Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

Lecture 9: Robust Coding

Contents

This lecture will include:-

- Rollover-safe timing
- Preserving D0 and D1
- Preventing race conditions on the seven-segment display

Time and Timekeeping

- Fundamentally time can be defined by the “Arrow of time” concept.

  https://en.wikipedia.org/wiki/Arrow_of_Time
- To tell time we need an energy source and a counting mechanism.

  https://www.youtube.com/watch?v=URK9Z2G71j8
- The counting mechanism needs to “remember the count”
- No mechanism can have unlimited memory
- Limited memory has the overflow problem.

Arduino Timekeeping

- Arduino uses **`millis()`** or **`micros()`** which respectively return the number of milliseconds or microseconds since last power-up.
- Both of these functions return **`unsigned long`** values.
- In the case of **`millis()`**, the returned value overflows in 2^{32}-1 milliseconds; around 49.7 days.
- The problem here is that embedded systems ***must*** work for as long power is applied.
- The problem is fundamental to timekeeping as no mechanism, including hardware timers, can have unlimited memory.

Module timekeeping

- We need to know when to run our modules.
- For that we need to measure time durations, i.e. differences in time instances.
- We will need to use 2's complement
- Represent numbers on a number circle rather than a number line.

Rollover-safe Timing

Up to now, in our modules we have used code like:-

```
...
    {
      unsigned long m = millis();
      if ((m - module_time) > module_delay) { ... }
      else ...
    }
...
```

We can now make our code rollover-safe

```
...
    {
      unsigned long m = millis();
      if (((long)(m - module_time)) > module_delay) { ... }
      else ...
    }
...
```

Rollover-safe Timing

- The cast `((long)(m - module_time))` forces the compiler to use 2's-complement.

- This limits our range since 2's complement uses 1 bit for the sign. i.e. we have halved the maximum of what the difference `(m - module_time)` can be.

- The calculation works as long as `module_delay` is less than 2^{31}-1 milliseconds; which is around 24.9 days.

- With this method we can measure any duration in perpetuity as long as the duration is less than 2^{31}-1 milliseconds.

Preserving D0 and D1.

- Arduino digital pins D0 and D1 are respectively controlled by bits 0 and 1 of Port D.
- Used respectively as TXD and RXD for serial communications.
- For our coursework, regarding PORT D, we have
  - Bit 7 – Green LED 2
  - Bit 6 – Yellow LED 2
  - Bit 5 – Red LED 2
  - Bit 4 – Green LED 1
  - Bit 3 – Yellow LED 1
  - Bit 2 – Red LED 1
  - Bit 1 – **RXD**
  - Bit 0 – **TXD**

Preserving D0 and D1.

- Suppose the following bare-metal approach is used to turn on Red LED 2:

```
PORTD |= B00100000;
```

- This is the equivalent of

```
PORTD = PORTD | B00100000;
```

- This takes the current state of PORTD, logically ORs bit 5 (i.e. switches it on), then stores the result back in PORTD.
- Net result: bit 5 becomes a ‘1’, the status of the other 7 bits is retained.
- This approach therefore does not affect D0 and D1.

Preserving D0 and D1.

- The following bare-metal approach is used to control all the LEDs in a single operation.
- Suppose we turn on both Red LEDs and Yellow LED 2:

```
PORTD = B01100100;
```

- This value `B01100100` is written directly to PORTD.

- The result of this is:-
  - The Red LEDs and the Yellow LED 1 turn on.
  - The other LEDs turn off (a desired result).
  - However, the states of D0 and D1 are overwritten.

- Whilst (for this application) we can get away with this, care should be taken to preserve the status of D0 and D1.

Preserving D0 and D1.

The solution:

- Obtain the states of D0 and D1.

```
(PORTD & B00000011)
```

- Bitwise OR it with the required bit-pattern.
- Store the result in PORTD

```
PORTD = (PORTD & B00000011) | B01100100;
```

- This 6 most significant bits of `B01100100` are written to Bits 7 through to 2 of PORTD, whilst D0 and D1 are retained.
- Further care could be applied by masking off the 2 least significant bits of the required bit-pattern:

```
PORTD = (PORTD & B00000011) | (B01100100 & b11111100);
```

Avoiding race conditions on 7-segment display

- For our coursework we have several modules requiring access to the seven segment display.
- The counter and orientation detectors place characters on the main seven segments.
- The heartbeat module blinks the dot point.
- Without care, the counter and orientation modules could overwrite the dot point when writing a character.
- Without care, the heartbeat module could overwrite the character when updating the dot point.

Avoiding race conditions on 7-segment display

There are several ways to achieve this. One of the simplest involves:

- Having only a single call to the shift register. This could be placed as the last statement of the `loop()` function.
- Declare an `unsigned char` (or `byte`) variable to store the current state of heartbeat (in its most significant bit).
- Declare a second `unsigned char` (or `byte`) variable to store the current character (in its 7 least significant bits).
- Bitwise OR the two variables and send the combined result to the shift register.

Summary

During this session we have considered some coding techniques that can be used to improve the robustness of your solution.

- Ensure that timings are rollover-safe.
- Ensure that D0 and D1 are not inadvertently overwritten.
- Take steps to prevent race conditions on the seven segment display.