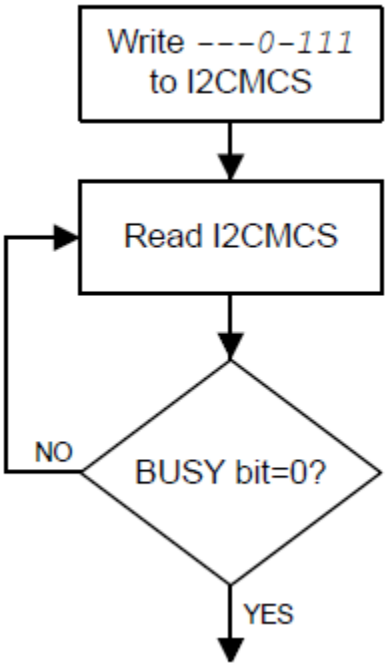
Solution for I2C BUSY Status Latency

July 27, 2018

One issue I ran into while working with I2C for the Tiva C LaunchPads was being able to reliably detect when the master device has finished sending or receiving a byte of data. The datasheets for the TM4C123GH6PM and TM4C1294NCPDT microcontrollers contain a flowchart showing a loop in which the BUSY flag in the Master Control/Status (MCS) register is tested . The implication is that after setting the MCS register, the byte transfer begins and the master immediately becomes busy, as indicated by the BUSY flag. After the byte transfer is completed, the BUSY bit clears and the MCS register can be read back to check for error status. If no error occurs, then the next byte may be transferred.



Waiting for the BUSY bit to clear.

What isn't mentioned in the datasheet is that there is some latency between the time the MCS register is assigned and when the BUSY flag turns on. If you're not careful, your code will detect BUSY bit = 0, and follow the 'YES' branch of the flowchart before the byte transfer has actually completed. In my own experience, when this happens during a multi-byte write transaction, only the last byte of data will be sent on the bus. In short, waiting for the BUSY flag to turn off immediately after assigning control flags to the MCS register is unreliable.

Other users have complained about this problem as well. A web search for 'tiva c i2c busy flag latency' will turn up a few examples. One solution mentioned is to wait for the BUSY flag to set, then wait for it to clear, in two successive while loops, like this:

```
// Set the control flags
i2c->MCS = 0x3;

// Wait for the BUSY bit to set...
while (!(i2c->MCS & I2C_MCS_BUSY));

// Wait for the BUSY bit to clear...
while (i2c->MCS & I2C_MCS_BUSY);
```

The drawback that I encountered with this solution is that sometimes the BUSY flag clears immediately, causing the first loop to go on forever. In other words, if the BUSY flag *never* sets, the first loop becomes infinite and execution hangs. No good.

Another solution says to read back the MCS register to flush the preceding write. The readback is supposed to overcome the latency caused by the TM4C's buffered write architecture. After you perform the 'flush' read, then the next read should (in theory) reflect the true status of the BUSY flag, like so:

```
// Set the control flags
i2c->MCS = 0x3;

// Read back the MCS to flush it.
volatile uint32_t readback = i2c->MCS;

// Wait for the BUSY bit to clear...
while (i2c->MCS & I2C_MCS_BUSY);
```

This solution didn't work for me either, and I still can't see why it should. First, the MCS register has a dual-purpose: you write one set of flags to it for control, and read a different set of flags from it for status. The bits in MCS appear more like independent sets of inputs and outputs to logic gates, rather than storage bits that might trigger a flush when read back after a write. But also, the MCS register is *read-senstive*, which means that when you read back the status bits, they clear immediately. So if you were reading MCS just to flush it, you might end up clearing valuable status flags and miss the result of the transaction. Finally, the original problem is that due to latency, reading back MCS may falsely return BUSY = 0 after writing to it. If it's read once to flush and then multiple times in a loop afterwards, isn't that the same thing as a loop with one extra iteration? The first iteration of a loop should therefore be sufficient to flush the buffered write and, if that works, subsequent iterations should properly detect when BUSY turns off. However, this is simply the original loop in the datasheet flowchart, which has already been reported to work unreliably. Pass.

A third solution is to insert a delay after writing to MCS, to give some time for the latency period to complete and the BUSY status to properly settle, e.g.:

```
// Set the control flags
i2c->MCS = 0x3;

// Delay to compensate for the buffered write latency.
int i = 0;
while (i < 100) { i++; }

// Wait for the BUSY bit to clear...
while (i2c->MCS & I2C_MCS_BUSY);
```

Depending on the length of the delay, this could mean that the BUSY status returns 1 when first read, indicating that the byte transfer is still in progress, or 0 indicating that the transfer is complete. Either way, the delay must exceed the latency time, or the same problems will occur.

The main issue I have with the 'delay' solution is that it means having to guess how long the latency period will last in order to choose a suitable delay time. Quite possibly, the latency may differ based on the system clock, i.e. the frequency of the microcontroller's execution. If one were to hardcode a spin loop that iterated N number of times, would that provide the same, sufficient delay at 16MHz versus 120MHz? Alternatively, an absolute delay in microseconds or nanoseconds could be inserted using the SYSTICK timer or a Timer block, but this likewise might have different results if the latency period differs according to the system clock. Either way, SYSTICK is best reserved for use by an RTOS (e.g. FreeRTOS), and I don't like having to use up a Timer block to work around an issue with I2C. Frankly, while delay workarounds like this may work under fixed conditions, it still sucks having to use them. ☹

What I really wanted was a reliable solution that didn't involve guesswork. Fortunately, I found that solution in the Master Raw Interrupt Status (MRIS) register. As mentioned in the TM4C1294NCPDT datasheet, the RIS flag (bit 0) of MRIS turns on whenever 'the master transaction is completed, or the next byte transfer is requested'. While this description is not present in the TM4C123GH6PM datasheet, which says only that the RIS flag is set when a master interrupt is pending, one can infer from the list of possible causes for master interrupts that the RIS flag will be set due to either of the above two conditions or a bus error.

Now, the 'R' in RIS means 'raw' interrupt status, so whether or not you have an actual interrupt service routine set up for I2C, the raw status bit will be set if a condition that causes an interrupt is pending. Therefore, a raw interrupt bit can be checked synchronously, for example, in a loop. So instead of checking for the BUSY flag of MCS to be cleared, that same loop can be replaced with one that checks for the RIS flag to set, as follows:

```
int error = 0;

// Clear the RIS bit (master interrupt)
i2c->MICR |= I2C_MICR_IC;

// Set the control flags.
i2c->MCS = 0x3;
```

```
// Wait until the RIS bit is set, which indicates the next byte to transfer is being
requested.
while (!(i2c->MRIS & I2C_MRIS_RIS));

// Check the error status.
uint32_t mcs = i2c->MCS;
error |= mcs & (I2C_MCS_ARBLST | I2C_MCS_DATACK | I2C_MCS_ADRACK | I2C_MCS_ERROR);
```

The first thing to note in the example above is that the RIS flag must be cleared before assigning the control flags to MCS. To clear RIS, you set the IC flag (bit 0) in the Master Interrupt Clear (MICR) register. After writing to MCS, you then loop until RIS sets. Afterwards, the status flags in MCS should be tested to see if any errors occurred. Checking for error status is the same step you take if you had been looping until BUSY = 0, and from this point onward the flow chart steps described in the datasheet will be the same.

Conclusion

I have been running this solution on both the TM4C123 and TM4C1294 boards for many cycles now, at 80MHz and 120MHz system speeds, and have had reliable success. An I2C transaction has never once failed, nor has the CPU become stuck in an infinite loop. Based on comments in the TM4C support forums, as well as examining the TivaWare I2C examples, it may be that an interrupt-based I2C solution is preferred for the TM4C1294 processor and/or for higher system clock speeds anyway. However, if you don't want to use an interrupt service routine for the I2C master, as I didn't need or want to do either, then testing the RIS flag instead of the BUSY flag should help you get the same, reliable result when following the command sequence flowcharts.