

Tutorial 2

ENGN8537

Embedded Systems

Tutorial Question

Consider the following C code, written for an AVR microcontroller on the bare metal. The `_delay_ms()` macro works by simply executing a number of “no operation” instructions to kill the right amount of time. This number is computed at compile-time based on the processor’s clock speed.

1. Will enabling interrupts have any effect on how long the delay will actually take to complete? Is the delay likely to be longer or shorter than intended?
2. Say the processor is connected to a motor and that an interrupt is triggered each time the motor completes a revolution. The motor has a top speed of 1000RPM and the interrupt, when triggered, takes 630uS to complete. There is no coupling between the waveform and the motor speed. What is the worst-case error in timing, as a percentage, of the output waveform?
3. Assume the motor is equally likely to be at any speed in its range at any time, what’s the average delay actually going to be? What’s the standard deviation going to be? What’s the likelihood, in percent, that any given delay is going to be off by more than 1%?

Tutorial Question

```
while(1) {  
    pin_high();  
    _delay_ms(1);  
    pin_low();  
    _delay_ms(1);  
}
```

Tutorial Question

Consider the following C code, written for an AVR microcontroller on the bare metal. The `_delay_ms()` macro works by simply executing a number of “no operation” instructions to kill the right amount of time. This number is computed at compile-time based on the processor’s clock speed.

1. Will enabling interrupts have any effect on how long the delay will actually take to complete? Is the delay likely to be longer or shorter than intended?

The delay will be longer than expected; it will be exactly the normal delay time, plus the total amount of time spent in interrupts during the delay.

The `_delay_ms` macro tries to keep track of time within the thread of execution that called it (there are no hardware timers for example). As such, it doesn’t delay for any amount of real time, but delays for an amount of time that the thread is actually executing.

Tutorial Question

Say the processor is connected to a motor and that an interrupt is triggered each time the motor completes a revolution. The motor has a top speed of 1000RPM and the interrupt, when triggered, takes 630uS to complete. There is no coupling between the waveform and the motor speed. What is the worst-case error in timing, as a percentage, of the output waveform?

As on the previous slide, the delay will be the actual delay plus the amount of time spent in interrupts. At maximum speed, the motor will generate 1000 interrupts per minute, or 630mS per minute.

$$0.630/60 = 1.05\%$$

Tutorial Question

Assume the motor is equally likely to be at any speed in its range at any time, what's the average delay actually going to be? What's the standard deviation going to be? What's the likelihood, in percent, that any given delay is going to be off by more than 1%?

The minimum delay is 1ms, the maximum delay is 1.0105 ms and the distribution of delays within that is uniform. The 1% error mark is 1.01ms. As such, the likelihood is simply $0.0005/0.01 = 5\%$

Key point: With typical numbers, most Embedded Systems are sufficiently close to real time that they get away with it. Depending on the project priorities, getting away with it might be the right thing to do, but at least do it consciously.

Tutorial Question

Consider the following two threads. Is access to all variables safe? Can either thread deadlock? If so, give one example set of preemption points that would trigger it.

```
Semaphore lock_chick, lock_fox, lock_egg;  
Integer Fox, Chicken, Egg, Snake;
```

Thread 1

```
wait(lock_chick);  
wait(lock_fox);  
Fox += Chicken;  
wait(lock_egg);  
signal(lock_chick);  
Egg -= Snake + Fox;  
signal(lock_egg);  
signal(lock_fox);
```

Thread 2

```
wait(lock_egg);  
wait(lock_fox);  
Fox = Egg;  
signal(lock_fox);  
wait(lock_chick);  
Egg = Chicken;  
signal(lock_chick);  
signal(lock_egg);
```

Tutorial Question

All of the locked variables are correctly bracketed by their own lock. The Snake variable doesn't have a lock, but is only accessed from a single thread and therefore can't race. Access is safe from races.

```
Semaphore lock_chick, lock_fox, lock_egg;  
Integer Fox, Chicken, Egg, Snake;
```

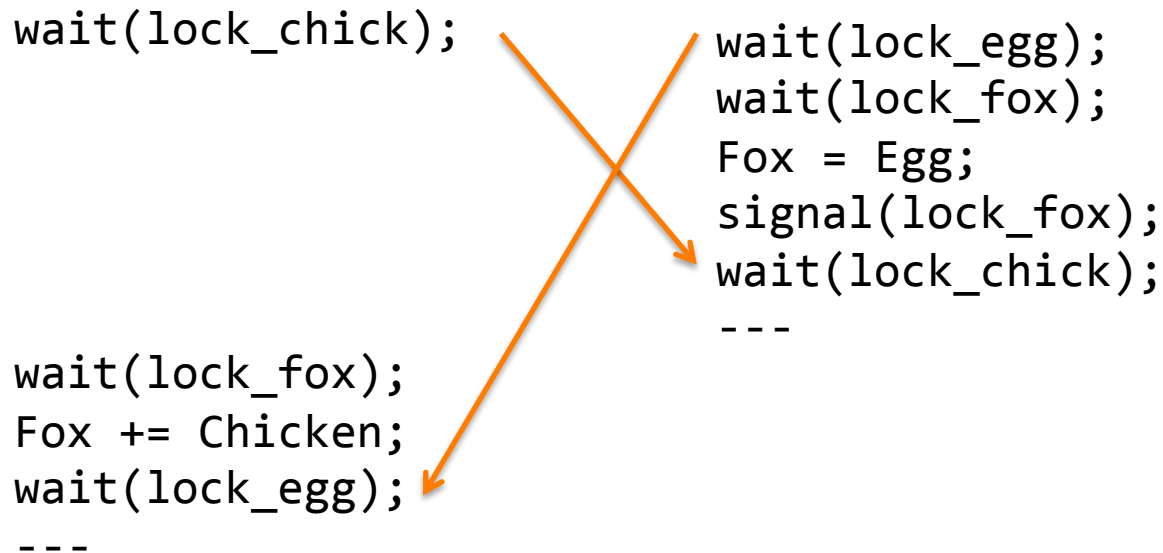
Thread 1

```
wait(lock_chick);  
wait(lock_fox);  
Fox += Chicken;  
wait(lock_egg);  
signal(lock_chick);  
Egg -= Snake + Fox;  
signal(lock_egg);  
signal(lock_fox);
```

Thread 2

```
wait(lock_egg);  
wait(lock_fox);  
Fox = Egg;  
signal(lock_fox);  
wait(lock_chick);  
Egg = Chicken;  
signal(lock_chick);  
signal(lock_egg);
```


Tutorial Question



```
wait(lock_chick);  
  
wait(lock_fox);  
Fox += Chicken;  
wait(lock_egg);  
---  
  
wait(lock_egg);  
wait(lock_fox);  
Fox = Egg;  
signal(lock_fox);  
wait(lock_chick);  
---
```

Thread 1 holds `lock_chick` and is waiting on `lock_egg`. Thread 2 holds `lock_egg` and is waiting on `lock_chicken`. This is an ABBA deadlock, just slightly obscured, as Thread 1 takes chick/egg and Thread 2, egg/chick. If the ordering of locks was changed to be consistent, the deadlock couldn't occur. Note this is one of two possible locks, the other being an ABBA deadlock between fox and egg.