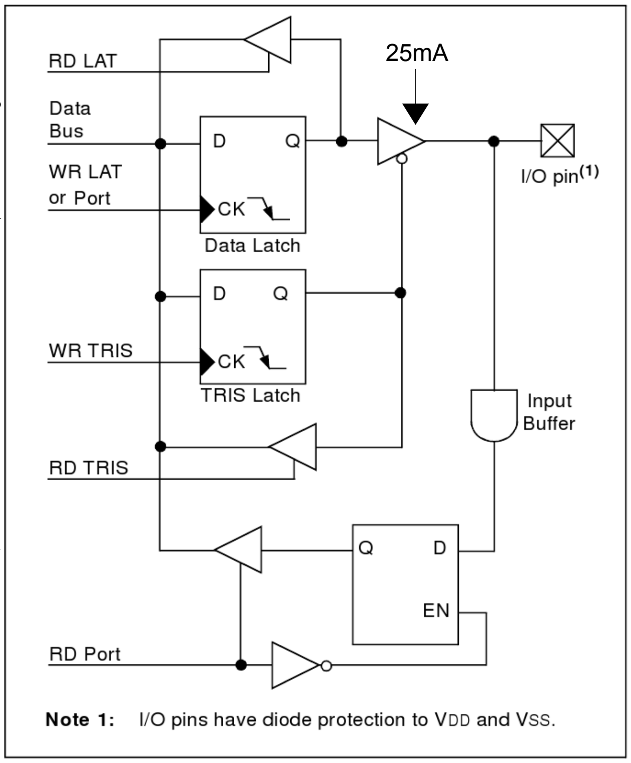
The Read-Modify-Write Problem
(And How to Avoid It)

Hector Martín, 11-01-2007

Almost every PIC programmer has had to deal with the Read-Modify-Write (RMW) problem at one point or another. Nowadays, it is (or should be) documented in most PIC tutorials and books, but the problem keeps coming up. The RMW problem could almost be considered a design flaw in the mid-range PICs (PIC16Fxxx) and programmers had to work around it using software techniques, but nowadays the PIC18Fxxxx series has built-in support to avoid it. Here's how you can use the new features of PIC ports to never have to deal with the issue.

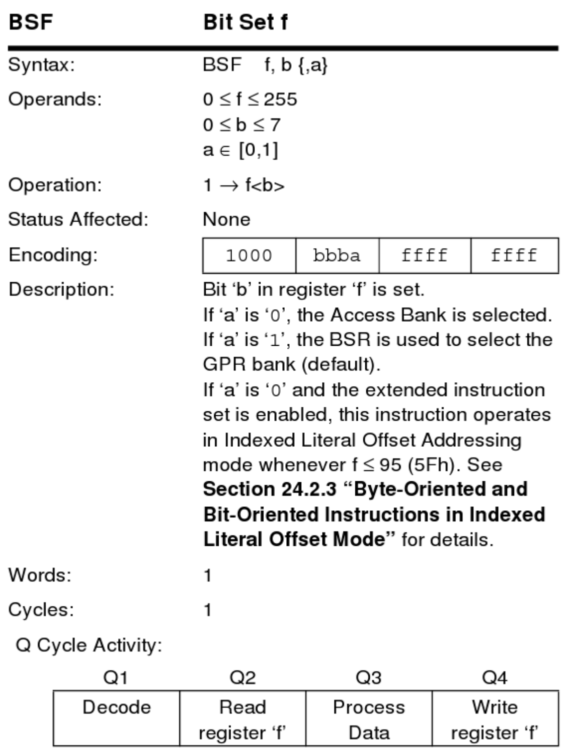
To understand the RMW problem, first we have to realize that PIC Special Function Registers (SFRs) are not necessarily registers, in the memory storage sense. Indeed, they are accessed like memory, but there is no guarantee that there actually is some memory (or even a set of flip-flops) behind them. In general, a register is simply a location in the data address space that accepts data written to it and returns data when read. These two sets of data may or may not have anything to do with each other, and in some cases even the mere action of reading a register will cause changes to it (this is the case with registers like RCREG, which is actually a window into an internal data queue: reading it advances to the next element in the queue).



The PORT registers are one of those registers. In fact, it should be obvious: when we read a PORT register, for the pins set to inputs, we will read the value that is being read by that pin. To the right is the standard schematic for a PIC port pin. Notice that the value read when we read the port pin (RD Port) comes from a latch fed by the port pin input buffer directly. Always. Even when then pin is set as an output. That means that, no matter what, when we read the PORT register, we will be reading the value present on the physical pin on the PIC, just as if we were to hook a multimeter to it and turn the resulting reading into a binary '1' or a '0'.

This may not seem relevant: why would an output pin ever be at a voltage that contradicts the value that was written to it? The answer lies in the output buffer, marked with an arrow. The PIC pins have limits, and, as specified in the datasheet, the maximum current that a PIC pin can source or sink is 25mA. That's not a lot of current. It wouldn't take a lot of drive on the outside to force a pin to the opposite state, and the PIC's output buffer would not be able to keep the pin at its intended state. In the long term, this can destroy the buffer. In the short term however, this is expected and it will happen when there are capacitive loads on the pin. For example, if a capacitor is attached between a pin and ground, it will take a short while to charge when the pin is turned on. While it is discharged, a capacitor acts like a short circuit, forcing the pin to a '0' state (even though we wrote a '1' to it) and, therefore, a read of the PORT register will return a '0' even though the Data Latch is at '1'.

Now, we clearly can't trust the data read from the PORT register when a pin is set as an output. However, up to 8 pins are accessed using the very same PORT register on a PIC. The PIC's data path is 8 bits wide, and therefore that is the smallest unit that can be read or written by the PIC. To read a pin, we must read all pins. To write to a pin, we must write to all pins. How do instructions like BSF and BCF work, when they are only supposed to change one bit? Look at the Q Cycle Activity in the instruction description to the left. On Q2 the register is read, on Q3 the data is modified (this happens inside the PIC's temporary data registers), and on Q4 the **entire** register is written with the new data. We just mentioned how register reads may or may not have something to do with register writes, and we've showed how this is the case with the PORT registers, but BSF assumes that the data read from a register is indeed the proper data that should be written to it for bits that are not to change.



BSF	**Bit Set f**
Syntax:	BSF f, b {,a}
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	1 → f<b>
Status Affected:	None
Encoding:	`1000` | `bbba` | `ffff` | `ffff`
Description:	Bit 'b' in register 'f' is set. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Consider the following program fragment:

```
1    BSF  PORTB, 0
2    BSF  PORTB, 1
```

Assume that PORTB is initially all zero (00h), and that all pins are set to output. Let's say we hook up a capacitor to pin RB0 (and that capacitor is initially discharged). What happens?

- 1 Q1: BSF instruction is decoded
- 1 Q2: PORTB is read. Result: 00000000b
- 1 Q3: The data is modified to set the bit. Result: 00000001b (this is stored inside a temporary internal register in the PIC)
- 1 Q4: PORTB is written with the new data, 00000001b. The output driver for RB0 turns on, and the capacitor starts to charge at 25mA.
- 2 Q1: BSF instruction is decoded
- 2 Q2: PORTB is read. Since the capacitor is still charging, the voltage at RB0 is still low and reads as a '0' (since we're reading from the pins directly, not from the data register). Result: 00000000b
- 2 Q3: The data is modified to set the bit. Result: 00000010b
- 2 Q4: PORTB is written with the new data, 00000010b. The output driver for RB1 turns on, but the driver for RB0 turns back off!

Therefore, the second BSF unintentionally undoes the actions of the first if the outside conditions are right. How do we avoid this? Easy. Use the LAT registers when writing to ports. Writing to a LAT register is equivalent to writing to a PORT register, but reads from LAT registers return the Data Latch data, regardless of the state of the actual pin. In other words, Read-Modify-Write instructions like BSF will operate correctly. The rule of thumb: never use PORT registers when writing to pins. Use LAT for writing, and PORT for reading:

```
1    BSF  LATB, 0
2    BSF  LATB, 1
```