#define LINE_2_POS_2 0xC1
#define LINE_2_POS_3 0xC2
#define LINE_2_POS_4 0xC3
```

```
#define LINE_2_POS_7 0xC6
#define LINE_2_POS_8 0xC7
#define LINE_2_POS_9 0xC8
#define LINE_2_POS_10 0xC9
#define LINE_2_POS_11 0xCA
#define LINE_2_POS_12 0xCB
#define LINE_2_POS_13 0xCC
#define LINE_2_POS_14 0xCD
#define LINE_2_POS_15 0xCE
#define LINE_2_POS_16 0xCF
#define LINE_2_POS_17 0xD0
#define LINE_2_POS_18 0xD1
#define LINE_2_POS_19 0xD2
#define LINE_2_POS_20 0xD3

#define LINE_3_POS_1 0x94
#define LINE_3_POS_2 0x95
#define LINE_3_POS_3 0x96
#define LINE_3_POS_4 0x97
#define LINE_3_POS_5 0x98
#define LINE_3_POS_6 0x99
#define LINE_3_POS_7 0x9A
#define LINE_3_POS_8 0x9B
#define LINE_3_POS_9 0x9C
#define LINE_3_POS_10 0x9D
#define LINE_3_POS_11 0x9E
#define LINE_3_POS_12 0x9F
#define LINE_3_POS_13 0xA0
#define LINE_3_POS_14 0xA1
#define LINE_3_POS_15 0xA2
#define LINE_3_POS_16 0xA3
#define LINE_3_POS_17 0xA4
#define LINE_3_POS_18 0xA5
#define LINE_3_POS_19 0xA6
#define LINE_3_POS_20 0xA7

#define LINE_4_POS_1 0xD4
#define LINE_4_POS_2 0xD5
#define LINE_4_POS_3 0xD6
#define LINE_4_POS_4 0xD7
#define LINE_4_POS_5 0xD8
#define LINE_4_POS_6 0xD9
#define LINE_4_POS_7 0xDA
#define LINE_4_POS_8 0xDB
#define LINE_4_POS_9 0xDC
#define LINE_4_POS_10 0xDD
#define LINE_4_POS_11 0xDE
#define LINE_4_POS_12 0xDF
#define LINE_4_POS_13 0xE0
#define LINE_4_POS_14 0xE1
#define LINE_4_POS_15 0xE2
#define LINE_4_POS_16 0xE3
#define LINE_4_POS_17 0xE4
#define LINE_4_POS_18 0xE5
#define LINE_4_POS_19 0xE6
#define LINE_4_POS_20 0xE7

#endif
```

driving.h

```
#ifndef DRIVING
#define DRIVING

//FUNCTION PROTOTYPES
void InitPWM(void);
void SetMotor(char motorID, char direction, char duty);
void Move(char direction, char duty); //moves the vehicle in a given direction
void Turn(char direction, char duty); //turns the vehicle about its pivot in a given direction
void Stop(void); //stops both motors

#endif DRIVING
```

driving.c

```
//----------- driving.c ------------//
//-- code courtesy of BurgerStache -//
//------------------////-----------//

#include "headers.h"

//set MirrorFlag to TRUE if we're mirroring (playing on side B)
extern char MirrorFlag;

//Initializes the PWM subystem on the E128
void InitPWM(void){
	//Initialize the clock
	PWMSCLA = POSTSCALER; //scale the A clock by / (3*2)
	PWMPRCLK |= 1; //use clock A with M/4 scalar (write to bit 1)

	//Initialize PWM for motor 1 (T0)
	PWME |= BIT0HI; //enable PWM on bit 0
	MODRR |= BIT0HI; //map T0 to PWM
	PWMCLK |= BIT0HI; //use SA (scaled clock)
	PWMPOL |= BIT0HI; //select the PWM polarity. 1 = output initially high
	PWMPER0 = MOTOR_PWM_PERIOD; //contains the count of the total number of cycles on clock A or SA 
that will constitute the total period for PWM channel 0
	PWMDTY0 = DEFAULT_MOTOR_DUTY; //contains the count of the total number of cycles on either clock 
A or SA that will constitute the active period for PWM channel 0

	//Initialize PWM for motor 2 (T1)
	PWME |= BIT1HI; //enable PWM on bit 1
	MODRR |= BIT1HI; //map T1 to PWM
	PWMCLK |= BIT1HI; //use SA (scaled clock)
	PWMPOL |= BIT1HI; //select the PWM polarity. 1 = output initially high
	PWMPER1 = MOTOR_PWM_PERIOD; //contains the count of the total number of cycles on clock A or SA 
that will constitute the total period for PWM channel 0
	PWMDTY1 = DEFAULT_MOTOR_DUTY; //contains the count of the total number of cycles on either clock 
A or SA that will constitute the active period for PWM channel 0
}

//Sets the duty cycle of the given motor and sets the direction output
//motorID = LEFT or RIGHT
//direction = FORWARD or BACKWARD
//duty = 0 to 100
void SetMotor(char motorID, char direction, char duty){
	//calculate the number of clock ticks to give powers to the motor
	unsigned int dutyTicks;
	dutyTicks = (MOTOR_PWM_PERIOD * duty)/100;

	//check to make sure the parameters are in bounds
	if(duty < 0 || duty > 100){
		printf("ERR: duty out of bounds in SetMotor \r\n");
		return; //failure
	}
	if(!((direction == FORWARD) || (direction == BACKWARD))){
		printf("ERR: direction must be forward or backward \r\n");
		return; //failure
	}
	if(!((motorID == R_MOTOR) || (motorID == L_MOTOR) || (motorID == BOTH_MOTORS))){
		printf("ERR: unknown motorID given \r\n");
		return; //failure
	}

	//Set the direction and PWM based on which motor and which direction are selcted
	if(((motorID == L_MOTOR) && (MirrorFlag == FALSE))|| //non-mirrored, side A
	   ((motorID == R_MOTOR) && (MirrorFlag == TRUE))|| //when mirrored, switch left and right motors
```

```
                        PTU |= BIT5HI; //set direction pin output
                    PWMDTY1 = (char)(MOTOR_PWM_PERIOD-dutyTicks); //set motor PWM registers as prescribed
by the PWM subsystem. Invert duty when direction pin is high.
                        //printf("I'm setting left motor duty to: %d  (INVERSE) \n\r",dutyTicks);
                }else{
                        PTU &= BIT5LO;
                        PWMDTY1 = (char)dutyTicks;
                        //printf("I'm setting left motor duty to: %d \n\r",dutyTicks);
                }
        }
        if(((motorID == R_MOTOR) && (MirrorFlag == FALSE))|| //non-mirrored, side A
           ((motorID == L_MOTOR) && (MirrorFlag == TRUE))|| //when mirrored, switch motors
            (motorID == BOTH_MOTORS)){
                //set direction pin output
                if(direction == FORWARD){
                        PTU |= BIT4HI;
                    PWMDTY0 = (char)(MOTOR_PWM_PERIOD-dutyTicks); //set motor PWM registers as prescribed
by the PWM subsystem. Invert duty when direction pin is high.
                        //printf("I'm setting right motor duty to: %d  (INVERSE) \n\r",dutyTicks);
                }else{
                        PTU &= BIT4LO;
                        PWMDTY0 = (char)dutyTicks;
                        //printf("I'm setting right motor duty to: %d \n\r",dutyTicks);
                }
        }
}

//------------ THESE FUNCTIONS SIMPLIFY OUR MOVING LIFE -----------------//
void Move(char direction, char duty) {
        if(direction == FORWARD){
        //      printf("   Now moving forward\r\n");
        }else{
        //      printf("   Now moving backward\r\n");
        }
        SetMotor(BOTH_MOTORS, direction, duty);
}

void Turn(char direction, char duty){
        if (direction == RIGHT) {
                //printf("   Now turning right\r\n");
                SetMotor(L_MOTOR, FORWARD, duty);
                SetMotor(R_MOTOR, BACKWARD, duty);
        } else if (direction == LEFT) {
                //printf("   Now turning left\r\n");
                SetMotor(L_MOTOR, BACKWARD, duty);
                SetMotor(R_MOTOR, FORWARD, duty);
        }
}

void Stop(void){
        //printf("   Now stopping\r\n");
        SetMotor(BOTH_MOTORS, FORWARD, 0);
}

//------------ TEST FUNCTION -----------------//

#ifdef DRIVING_TEST
void main(void){

        InitAll();

        //check for motor driver error flags
        if(!(PTU & BIT6HI)){   //L motor error flag (low = error)
                printf("ERR: Left motor driver error\r\n");
        }
        if(!(PTU & BIT7HI)){   //R motor error flag (low = error)
                printf("ERR: Right motor driver error\r\n");
        }

        //30 percent power
        Turn(LEFT, 30);
        Wait(2000);
        Turn(RIGHT, 30);
        Wait(2000);
        Move(FORWARD, 30);
        Wait(2000);
        Move(BACKWARD, 30);
        Wait(2000);
        Stop();

        //full fifty percent power
        Turn(RIGHT, 50);
        Wait(2000);
        Move(FORWARD, 50);
        Wait(2000);
        Move(BACKWARD, 50);
        Wait(2000);
        Stop();
        Wait(2000);

}
#endif

#ifdef PWM_TEST
//should see a PWM duty cycle of 70% coming out of both ports
void main(void){
        InitAll();
        Move(FORWARD, 70);
}
#endif
```

headers.h

```c
#ifndef HEADERS
#define HEADERS

//Standard Libraries
#include "ME218_E128.h"
#include <hidef.h>
#include <mc9s12e128.h>
#include <bitdefs.h>

#include "S12eVec.h"          /* vector addresses for interrupts */
#include <S12e128bits.h>    /* bit definitions  */

#include <timerS12.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "ADS12e.h"

//Our libraries
#include "balldispense.h"
#include "beacon.h"
#include "defines.h"
#include "driving.h"
#include "helpers.h"
#include "main.h"
#include "SMGame.h"
#include "SMTape.h"
#include "superLCD.h"

#endif
```

helpers.h

```
#ifndef HELPERS
#define HELPERS

//Function Prototypes
void Wait(int ticks);
void SetTimer(char timer, int ticks);
char CheckTimerExpired(char timer);
void InitializeGameTimer(void);
int GetGameTime(void);
char CheckGameTimerExpired(void);
void CheckBattVoltages(void);
int AngleToTime(int degrees, char duty);
int DistanceToTime(int inches, char duty);

#endif
```

helpers.c

```
//------------ helpers.c -----------//
//-- code courtesey of BurgerStache -//

#include "headers.h"

//declare MODULE LEVEL VARIABLES
static int gameTimeSeconds = 0; //ACTUALLY TENTHS OF SECONDS!

//set MirrorFlag to TRUE if we're mirroring (playing on side B)
extern char MirrorFlag;

//Waits for a number of milliseconds given by ticks (blocking)
void Wait(int ticks){
        //uses timer 0 for blocking WAIT, which is one of 8 possible timers
        TMRS12_InitTimer(0,ticks);
        while(TMRS12_IsTimerExpired(0) != TMRS12_EXPIRED);
}

//sets a timer to count down
//input the length of the timer in MS and the ID of the timer
void SetTimer(char timer, int ticks){
        printf("  Timer %d set with ticks = %d\r\n", timer, ticks);
        TMRS12_InitTimer(timer,ticks);
}

//returns true if the given timer is expired
char CheckTimerExpired(char timer){
        char timex = (TMRS12_IsTimerExpired(timer) == TMRS12_EXPIRED);
        if(timex == TRUE){
                printf("  Timer %d expired\r\n", timer);
                TMRS12_ClearTimerExpired(timer); //clear the timer so we don't keep creating events
        }
        return timex;
}

//because a 120 second timer would overflow the limits of our timers,
//we are keeping track of the game time in one second increments with a counter
void InitializeGameTimer(void){
        //set initial countdown clock for game time
        TMRS12_InitTimer(TIMER_GAME,100); //time for one second
}

//returns the game time in tenths of seconds
int GetGameTime(void){
        return gameTimeSeconds;
}

//check game timer must be continually called in the check function
char CheckGameTimerExpired(void){
        char timex = (TMRS12_IsTimerExpired(TIMER_GAME) == TMRS12_EXPIRED);
        if(timex == TRUE){
                gameTimeSeconds++;
                TMRS12_InitTimer(TIMER_GAME,100); //time for one second
                //SendToDebugRug(gameTimeSeconds, DEBUG_GAME_TIME);                          //USE FOR
DEBUG
        }
        //if the game is over, clear the timer and return true
        if(gameTimeSeconds >= TIME_GAME) {
```

```
			return TRUE;
		}
		return FALSE;
}

//Prints the battery voltages to the debug rug
//batt1 gives the voltage of the battery connected to ground on port AD1
//batt2 gives the total voltage of the two batteries in series on port AD2
void CheckBattVoltages(void){
		short battAD1,battAD2;
		int battV1,battV2;

		battAD1 = ADS12_ReadADPin(1);
	battAD2 = ADS12_ReadADPin(0);
	battV1 = (((long)battAD1)*BATT_V1)/BATT_AD1; //convert AD value to calibrated voltage
	battV2 = (((long)battAD2)*BATT_V2)/BATT_AD2;

		//send the battery voltages to the debug rug
		//SendToDebugRug(battV1, DEBUG_BATTV1);
		//SendToDebugRug(battV2, DEBUG_BATTV2);
		printf("lower batt V  = %d dV\r\n",battV1);
		printf("batt series V = %d dV\r\n",battV2);
}

//converts an angle to an amount of time to turn
//remember to take duty cycle and battery voltage into account!
//lower batt V  = 78 dV
//batt series V = 157 dV
int AngleToTime(int degrees, char duty){
		//this function is calibrated to work best at 30% duty, but it extrapolates as well
		return (((degrees*10)/(duty-20))*15);  //old = *20, *22, no subtraction from duty
}

//given a number of inches to travel, how many ticks should we run for?
int DistanceToTime(int inches, char duty){
		return (((inches*104)/(duty-20))*15);
}
```

main.h

```
#ifndef MAIN
#define MAIN

//Function Prototypes
void InitAll(void);
int CheckEvents(void);

//private parts
char CheckMirrorSwitch(void);
char CheckBump(void);
static char CheckRTape(void);
static char CheckLTape(void);
static char CheckCTape(void);
static char CheckEstop(void);
static char CheckNext(void);
static char CheckFlash(void);
static char CheckBeacon(void);
static char CheckTape(void);
static char CheckLCDButton(void);

#endif
```

main.c

```
//-------------- main.c -------------//
//-- code courtesey of BurgerStache -//

#include "headers.h"

//DECLARE GLOBAL VARIABLES
//set MirrorFlag to TRUE if we're mirroring (playing on side B)
char MirrorFlag = FALSE;

//MODULE LEVEL VARIABLES
//create persistent variables that hold the previous states of all sensors
///when the state of a sensor changes, then we create an event accordingly
//variables are initialized only the first time this code is run
static char estopState;
static char nextState;
static char flashState;
static char bumpState;
static char tapeState = 0; //a character which holds bits corresponding to the L, C, and R tape
static char newTapeState;
static char lcdState;
static short flashInitialAD; //value of the flash analog input when robot is started

#ifdef REAL_DEAL
void main(void){
        //Initialize all variables
        InitAll();
        //start the master state machine initialization
        StartGameSM();
        //check for and handle events
        while(TRUE){
                RunGameSM(CheckEvents());
        }
}
#endif

//Initialize ports, timers, etc. when the program begins
//Called only once when the program beings
void InitAll(void) {
        printf("Welcome to me.\r\n");

        //Initializes AD ports
    if(ADS12_Init(AD_PIN_ASSIGN) != ADS12_OK)
        printf("ERR: AD Initialization unsuccessful\r\n");

        //Initialize timer
        TMRS12_Init(TMRS12_RATE_1MS);

        //Set port directions
        DDRP = (BIT0HI | BIT1HI); // Set pins 0 and 1 to outputs, the rest are inputs
    DDRT = 0x00; //Set port T to be inputs
        DDRU = 0xFF; //All port U are outputs
        DDRS = 0x00; //Set port S to be inputs
        PTP &= (BIT0LO & BIT1LO);     // Set pins 0 and 1 to low to make sure we are set for the LCD 
initialization

        //Initialize various functionalities
        InitSPI();             //Initialize SPI to talk to our shift register to enable LCD printouts
        InitLCD();             //Initialize our LCD screen by sending a series of commands (must occur 
after SPI is initialized)
        InitPWM();     //Initialize PWM (in driving module)
        InitServoPWM(); //Initialize ball dispensing servo PWM
        InitBEACON();  //Initialize input captures for measuring Duty Cycle of signal (in beacon module)

        //Initialize initial state of actuators
        lcdState = CheckLCDButton();
        estopState = CheckEstop();
        nextState = CheckNext();
        flashState = CheckFlash();
        bumpState = CheckBump();
        tapeState = CheckTape(); //a character which holds bits corresponding to the L, C, and R tape

        //set initial flash AD value
        flashInitialAD = ADS12_ReadADPin(7);

        //Check and print battery voltages
        CheckBattVoltages();

        //Set motors to begin at a stop
        Stop();
}

//main event checker for the state machines
int CheckEvents(void)
{
        int CurrentEvent = EV_NO_EVENT;
        int KeyStroke;

        //Check for events
        //These events should be arranged in order of priority, since
        //if two events are encountered at once, only process the first one so the second is processed 
the next time around

        //check timers
        if(CheckTimerExpired(TIMER_STATE)){
                CurrentEvent = EV_STATE_TIMEUP;
        }
        else if(CheckGameTimerExpired()){
                CurrentEvent = EV_ESTOP;
        }
        else if(estopState != CheckEstop()){
                if(estopState == 0){
                        estopState = 1; //toggle the state variable
                        CurrentEvent = EV_ESTOP;
                } else
                        estopState = 0;
        }
        else if(nextState != CheckNext()){
                if(nextState == 0){
                        nextState = 1; //toggle the state variable
                        CurrentEvent = EV_NEXT;
                } else
                        nextState = 0;
        }
        else if(bumpState != CheckBump()){
                if(bumpState == 0){
                        bumpState = 1;//toggle the state variable
                        CurrentEvent = EV_BUMP;
                } else
                        bumpState = 0;
        }
        else if(tapeState != CheckTape()){
                tapeState = newTapeState;
                //if(tapeState != 0) //ignore "no-tape" events
                        CurrentEvent = tapeState;//tapeState holds all the bit values, we return the event 
as #defined
        }
        else if(flashState != CheckFlash()){
                if(flashState == 0){
                        flashState = 1;//toggle the state variable
                        CurrentEvent = EV_FLASH;
                } else
                        flashState = 0;
        }
        if(lcdState != CheckLCDButton()){
                if(lcdState == 0){
                        lcdState = 1;//toggle the state variable
                        PrintDebugToLCD();
                        ToggleLCDScreen();
                } else
                        lcdState = 0;
        }

        #ifdef SIMULATE_EVENTS //this allows us to simulate our state machine using keyboard presses
        if (kbhit() != 0){ //there was a key pressed
                KeyStroke = getchar();
                switch(toupper(KeyStroke)){
                        case 'E' : CurrentEvent = EV_ESTOP; break;
                        case 'F' : CurrentEvent = EV_FLASH; break;
                        case 'T' : CurrentEvent = EV_BEACON_BLIP; break;
                        case 'R' : CurrentEvent = EV_BEACON; break;
                        case 'B' : CurrentEvent = EV_BUMP; break;
                        case 'N' : CurrentEvent = EV_NEXT; break;
                        case 'W' : CurrentEvent = EV_RLC; break;
                        case 'S' : CurrentEvent = EV_RL; break;
                        case 'D' : CurrentEvent = EV_RC; break;
                        case 'A' : CurrentEvent = EV_LC; break;
                        case 'C' : CurrentEvent = EV_R; break;
                        case 'Z' : CurrentEvent = EV_L; break;
                        case 'X' : CurrentEvent = EV_C; break;
                }
        }
        #endif

        //update displays
        SendToDebugRug(CurrentEvent, DEBUG_EVENT);

        //print debug rug every time there is an event
        //ALSO, skip any tape events, so the term isn't swamped
        if((CurrentEvent != EV_NO_EVENT) && (CurrentEvent > 7)) {

                PrintDebugToTerm();
        }

        return(CurrentEvent);
}


static char CheckTape(void) {
        newTapeState = 0;
        if (CheckRTape()) newTapeState |= BIT0HI;
        if (CheckLTape()) newTapeState |= BIT1HI;
        if (CheckCTape()) newTapeState |= BIT2HI;
        //we use bit operators so as not to disturb the tapeState variable if there are no changes

        //CTAPE LTAPE   RTAPE
        //0             0               0               NO_EVENT        0x00
        //0             0               1               EV_R            0x01
        //0             1               0               EV_L            0x02
        //0             1               1               EV_RL           0x03
        //1             0               0               EV_C            0x04
        //1             0               1               EV_RC           0x05
        //1             1               0               EV_LC           0x06
        //1             1               1               EV_RLC          0x07
        return newTapeState;
}

//check for a bumper press. Returns true if depressed
char CheckBump(void)
{
        if(PTP & BIT3HI) return TRUE;
        else return FALSE;
}

static char CheckRTape(void)
{
        short Tape_Level;

        //Part 1 of 2 replicas of "silly mirror code"
        if(MirrorFlag == FALSE)
                Tape_Level = ADS12_ReadADPin(RIGHT_TAPE);
        else
                Tape_Level = ADS12_ReadADPin(LEFT_TAPE);

        //printf("RTapeADLevel = %d \r\n", Tape_Level);
        else return FALSE;
}

static char CheckLTape(void)
{
        short Tape_Level;

        //Here's some silly mirror code
        if(MirrorFlag == FALSE)
                Tape_Level = ADS12_ReadADPin(LEFT_TAPE);
        else
                Tape_Level = ADS12_ReadADPin(RIGHT_TAPE);

        //printf("LTapeADLevel = %d \r\n", Tape_Level);

        if((Tape_Level>TAPE_THRESH_LOW) && (Tape_Level<TAPE_THRESH_HIGH)) return TRUE;
        else return FALSE;
}

static char CheckCTape(void)
{
        short Tape_Level;
        Tape_Level = ADS12_ReadADPin(CENTER_TAPE);

        //printf("CTapeADLevel = %d \r\n", Tape_Level);

        if((Tape_Level>TAPE_THRESH_LOW) && (Tape_Level<TAPE_THRESH_HIGH)) return TRUE;
        else return FALSE;
}

char CheckMirrorSwitch(void){
        if(PTP & BIT4HI) return TRUE;
        else return FALSE;
}

static char CheckEstop(void)
{
        if(PTS & BIT3HI) return TRUE;
        else return FALSE;
}

static char CheckNext(void)
{
        if(PTS & BIT2HI) return TRUE;
        else return FALSE;
}

static char CheckFlash(void)
{
        if(ADS12_ReadADPin(7) > (flashInitialAD + FLASH_TOLERANCE)) return TRUE;
        else return FALSE;
}

static char CheckLCDButton(void) {
        if(PTP & BIT5HI) return TRUE;
        else return FALSE;
}

//check all the sensors to see if they're connected
#ifdef SENSORS_TEST
void main(void){
        InitAll();
        while(TRUE){
                printf("RTape = %d\r\n", CheckRTape());
                printf("LTape = %d\r\n", CheckLTape());
                printf("CTape = %d\r\n", CheckCTape());
                printf("MirrorSwitch = %d\r\n", CheckMirrorSwitch());
                printf("Estop = %d\r\n", CheckEstop());
                printf("Next = %d\r\n", CheckNext());
                printf("Flash = %d\r\n", CheckFlash());
                printf("Bump = %d\r\n", CheckBump());
                printf("LCDButton = %d\r\n", CheckLCDButton());
                printf("\r\n");
                Wait(250);
        }
}
#endif


#ifdef LINE_FOLLOW_TEST
void main(void){
        StartTapeSM(RIGHT);
        while(TRUE){
                RunTapeSM(CheckEvents());
                if(QueryTapeSM() == EV_ATT){
                        Stop();
                        printf("Tape sensing done\r\n");
                }
        }
}
#endif

#ifdef CALIBRATION_MOVE_TEST
void main(void){
        InitAll();
        printf("Starting a calibrated move test\r\n");
        while(TRUE){    //repeat
                Move(FORWARD, DUTY_DEFAULT_MOVE);
                Wait(DistanceToTime(24, DUTY_DEFAULT_MOVE)); //go two feet
                Stop();
                Wait(2000);
                Turn(RIGHT, DUTY_DEFAULT_TURN); //turn around
                Wait(AngleToTime(180, DUTY_DEFAULT_TURN));
                Stop();
                Wait(2000);
        }
}
#endif
```

SMGame.h

```c
#ifndef SMGAME
#define SMGAME

#include "headers.h"

// Public Function Prototypes
int RunGameSM(int CurrentEvent);
void StartGameSM (void);
int QueryGameSM (void);

//Private function prototypes
static int During_START(int Event);
static int During_BEACON(int Event);
static int During_BEACON_1(int Event);
static int During_BEACON_2(int Event);
static int During_BEACON_3(int Event);
static int During_CLEAR_SQUARE(int Event);
static int During_MIRROR_CHECK(int Event);
static int During_MEET_DISPENSER(int Event);
static int During_MEET_DISPENSER_1(int Event);
static int During_MEET_DISPENSER_2(int Event);
static int During_COLLECT_BALLS(int Event);
static int During_COLLECT_BALLS_1(int Event);
static int During_COLLECT_BALLS_2(int Event);
static int During_GOTO_GOAL(int Event);
static int During_GOTO_GOAL_1(int Event);
static int During_GOAL_BOOST(int Event);
static int During_GOTO_GOAL_2(int Event);
static int During_GOTO_GOAL_4(int Event);
static int During_GOTO_GOAL_5(int Event);
static int During_GOTO_GOAL_6(int Event);
static int During_DUMP(int Event);
static int During_REVISIT_DISPENSER(int Event);
static int During_REVISIT_DISPENSER_1(int Event);
static int During_REVISIT_DISPENSER_2(int Event);
static int During_RETURN_BOOST(int Event);
static int During_REVISIT_DISPENSER_3(int Event);
static int During_END(int Event);
#endif
```

SMGame.c

```c
#include "headers.h"

//set MirrorFlag to TRUE if we're mirroring (playing on side B)
extern char MirrorFlag;

/*---------------------------- Module Variables ---------------------------*/
// everybody needs a state variable, you may need others as well
static int CurrentState;
static char BallsCarried; //holds the number of balls in the robot at the moment
static char BallsDelivered; //total balls delivered to the goal (or not)

/*------------------------------ Module Code ------------------------------*/
// make recursive call warning into info
#pragma MESSAGE INFORMATION C1855
int RunGameSM( int CurrentEvent )
{

   unsigned char MakeTransition = FALSE;/* are we making a state transition? */
   int NextState = CurrentState;

   //send the current state to the debugger
   SendToDebugRug(CurrentState, DEBUG_GAME_STATE);

   switch ( CurrentState )
   {

       case ST_START :       // waiting for electronic flash
         // Execute During function for state one. EV_ENTRY & EV_EXIT are
         // processed here
         CurrentEvent = During_START(CurrentEvent);
         //process any events
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
```

```
                          NextState = ST_BEACON;
                          MakeTransition = TRUE;
            break;
         case EV_NEXT : //If we're skipping to the next stage
                          Stop(); //YES. WE KNOW THIS IS POOR FOR. OH WELL.
                          Wait(1000);
                          NextState = ST_BEACON;
                          MakeTransition = TRUE;
            break;
         case EV_ESTOP : //If we're emergency stopping
                          NextState = ST_END;
                          MakeTransition = TRUE;
            break;
                 }
  }
  break;
case ST_BEACON :       //scan left until you see a blip
  CurrentEvent = During_BEACON(CurrentEvent);
  if ( CurrentEvent != EV_NO_EVENT )
  {
     switch (CurrentEvent)
     {
       case EV_BEACON_BLIP : //If we found a potential beacon
                          NextState = ST_BEACON_1;
                          MakeTransition = TRUE;
            break;
         case EV_NEXT : //If we're skipping to the next stage
                          Stop();
                          Wait(1000);
                          NextState = ST_BEACON_1;
                          MakeTransition = TRUE;
            break;
         case EV_ESTOP : //If we're emergency stopping
                          NextState = ST_END;
                          MakeTransition = TRUE;
            break;
      }
  }
  break;
case ST_BEACON_1 :     //turning back to correct for overshoot
  CurrentEvent = During_BEACON_1(CurrentEvent);
  if ( CurrentEvent != EV_NO_EVENT )
  {
     switch (CurrentEvent)
     {
       case EV_STATE_TIMEUP : //If we're done overshooting, says the timer
                          NextState = ST_BEACON_2;
                          MakeTransition = TRUE;
            break;
         case EV_NEXT : //If we're skipping to the next stage
                          Stop();
                          Wait(1000);
                          NextState = ST_BEACON_2;
                          MakeTransition = TRUE;
            break;
         case EV_ESTOP : //If we're emergency stopping
                          NextState = ST_END;
                          MakeTransition = TRUE;
            break;
      }
  }
  break;
 case ST_BEACON_2 :    //verify beacon
  CurrentEvent = During_BEACON_2(CurrentEvent);
  if ( CurrentEvent != EV_NO_EVENT )
  {
     switch (CurrentEvent)
     {
       case EV_FAKE_BEACON : //If it's not the beacon we want
                          NextState = ST_BEACON_3;
                          MakeTransition = TRUE;
            break;
       case EV_BEACON : //we found it!
                          NextState = ST_CLEAR_SQUARE;
                          MakeTransition = TRUE;
            break;
       case EV_NEXT : //If we're skipping to the next stage
                          Stop();
                          Wait(1000);
                          NextState = ST_CLEAR_SQUARE;
                          MakeTransition = TRUE;
```

```
                        NextState = ST_END;
                        MakeTransition = TRUE;
          break;
     }
 }
 break;
case ST_BEACON_3 :     //pass the fake beacon
 CurrentEvent = During_BEACON_3(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
      case EV_STATE_TIMEUP :
                        NextState = ST_BEACON;
                        MakeTransition = TRUE;
          break;
      case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_BEACON;
                        MakeTransition = TRUE;
          break;
       case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
          break;
     }
 }
 break;

case ST_CLEAR_SQUARE : //get off the green square
 CurrentEvent = During_CLEAR_SQUARE(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_STATE_TIMEUP :
          NextState = ST_MIRROR_CHECK;
          MakeTransition = TRUE;
          break;
       case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_MIRROR_CHECK;
                        MakeTransition = TRUE;
          break;
       case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
          break;
     }
 }
 break;

        case ST_MIRROR_CHECK :       //hit L or R so we know what side we're on
 CurrentEvent = During_MIRROR_CHECK(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
      case EV_L :
                 //if we're on side B, this code gets executed
          MirrorFlag = TRUE;

          printf("   ON SIDE B\r\n");
          NextState = ST_MEET_DISPENSER;
          MakeTransition = TRUE;
          break;
                 case EV_R :
                 //if we're on side A, this code gets executed
          MirrorFlag = FALSE;
          printf("   ON SIDE A\r\n");
          NextState = ST_MEET_DISPENSER;
          MakeTransition = TRUE;
          break;
       case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_MEET_DISPENSER;
                        MakeTransition = TRUE;
          break;
```

```
                            MakeTransition = TRUE;
                 break;
            }
        }
        break;

       case ST_MEET_DISPENSER :      //going forward until the first black line is passed
        CurrentEvent = During_MEET_DISPENSER(CurrentEvent);
        if ( CurrentEvent != EV_NO_EVENT )
        {
           switch (CurrentEvent)
           {
             case EV_C :
                 NextState = ST_MEET_DISPENSER_1;
                 MakeTransition = TRUE;
                 break;
             case EV_LC :
                 NextState = ST_MEET_DISPENSER_1;
                 MakeTransition = TRUE;
                 break;
                        case EV_RC :
                 NextState = ST_MEET_DISPENSER_1;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                            Stop();
                            Wait(1000);
                            NextState = ST_MEET_DISPENSER_1;
                            MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                            NextState = ST_END;
                            MakeTransition = TRUE;
                 break;
            }
        }
        break;
       case ST_MEET_DISPENSER_1 : //correction factor depending on side A or B
        CurrentEvent = During_MEET_DISPENSER_1(CurrentEvent);
        if ( CurrentEvent != EV_NO_EVENT )
        {
           switch (CurrentEvent)
           {
              case EV_STATE_TIMEUP :
                 NextState = ST_MEET_DISPENSER_2;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                            Stop();
                            Wait(1000);
                            NextState = ST_MEET_DISPENSER_2;
                            MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                            NextState = ST_END;
                            MakeTransition = TRUE;
                 break;
            }
        }
        break;
        case ST_MEET_DISPENSER_2 :           //approaching tape from the right until T is hit (nested
tape SM)
        CurrentEvent = During_MEET_DISPENSER_2(CurrentEvent);
        if ( CurrentEvent != EV_NO_EVENT )
        {
           switch (CurrentEvent)
           {
              case EV_ATT :  //if we're at the T
                 NextState = ST_COLLECT_BALLS;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                            Stop();
                            Wait(1000);
                            NextState = ST_COLLECT_BALLS;
                            MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                            NextState = ST_END;
                            MakeTransition = TRUE;
                 break;
```

```
         break;
        case ST_COLLECT_BALLS :       //moving forward until you hit the front bumper OR if bumper is
already hit, create a bump event
         CurrentEvent = During_COLLECT_BALLS(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
               case EV_BUMP :
                  NextState = ST_COLLECT_BALLS_1;
                  MakeTransition = TRUE;
                  break;
               case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_COLLECT_BALLS_1;
                              MakeTransition = TRUE;
                  break;
               case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                  break;
             }
         }
         break;
        case ST_COLLECT_BALLS_1 : //sitting still until a ball is collected
         CurrentEvent = During_COLLECT_BALLS_1(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
               case EV_STATE_TIMEUP :
                  NextState = ST_COLLECT_BALLS_2;
                  MakeTransition = TRUE;
                  break;
               case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_COLLECT_BALLS_2;
                              MakeTransition = TRUE;
                  break;
               case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                  break;
             }
         }
         break;
        case ST_COLLECT_BALLS_2 : //moving backwards
         CurrentEvent = During_COLLECT_BALLS_2(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
               case EV_STATE_TIMEUP :
                  if(BallsCarried < 5)
                       NextState = ST_COLLECT_BALLS;
                  else
                       NextState = ST_GOTO_GOAL;

                  MakeTransition = TRUE;
                  break;
               case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_GOTO_GOAL;
                              MakeTransition = TRUE;
                  break;
               case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                  break;
             }
         }
         break;
        case ST_GOTO_GOAL :    //turning 90 degrees left (or right)
         CurrentEvent = During_GOTO_GOAL(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_GOTO_GOAL_1;
                              MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                 break;
            }
         }
         break;
               case ST_GOTO_GOAL_1: //GO FORWARD A LITTLE BIT UNTIL WE'RE OFF THE TAPE, THEN ENABLE TAPE
SENSING
         CurrentEvent = During_GOTO_GOAL_1(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
              case EV_STATE_TIMEUP :
                 NextState = ST_GOAL_BOOST;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_GOAL_BOOST;
                              MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                 break;
            }
         }
         break;
        case ST_GOAL_BOOST: //let's go to the goal. FAST!
         CurrentEvent = During_GOAL_BOOST(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
              case EV_STATE_TIMEUP :
                 NextState = ST_GOTO_GOAL_2;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_GOTO_GOAL_2;
                              MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
                 break;
            }
         }
         break;
               case ST_GOTO_GOAL_2: //GOING FWD UNTIL TAPE IS HIT (approach from left)
         CurrentEvent = During_GOTO_GOAL_2(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
              case EV_ATT :
                 NextState = ST_GOTO_GOAL_4;
                 MakeTransition = TRUE;
                 break;
              case EV_NEXT : //If we're skipping to the next stage
                              Stop();
                              Wait(1000);
                              NextState = ST_GOTO_GOAL_4;
                              MakeTransition = TRUE;
                 break;
              case EV_ESTOP : //If we're emergency stopping
                              NextState = ST_END;
                              MakeTransition = TRUE;
```

```
 }
 break;

case ST_GOTO_GOAL_4 : //backing up a bit
 CurrentEvent = During_GOTO_GOAL_4(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_STATE_TIMEUP :
          NextState = ST_GOTO_GOAL_5;
          MakeTransition = TRUE;
          break;
       case EV_NEXT : //If we're skipping to the next stage
                         Stop();
                         Wait(1000);
                         NextState = ST_GOTO_GOAL_5;
                         MakeTransition = TRUE;
          break;
       case EV_ESTOP : //If we're emergency stopping
                         NextState = ST_END;
                         MakeTransition = TRUE;
          break;
     }
 }
 break;
case ST_GOTO_GOAL_5 :  //turning R (or L) 90 degrees
 CurrentEvent = During_GOTO_GOAL_5(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_STATE_TIMEUP :
          NextState = ST_GOTO_GOAL_6;
          MakeTransition = TRUE;
          break;
       case EV_NEXT : //If we're skipping to the next stage
                         Stop();
                         Wait(1000);
                         NextState = ST_GOTO_GOAL_6;
                         MakeTransition = TRUE;
          break;
       case EV_ESTOP : //If we're emergency stopping
                         NextState = ST_END;
                         MakeTransition = TRUE;
          break;
     }
 }
 break;
case ST_GOTO_GOAL_6 : //going forward until bumper is pressed
 CurrentEvent = During_GOTO_GOAL_6(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_BUMP :
          NextState = ST_DUMP;
          MakeTransition = TRUE;
          break;
       case EV_NEXT : //If we're skipping to the next stage
                         Stop();
                         Wait(1000);
                         NextState = ST_DUMP;
                         MakeTransition = TRUE;
          break;
       case EV_ESTOP : //If we're emergency stopping
                         NextState = ST_END;
                         MakeTransition = TRUE;
          break;
     }
 }
 break;
case ST_DUMP : //dump till you're pooped
 CurrentEvent = During_DUMP(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_STATE_TIMEUP :
          NextState = ST_REVISIT_DISPENSER;
          MakeTransition = TRUE;
```

```
                        Stop();
                        Wait(1000);
                        NextState = ST_REVISIT_DISPENSER;
                        MakeTransition = TRUE;
             break;
          case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
             break;
        }
    }
    break;
   case ST_REVISIT_DISPENSER :   //backing up a bit
    CurrentEvent = During_REVISIT_DISPENSER(CurrentEvent);
    if ( CurrentEvent != EV_NO_EVENT )
    {
       switch (CurrentEvent)
       {
          case EV_STATE_TIMEUP :
             NextState = ST_REVISIT_DISPENSER_1;
             MakeTransition = TRUE;
             break;
          case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_REVISIT_DISPENSER_1;
                        MakeTransition = TRUE;
             break;
          case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
             break;
        }
    }
    break;
   case ST_REVISIT_DISPENSER_1 : //turning around 135 degrees
    CurrentEvent = During_REVISIT_DISPENSER_1(CurrentEvent);
    if ( CurrentEvent != EV_NO_EVENT )
    {
       switch (CurrentEvent)
       {
          case EV_STATE_TIMEUP :
             NextState = ST_REVISIT_DISPENSER_2;
             MakeTransition = TRUE;
             break;
          case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_REVISIT_DISPENSER_2;
                        MakeTransition = TRUE;
             break;
          case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
             break;
        }
    }
    break;
   case ST_REVISIT_DISPENSER_2 : //goin' forward till we pass the green line
    CurrentEvent = During_REVISIT_DISPENSER_2(CurrentEvent);
    if ( CurrentEvent != EV_NO_EVENT )
    {
       switch (CurrentEvent)
       {
          case EV_STATE_TIMEUP :
             NextState = ST_RETURN_BOOST;
             MakeTransition = TRUE;
             break;
          case EV_NEXT : //If we're skipping to the next stage
                        Stop();
                        Wait(1000);
                        NextState = ST_RETURN_BOOST;
                        MakeTransition = TRUE;
             break;
          case EV_ESTOP : //If we're emergency stopping
                        NextState = ST_END;
                        MakeTransition = TRUE;
             break;
        }
    }
    break;
```

```
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
              case EV_STATE_TIMEUP :
                  NextState = ST_REVISIT_DISPENSER_3;
                  MakeTransition = TRUE;
                  break;
               case EV_NEXT : //If we're skipping to the next stage
                                 Stop();
                                 Wait(1000);
                                 NextState = ST_REVISIT_DISPENSER_3;
                                 MakeTransition = TRUE;
                  break;
               case EV_ESTOP : //If we're emergency stopping
                                 NextState = ST_END;
                                 MakeTransition = TRUE;
                  break;
             }
         }
         break;
        case ST_REVISIT_DISPENSER_3 : //approaching tape from the left (or right) side until T is hit
         CurrentEvent = During_REVISIT_DISPENSER_3(CurrentEvent);
         if ( CurrentEvent != EV_NO_EVENT )
         {
            switch (CurrentEvent)
            {
               case EV_ATT :
                  NextState = ST_COLLECT_BALLS;
                  MakeTransition = TRUE;
                  break;
               case EV_NEXT : //If we're skipping to the next stage
                                 Stop();
                                 Wait(1000);
                                 NextState = ST_COLLECT_BALLS;
                                 MakeTransition = TRUE;
                  break;
               case EV_ESTOP : //If we're emergency stopping
                                 NextState = ST_END;
                                 MakeTransition = TRUE;
                  break;
             }
         }
         break;
       case ST_END : //timer's up! we're done
         //do some sort of LED light show
           switch (CurrentEvent)
           {
               case EV_NEXT : //If we're skipping to the next stage
                                 NextState = ST_START;
                                 MakeTransition = TRUE;
                  break;
           }
       break;

    }

    // Check for error events, and printout
    if(CurrentEvent == EV_ERROR)
       printf("EV_ERROR FOUND!\r\n");

    //   If we are making a state transition
    if (MakeTransition == TRUE)
    {
       //   Execute exit function for current state
       RunGameSM(EV_EXIT);
       CurrentState = NextState; //Modify state variable
       //   Execute entry function for new state
       RunGameSM(EV_ENTRY);
     }

     return(CurrentEvent);
}
/****************************************************************************
 Function
     StartGameSM
****************************************************************************/
void StartGameSM ( void )
{
   CurrentState = ST_START;
   // call the entry function (if any) for the ENTRY_STATE
```

```
int QueryGameSM ( void )
{
   return(CurrentState);
}

/***************************************************************************
 private functions
 ***************************************************************************/

//WAITING FOR ELECTRONIC FLASH
static int During_START(int Event){
    // process EV_ENTRY & EV_EXIT events
    //these events must all occur when the game starts over
    if ( Event == EV_ENTRY)
    {
               //Initialize our mirror flag to false
               MirrorFlag = FALSE;

        //clean up all variables for next run!
        BallsCarried = 0;
        BallsDelivered = 0;

    }else if ( Event == EV_EXIT)
    {
        //Initialize game timer as soon as the flash is recognized
               InitializeGameTimer();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//scan L toward the beacon, stop when we see a beacon blip on our radar
static int During_BEACON(int Event){

    //temp variable to store beacon blip
    int blipDuty;

    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
               Turn(LEFT, DUTY_BEACON_SEEK);
    }else if ( Event == EV_EXIT){
               Stop();
    }else
    // do the 'during' function for this state
    {
        blipDuty = CheckForBeacon();

        //check for a beacon blip event, which only matters to us at this stage
        if((blipDuty != 0) && (GetGameTime() > 5)){ //if we see any sort of signal
               //printf("   BEACON BLIP DUTY = %d\r\n", blipDuty);
               return EV_BEACON_BLIP;
        }
    }
    return Event;
}

//turn back LEFT a small ammount to boost the robot into the center of the beacon
static int During_BEACON_1(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                SetTimer(TIMER_STATE, AngleToTime(ANGLE_BEACON_OVERSHOOT, DUTY_DEFAULT_TURN));
                Turn(LEFT, DUTY_BEACON_SEEK); //notice this is in the same direction of travel
    }else if ( Event == EV_EXIT){
               Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}


//verify that this is the beacon we're looking for
static int During_BEACON_2(int Event){
    //create a variable to count how many verification cycles we have performed
    static int beaconVerifyCounter = 0;
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
    }else if ( Event == EV_EXIT){
    }else
    // do the 'during' function for this state
    {
       //otherwise, keep checking (repeat state)
       if(CheckForSpecificBeacon(BEACON_TO_FIND)){
              beaconVerifyCounter++;
              printf("   THIS IS BEAC0N BLIP NUMBER %d\r\n", beaconVerifyCounter);
       } else{
       //this is not a verified beacon. Count up one more error
           beaconVerifyErrors++;
           printf("   THIS IS FAKE BEAC0N ERROR NUMBER %d\r\n", beaconVerifyErrors);
       }

       //if we're done verifying, we've found our beacon!
       if(beaconVerifyCounter >= BEACON_VERIFY_CYCLES){
              printf("   THIS IS A VERIFIED BEAC0N\r\n");
              beaconVerifyCounter = 0; //reset for next time
              beaconVerifyErrors = 0; //reset for next time
              return EV_BEACON;
       }
       if(beaconVerifyErrors >= BEACON_VERIFY_ERRORS_ALLOWED){
              printf("   THIS IS A FAKE BEAC0N\r\n");
              beaconVerifyCounter = 0;
              beaconVerifyErrors = 0;
              return EV_FAKE_BEACON;
       }
    }
    return Event;
}

//move past the false prophet (beacon)
static int During_BEACON_3(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY){
        SetTimer(TIMER_STATE, AngleToTime(ANGLE_PASS_FAKE_BEACON, DUTY_DEFAULT_TURN));
        Turn(LEFT, DUTY_DEFAULT_TURN);
    }else if ( Event == EV_EXIT){
       Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//GO FORWARD A LITTLE BIT TO GET OFF THE SQUARE
static int During_CLEAR_SQUARE(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
              //Init + Start state timer
              SetTimer(TIMER_STATE, DistanceToTime(DIST_CLEAR_SQUARE, DUTY_DEFAULT_MOVE));
              Move(FORWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
              Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//check L and R to see what side we're on
static int During_MIRROR_CHECK(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
       Move(FORWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
       Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
```

```
//GOING FORWARD UNTIL WE PASS THE BLACK LINE
static int During_MEET_DISPENSER(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
       Move(FORWARD, DUTY_FIRST_APP);
    }else if ( Event == EV_EXIT)
    {
       Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//TURN FUDGE FACTOR (DEPENDING ON SIDE A or B)
static int During_MEET_DISPENSER_1(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                if(MirrorFlag == FALSE){ //SIDE A
                        SetTimer(TIMER_STATE, AngleToTime(35, DUTY_DEFAULT_TURN)); //old = 30, older = 35,
really old = 43
                        Turn(LEFT, DUTY_DEFAULT_TURN);
                } else {        //SIDE B
                        SetTimer(TIMER_STATE, AngleToTime(18, DUTY_DEFAULT_TURN)); //old = 17
                    Turn(LEFT, DUTY_DEFAULT_TURN); //actually right
                }

    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//APPROACHING TAPE FROM THE RIGHT UNTIL THE T IS HIT
static int During_MEET_DISPENSER_2(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        StartTapeSM(RIGHT);
    }else if ( Event == EV_EXIT){
        Event = RunTapeSM(Event);
    }else{
    // do the 'during' function for this state
        Event = RunTapeSM(Event);
    }
    return Event;
}

//MOVING FORWARD TILL WE HIT THE BUMPER
static int During_COLLECT_BALLS(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //if somehow we're already against the bumper, move on
                if(CheckBump() == TRUE)
                        return EV_BUMP;

                Move(FORWARD, DUTY_GENTLEBUMP);
    }else if ( Event == EV_EXIT){
                Stop();
    }else{
    // do the 'during' function for this state

    }
    return Event;
}

//SITTING STILL TILL WE GET ONE BALL
static int During_COLLECT_BALLS_1(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
        BallsCarried++;
                printf("  Balls carried = %d\r\n", BallsCarried);
    }else{
        // do the 'during' function for this state
    }
    return Event;
}

//BACK UP
static int During_COLLECT_BALLS_2(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                SetTimer(TIMER_STATE, DistanceToTime(DIST_BACKUP, DUTY_GENTLEBUMP));
                Move(BACKWARD, DUTY_GENTLEBUMP);
        }else if ( Event == EV_EXIT){
                Stop();
    }else{
        // do the 'during' function for this state
    }
    return Event;
}

//TURN 90 DEGREES LEFT (or right for side B)
static int During_GOTO_GOAL(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                if(MirrorFlag == FALSE){ //SIDE A
                        SetTimer(TIMER_STATE, AngleToTime(87, DUTY_DEFAULT_TURN)); //old = 90
                        Turn(LEFT, DUTY_DEFAULT_TURN);
                } else {        //SIDE B
                        SetTimer(TIMER_STATE, AngleToTime(75, DUTY_DEFAULT_TURN)); //old = 75
                    Turn(LEFT, DUTY_DEFAULT_TURN); //actually right
                }

    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//GO FORWARD A LITTLE BIT UNTIL WE'RE OFF THE TAPE, THEN ENABLE TAPE SENSING
static int During_GOTO_GOAL_1(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                SetTimer(TIMER_STATE, DistanceToTime(DIST_OFFTAPE_CLEARANCE, DUTY_DEFAULT_MOVE));
                Move(FORWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}


//GOAL BOOST!
static int During_GOAL_BOOST(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                SetTimer(TIMER_STATE, DistanceToTime(DIST_BOOST, DUTY_BOOST));
                Move(FORWARD, DUTY_BOOST);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
```

```
//GOING FWD UNTIL TAPE IS HIT (approach from left)
static int During_GOTO_GOAL_2(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        StartTapeSM(LEFT);
    }else if ( Event == EV_EXIT)
    {
        Event = RunTapeSM(Event);
    }else
    // do the 'during' function for this state
    {
        Event = RunTapeSM(Event);
    }
    return Event;
}

//BACKING UP A BIT
static int During_GOTO_GOAL_4(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
               //Init + Start state timer
               SetTimer(TIMER_STATE, DistanceToTime(DIST_BACK_OFF_GREEN_T, DUTY_DEFAULT_MOVE));
               Move(BACKWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
               Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//TURNING RIGHT 90 DEGREES
static int During_GOTO_GOAL_5(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
               //Init + Start state timer
               SetTimer(TIMER_STATE, AngleToTime(85, DUTY_DEFAULT_TURN)); //old=90
               Turn(RIGHT, DUTY_DEFAULT_TURN);
    }else if ( Event == EV_EXIT)
    {
               Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//GOING FORWARD TILL BUMPER IS PRESSED
static int During_GOTO_GOAL_6(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
               Move(FORWARD, DUTY_GENTLEBUMP);
    }else if ( Event == EV_EXIT)
    {
               Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//DOIN' THE DUMP THANG
static int During_DUMP(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
               //Init + Start state timer
               SetTimer(TIMER_STATE, TIME_DUMP);
               //Set servo voltage so that the 'stache lets the balls out
               Open_Ball_Dispenser();
    }else if ( Event == EV_EXIT)
    {
        //update the ball count
        BallsCarried -= 5;
        BallsDelivered += 5;
        printf("  Balls carried = %d\r\n", BallsCarried);
        //if(BallsDelivered >= 20)
        //      return EV_ESTOP;
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//BACKING UP A BIT (PAST GREEN TAPE!)
static int During_REVISIT_DISPENSER(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                SetTimer(TIMER_STATE, DistanceToTime(DIST_BACKUP, DUTY_DEFAULT_MOVE));
                Move(BACKWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//TURNING AROUND TO GET MORE BALLS
static int During_REVISIT_DISPENSER_1(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        //v1=7.6, v2=7.9
                if(MirrorFlag == FALSE){ //SIDE A
                        SetTimer(TIMER_STATE, AngleToTime(130, DUTY_DEFAULT_TURN)); //old = 132, older =
129, oldest = 127
                        Turn(RIGHT, DUTY_DEFAULT_TURN);
                } else {        //SIDE B
                        SetTimer(TIMER_STATE, AngleToTime(145, DUTY_DEFAULT_TURN)); //old = 142, older =
127
                    Turn(RIGHT, DUTY_DEFAULT_TURN); //actually right
                }

    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//GOING FORWARD UNTIL THE GREEN LINE IS PASSED
static int During_REVISIT_DISPENSER_2(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
                //Init + Start state timer
                SetTimer(TIMER_STATE,  DistanceToTime(DIST_OFFTAPE_CLEARANCE, DUTY_DEFAULT_MOVE));
                Move(FORWARD, DUTY_DEFAULT_MOVE);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//RETURN BOOST!
static int During_RETURN_BOOST(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
```

```
                Move(FORWARD, DUTY_BOOST);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}

//APPROACHING TAPE FROM THE LEFT SIDE UNTIL T IS HIT
static int During_REVISIT_DISPENSER_3(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        StartTapeSM(LEFT);
    }else if ( Event == EV_EXIT)
    {
        Event = RunTapeSM(Event);
    }else
    // do the 'during' function for this state
    {
        Event = RunTapeSM(Event);
    }
    return Event;
}

//LED LIGHT SHOW!!!!!!!!!!!!!!
static int During_END(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
       printf("GAME OVER! END END. WE WIN (MAYBE)\r\n");
       Wait(1000);
    }else if ( Event == EV_EXIT)
    {
    }else
    // do the 'during' function for this state
    {
    }
    return Event;
}
```

SMTape.h

```
#ifndef SMTAPE
#define SMTAPE

#include "headers.h"

// Public Function Prototypes
int RunTapeSM( int CurrentEvent);
void StartTapeSM (char appDir);
int QueryTapeSM (void);

//Private function prototypes
static void During_APPR( int Event);
static void During_APPL( int Event);
static void During_PIVOTR( int Event);
static void During_PIVOTL( int Event);
static void During_BOPR( int Event);
static void During_BOPL( int Event);
static void During_GO( int Event);

#endif
```

SMTape.c

```
#include "headers.h"

//set MirrorFlag to TRUE if we're mirroring (playing on side B)
extern char MirrorFlag;

/*---------------------------- Module Variables ---------------------------*/
// everybody needs a state variable, you may need others as well
static int CurrentState;

/*------------------------------ Module Code ------------------------------*/
// make recursive call warning into info
#pragma MESSAGE INFORMATION C1855
int RunTapeSM(int CurrentEvent)
{
   unsigned char MakeTransition = FALSE;/* are we making a state transition? */
   int NextState = CurrentState;

   //send the current state to the debugger
   SendToDebugRug(CurrentState, DEBUG_TAPE_STATE);

   switch (CurrentState)
   {
       case ST_APPR :       // If current state is state one
         // Execute During function for state one. EV_ENTRY & EV_EXIT are
         // processed here
         During_APPR(CurrentEvent);
         //process any events
         if ( CurrentEvent != EV_NO_EVENT )        //If an event is active
         {
            switch (CurrentEvent)
            {
               case EV_RLC : //If event is event one
                  // Execute action function for state one : event one
                         CurrentEvent = EV_ERROR;
                  break;
                // repeat cases as required for relevant events
               case EV_RL :
                  break;
               case EV_RC :
                  NextState = ST_BOPR;
                  MakeTransition = TRUE;
                  break;
               case EV_LC :
                  NextState = ST_BOPL;
                  MakeTransition = TRUE;
                  break;
               case EV_R :
                  break;
               case EV_L :
                  break;
               case EV_C :
                  NextState = ST_PIVOTR;
                  MakeTransition = TRUE;
```

```
 }
 break;
case ST_APPL :
 During_APPL(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_RLC :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_RL :
          break;
       case EV_RC :
          NextState = ST_BOPR;
          MakeTransition = TRUE;
          break;
       case EV_LC :
          NextState = ST_BOPL;
          MakeTransition = TRUE;
          break;
       case EV_R :
          break;
       case EV_L :
          break;
       case EV_C :
          NextState = ST_PIVOTL;
          MakeTransition = TRUE;
          break;
    }
 }
 break;
case ST_PIVOTR :
 During_PIVOTR(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_RLC :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_RL :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_RC :
          NextState = ST_BOPR;
          MakeTransition = TRUE;
          break;
       case EV_LC :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_R :
          NextState = ST_BOPR;
          MakeTransition = TRUE;
          break;
       case EV_L :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_C :
          break;
    }
 }
 break;
case ST_PIVOTL :
 During_PIVOTL(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_RLC :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_RL :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_RC :
                         CurrentEvent = EV_ERROR;
          break;
       case EV_LC :
          NextState = ST_BOPL;
          MakeTransition = TRUE;
```

```
                        CurrentEvent = EV_ERROR;
          break;
       case EV_L :
          NextState = ST_BOPL;
          MakeTransition = TRUE;
          break;
       case EV_C :
          break;
    }
 }
 break;
case ST_BOPR :
 During_BOPR(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_RLC :
                        CurrentEvent = EV_ATT;
          break;
       case EV_RL :
                        CurrentEvent = EV_ATT;
          break;
       case EV_RC :
          break;
       case EV_LC :
                        CurrentEvent = EV_ERROR;
          break;
       case EV_R :
          break;
       case EV_L :
                        CurrentEvent = EV_ERROR;
          break;
       case EV_NOTAPE :
          NextState = ST_GO;
          MakeTransition = TRUE;
       case EV_C :
          NextState = ST_GO;
          MakeTransition = TRUE;
          break;
    }
 }
 break;
case ST_BOPL :
 During_BOPL(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
       case EV_RLC :
                        CurrentEvent = EV_ATT;
          break;
       case EV_RL :
                        CurrentEvent = EV_ATT;
          break;
       case EV_RC :
                        CurrentEvent = EV_ERROR;
                  break;
       case EV_LC :
          break;
       case EV_R :
                        CurrentEvent = EV_ERROR;
          break;
       case EV_L :
          break;
       case EV_NOTAPE :
          NextState = ST_GO;
          MakeTransition = TRUE;
       case EV_C :
          NextState = ST_GO;
          MakeTransition = TRUE;
          break;
    }
 }
 break;
case ST_GO :
 During_GO(CurrentEvent);
 if ( CurrentEvent != EV_NO_EVENT )
 {
    switch (CurrentEvent)
    {
```

```
                 break;
              case EV_RL :
                 CurrentEvent = EV_ATT;
                 break;
              case EV_RC :
                 NextState = ST_BOPR;
                 MakeTransition = TRUE;
                 break;
              case EV_LC :
                 NextState = ST_BOPL;
                 MakeTransition = TRUE;
                 break;
              case EV_R :
                 NextState = ST_BOPR;
                 MakeTransition = TRUE;
                 break;
              case EV_L :
                 NextState = ST_BOPL;
                 MakeTransition = TRUE;
                 break;
              case EV_C :
                 break;
           }
        }
        break;
    }
    //   If we are making a state transition
    if (MakeTransition == TRUE)
    {
       //   Execute exit function for current state
       RunTapeSM(EV_EXIT);
       CurrentState = NextState; //Modify state variable
       //   Execute entry function for new state
       RunTapeSM(EV_ENTRY);
     }
     return(CurrentEvent);
}

//Start the tape state machine by giving it an approach direction
//take care of any mirroring here!
void StartTapeSM (char appDir)
{
   printf("Starting tape sensing\r\n");

   //Don't take mirroring into account because we SWITCH TAPE SENSORS
   if(appDir == RIGHT)
               CurrentState = ST_APPR;
   else
               CurrentState = ST_APPL;

   // call the entry function (if any) for the ENTRY_STATE
   RunTapeSM(EV_ENTRY);
}

int QueryTapeSM ( void )
{
   return(CurrentState);
}

/****************************************************************************
 private functions
 ***************************************************************************/

static void During_APPR( int Event)
{
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        // implement any entry actions required for this state machine
               Move(FORWARD, DUTY_APP);
    }else if ( Event == EV_EXIT)
    {
        Stop();
}else
{
       // no activity for during
}
    return;
}

static void During_APPL( int Event)
```

```
    {
                Move(FORWARD, DUTY_APP);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    {
                //no activity for during
    }
    return;
}

static void During_PIVOTR( int Event)
{
    if ( Event == EV_ENTRY)
    {
                Turn(RIGHT, DUTY_PIVOT);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    {
                //no activity for during
    }
    return;
}

static void During_PIVOTL( int Event)
{
    if ( Event == EV_ENTRY)
    {
                Turn(LEFT, DUTY_PIVOT);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    {
                //no activity for during
    }
    return;
}

static void During_BOPR( int Event)
{
    if ( Event == EV_ENTRY)
    {
                SetMotor(L_MOTOR, FORWARD, DUTY_BOP_OUTER);
                SetMotor(R_MOTOR, FORWARD, DUTY_BOP_INNER);
    }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    {
                //no activity for during
    }
    return;
}

static void During_BOPL( int Event)
{
    if ( Event == EV_ENTRY)
    {
                SetMotor(R_MOTOR, FORWARD, DUTY_BOP_OUTER);
                SetMotor(L_MOTOR, FORWARD, DUTY_BOP_INNER);
        }else if ( Event == EV_EXIT)
    {
                Stop();
    }else
    {
                //no activity for during
    }
    return;
}

static void During_GO( int Event)
{
    if ( Event == EV_ENTRY){
                Move(FORWARD, DUTY_GO);
    }else if ( Event == EV_EXIT){
                Stop();
    }else{//no activity for during}
```