MSP430 microcontroller basics: Solutions to odd-numbered examples

John H. Davies

June 25, 2008

CHAPTER 1

Embedded electronic systems and microcontrollers

Example 1.1 I counted: remote controls (4), digital TV box, the TV itself, videocassette recorder, DVD player, cordless telephone and its base, which includes an answering machine. That's 10, which I suspect is a modest total – no MP3 players or games consoles! The base unit for the cordless telephone may contain two embedded systems, one for the telephone and another for the answering machine. Similarly, the television may have separate systems for controlling reception and teletext (a system for broadcasting information between the visible frames of a television signal).

Example 1.3 The data memory has a 9-bit address bus so there are $2^9 = 512 = 0x200$ possible locations, with addresses from 0x000–0x1FF. Similarly, the program memory has a 13-bit bus and can therefore address up to $2^{13} = 8\,\mathrm{K} = 8096 = 0x2000$ locations.

CHAPTER 2

The Texas Instruments MSP430

Example 2.1 Unfortunately you won't be able to answer this accurately until you have reached the end of the book because it requires knowledge of all the modules. Here's my suggestion, which comes to a total of 40 or 42 pins – a lot more than the 14 that are actually provided!

- Two pins for power, V_{CC} and V_{SS}. A further two pins could be added if the analog and digital supplies were separate.
- Two pins for the crystal, XIN and XOUT.
- Two pins for the clock outputs, ACLK and SMCLK.
- One pin for the reset or non-maskable interrupt, $\overline{\text{RST}}$/NMI.
- Ten pins for the input/output ports, P1.0–P1.7 and P2.6–P2.7.
- Five pins for Timer_A: the input clock TACLK and an input and output for each of channel 0 and channel 1.
- Eleven pins for the ADC: two for each of the five input channels, such as A0±, plus one for the reference VREF.
- Three pins for communications, because there is only one peripheral (USI) so it cannot drive SPI and I²C simultaneously. This means that SCLK, SDO/SCL and SDI/SDA are sufficient.

- Four pins for debugging because you could not use the four-wire and two-wire (Spy-Bi-Wire) interfaces simultaneously, nor would the TEST pin be needed if the pins had only one function. Therefore TCK, TMS, TDI/TCLK and TDI would be sufficient and two of these could be used for Spy-Bi-Wire.

Example 2.3 Reading the input register, such as P1IN, always shows the state of the pins. This can obviously be different from P1OUT for a pin that is configured as an input. They can also differ if an output pin is overloaded and is unable to provide the voltage desired.

If a single register is used, such as P1DATA, reading and writing do not work in the same way. Writing to P1DATA corresponds to writing to P1OUT and reading from P1DATA is like reading P1IN for a pin that is configured as an input. That is straightforward but there are two options for reading from P1DATA for a pin configured as an output. It may return the value on the pin itself (P1IN) or the value in the output buffer (P1OUT). These should be the same unless the pin is overloaded, in which case they may differ.

The shared register for input and output leads to the curious situation that the assignment `P1DATA = P1DATA` is not a null operation: It may change the value in the output buffer because it is equivalent to `P1OUT = P1IN`. There are clear advantages to having separate registers for input and output.

Example 2.5 I'll start with the three suggested conditions.

(a) The current I_{AM} in active mode at 1 MHz is given in a table but only for $V_{CC} = 2.2\,V$ and 3.0 V. These show that the current appears to be proportional to V_{CC} so we should use the lowest permissible voltage of 1.8 V, in which case the current is presumably close to 0.18 mA.

(b) Staying at the minimum supply voltage of 1.8 V, the maximum specified frequency is $f_{MCLK} = 6\,MHz$. Unfortunately there are no data for this frequency and voltage. The current is 1.0 mA at 2.2 V and will presumably be less at 1.8 V, so let's go for 0.9 mA. This is 5 times higher than the current at 1 MHz but gives six times the speed and is therefore more economical.

(c) The maximum speed of 16 MHz needs V_{CC} to be raised to 3.3 V. The current is about 4.2 mA and dividing by 16 gives 0.27 mA for comparison with 0.18 mA at 1 MHz. This is less economical.

This shows that the most economical condition seems to be the minimum supply voltage with the maximum clock frequency permitted at that voltage. Of course this will work only if there is time to complete the operations; otherwise we will have to raise V_{CC} to allow a faster MCLK.

Can this be made more precise? The plots show that $I_{AM} \propto f_{MCLK}$ fairly well and that $I_{AM} \propto V_{CC}$ rather less well – the current increases more than in direct proportion. Putting these together we can write $I_{AM} \approx CV_{CC} f_{MCLK}$. The constant has the dimensions of a capacitance, hence the choice of the symbol C, and I estimated $C \approx \frac{1}{12}$ nF. The charge that flows for each cycle of MCLK – each operation, in other words – is given by $Q = I_{AM}/f_{MCLK} \approx CV_{CC}$. This shows that the main strategy for saving energy is to run at as low a supply voltage as possible. The value of f_{MCLK} has no direct effect but determines the minimum value of V_{CC}.

The relation $I_{AM} \approx CV_{CC} f_{MCLK}$ is a standard one for CMOS digital systems because most of the current is used to charge the gates of the transistors and will be familiar to anybody who has taken a course on digital VLSI design. It does not account for other parts of the system, such as the DCO and memory, nor for leakage through the transistors. This is why it seems better to use 6 MHz than 1 MHz at $V_{CC} = 1.8$ V. A full calculation should also take account of the current while the MSP430 is in a low-power mode between these computational tasks. This is around 1 μA for low power mode 3, when only a low-frequency oscillator for ACLK is running.

CHAPTER 3

Development

There aren’t any examples in this chapter.

CHAPTER 4

A simple tour of the MSP430

Example 4.1 The voltage across the resistor is $V_R = V_{CC} - V_{LED} = 3.0 - 1.8 = 1.2\,V$. The current through the resistor is specified as $I_{LED} = 4\,mA$ so the resistance should be $R = V_R/I_{LED} = 1.2/0.004 = 300\,\Omega$. This should be rounded up to the nearest conventional value of $330\,\Omega$ so the current is slightly lower than specified.

On the board itself I measured 1.2 V across $R_6 = 330\,\Omega$ and 1.75 V across the LED, which gives $I = 3.6\,mA$.

Example 4.3 Again, I hope that appendix A will solve any problems. You may get a complaint that The stack plug-in failed to set a breakpoint on "main". The stack window will not be able to display stack contents. You can safely ignore it because we do not use the stack.

Example 4.5 The simplest modification so that the LED lights when the button is up rather than down is to change `if ((P2IN & B1) == 0)` to `if ((P2IN & B1) != 0)`. See `butledback.c`. Alternatively you could swap the lines to turn the LED on and off.

Example 4.7 This can be done with nested `if-else` statements. Alternatively, both buttons can be tested in the same statement with a logical AND && between them. See `butledand.c`.

Example 4.9 This also needs the structure with two `while` loops. See `count1.c`.

Example 4.11 See `dice1.s43`, `dice2.s43` and `dice3.s43` for different approaches to an unbiassed solution. It is vital to count cycles in the simulator because some instructions are automatically replaced by more efficient emulated instructions but others are not.

For example, `mov.w #1,R5` can be emulated using the constant generator but `mov.w #6,R5` cannot. The second instruction therefore takes longer to execute. I explain this in section 5.3.

Example 4.13 It is chastening to see your code optimized out of existence!

Example 4.15 See `flashled3.c`. This has only a single loop in the subroutine, which limits it to fairly short delays. I decided that 0.5 s was long enough to wait! Longer delays could be obtained by putting the loop for a 0.1 s delay inside another, whose number of iterations is given by the passed parameter. See `flashled31.c`.

Example 4.17 See `dicepat1.s43`.

Example 4.19 See `timrled4.c`.

Example 4.21 See the Excel file `simulate PRBS.xls`, which contains simulations of several lengths.

CHAPTER 5

Architecture of the MSP430 processor

Example 5.1 Two instructions are required for an increment because the value of the carry bit must be fixed before using `dadd`. Do not forget that this is really decimal add *with carry*! You can either clear the carry bit (`clrc`) and add 1 decimally (`dadd #1,dst`), or set the carry bit (`setc`) and add 0 (`dadd #0,dst` or `dadc dst`).

Decrementing is more tricky because there is no decimal subtraction instruction, nor signed BCD values. We can decrement a single digit by adding 9 and discarding the carry. For example, $7+9=16\rightarrow 6$. Similarly, adding 99 works for a two-digit number: $27+99=126\rightarrow 26$. This can be applied directly to a byte, which holds two BCD digits, giving `dadd.b #99,dst`. As usual you must clear the carry flag first. The carry flag will be set after this operation because it must overflow. Similarly, `dadd.w #9999,dst` decrements a word.

Example 5.3 There is no way of doing this with a single instruction. The definition of the twos complement is that the value is inverted and incremented, so it can be implemented using the two instructions `inv` followed by `inc`.

Another obvious route is to subtract the given value from zero. A snag is that the subtraction instruction performs `dst -= src`. You must first clear `dst`, then subtract `src` from it. This is useful if `src` should not be modified.

Example 5.5 No solution required.

Example 5.7 See `shiftreg1.c` for a straightforward program; `shiftreg2.c` uses a `union` as suggested.

CHAPTER 6

Functions, interrupts and low-power modes

Example 6.1 No solution required.

Example 6.3 The value on top of the stack will be used as the return address. Unfortunately this is the little loop counter instead!

Example 6.5 No solution required.

Example 6.7 No solution required. Another way of triggering interrupts is to select Simulator > Interrupt Setup... from the menu and click the New... button. This generates periodic interrupts, which is a little more faithful for a timer. You can even add a stochastic element with the Variance and Probability fields.

Example 6.9 No solution required. I had trouble simulating this in C because the forced interrupts seemed not to work, although there was no difficulty in assembly language.

Example 6.11 No solution required. I had trouble with the simulator again and was unable to trigger any interrupts. There were no problems with the emulator.

Example 6.13 See `morsetim1.c`.

CHAPTER 7

Digital input, output and displays

Example 7.1 The configuration of the ports is given in table 7.1 on the facing page. We are also told that ACLK should be derived from VLO. This requires XTS = 0 (default) in BCSCTL1 and LFXT1Sx = 10 in BCSCTL3 (not the default, which is 00 for a 32 KHz crystal on LFXT1).

Example 7.3 We could leave the pins as inputs and activate the pull resistors with

```
P1REN |= BIT6|BIT7; // activate internal pulls on P1.6 and P1.7
```

The resistors will be pullups or pulldowns according to the values in P1OUT but it doesn't matter which. This solution works on the MSP430F2xx family and some other recent devices. Earlier devices don't have pull resistors and the pins should be switched to output instead. Again the value in P1OUT doesn't matter, assuming that nothing is connected to the pins.

```
P1DIR |= BIT6|BIT7; // configure P1.6 and P1.7 as outputs
```

Example 7.5 See `quiz1.c` and `quiz2.c`.

Example 7.7 See `keypad3.c`. Again this has been tested only with the simulator, not hardware.

Example 7.9 See `cntint1.s43`.

Table 7.1: Configuration of ports P1 and P2 for a F2013.

pin	function	PnDIR	PnSEL	PnOUT	PnREN	SD16AE
P1.0	input A0+ to SD16_A	x	x	x	0	1
P1.1	input A0− to SD16_A	x	x	x	0	1
P1.2	input CCI1A to Timer_A	0	1	x	0	0
P1.3	VREF of SD16_A	x	1	x	0	0
P1.4	digital output, initially low	1	0	0	0	0
P1.5	output TA0 from Timer_A	1	1	x	0	0
P1.6	not used	0	0	x	0	0
P1.7	not used	0	0	x	0	0
P2.6	digital input with external pullup	0	0	1	0	—
P2.7	digital input with external pullup	0	0	1	0	—

Example 7.11 See figure 7.1 on the next page. This shows a similar input to figure 7.8. The essential difference is that the output voltage falls immediately to zero when the contacts of the switch close.

To eliminate the effects of bounce, we must ensure that the voltage never rises above the upward threshold V_{IT+} until the contacts finally close. The critical time T is therefore the duration for which the contact is closed during a bounce. The time-constant of the circuit can then be found as before. For example, take $T = 5\,\text{ms}$ again. Then we need $V_{CC}[1 - \exp(-T/\tau)] < V_{IT+}$. For safety we should use the *minimum* value of V_{IT+}, which is 1.35 V for $V_{CC} = 3\,\text{V}$. This leads to $\tau > 8\,\text{ms}$ and $R = 84\,\text{k}\Omega$ with $C = 0.1\,\mu\text{F}$. In practice we would round this up to $100\,\text{k}\Omega$ for safety.

Example 7.13 I couldn't come up with anything significantly simpler than listing 7.2. At first I thought that it would be sufficient to signal a transition when the shift register contained 00000111b, assuming that a run of 5 zeroes is the criterion for a valid press. The problem is that the remaining 3 ones might be a short pulse of noise, rather than the end of a long train of ones.

Example 7.15 This needs one common-cathode and one common-anode display. The common anode is connected to V_{CC} and the common cathode to V_{SS}. The corresponding segment pins are connected together and to a pin of the MSP430 through the usual resistor to limit the current. An LED in the common-cathode display is lit by driving the pin of the MSP430

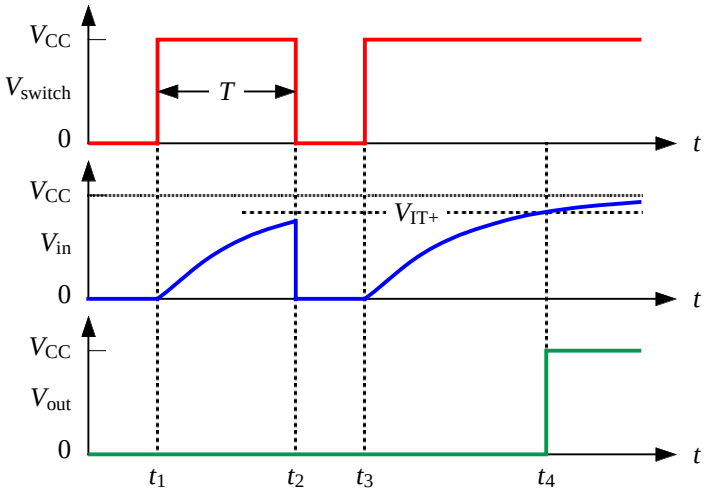

Figure 7.1: Voltages across the switch, input to the Schmitt trigger and output of the Schmitt trigger for a bounce using the circuit without R_2.

high. Driving the pin low illuminates an LED in the common-anode display instead. Neither LED lights if the pin is an input; the LEDs are connected in forward-biassed pairs from V_{CC} to V_{SS} but the voltage across each is too low for a significant current to flow, as we saw earlier. Effectively we are using the pin as a three-state output.

I'm not sure that this is good engineering because it depends critically on the current–voltage relation of the LEDs – test it carefully! Another defect is that the pins are effectively floating when they are configured as inputs, which brings the problems described in section 7.1.2.

Example 7.17 It is the inverse (complement) of the waveform for S0: low, high, high, low in each frame.

Example 7.19 See `testlcd3.c`.

Example 7.21 See `sclock4.c`.

Example 7.23 See `sclock3.c`.

Example 7.25 See `testcp1.c` and `testcp2.c`. I found that the voltage needed to be greater than 3.2 V for a clear, sharp display, viewed normally. The optimum viewing angle moved away from normal as the voltage was reduced. My batteries gave 3.1 V so the display was slightly indistinct without the charge pump.

CHAPTER 8

Timers

Example 8.1 Some of the mistakes are shown in `wdtest0.c`.

Example 8.3 See `sclock6.c`.

Example 8.5 See `rtclock4.c`.

Example 8.7 It may be easier to single-step in the disassembly window rather than through the C code. The problem is again that TAIV can be read only once.

Example 8.9 See `press3.c`.

Example 8.11 See `press4.c`. This suffers from some of the problems described in the paragraph of the main text before the example.

Example 8.13 See `react4.c`. This time I have put the actions in the main loop as in `react2.c`. Again it is a bit clumsy because the main loop must distinguish the source of the interrupt. This would be easy if the flags were still available but they are either cleared automatically when the interrupt is serviced or must be cleared by software to prevent the interrupt recurring immediately. I have therefore checked the state of the appropriate button instead. The start button S2 can be checked using P1IN.1 as usual but I have tested the reaction-time button S1 by using the CCI bit of TACCR0, which shows the current state of the input.

The program could have been based on interrupt service routines instead. The ISR for TAIFG would be even more complicated than before because the interrupt could arise while waiting for the start button, the random delay or the reaction-time button.

Example 8.15 I found that the frequency went down by about 20 KHz from 2 MHz, roughly -1%. My finger wasn't very hot so the change in temperature was probably about 5°C. Thus the temperature coefficient of the oscillator is about $-0.2\%/°\text{C}$, which is consistent with the data sheet.

You could test the sensitivity to V_{CC} by powering the board from the FET and varying the supply voltage in the debugger. I didn't try this.

Example 8.17 See `clkcmp3.c`. You must keep SMCLK running or the timer won't be able to perform the measurements! I used LPM0 but it might be possible to use LPM1. To be honest, I have trouble working out what happens in LPM1 and LPM2.

Example 8.19 See `contall2.c`.

Example 8.21 I found that the apparent musical note altered although the oscilloscope showed that the average frequency remained at 440 Hz. This isn't surprising if you look at the power spectra. The plot for a 1 KHz clock looks more like that for a square wave of around 500 Hz although the highest peak is correctly at 440 Hz. The ear does some complicated signal processing!

Example 8.23 The obvious change is that the PWM should be changed from negative to positive, which means that output mode 7 (Reset/Set) should be selected instead of mode 3.

The less obvious change is that the LEDs should be turned *on* before PWM is started so that the first cycle is correct. This needs `TACCTL1 = OUTMOD_0 | OUT` and similarly for channel 2.

Example 8.25 This is explained in the main text.

Example 8.27 See `ezpwm2.c`, where I have added a test so that the LED is not turned on if $D = 0$.

I do not think that there are any problems for $D = 1$: The LED remains on continuously as it should because the compare event for CCIFG1 does not occur at all.

Example 8.29 See `butpwm1.c`. This polls the buttons in every cycle of PWM using the CCIFG0 interrupt and adjusts the duty cycle if either is pressed. There is a more sophisticated program in `butpwm2.c`, which turns the CCIFG0 interrupts off if neither button is down and no change is needed. These interrupts are restarted by interrupts from the ports when a button is pressed.

Example 8.31 See `butstmc2.c`.

CHAPTER 9

Mixed-signal systems: Analog input and output

Example 9.1 I saw a small error with a factor of 8 and a large error with a factor of 4. It is hard to check both the discharge and charge measurements simultaneously because a breakpoint allows lots of time for the capacitor to charge or discharge.

Example 9.3 The problem is that the algorithm always rounds *down* to the digital value below the input. For example, suppose that the input is raised slightly from $0.40V_{\rm FS}$ to $0.43V_{\rm FS}$. This is $0.43\times 16 = 6.9$LSB, which should be rounded to the nearest integer of 7. However, the final comparison in the illustration is with $\frac{7}{16}V_{\rm FS} = 0.4375V_{\rm FS}$ so $V_{\rm in} < \frac{7}{16}V_{\rm FS}$ and the bit is 0.

Example 9.5 See `ad10led3.c`. There is no crystal so I used the VLO for ACLK. I put everything in the ISR and used a polling loop again instead of interrupts. I have assumed that nothing else uses the ADC10, which is a bit lazy. The ADC10 is turned off between conversions, which may save a little current.

Example 9.7 This threshold gives $V_{\rm mid} = 1.35$ V so the 1.5 V reference would be adequate. On the other hand, it is tempting to use 2.5 V so that the whole operating range up to 3.6 V can be measured. The snag is that the 2.5 V reference needs a little 'headroom' in the supply voltage $V_{\rm CC}$ and loses accuracy when this falls below 2.8 V. Of course there is no hope of it working when $V_{\rm CC} < 2.5$ V! Therefore it is better to use the 1.5 V reference after all. The corresponding value in ADC10MEM is given by $N_{\rm mid} = {\rm nint}(1024V_{\rm mid}/V_{\rm ref}) = (1024\times$

$1350/1500) = (1024 \times 2700/3000) = 922$ for 1.35 V. See `ad10bat1.c`. The compiler didn't like the expression for the threshold voltage so I had to scale it to $1024 \times 27/30$. This can easily be changed with a resolution of 0.1 V.

Example 9.9 There are several possibilities to remove the dependence on V_{CC}.

- If the circuit cannot be changed, you could take a reading of V_{CC} using channel 5 and divide the external voltage by this.
- Assuming that you can modify the circuit, take V_{ref} off a potential divider from V_{CC}.
- Alternatively, continue to use the internal reference but drive it off-chip and use it to supply the potential divider.

I discuss this issue in section 9.11.

Example 9.11 See `sd16bat2.c`.

Example 9.13 Here are the usual three steps.

1. Work out the voltage on the non-inverting input, V_+. This is set by the potential divider so $V_+ = V_{CC} \times R_3/(R_3 + R_4) = (5/11)V_{CC}$. We assume that no current flows into the non-inverting input because an ideal op-amp has infinite input resistance.
2. An ideal op-amp has infinite gain and does whatever is necessary with negative feedback to bring its inputs to the same potential, so $V_- = V_+ = (5/11)V_{CC}$.
3. We now know V_- and can apply nodal analysis at the inverting input, again assuming that no current flows into the op-amp. We also assume that the op-amp has zero output resistance, which means that the load has no effect on the output voltage V_{out}. Here the load is just the feedback circuit.

The last step is, as usual, the only one to require any work. Nodal analysis gives

$$\frac{V_{in} - V_-}{R_1} + \frac{V_{out} - V_-}{R_2} + 0 = 0. \tag{9.1}$$

I've added a 0 as a reminder that no current flows into the op-amp. Rearranging this gives

$$\begin{aligned} V_{\text{out}} &= -\frac{R_2}{R_1}\left(V_{\text{in}} - \frac{R_1+R_2}{R_2}V_-\right) \\ &= -10(V_{\text{in}} - \tfrac{11}{10}V_-) \\ &= -10(V_{\text{in}} - \tfrac{1}{2}V_{\text{CC}}). \end{aligned} \tag{9.2}$$

For correct operation, the output predicted by this equation must lie between ground and V_{CC}, assuming that the op-amp has a true rail-to-rail output stage. This determines the range of inputs. Clearly $V_{\text{in}} = \frac{1}{2}V_{\text{CC}} = 1.5\,\text{V}$ gives $V_{\text{out}} = 0$. The other limit is $V_{\text{out}} = V_{\text{CC}}$, which needs $V_{\text{in}} = 0.4V_{\text{CC}} = 1.2\,\text{V}$. Thus the input must lie between 1.2 and 1.5 V for correct operation. There would be no point in specifying rail-to-rail inputs.

CHAPTER 10

Communication

Example 10.1 See `usispiloop5.c` for mode 0 . Thisis easier than mode 3 because there is a falling edge on SCLK to update the output after the last rising edge, at which point we can read the input from the shift register. I wrote 0xFFFF to the shift register (after reading the received value!) to get a 1 in the bit that feeds the output latch. This will be updated correctly on the final falling edge of SCLK. This will work cleanly only if there is time to perform these actions in a half-cycle of SCLK, so MCLK must be much faster than SCLK.

It is more complicated to get this behavior in mode 3 because there are no more clock edges to come. Instead you need to write 0xFFFF to the shift register (after reading the received value of course), then wait for about $\frac{1}{2}$ cycle of SCLK. You then set the USIGE bit, which makes the output latch transparent and transfers the 1 from the shift register onto SDO. USIGE should then be cleared again to 'lock' the latch, or the next transaction won't work correctly. See `usispiloop6.c`. I have also adjusted the $\overline{\text{SS}}$ output to go high at the end of all this. The delay is estimated by subtracting the time required for the other instructions and is only approximate. If the divider for SCLK from (S)MCLK were smaller, the other instructions alone would give sufficient delay and the loop would not be needed. Even 64 hardly needs a loop and 32 would clearly be fine.

The manipulation of USIGE is similar to the procedure for generating start and stop conditions on an I²C bus and is described in section 10.11.2.

Example 10.3 See `usi2cmastsm3.c`, whose structure is similar to listing 10.9. There are a couple of awkward points (as always with the USI).

- The main function sets the USISTTIFG flag for a start condition to request and interrupt and start the state machine for a new transaction. It is easy to forget that USISTTIFG will be raised again when the actual start condition is put on the bus. This must not be allowed to generate a further interrupt or the program will become trapped in an infinite loop. I have therefore cleared USISTTIFG as late as possible, after both the fictitious start condition from the main function and the real start condition on the bus.
- I planned to request an interrupt by setting USISTTIFG from software. However, this didn't work: I had to set USIIFG as well or nothing happened. I don't know why. There's nothing in the errata, although the USI3 bug looks vaguely related. This is probably not how USISTTIFG is intended to be used but most interrupt flags in the MSP430 work equally well when set by hardware or software.

See also `usi2cmastsm4.c` for a combined transaction, based on `usi2cmastsm2.c`. A further start condition is produced in a combined transaction and USISTTIFG must be cleared to avoid an undesired interrupt.

Example 10.5 See `usi2cmastsm2.c`. There is an oscilloscope trace of the transaction in `usi2comb.pdf`.

Example 10.7 Start with the division $f_{\mathrm{BRCLK}}/f_{\mathrm{baud}} = 1\,000\,000/9600 = 104.2$, which rounds to 104. Then $104 = 16 \times 6 + 8$ so UCBRx = 6 and UCBRFx = 8. This agrees with the user's guide.

Example 10.9 These are the events that cause the crowd of five interrupts.

1. Sampling of first stop bit by receiver.
2. Setup of start bit by transmitter.
3. Capture of second start bit by receiver.
4. Sampling of second start bit by receiver.
5. Setup of first data bit by transmitter.

The last two are repeated while one byte is being received and another transmitted.