Project Mini
Pseudocode

Blake English
Nina Joshi
Peter Miller

Board module

```
Event_t BOARD_CheckForBoardEvent(void) {
Begin
    Event is NO_EVENT
    If the left photo bit is greater than the photo interrupt threshold,
        Event is LEFT_SENSED
    Else if the right photo bit is greater than the photo interrupt threshold,
        Event is RIGHT_SENSED
    End

    If the event is not equal to the last event,
        Set the last event to current event
        Return current event
    End

    Return NO_EVENT
End

void BOARD_Debug(void) {
Forever
    While no key is hit,
        Check for events
        If LEFT_SENSED, print statement
        If RIGHT_SENSED, print statement
    If a key is hit, respond to the appropriate key,
        If l, set left light
        If r, set right light
        If c, clear both lights
        If b, set both lights
        If d, set right building as depot
        If k, set right building as bank
        If q, exit out of loop
    End
End
```

Driving module

```
Event_t DRIVING_CheckForDrivingEvent(void) {
Begin
    Event is NO_EVENT
    If the ignition button is hit,
        Event is IGNITION_ON
    End

    If the event is not equal to the last event,
```

```
            Set the last event to current event
            Return current event
        End

        Return NO_EVENT
End

/* Finds a step that divides the end timer value by the limits of the servo PWM values.
 * This step is used later to calculate the servo PWM value based on the current time.
 * For example, 50-100% PWM values corresponding to 1-450000 timer values. */
void DRIVING_InitFuelGauge(void) {
Begin
        Get current time and set it to last time
        Set timer tick to end time of game / (start value – end value)
End

/* Sets the fuel gauge servo voltage to the appropriate value for the time. */
void DRIVING_SetFuelGauge(void) {
Begin
        Get current time
        Set time step to current time – last time
        Time is set to time plus time step

        Set dutycycle to start value – time / timer tick

        If dutycycle is less than1,
            set dutycycle to 1
        End

        Set duty cycle
        Set last time to current time
End

void DRIVING_DriveCar(void) {
Begin
        Get state of throttle switch
        If REV_BUTTON,
            Reverse car motor
        If FWD_BUTTON,
            Forward car motor
        If BOTH_BUTTONS or NO_BUTTON,
            Stop car motor
        End

        Get steering wheel position
        If RIGHT,
            Right car steering
        If LEFT,
            Left car steering
        If CENTERED,
            Center car steering
        End
End

void STEERING_initSteeringWheelSwitchPins(void) {
Begin
```

```
        Set forward switch to input
        Set reverse switch to input
End

void STEERING_initCarControlPins(void) {
Begin
        Set steering left to output
        Set steering right to output
        Set wheel vibrate to output
End

/* Figures out current steering wheel position: centered, left, or right. */
static SteeringWheel_t STEERING_getSteeringWheelPosition(void) {
Begin
        Set current wheel positon to CENTERED
        Read steering potentiometer value

        If the potentiometer value is greater than the right threshold,
            Wheel position is RIGHT
        If the potentiometer value is less than the left threshold,
            Wheel position is LEFT
        If the potentiometer value is between the two thresholds,
            Wheel position is CENTERED
        End

        Return wheel position
End

/* Figures out current throttle switch state: forward, reverse, both, none. */
static SteeringWheel_t STEERING_getSteeringWheelThrottleSwitchStates(void) {
Begin
        Set current switch state to NO_BUTTON
        Check forward switch
        Check reverse switch

        If both forward and reverse are active,
            Switch state is BOTH_BUTTONS
        If just forward,
            Switch state is FWD_BUTTON
        If just reverse,
            Switch state is REV_BUTTON
        If neither,
            Switch state is NO_BUTTON
        End

        Return switch state
End

void DRIVING_Debug(void) {
Forever
        While no key is hit,
            Check for events
            If IGNITION_ON, print statement
        If a key is hit, respond to the appropriate key,
            If h, set ignition light
            If l, clear ignition light
```

```
            If s, vibrate wheel
            If c, stop vibrating wheel
            If f, car forward motor
            If b, car reverse motor
            If t, car left steering
            If r, car right steering
            If q, exit out of loop
        End
End
```

Events module

```
typedef enum {
        NO_EVENT,
        OPTOISOLATOR_ON,
        FORCE_START_HIT,
        IGNITION_ON,
        FINAL_DESTINATION_REACHED,
        END_TIMER_EXPIRED,
        OPTO_TIMER_EXPIRED,
        SAFE_CLEARED,
        LEFT_SENSED,
        RIGHT_SENSED,
        SET_UP_TIMER_EXPIRED,
        BLINK_TIMER_EXPIRED,
        BUZZER_TIMER_EXPIRED,
        CELEBRATION_TIMER_EXPIRED
} Event_t;

/* Checks and returns any events that have occurred. */
Event_t EVENTS_CheckForEvents(void) {
Begin
        Set event to NO_EVENT

        If timer has expired, return event
        If opto event has occurred, return event
        If driving event has occurred, return event
        If board event has occurred, return event
        If safe event has occurred, return event

        Return event
End

void EVENTS_Debug(void) {
Forever
        While no key is hit,
            Check for events
            If any event has occurred, print statement
        If a key is hit, respond to the appropriate key,
            If e, set end timer
            If o, set opto timer
            If z, set buzzer timer
            If b, set blinking timer
            If c, set celebration timer
            If q, exit out of loop
```

```
    End
End
```

Main module

```
void main (void) {
Begin
    Initialize timers
    Initialize routine for the PWMS functions
    Initialize ports to be outputs or inputs

    //Allow for debugging to be called

    Initialize the state machine and loop forever while calling for event checker to run it.
End

static void MAIN_Debug(void) {
Forever
    Debug driving
    Debug board
    Debug opto
    Debug events
End
```

Opto Module

```
Event_t OPTO_CheckForStartEvent(void) {
    Begin
    Event is NO_EVENT
    If the opto input bit is not equal to zero,
        Event is OPTOISOLATOR_ON
    End

    If the event is not equal to the last event,
        Set the last event to current event
        Return current event
    End

    Return NO_EVENT
End

void OPTO_Debug(void) {
Forever
    While no key is hit,
        Check for events
        If OPTOISOLATOR_ON, print statement
    If a key is hit, respond to the appropriate key,
        If q, exit out of loop
    End
End
```

State module

```
/* Initialize state machine by setting initial state and final destination. */
void STATE_InitStateMachine() {
Begin
        set final destination to RIGHT
        set state to WAITING_FOR_SWITCH
End

/* Figure out which state we are in, and call the corresponding function to handle the event. */
void STATE_RunStateMachine(Event_t event) {
Begin
        If current state is WAITING_FOR_SWITCH,
            Waiting for switch state
        If current state is WAITING_FOR_IGNITION,
            Waiting for ignition state
        If current state is WAITING_TO_REACH_DESTINATION,
            Waiting to reach destination state
        If current state is DEALING_WITH_SAFE,
            Dealing with safe state
        If current state is PULSING_OPTOISOLATOR,
            Pulsing optoisolator state
        End
End

/* If optoisolator input is on or force start is hit, prepare to begin the game.*/
static void WaitingForSwitchState(Event_t event) {
Begin
        If event is OPTOISOLATOR_ON or FORCE_START_HIT,
            Set opto output low
            Initilalize safe
            Set fuel gauge full
            Reset board
            Set ignition light
            Start blink timer
            Set state to WAITING_FOR_IGNITION
        End
End

/* If ignition button is hit, begin driving state. Until then, blink ignition indicator light and
 * continually adjust fuel gauge. */
static void WaitingForIgnitionState(Event_t event) {
Begin
        If event is IGNITION_ON,
            Clear ignition indicator
            Start end timer
            Initialize fuel gauge
            Allow driving
            Vibrate steering wheel
            Start blink timer
            Set state to WAITING_TO_REACH_DESTINATION
        If event is BLINK_TIMER_EXPIRED,
            Start blink timer
            Switch ignition light state
        Default
            Set fuel gauge
        End
```

```
End

/* If the car is sensed in the final destination building, stop all driving and begin dealing with the safe.
 * If end timer expires, stop all driving, pulse opto output high, and begin failure sequence. Otherwise,
 * blink final destination building lights, allow driving, and adjust fuel gauge. */
static void WaitingToReachDestinationState(Event_t event) {
Begin
        If event is RIGHT_SENSED and final destination is RIGHT,
            Stop car motor
            Center car steering
            Stop wheel vibrate
            Set final destination light
            Set state to DEALING_WITH_SAFE
            Set final destination to LEFT
        If event is LEFT_SENSED and final destination is LEFT,
            Stop car motor
            Center car steering
            Stop wheel vibrate
            Set final destination light
            Set state to DEALING_WITH_SAFE
            Set final destination to RIGHT
        If event is BLINK_TIMER_EXPIRED,
            Start blink timer
            Switch ignition light state
        If event is END_TIMER_EXPIRED,
            Stop blink timer
            Clear blink timer
            Stop car motor
            Stop wheel vibrate
            Clear end timer
            End safe
            Start buzzer timer
            Turn buzzer on
            Start opto timer
            Pulse opto output high
            Set state to PULSING_OPTOISOLATOR
        Default
            Drive car
            Update fuel gauge
        End
End

/* If safe is cleared, reset safe, begin celebration sequence, and pulse opto ouptut high.
 * If end timer expires, reset safe, turn buzzer on, pulse output high, and begin failure sequence.
 * Otherwise, deal with the safe and adjust the fuel gauge. */
static void DealingWithSafeState(Event_t event) {
Begin
        If event is SAFE_CLEARED,
            End safe
            Start celebration timer
            Start opto timer
            Pulse opto output high
            Set state to PULSING_OPTOISOLATOR
        If event is END_TIMER_EXPIRED,
            Clear end timer
            End safe
```

```
            Start buzzer timer
            Turn buzzer on
            Start opto timer
            Pulse opto output high
            Set state to PULSING_OPTOISOLATOR
        Default
            Deal with safe
            Update fuel gauge
        End
End

/* If the celebration timer expires, return to waiting for switch state. If buzzer timer expires,
 * first turn the buzzer off and then return to waiting for switch state. If opto timer expires,
 * clear opto output to low and clear board lights. Otherwise, if the celebration timer is active,
 * continue to celebrate the win by chiming the safe solenoid. */
static void PulsingOptoisolatorState(Event_t event) {
Begin
        If event is OPTO_TIMER_EXPIRED,
            Clear opto timer
            Pulse opto output low
            Clear board lights
        If event is CELEBRATION_TIMER_EXPIRED,
            Set state to WAITING_FOR_SWITCH
        If event is BUZZER_TIMER_EXPIRED,
            Turn buzzer off
            Set state to WAITING_FOR_SWITCH
        Default
            If celebration timer is active,
                    Chime safe celebration
            End
        End
End
```

Safe Module

```
void main(void){

 DO FOREVER:
   if the GAME_TIMER expires:
     SEND A SIGNAL TO THE NEXT MACHINE
     reset the game;


   if STAGE == REACH_DEST_B:
     if the player reaches destination B:
       signal to the user to unlock the safe:
         turn the "LOCKED" light ON;
         turn on a FlashSafeState_Timer;
       STAGE  = CRACK_SAFE;


   if STAGE = CRACK_SAFE:
     reset the safe module;
     signal to the user to spin the safe dial CW:
       turn the "CW" light ON;
```

```
      update the safe register;
      turn on a FlashSpinDirection_Timer;
    call a function RandTick() to store a randomly generated number of ticks to count on the safe dial; call this TargetTicks;
    reset the users ticks; call the UserTicks = 0;
    STAGE  = CRACK_FIRST_NUM;


  if STAGE == CRACK_FIRST_NUM:
    if a dial tick event occurred
      update the number of user ticks that have occurred; (UserTicks);
    if UserTicks == Target Ticks:
      pulse the ticker solenoid to alert the player;
      turn on Overshoot_Timer;
    if the Overshoot_Timer expires:
      clear Overshoot_Timer;
      if UserTicks within a certain range of TargetTicks:
         set STAGE  = CRACK_SECOND_NUM;
      otherwise:
        reset a module-level variable called WarningSequence to 1;
        turn all indicator lights ON;
        turn on OvershootWarning_Timer;
        set STAGE = OVERSHOOT;


  if STAGE == CRACK_SECOND_NUM:
    if a dial tick event occurred
      update the number of user ticks that have occurred; (UserTicks);
    if UserTicks == Target Ticks:
      pulse the ticker solenoid to alert the player;
      turn on Overshoot_Timer;
    if the Overshoot_Timer expires:
      clear Overshoot_Timer;
      if UserTicks within a certain range of TargetTicks:
         set STAGE  = CRACK_THIRD_NUM;
      otherwise:
        reset a module-level variable called WarningSequence to 1;
        turn all indicator lights ON;
        turn on OvershootWarning_Timer;
        set STAGE = OVERSHOOT;


  if STAGE == CRACK_THIRD_NUM:
    if a dial tick event occurred
      update the number of user ticks that have occurred; (UserTicks);
    if UserTicks == Target Ticks:
      pulse the ticker solenoid to alert the player;
      turn on Overshoot_Timer;
    if the Overshoot_Timer expires:
      clear Overshoot_Timer;
      if UserTicks within a certain range of TargetTicks:
         set STAGE  = SAFE_CRACKED;
      otherwise:
        reset a module-level variable called WarningSequence to 1;
        turn all indicator lights ON;
        turn on OvershootWarning_Timer;
```

```
        set STAGE = OVERSHOOT;


    if STAGE == SAFE_CRACKED:
      open the lock solenoid to allow the player to open the safe;
      turn the "UNLOCKED" light ON;
      STAGE  = OPEN_SAFE;


    if STAGE == OPEN_SAFE:
      if the player opens the safe:
        pulse the opto output to signal the next game;
        initiate the celebration sequence --> THIS CAN BE BLOCKING CODE SINCE THE GAME IS
OVER
        STAGE  = GAME OVER;


    if STAGE == GAME_OVER:
        reset the game;
          1) reset all timers;
          2) return all lights to their proper state
          3) etc.
        STAGE  = GAME_READY;


    if FlashSafeState_Timer expires:
        invert the state of the "LOCKED" light off;
        reset the FlashSafeState_Timer and start it;


    if FlashSpinDirection_Timer expires:
      invert the state of the "LOCKED" light off;
      reset the FlashSpinDirection_Timer and start it;


    if OvershootWarning_Timer expires:
      increment WarningSequence by 1;
      if WarningSequence <= 6:
        invert the state of all indicator lights;
        reset the OvershootWarning_Timer;
      otherwise:
        clear the OvershootWarning_Timer;
        set STAGE = CRACK_SAFE;


    if UpdateSafeRegister_Timer expires:
      update the safe register;
}
```