Design Report for a New Car Anti-Theft System

Tony Kim

March 6, 2008

Abstract

We describe the implementation of an automobile anti-theft system, featuring a unique security layer by controlling power to the fuel pump.

Contents

List of Figures

1 Overview

A reliable and robust anti-theft system is a necessary component in modern cars. That both manufacturers and consumers have made this realization is obvious, given that virtually every newly produced automobile comes equipped with this safety feature. However, there is a natural disadvantage to the standard-issue anti-theft system, since it is a standard factory unit. Due to its wide use, many people will have learned how to disable it. Our proposed system, being custom designed, does not share this defect. In addition, we have implemented an additional safety measure that is unique to our system.

We have implemented the basic car alarm system based on the client's wishes that it is highly automated. We first give a casual description of our anti-theft system by showing the action of our system as the client exits and later returns to the car. In this scenario, the system is initially in the "disarmed" state. Please refer to Figure 1, where we have illustrated the various inputs and outputs to our alarm system as they might be physically implemented in a typical car.

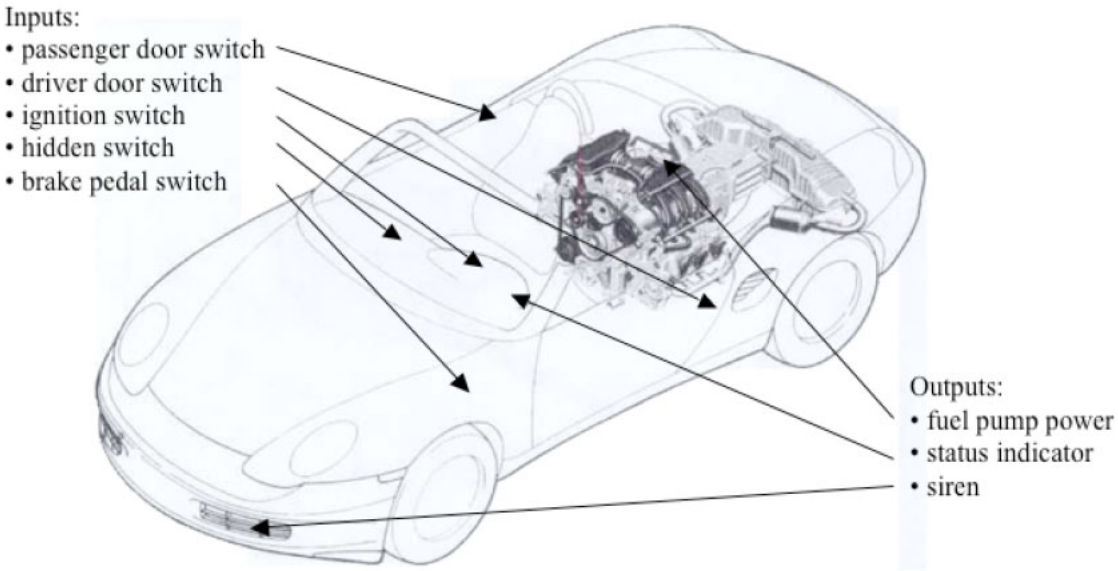

Figure 1: Inputs and outputs of the alarm system[1]

Once the client has turned the ignition off, and exits the car (which is indicated by the driver door being open, then closed), the system begins a countdown of T_ARM_DELAY seconds (e.g. when T_ARM_DELAY = 6, the countdown lasts for 6 seconds). When the countdown is over, the anti-theft system enters its "armed" state. The countdown is restarted if the driver door is opened and reclosed before the system has been armed.

Once the system has been armed, opening the driver door *or* the passenger door begins a countdown. If the ignition is not turned on within the respective countdown interval (T_DRIVER_DELAY and T_PASSENGER_DELAY), the siren sounds. The siren remains on as long as any door is open. After all doors are closed, the alarm remains on for an additional interval T_ALARM_ON. At this point the system resets to the armed but silent state.

Our unique, additional deterrent involves the power to the fuel pump of the car. (Henceforth we refer to this subunit as "FuelPumpLogic.") Of course, it is understood that when the fuel pump power is cut, the car will be immobile even if the car's ignition were on. The functional logic of this subunit is as follows: whenever the ignition is turned off, the fuel pump power is turned *off* regardless of its previous state. When the ignition is on, the driver can turn *on* the fuel pump power by simultaneously stepping on the brake pedal and by pressing a hidden switch. Note that turning on the car's ignition alone will *not* restore power to the fuel pump. Rather, one must first turn on the car, and then apply the above sequence of inputs.

It is worthwhile to mention that the additional FuelPumpLogic neutralizes a theft scenario that is not covered by the standard alarm system. Consider the case in which the thief has first stolen the *keys* to the car. The typical alarm system, in such a case, can be entirely bypassed. However, even if the thief were to possess the key, the car would still be immobile since in all likelihood he will not know the method to restore power to the fuel pump.

In addition, our alarm system provides a status indicator LED which reflects the status of the alarm system (Disarmed, Armed, Triggered, Sound Alarm). Details regarding the specific outputs of the LED are relegated to the next section.

Finally, all of the timing parameters T_ARM_DELAY, T_DRIVER_DELAY, T_PASSENGER_DELAY and T_ALARM_ON can be reprogrammed to take values between 0 to 15 (seconds). This reprograming feature is not brought out to the client in the car. However, this reprogrammability means that if the client is not satisfied with some of the timing specifications of the system, then he or she may easily consult a technician and have the parameters reconfigured without major adjustments to the system.

2 Description

In this section, we assume that all signals coming from the car are properly debounced. This is achieved by a module (debounce.v) that latches onto an output value only if the input value has been stable for 0.01 seconds. Note that the signals associated with the resetting of the time parameters are not debounced, since their intended use is not bounce-sensitive.

We begin this section with a high-level block diagram (Figure 2) that illustrates how we have organized the above design into smaller modules. The arrows associated with each signal indicate whether it is an input or an output for the module. The notation "/n" indicates that the wire is a bus of n bits. All the inputs to the overall system are placed on the left end of the diagram. The outputs that it generates are placed on the right.

Note that we have chosen to implement FuelPumpLogic and AntiTheftFSM using Moore-type finite state machines (FSM).

In examining this diagram, we see an intermediate-level organization that consists of three major units. They are:

1. FuelPumpLogic;
2. AntiTheftFSM;
3. Programmable one-second timer.

It is clear that the first two units are "distinct" in the sense that the behavior of FuelPumpLogic does not

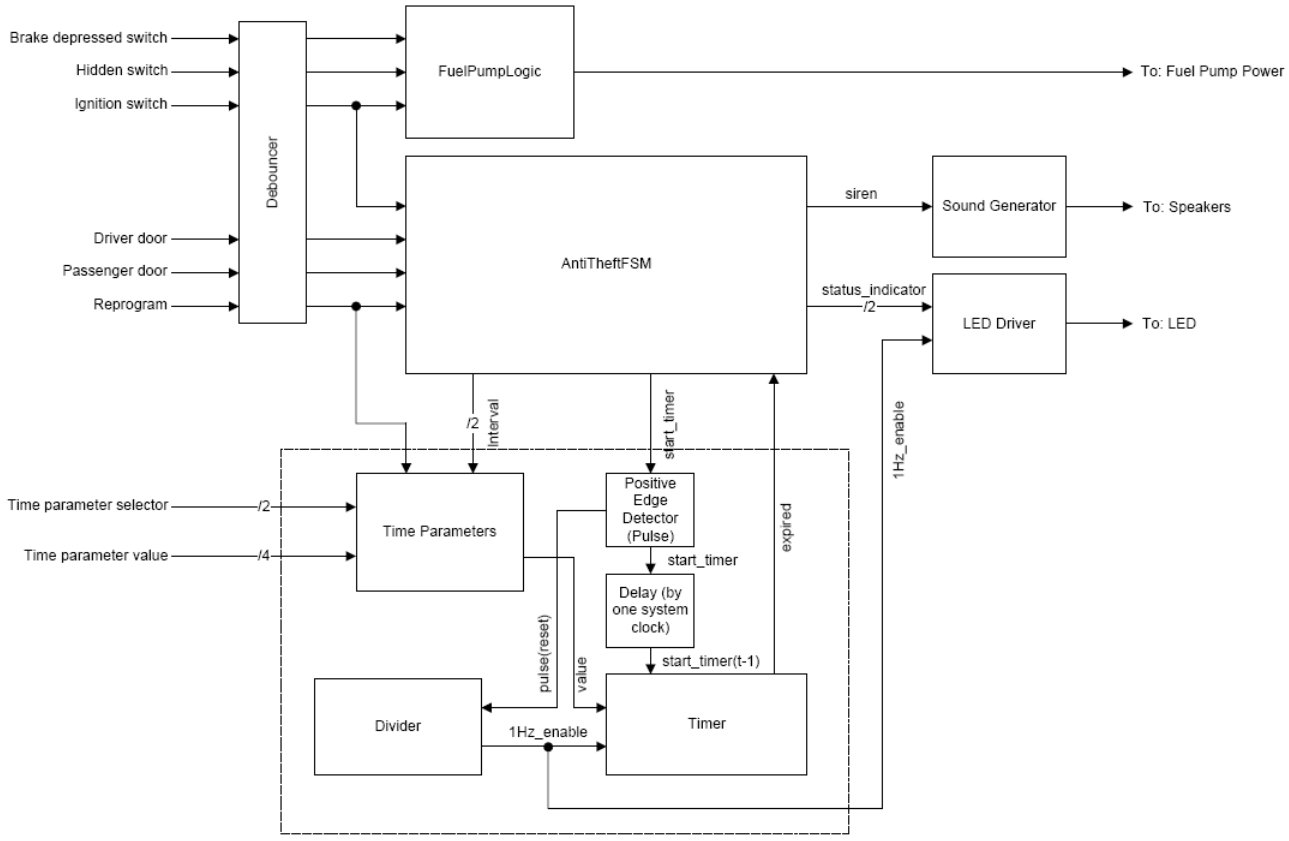

Figure 2: Organization of the design into small modules

depend on any information from AntiTheftFSM, and visa versa. We claim that the set of modules contained in the dashed box can also be recognized as a single entity (which we call "Programmable one-second timer.").

Among the above-mentioned major units, the Programmable one-second timer interacts only with AntiTheftFSM. Furthermore, there is only one pattern of interaction. Namely:

1. AntiTheftFSM puts a two-bit value on the 'interval' line that indicates which timing parameter is currently pertinent. In other words, it selects among T_ARM_DELAY, T_DRIVER_DELAY, T_PASSENGER_DELAY and T_ALARM_ON.
2. AntiTheftFSM drives the 'start_timer' line high, indicating to the timer subunit that it should begin a countdown towards zero using as the initial value the parameter selected in the previous step. The countdown proceeds at a rate of one decrement per second.
3. Once the timer has reached zero in its countdown, the Programmable one-second timer sends a high on the 'expired' line, indicating to the AntiTheftFSM that the requested amount of time has passed.

The additional details within the Programmable one-second timer module are due to the particular imple-

mentation of this functionality. Precisely because these extraneous details are irrelevant for the operation of the other major units, we have abstracted the programmable one-second timer as an important organizational entity. With these boundaries established, we now discuss each of the major units in greater depth.

2.1 Fuel Pump Logic ("FuelPumpLogic")

The FuelPumpLogic module is implemented as a simple two-state finite state machine. The two states are FUEL_PUMP_OFF and FUEL_PUMP_ON, whose outputs to the 'fuel_pump_power' line are exactly as the labels suggest. The transitions between the two states are as follows.

- If ignition is off, then the FSM transitions to FUEL_PUMP_OFF.
- If ignition is on, *and* 'brake_switch' and 'hidden_switch' are *simultaneously* high, then the FSM transitions to FUEL_PUMP_ON.

In all other cases the FSM retains the previous state.

2.2 Anti-Theft FSM ("AntiTheftFSM")

The Anti-Theft FSM is implemented as an eight-state FSM. Each state fully specifies the four output lines. (Two towards the transducers: status_indicator, siren. Two for communication with the Programmable one-second timer: start_timer, expired.) The transition diagram looks as in Figure 3. The indicated transitions are a literal translation of the description we gave in Section 1.

Furthermore, please note that it is possible to transition, from any initial state, to S_DISARMED by turning 'ignition' on. We have omitted this transition for the clarity of the diagram.

In each of the waiting states, the AntiTheftFSM drives a value onto the 'interval' bus, and sets 'start_timer' high. This indicates to the Programmable one-second timer unit to begin the countdown. The 'expired' signal is returned once the countdown is complete.

In particular, note that the state S_ALARM_ON catches for the 'expired' signal. This is necessary due to the particular implementation of the Programmable one-second timer. Due to its particular construction, it is possible for the 'expired' signal to remain high through a state transition. Hence, when more than one waiting states are visited in rapid sequence, they may both be cleared by a single 'expired' signal. (The correct procedure would be for the timer unit to countdown again.) We have modified the S_ALARM_ON state to prevent such errors.

2.3 Programmable one-second timer

2.3.1 TimeParameters

This module stores the four different timing parameter values that are used in the system. Its functionality is identical to a four-location 4-bit memory that comes preloaded with default values. The values present on the 'value' output bus is the parameter selected by 'interval' from the AntiTheftFSM.

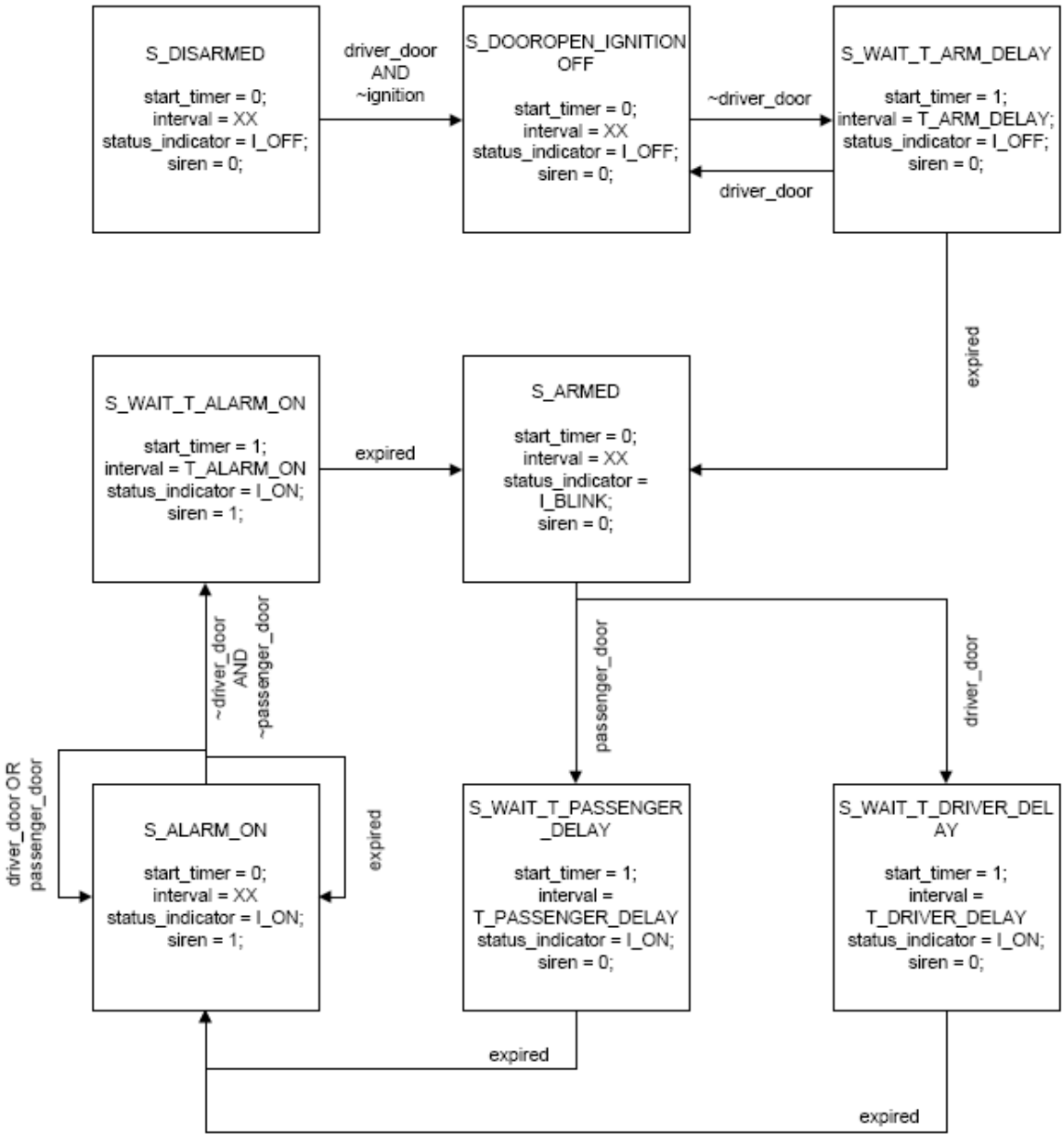

Figure 3: AntiTheftFSM State Transition Diagram

The user is able to modify the values by using 'Time_Parameter_Selector' and 'Time_Value', then pressing the Reprogram button. (See Figure 2.) When TimeParameters is reprogrammed, the AntiTheftFSM is reset to the S_ARMED state.

2.3.2 Divider

The Divider consists of an internal 25-bit counter that increments from 0 to 27,000,000. Since the counting proceeds at 27 MHz, this procedure provides a 1 Hz clock. As configured, this module produces a positive pulse with a period equal to one clock cycle of the 27 MHz clock. This occurs when the counter is at 27,000,000. We have also arranged for the counting to occur at the negative edge of the underlying clock, although the rest of the system proceeds at the positive edge. This design was so that the other modules will not fail to catch the narrow high in the 1 Hz pulse.

This module also admits a level reset that forces the internal counter to zero.

2.3.3 Timer

The timer is implemented as a two-state FSM. Its two modes are TIMER_SLEEP and TIMER_COUNT. By default, it lies in the sleeping state.

When the 'start_timer' signal from the AntiTheftFSM is high, the timer's response depends on its state. If it is TIMER_SLEEP, then it clears the 'expired' line, and obtains the initial value for the countdown via the 'count_value' lines from TimeParameters. It then transitions into the TIMER_COUNT state.

If 'start_timer' is high and we are in the TIMER_COUNT mode, we first check whether the countdown is over. If so, we output a high on the 'expired' line. Otherwise, we wait for the 1 Hz pulse from the Divider, in order to decrement our value.

2.3.4 Glue logic

In addition to the three modules above, this section of the system also employs two small components to provide glitch-free performance. First, on the positive edge of 'start_timer', we produce a positive edge pulse that goes to the reset pin of the Divider. This is to ensure that whenever we wish to count for n seconds, we count the entire time in full. The issue is illustrated in Figure 4.

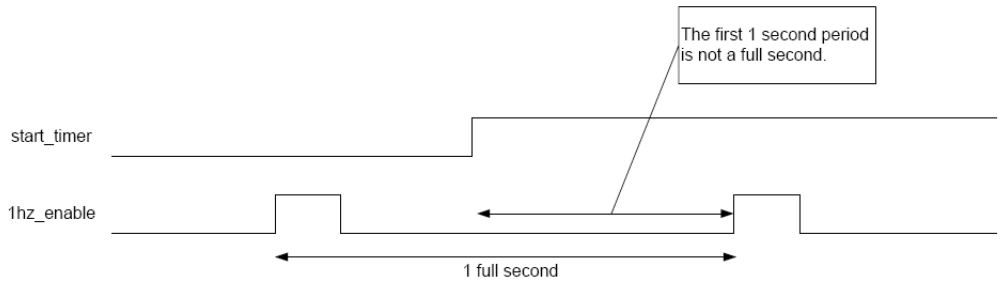

Figure 4: Unless the divider is properly reset with 'start_timer', we may not be waiting a full initial second

Positive Edge Pulse Generator. Our solution then is to reset the divider when 'start_timer' is asserted. We achieve this by producing a pulse at the positive edge of 'start_timer'. This is a standard part, producing a pulse using an AND between the input signal and the complement of the delayed version of that same signal.

Secondly, during testing, we encountered problems in which the correct timing value was not latched for the countdown. To understand this issue, suppose we have just entered the S_WAIT_T_DRIVER_DELAY state in the AntiTheftFSM. According to the FSM diagram, we then choose: 'interval = T_DRIVER_DELAY' and drive the 'start_timer' line high. Suppose 'interval' were previously T_ARM_DELAY. In the transition, we are simultaneously telling TimeParameters to change its output, and for Timer to latch onto the value that is present from TimeParameters. Hence, it is possible that we may latch onto the incorrect value of T_ARM_DELAY.

To circumvent this problem, we added a delay-by-one-clock module between 'start_timer' and Timer. This gives sufficient time for TimeParameters to provide the correct data on 'value'.

During our discussion of the AntiTheftFSM, we noted that we must break the path between two consecutive waiting states by an intermediate state that makes sure that 'expired' in unasserted. The above design choice in Programmable one-second timer is the root of that problem. Because of the delay unit, Timer may hold the 'expired' signal high for a longer time than is necessary for AntiTheftFSM. If 'expired' is retained through a second waiting state, the signal will instantly clear the second waiting state without starting the appropriate countdown.

2.4 Transducers

2.4.1 LED Driver

Each state in the AntiTheftFSM has a 2-bit 'status_indicator' output. The job of the LED Driver module is to interpret this signal from AntiTheftFSM and then to properly light the indicator LED. Two of the modes I_OFF and I_ON require no further explanation. In the third I_BLINK state, we use an internal 1-bit counter that increments when the 1 Hz pulse from Divider is high. We use the value of this counter to drive the LED. The result is an LED blinking at 2 Hz.

2.4.2 Siren Generator

We have implemented a tone generator that toggles at a rate of 2Hz between 800Hz and 1000Hz tones. The implementation involves two counters, one to control the transitions between the states and the other to produce the high frequency audio signal. The audio effect is similar to that of a police car siren.

3 Conclusion

In this document we have addressed the need for an anti-theft system that is unique from the standard factory unit. In particular, we described a system that, in addition to basic functionality, has an additional security layer through control of power to the fuel pump. We then detailed its implementation.

We suggest that in an upcoming implementation, the transition between S_DOOROPEN_IGNITIONOFF

and S_WAIT_T_ARM_DELAY involve 'passenger_door' as well, since as currently implemented, the countdown does not restart if the passenger leaves the car after the arming delay has been triggered.

Otherwise, we feel that the current state of the system faithfully achieves the client's specifications.

References

[1] MIT 6.111 Staff. *Laboratory Manual - Car Anti-Theft System.* 2008

A FuelPumpLogic

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
//
// Module name: FuelPumpLogic
// Description: Implements a very simple FSM for turning off/on the fuel pump.
//
//////////////////////////////////////////////////////////////////////////////////

module FuelPumpLogic(clk_27mhz, brake_switch, hidden_switch, ignition, fuel_pump_power);
 input clk_27mhz;
    input brake_switch;
    input hidden_switch;
    input ignition;
    output fuel_pump_power;

parameter FUEL_PUMP_OFF = 1’b0;
parameter FUEL_PUMP_ON  = 1’b1;

reg fuel_pump_power = FUEL_PUMP_ON;

always @ (posedge clk_27mhz)
begin
if(ignition)
begin
if(brake_switch && hidden_switch)
fuel_pump_power = FUEL_PUMP_ON;
end
else
fuel_pump_power = FUEL_PUMP_OFF;
end

endmodule
```

B AntiTheftFSM

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
//
// Module name: AntiTheftFSM
// Description: Implementation of the AntiTheft finite state machine. See documentation
//  for more details.
```

```
//
////////////////////////////////////////////////////////////////////////////////

module AntiTheftFSM(clk_27mhz, driver_door, passenger_door, ignition, reprogram, interval,
  start_timer, expired, status_indicator, siren, state);
    input clk_27mhz;
    input driver_door;
    input passenger_door;
    input ignition;
    input reprogram;
    output [1:0] interval;
    output start_timer;
    input expired;
    output [1:0] status_indicator;
    output siren;
 output [2:0] state;

//-----------------------------------------------------------------------
// The states of our AntiTheftFSM
//-----------------------------------------------------------------------
parameter S_DISARMED  = 3'd0;
parameter S_DOOROPEN_IGNITIONOFF  = 3'd1;
parameter S_WAIT_T_ARM_DELAY  = 3'd2;
parameter S_ARMED  = 3'd3;
parameter S_WAIT_T_PASSENGER_DELAY  = 3'd4;
parameter S_WAIT_T_DRIVER_DELAY     = 3'd5;
parameter S_ALARM_ON  = 3'd6;
parameter S_WAIT_T_ALARM_ON  = 3'd7;

//-----------------------------------------------------------------------
// Timing parameters
//-----------------------------------------------------------------------
parameter T_ARM_DELAY       = 2'b00;
parameter T_DRIVER_DELAY    = 2'b01;
parameter T_PASSENGER_DELAY = 2'b10;
parameter T_ALARM_ON        = 2'b11;

//-----------------------------------------------------------------------
// Status indicator modes
//-----------------------------------------------------------------------
parameter I_OFF   = 2'b00;
parameter I_BLINK = 2'b01;
parameter I_ON    = 2'b10;

reg [2:0] state = S_ARMED;
reg [2:0] next_state;

reg [1:0] interval = T_ARM_DELAY;
reg start_timer;
```

```
reg [1:0] status_indicator;
reg siren;

always @ (posedge clk_27mhz)
begin
if(reprogram) next_state <= S_ARMED;
else
begin
//-------------------------------------------------------------------------
// By default, we will stay at our current state. Transitions are
// explicitly coded in.
//-------------------------------------------------------------------------
next_state <= state;
case(state)
S_DISARMED:
begin
// Current outputs
start_timer <= 1'b0;
status_indicator <= I_OFF;
siren <= 1'b0;

// Next states
if(ignition) next_state <= S_DISARMED;
if(driver_door && ~ignition) next_state <= S_DOOROPEN_IGNITIONOFF;
end

S_DOOROPEN_IGNITIONOFF:
begin
// Current outputs
start_timer <= 1'b0;
status_indicator <= I_OFF;
siren <= 1'b0;

// Next states
if(ignition) next_state <= S_DISARMED;
if(~driver_door) next_state <= S_WAIT_T_ARM_DELAY;
end

S_WAIT_T_ARM_DELAY:
begin
// Current outputs
interval         <= T_ARM_DELAY;
start_timer      <= 1'b1;
status_indicator <= I_OFF;
siren            <= 1'b0;

// Next states
if(ignition)    next_state <= S_DISARMED;
if(driver_door) next_state <= S_DOOROPEN_IGNITIONOFF;
```

```
if(expired)     next_state <= S_ARMED;
end

S_ARMED:
begin
// Current outputs
start_timer      <= 1'b0;
status_indicator <= I_BLINK;
siren            <= 1'b0;

// Next states
if(ignition) next_state <= S_DISARMED;
if(passenger_door) next_state <= S_WAIT_T_PASSENGER_DELAY;
if(driver_door) next_state <= S_WAIT_T_DRIVER_DELAY;
end

S_WAIT_T_PASSENGER_DELAY:
begin
// Current outputs
interval         <= T_PASSENGER_DELAY;
start_timer      <= 1'b1;
status_indicator <= I_ON;
siren            <= 1'b0;

// Next states
if(ignition) next_state <= S_DISARMED;
else if(expired)  next_state <= S_ALARM_ON;
end

S_WAIT_T_DRIVER_DELAY:
begin
// Current outputs
interval         <= T_DRIVER_DELAY;
start_timer      <= 1'b1;
status_indicator <= I_ON;
siren            <= 1'b0;

// Next states
if(ignition) next_state <= S_DISARMED;
else if(expired)  next_state <= S_ALARM_ON;
end

S_ALARM_ON:
begin
// Current outputs
start_timer      <= 1'b0;
status_indicator <= I_ON;
siren            <= 1'b1;
```

```
// Next states
if(ignition) next_state <= S_DISARMED;
else if( expired ) next_state <= S_ALARM_ON;
else if( ~passenger_door && ~driver_door ) next_state <= S_WAIT_T_ALARM_ON;
end

S_WAIT_T_ALARM_ON:
begin
// Current outputs;
interval         <= T_ALARM_ON;
start_timer      <= 1'b1;
status_indicator <= I_ON;
siren            <= 1'b1;

// Next states
if(ignition) next_state <= S_DISARMED;
if(expired)  next_state <= S_ARMED;
end
endcase
end

//-----------------------------------------------------------------------
// Move to our new state
//-----------------------------------------------------------------------
state <= next_state;
end

endmodule
```

C TimeParameters

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
//
// Module name: TimeParameters
// Description: A four-location memory that holds 4-bit values.
//
//////////////////////////////////////////////////////////////////////////////////

module TimeParameters(clk, timeSelector, timeValue, reprogram, interval, value, selected_value);
    input clk;
 input [1:0] timeSelector;
    input [3:0] timeValue;
    input reprogram;
    input [1:0] interval;
```

```
    output [3:0] value;
 output [3:0] selected_value;

parameter T_ARM_DELAY       = 2'b00;
parameter T_DRIVER_DELAY    = 2'b01;
parameter T_PASSENGER_DELAY = 2'b10;
parameter T_ALARM_ON        = 2'b11;

parameter T_ARM_DELAY_DEFAULT       = 4'd6;
parameter T_DRIVER_DELAY_DEFAULT    = 4'd8;
parameter T_PASSENGER_DELAY_DEFAULT = 4'd15;
parameter T_ALARM_ON_DEFAULT        = 4'd10;

reg [3:0] t_arm_delay       = T_ARM_DELAY_DEFAULT;
reg [3:0] t_driver_delay    = T_DRIVER_DELAY_DEFAULT;
reg [3:0] t_passenger_delay = T_PASSENGER_DELAY_DEFAULT;
reg [3:0] t_alarm_on        = T_ALARM_ON_DEFAULT;
reg [3:0] value;
reg [3:0] selected_value;

always @ (posedge clk)
begin
if(reprogram)
case(timeSelector)
T_ARM_DELAY:       t_arm_delay       <= timeValue;
T_DRIVER_DELAY:  t_driver_delay    <= timeValue;
T_PASSENGER_DELAY: t_passenger_delay <= timeValue;
T_ALARM_ON:        t_alarm_on        <= timeValue;
endcase

case(interval)
T_ARM_DELAY:       value <= t_arm_delay;
T_DRIVER_DELAY:    value <= t_driver_delay;
T_PASSENGER_DELAY: value <= t_passenger_delay;
T_ALARM_ON:        value <= t_alarm_on;
endcase

//------------------------------------------------------------------------
// This functionality is to print the appropriate parameter to
// the hex displays.
//------------------------------------------------------------------------
case(timeSelector)
T_ARM_DELAY:       selected_value <= t_arm_delay;
T_DRIVER_DELAY:    selected_value <= t_driver_delay;
T_PASSENGER_DELAY: selected_value <= t_passenger_delay;
T_ALARM_ON:        selected_value <= t_alarm_on;
endcase
end
```

```
endmodule
```

D Divider

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 PS2, Problem 1c
//
// Module name: counter
// Description: counter accepts a 27mhz incoming ’clock’, and produces a positive
//  pulse ’enable’ for one clock period every 1 second. The internal
//  counter can be reset by a level input ’reset’.
//
//////////////////////////////////////////////////////////////////////////////////

module counter(clock, reset, enable);
   input clock;
   input reset;
   output enable;
reg [24:0] q = 0;

always @ (negedge clock)
begin
if(reset)
q <= 0;
else
q <= q+1;

// This counter counts up to 27,000,000 in hex, and then resets.
if( q > 25’d27_000_000 )
q <= 25’b0;
end

assign enable = (q == 25’d27_000_000);

endmodule
```

E Timer

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
```

```
//
// Module name: Timer
// Description: This module when triggered by 'start_timer', will count down from
// the initial 'count_value' towards zero. The count occurs whenever
// 'clk_1hz' is high. When it has reached zero, will output a high
// 'expired'. The 'current_count' output is used for HEX displays on
// the labkit.
//
//////////////////////////////////////////////////////////////////////////////////

module Timer(clk_27mhz, clk_1hz, count_value, start_timer, expired, current_count);
    input clk_27mhz;
    input clk_1hz;
    input [3:0] count_value;
    input start_timer;
    output expired;
 output [3:0] current_count;

reg expired = 0;
reg [3:0] current_count;

reg [3:0] count;

parameter TIMER_SLEEP = 1'b0;
parameter TIMER_COUNT = 1'b1;
reg timer_mode = TIMER_SLEEP;

always @ (posedge clk_27mhz)
begin
if(start_timer)
begin
case(timer_mode)
// If we get the start_timer signal, and the timer is currently sleeping,
// then we initialize the value of 'count' and change our state.
TIMER_SLEEP:
begin
timer_mode <= TIMER_COUNT;
expired    <= 0;
count      <= count_value;
end
// We are already in the counting mode, then we see if the counter has
// reached zero. If so, we set the expired signal high, and change the timer
// mode back into sleep.
//
// If the count is not zero, then we see if clock_1hz is high. (Recall that the
// 1Hz signal is high for one period of the 27MHz clock.)
TIMER_COUNT:
begin
current_count <= count;
```

```
if(count == 0)
begin
expired    <= 1;
end
else
begin
if(clk_1hz) count <= count - 1;
end
end
endcase
end
else
begin
// The timer is now turned off.
expired       <= 0;
timer_mode    <= TIMER_SLEEP;
current_count <= count;
end
end
endmodule
```

F LED Driver

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
//
// Module name: StatusLEDDriver
// Description: Appropriately lights the Status LED depending on ‘status’. It
// currently implements OFF, BLINK (0.5Hz), ON.
//
//////////////////////////////////////////////////////////////////////////////////

module StatusLEDDriver(clock_27mhz, clock_1hz, status, led_out);
    input clock_27mhz;
    input clock_1hz;
    input [1:0] status;
    output led_out;

// Status indicator modes
parameter I_OFF   = 2’b00;
parameter I_BLINK = 2’b01;
parameter I_ON    = 2’b10;

reg blink_state = 1’b0;
reg led_out     = 1’b0;
```

```
always @ (posedge clock_27mhz)
begin
case(status)
I_OFF: led_out <= 1'b0;
I_ON:  led_out <= 1'b1;
I_BLINK:
begin
if(clock_1hz) blink_state <= blink_state + 1;
led_out <= blink_state;
end
default: led_out <= 1'b0;
endcase
end
endmodule
```

G Siren Generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Tony Kim
// 6.111 Lab2
//
// Module name: SirenGenerator
// Description: Produces a warbling tone between 800 and 1000Hz. The two tones
// alternate every 0.25 second.
//
//////////////////////////////////////////////////////////////////////////////////

module SirenGenerator(clock_27mhz, siren_in, siren_out);
    input clock_27mhz;
    input siren_in;
    output siren_out;

parameter STATE_800HZ  = 1'b0;
parameter STATE_1000HZ = 1'b1;

//---------------------------------------------------------------------
// The following periods are computed with a 27MHz base clock
// The frequency of state toggling is 2Hz
//---------------------------------------------------------------------
parameter STATE_TOGGLE_PERIOD = 25'd6750000;
parameter PERIOD_800HZ  = 25'd33750;
parameter PERIOD_1000HZ = 25'd27000;

reg state = STATE_800HZ;
reg [25:0] state_count  = 25'b0;
```

```
reg siren_out = 1;

reg [25:0] siren_count  = 25'b0;

always @ (posedge clock_27mhz)
begin
if(siren_in)
begin
state_count <= state_count + 1;
// We first check toggle between the 800Hz and the 1000Hz sounds
if(state_count > STATE_TOGGLE_PERIOD)
begin
case(state)
STATE_800HZ:  state <= STATE_1000HZ;
STATE_1000HZ: state <= STATE_800HZ;
endcase
state_count <= 25'b0;
end
// Otherwise we produce the sound at the current frequency;
else
begin
case(state)
STATE_800HZ:
begin
if(siren_count > PERIOD_800HZ)
begin
siren_out   <= ~siren_out;
siren_count <= 25'b0;
end
else
siren_count <= siren_count + 1;
end
STATE_1000HZ:
begin
if(siren_count > PERIOD_1000HZ)
begin
siren_out   <= ~siren_out;
siren_count <= 25'b0;
end
else
siren_count <= siren_count + 1;
end
endcase
end
end
end
endmodule
```