



Discussion Topics: CF1, CF2, IRQ4, SCI UART, LPSTOP, Spurious Interrupts
Date: February 2007

Customer observation

My ISR turns off the interrupt-enable:

```
void Irq4RxISR(void)
{
    PinIO(IRQ4RXD);
    RTE();
}
```

But this leaves a window open in the main program between PinBus and LPStopCSE calls

```
PinBus(IRQ4RXD); // configuring as bus fnx turns on ISR
*(ushort *) 0xffffe00c = 0xF000; // force CF card into Card-1 active mode
LPStopCSE(FullStop);
```

This allows an Irq4 interrupt from the receive buffer to happen before the LPStopCSE, thus turning off the interrupt. Then the LPStop executes, with no way to wake it up!

Discussion

There are details of the seemingly simple IRQx and LPStop operations that you should be aware of and consider to make sure that this problem is truly eradicated. I will focus on IRQ4 for this discussion, but the basic principles apply to all of the other external interrupts with the exception of IRQ7 which is stranger still.

On the CF1/CF2, IRQ4 is connected directly to the 68332's SCI UART RXD input. However, IRQ4 plays no role in the UART operations and interrupts. The UART interrupts and processing are performed entirely within the SCI module after detection of the start bit and after waiting the baud appropriate time for the number of bits making up a serial word which is typically one start, eight data, and one stop bit. The IRQ4 connection exists solely to provide a simple mechanism to wake the CPU from LPStop on incoming serial words because the SCI UART cannot. When the MAX3222 RS232 driver's receiver is enabled, which it always is unless you explicitly turn it off with EIAEnableRx(false) in your program, RSRXD drives into the MAX3222 at RS232 levels and gets inverted and brought down to 3.3V CMOS to feed into the 68332's SCI UART RXD input and to IRQ4.

IRQ4 can be configured as a simple digital I/O pin, or as a negative-level sensitive interrupt request pin (as opposed to edge-sensitive). When it's configured as an interrupt, it will request a level-4 IRQ exception from the prioritizer logic which will ignore requests that are less than or equal to the current mask level. The default mask at application start is zero, enabling all interrupts, although no interrupts actually run at startup until you explicitly start or enable them. In normal operation, interrupt requests must remain low for two consecutive rising edges of the (typically 16MHz) system clock and will only be accepted between processor instructions. In LPStop, interrupts making it through the prioritizer are immediately recognized but the CPU may need between a few cycles (FastStop) and a few milliseconds (FullStop) to resume processing. If the interrupt goes away between the initial acceptance and the CPU coming back online, the CPU won't know what to do with the pending interrupt request and will trigger a spurious interrupt exception instead of the normal IRQx exception.



The default spurious interrupt exception is the hopefully-not-too-familiar CRASH display and program termination. This is why you will sometimes see simple CF2 UART examples that setup the spurious interrupt exception vector to use the same handler as the IRQ4 handler to deal with just such situations. This is only a valid practice if you know that your system will not have other interrupts or submodule operations with the potential to generate spurious interrupts and this can become very difficult to do for anything but simple example or limited functionality programs, and especially for programs with multiple authors that grow or change over time (i.e. real programs). In the scenario like the IRQ4 handler where the proper response is to revert the pin to a simple input after it has served its primary and critical purpose of waking the CPU, having it deactivated early can be devastating.

As mentioned above, IRQ4 plays no role in the normal UART operations and interrupts. It just provides an incidental wakeup stimulus by letting the start and subsequent low bits in the serial word pull the interrupt low, which if the mask level permits, will restart the CPU running. There is no inherent or explicit synchronization. The serial stream just barks repeatedly until the CPU wakes and throws a shoe at it. In an ideal system, the wakeup interrupt would be latched on the first falling edge, and the interrupt handler would stop future interrupts and then clear the latch for subsequent use. Some complex CFX systems use 7474-like logic to implement schemes like this. However, most applications involving the UART and LPStop do not do that but instead use the unsynchronized PinBus/PinIO technique and the handlers don't perform any interesting actions.

As you've unfortunately discovered, there's a non-obvious path to thwarting this approach which is to have an interrupt arrive after the PinBus makes IRQ4 interrupts possible but before execution of the LPStop instruction. When it does execute, which will in that case be after the handler has reverted IRQ4 to PinIO, there will be nothing to stimulate the CPU into waking from the LPStop. As long as IRQ4 is low, you will constantly be rerunning the handler and won't actually get to the PinIO until the first high bit in the serial stream. In the extreme case of BREAK, you won't get to the PinIO until the BREAK concludes, and in both cases your system won't be able to service any interrupts below level-4 until the PinIO stops the flood.

The CommLogger example and some other later code we've produced takes a different approach that you might want to consider. By disabling interrupts below level-5 just before the pre-LPStop setup code, you prevent interrupt recognition until the final LPSTOP instruction which atomically reenables interrupts at level-0 so that you won't get bitten by the inability to be woken from with either early or spurious interrupts. The CommLogger code should be on your hard drive at

C:\Program Files\Persistor\MotoCross Support\CFX\Examples\CommLogger.

Search on the keyword IRQ4RXD and see if that makes sense for your application.