BEACON MODULE:

```
#include

/* Variables */
// The previous beacon event.
static Event_t lastEvent = NO_EVENT;
// The current beacon event.
static Event_t thisEvent = NO_EVENT;

/* Functions */
/* Returns a BEACON_ON event if the phototransistor signal is between the specified thresholds.
 * Otherwise, will return BEACON_OFF event. Will only return event if different from previous.*/
Event_t BEACON_CheckForBeacon() {
        If AD readout on BEACON_BIT is between high and low thresholds, return BEACON_ON event.
        Else, return BEACON_OFF.

        If event is same as before, return NO_EVENT.
```

EVENTS MODULE:

```
#include

/* Functions */
/* Checks and returns any events that have occurred. */
Event_t EVENTS_CheckForEvents(void) {
        //Check if command given
        //Check if a timer expired
        //Check if tape event has occured
        //Check if beacon event has occured
        //otherwise return NO_EVENT


/* Checks all timers and if any has expired, clear the timer and return the corresponding event. */
static Event_t CheckForTimerExpired() {
        if timer is expired,
            clear timer
            return expired event

        otherwise return NO_EVENT
}
```

MAIN MODULE:

```
#include
#pragma LINK_INFO DERIVATIVE "SampleS12"

/* Functions */
void main (void) {
        //Initialize timers
   //Initialize PWM
   //Initialize ports
```

```
    //Initialize SPI communication

    //Uncomment the following line to debug and not run the state machine.
    //MAIN_Debug();

    //Initialize the state machine and loop forever to run it.
        STATE_InitStateMachine();
    for(;;) {
            STATE_RunStateMachine(EVENTS_CheckForEvents());
        }
}

/* Initialize port data directions for outputs and inputs, initialize pwms, and set opto output low. */
static void MAIN_Init(void) {
        //set outputs
        //clear inputs
        //ad port initialization string

        //set initial wheel state
}
```

PWM MODULE:

```
#include

/* Prototypes */
static void SetDuty(unsigned char duty, unsigned char wheel);
static void SetDirection(unsigned char direction, unsigned char wheel);

/* Functions */
/* Initializes PWM functions for wheels. */
void PWM_Init(void) {
        //initializes port U0 to be used by the PWM subsystem
        //enable PWM channel 0 for left wheel
        //initializes port U1 to be used by the PWM subsystem
        //enable PWM channel 1 for right wheel

        //give U0 pin control to PWM subsystem for left wheel
        //give U1 pin control to PWM subsystem for right wheel

        //10 kHz
        //prescale factor 16
        //set to clock A

        //set left wheel period in ticks
        //set right wheel period in ticks
}

/* Sets PWM specifically for wheels: which wheel, duty cycle, and which direction. */
void PWM_SetDuty(unsigned char wheel, unsigned char duty, unsigned char direction) {
  SetDirection(direction, wheel);
  SetDuty(duty, wheel);
}
```

```
/* Sets rotational direction of a particular wheel. */
static void SetDirection(unsigned char direction, unsigned char wheel) {
  if left wheel,
        if direction is forward,
            set LEFT_DIR_BIT
            set polarity to low
        else if direction is reverse,
            clear LEFT_DIR_BIT
            set polarity to high
  else if right wheel,
        if direction is forward,
            clear RIGHT_DIR_BIT
            set polarity to high
        else if direction is reverse,
            set RIGHT_DIR_BIT
            set polarity to low
}

/* Sets duty cycle of a particular wheel. */
static void SetDuty(unsigned char duty, unsigned char wheel) {
        if left wheel, set duty as a fraction of the period in clock ticks
        else if right wheel, set duty as a fraction of the peiod in clock ticks
}
```

SPI MODULE:

Module level variables
This command
Last command

Initialize the e128's PSI:
Set Baud rate to the slowest
    SPPR = 8 and SPR = 8
Enable SPI
Set MSB first
Set master
Set polarity active low
Set sample even edges
Enable SS pin (set MODFEN and SSOE)
Enable Receive register ready interrupt

Set up an output compare
 Set channel and period to query command chip

Enable interrupts

NEW COMMAND
If the latest command is different that the last time we checked
 Update the last command = command
 Return NEW_COMMAND
Else return NO_EVENT

GET COMMAND
 Return command

OC interrupt:
Clear OC flag
Set next output compare
Send 0xAA to SPI system

SPI interrupt:
Static ping
Static readnewcommand
If we had previously not read 0xFF
  If this is a ping READ garbage
        Set ping no
        Read SPISR
        Read the SPIDR
        Send for new request: by writing 0xAA again to SPIDR
  Else get the real data
        Set ping yes
        Read SPISR
        Read the SPIDR
        If we received 0xFF
          Then set readnewcommand = yes
If readnewcommand is yes (we just got 0xFF)
  If this is a ping READ garbage
        Set ping no
        Read SPISR
        Read the SPIDR
        Send for new request: by writing 0xAA again to SPIDR
  Else get the real data
        Set ping yes
        Read SPISR
        Read the SPIDR
        Switch on read to give us:
          Command
          Set readnewcommand = no;

STATE MODULE:

```
#include

/* Variables */
static State_t state;

/* Prototypes */
static void StopBothState(Event_t event);
static void RotateCW90State(Event_t event);
static void RotateCW45State(Event_t event);
static void RotateCCW90State(Event_t event);
static void RotateCCW45State(Event_t event);
static void ForwardHalfState(Event_t event);
static void ForwardFullState(Event_t event);
static void ReverseHalfState(Event_t event);
static void ReverseFullState(Event_t event);
static void AlignWithBeaconState(Event_t event);
```

```
static void ForwardUntilTapeState(Event_t event);

/* Initialize state machine by setting initial state and final destination. */
void STATE_InitStateMachine() {
        set initial state
}

/* Figure out which state we are in, and call the corresponding function to handle the event. */
void STATE_RunStateMachine(Event_t event) {
        switch (state) {
            if STOP_BOTH,
                    StopBothState(event);
            if ROTATE_CW_90,
                    RotateCW90State(event);
            if ROTATE_CW_45,
                    RotateCW45State(event);
            if ROTATE_CCW_90,
                    RotateCCW90State(event);
            if ROTATE_CCW_45,
                    RotateCCW45State(event);
            if FORWARD_HALF,
                    ForwardHalfState(event);
            if FORWARD_FULL,
                    ForwardFullState(event);
            if REVERSE_HALF,
                    ReverseHalfState(event);
            if REVERSE_FULL,
                    ReverseFullState(event);
            if ALIGN_WITH_BEACON,
                    AlignWithBeaconState(event);
            if FORWARD_UNTIL_TAPE,
                    ForwardUntilTapeState(event);
        }
}

/* On entering function, stops both wheels until a new command is given. */
static void StopBothState(Event_t event) {
        if first time this state is entered, stop both wheels
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates clockwise 90 degrees and sets timer. If the timer expires,
 * will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCW90State(Event_t event) {
        if first time this state is entered, rotate wheels clockwise and initialize timer
        if event is ROTATE90_TIMER_EXPIRED, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates clockwise 45 degrees and sets timer. If the timer expires,
 * will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCW45State(Event_t event) {
        if first time this state is entered, rotate wheels clockwise and initialize timer
        if event is ROTATE45_TIMER_EXPIRED, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}
```

```
/* On entering function, rotates counterclockwise 90 degrees and sets timer. If the timer
 *  expires, will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCCW90State(Event_t event) {
        if first time this state is entered, rotate wheels counterclockwise and initialize timer
        if event is ROTATE90_TIMER_EXPIRED, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates counterclockwise 45 degrees and sets timer. If the timer
 *  expires, will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCCW45State(Event_t event) {
        if first time this state is entered, rotate wheels counterclockwise and initialize timer
        if event is ROTATE45_TIMER_EXPIRED, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates both wheels forward at half duty until a new command is given. */
static void ForwardHalfState(Event_t event) {
        if first time this state is entered, rotate wheels forward with half duty
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates both wheels forward at full duty until a new command is given. */
static void ForwardFullState(Event_t event) {
        if first time this state is entered, rotate wheels forward with full duty
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates both wheels reverse at half duty until a new command is given. */
static void ReverseHalfState(Event_t event) {
        if first time this state is entered, rotate wheels reverse with half duty
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates both wheels reverse at full duty until a new command is given. */
static void ReverseFullState(Event_t event) {
        if first time this state is entered, rotate wheels reverse with full duty
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates clockwise. If beacon is seen, will switch to stop state.
 * Otherwise, will switch to next given command state. */
static void AlignWithBeaconState(Event_t event) {
        if first time this state is entered, rotate wheels clockwise
        if event is BEACON_ON, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}

/* On entering function, rotates both wheels forward. If tape is found, will switch
 * to stop state. Otherwise, will switch to next given command state. */
static void ForwardUntilTapeState(Event_t event) {
        if first time this state is entered, rotate wheels forward at full duty
        if event is TAPE_FOUND, set state to STOP_BOTH
        if event is NEW_COMMAND, get command and set it as state
}
```

TAPE MODULE:

```
#include libraries
#define
/* Variables */
static Event_t lastEvent = NO_EVENT;
static Event_t thisEvent = NO_EVENT;

/* Functions */
/* Returns a TAPE_FOUND event if the tape sensor signal greater than the specified threshold. */
 Event_t TAPE_CheckForTape() {
        if the TAPE_BIT A/D readout is higher than a specified threshold, return TAPE_FOUND.
        else return NO_EVENT.
}
```

TIMER MODULE:

Module Level Variables:
timer0 (unsigned int) used to keep track of Timer overflows by using the Input Capture
Period (unsigned int) based on the 43.69mS overflow, or 1.5 MHz clock
flag0_X (of type Flags_t) where X ranges from 0 to 7 used to keep track of whether the time has expired for each of the four timers
flagact0_X (of type Flags_t) where X ranges from 0 to 7 flags used to keep track of whether the time has expired for each of the four timers
NewTime0_X (unsigned int) where X ranges from 0 to 7 and the new time keeps track of the time each of the four timers will be checking to be expired.

Function: TIMER0_Init(unsigned char NewRate)
This function turns timer system on and sets the timer to count at a certain rate using the TIMER_RATE_XX definitions:
Turn the timer system on
Set TSCR2: divide by 16, so timer overflow occurs every 43.69mS
Use a switch statement to set the timer period, where 1mS = 1500, 2mS = 3000, 4ms = 6000, etc. and we set the variable Period equal to this number.
Setup OC4 to time the updates, and to trigger the first interrupt that will keep track of these times
Set IOC4 of the timer to be an output compare, the rest remain as inputs
Set no pin connected to IOC4, which means pin PT0 remains free
Set the first output capture to happen one "Period" into the future
Clear the flag for the IOC4
Enable the interrupt for IOC4
Enable interrupts

Interrupt Response for Timer IOC4:
This interrupt will be keeping track for timer0 channel 4, and will be setting flags for timer0 on channel 4-7 which will allow the TIMER0_InitTimer(TimerNumber, TicksToCount) function to let the user know if the timer is expired:
interrupt _Vec_tim0ch4 Timer0Counter (void)
Clear the flag for the IOC4
Update for the next interrupt to keep track of the ticking rate
EnableInterrupts
Add one to timer0 to indicate that one clock tick has passed by
Now we check to see if any of the flags have expired and update:

If timer0 is equal to newtime0_x and flagact0_X is active then
Set flag0_X to set
Set flagact0_X to not active
Repeat this if statement for all the X timers, 0 through 7

Function: TIMER0_InitTimer
This function will take in two paramaters, the first a char Num which will choose which timer we are starting/restarting, and the second an unsigned int NewTime which will give the number of ticks until that timer is expired. Everytime this function is called, that timer will start counting up to the the number of NewTime ticks that it is asked for. The user must be sure that this NewTime of ticks does not pass the overflow otherwise this will be inaccurate.

TIMER0_InitTimer(unsigned char Num, unsigned int NewTime)
Use a switch statement to choose which timer that the user wants by the timer number and be sure to set the NewTime for that timer as well as to clear its flag, then for Case X (0 through 7):
Set NewTime0_X equal to timer0 plus the input New Time;
Set flag0_X to be cleared
Set flagact0_X to be active

Function: TIMER0_IsTimerExpired
Checks to see if the the timer associated with the parameter input Num has expired (which clock, 0-7), the number of the timer to test. This function will return TIMER0_EXPIRED or TIMER0_NOT_EXPIRED, or TIMER0_ERR
TimerReturn_t TIMER0_IsTimerExpired(unsigned char Num)
Use a switch statement to choose which timer that the user wants to check and then return on the basis of whether it has expired or not, so for each case X:
If flag0_X is set then
Return timer0 is expired
Else if flag0_X is cleared
Return timer0 is not expired
Else
Return timer error

Function: TIMER0_ClearTimerExpired
This takes as a paramater an unsigned char Num which chooses which timer flag that we want to clear. This can be used to show that an event has been serviced.
TIMER0_ClearTimerExpired(unsigned char Num)
Again we use a switch statement to choose which timer that the user wants to clear and clear the according flag, so for each case X:
Set flag0_0 to be cleared

Function: TIMER0_GetTime
This function takes in no parameters, and will return an unsigned int representing the current clock count which will be between 0 and 65535. It simply returns the free-running counter of timer0.
unsigned int TIMER0_GetTime(void)
Return the value for timer0

WHEELS MODULE:

```
#include libraries
#define

/* Functions */
/* Stop both wheels */
```

```
void WHEELS_StopBoth(void) { //0x00
        set duty on left wheel to 0
        set duty on right wheel to 0
}

/* Rotate wheels clockwise */
void WHEELS_RotateCW(void) { //0x02 and 0x03 - adjust timers for degree
        set direction on left wheel to forward
        set direction on right wheel to reverse
}

/* Rotate wheels counterclockwise */
void WHEELS_RotateCCW(void) { //0x04 and 0x05 - adjust timers for degree
        set direction on left wheel to reverse
        set direction on right wheel to forward
}

/* Wheels forward at half duty cycle */
void WHEELS_ForwardBothHalf(void) { //0x08
        set left to half duty and forward
        set right to half duty and forward
}

/* Wheels forward at full duty cycle */
void WHEELS_ForwardBothFull(void) { //0x09
        set left to full duty and forward
        set right to full duty and forward
}

/* Wheels reverse at half duty cycle */
void WHEELS_ReverseBothHalf(void) { //0x10
        set left to half duty and reverse
        set right to half duty and reverse
}

/* Wheels reverse at full duty cycle */
void WHEELS_ReverseBothFull(void) { //0x11
        set left to full duty and reverse
        set right to full duty and reverse
}
```