Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

Lecture 7: Non-blocking debounce, heartbeat and counters

Contents

This lecture will include:-

- Non-blocking switch debouncer
- Heartbeat
- Button counter

- All these can be achieved using Finite State Machines (FSM)

The need for parallelism

The ‘heartbeat’ is code to blink the dot-point on the seven segment display to indicate all is well with the system.

- Consider some simple heartbeat code:-

```
void loop()
{
    writeToShiftRegister(SEG_DP);
    delay(500);
    writeToShiftRegister(0);
    delay(500);
}
```

This would implement a ‘heartbeat’ but it would be blocking.

The need for parallelism

- Consider the switch debouncer from the Formative Assignment Part 2.

```
void waitForKeypress()
{
  while (HAL_sw1Pressed);
  int count = DEBOUNCE_TIME;
  while (count > 0)
  {
    count--;
    if (HAL_sw1Released)
    {
      count = DEBOUNCE_TIME;
    }
    HAL_msDelay(1);
  }
}
```

This would implement a debounced switch but it would also be blocking.

The need for parallelism

Combining the two.

```
void loop()
{
    writeToShiftRegister(SEG_DP);
    delay(500);
    writeToShiftRegister(0);
    delay(500);

    waitForKeypress();
}
```

The need for parallelism

In this scenario:-

- The switch debouncer would have to wait for 1 second for the heartbeat to complete its cycle.
- Similarly, the heartbeat module would have to wait for the `waitForKeypress()` function, i.e. the debounce function to complete before it could commence its cycle.

This is clearly an undesirable situation which could be avoided if we could make both pieces of code run in a non-blocking fashion.

Formulating an FSM for the debouncer

See the DocCam presentation.

Non-blocking Debounce Code

Instead of using magic numbers 0, 1 and 2 for states, it is good practice to use more meaningful state names.

This could be done either by `#define`s, or my creating an enumerated type (as in the example below).

```
typedef enum
{
    NOT_PRESSED,        // Equivalent to 0
    PARTIAL_PRESS,      // Equivalent to 1
    DEBOUNCED_PRESS     // Equivalent to 2
} switch_state_t;
```

Non-blocking Debounce Code

The Finite State Machine (FSM) is shown below.

```
switch (state) {
 case NOT_PRESSED:
  if (HAL_SW1_RELEASED) state = NOT_PRESSED;
  else {
   debounce_count = millis();
   state = PARTIAL_PRESS;
  }
  break;

 case PARTIAL_PRESS:
  if (HAL_SW1_RELEASED) state = NOT_PRESSED;
  else if ((millis() - debounce_count) < debounce) state = PARTIAL_PRESS;
  else state = DEBOUNCED_PRESS;
  break;

 case DEBOUNCED_PRESS:
  if (HAL_SW1_RELEASED) state = NOT_PRESSED;
  else state = DEBOUNCED_PRESS;
  break;

 default:
  state = NOT_PRESSED;
  break;
 }
```

Non-blocking Debounce Code

Once the FSM has been designed and the code implemented, create the code to generate the timings. Either:-

1. Use the original in-line method (not recommended),
2. Use partial encapsulation (using the Concurrent module), or
3. Use full encapsulation.

You have two switches on the board. Each switch will need its own debouncer. You will need to develop the FSM for the second debouncer.

Food for thought: Do you opt for two timing modules, one per debouncer with one FSM in each? Or do you opt for a single timing module driving both FSMs?

Heartbeat

The heartbeat is an indication that the system is functioning properly and hasn’t frozen.

For this project, the dot point on the seven-segment display should repeatedly turn on and off for prescribed periods of time. Please refer to the main specification.

The heartbeat module is required to run concurrently with other modules. A finite state machine is required to control its operation. Please see DocCam.

Heartbeat FSM Implementation

The code below shows an overview of the heartbeat FSM code. The details of illuminating and extinguishing the dot-point are left to you!

```
switch (state)
{
  case 0:
    // Add code to illuminate the dot-point
    state = 1;
    break;
  case 1:
    // Add code to extinguish the dot-point
    state = 0;
    break;
  default:
    state = 0;
}
```

You will need to add code to control the timings. Either use the original method (not recommended), or partial or full encapsulation.

Button Counter

The button counter is specified count the number of valid debounced keypresses it recognizes. The button counter will not use the associated switch per-se, but instead will respond to the state of the debouncer.

Care needs to be taken here.

Suppose the counter is programmed to count when the debouncer is in the debounced press state and the button is held, then the count will rapidly increase. This is most undesirable.

The count needs to be incremented on the transition from partial press to debounced press. And this requires the development of a finite state machine……

Button Counter

- The button counter module should be written to allow concurrent operation with other modules.
- The code to control the timings will need to be developed. This could take the form of:
  1. the original in-line method (not recommended),
  2. partial encapsulation (using the Concurrent module), or
  3. full encapsulation
- The FSM for the button counter will also need to be developed.

Avoiding Race Conditions

- Whenever two or more processes compete to access a single resource there may be potential for a race condition that could lead to undesirable results.
- For example, the heartbeat module might correctly illuminate or extinguish the dot-point but inadvertently extinguish the other 7 segments in the process.
- For example, the counter module might correctly illuminate the segments corresponding to the number being displayed, but inadvertently extinguish the dot-point in the process.
- If this happens quickly, the seven-segment display will appear to perform erratically.

Avoiding Race Conditions

- To minimize the chance of a race condition, firstly there should only be one function call per loop that updates the seven-segment display. This should be placed at the end of `loop()`.

Either:-

- Have one variable containing the heartbeat status, and another variable containing the bit pattern for the character to be displayed. These could be combined as part of the display update. Or,
- Have a single variable containing the overall bit pattern. The heartbeat module ***only*** alters the bit associated with the dot-point. The counter module determines the existing state of the dot point then appends it to the character it is writing.

Summary

- During this session we saw how Finite State Machines (FSMs) can be used to develop code for:-
  - Debouncer
  - Heartbeat Module
  - Button Counter

- We now have the potential to have multiple modules running in parallel.
- Next week, we will look at how a task scheduler can be developed that will activate / deactivate code modules to select which ones will run.