Some Gnarly Bits

Posted on **April 24, 2012**

Given recent posts on Abstraction and its application in making devices (like the M25P80) easier to deal with, the reader might be under the impression that the workings of the system underneath the covers are not a real concern; A sort of “Out of sight, out of mind” attitude. Well the reality is much more like “What you don’t know *can* hurt you!”

A very common example of this sort of thing in embedded systems is the attachment of various devices to General Purpose Input/Output (GPIO) pins. In the MMB32 development system for example, the LCD CS pin is connected to Port F, bit number 12. Conveniently, the PIC32 “C” Peripheral Libraries define an easy-to-use variable for this called _LATF12. It is very easy to use a macro to give this bit a more meaningful name:

```
#define LCD_CS_LAT_BIT      _LATF12
```

Then in my code, to deselect the LCD I simply write:

```
LCD_CS_LAT_BIT = 1;
```

And that would appear to be the end of that! Of course, it is not; the above code is a “time-bomb” that could go off with serious consequences. A very nice feature of MPLAB-X is a dis-assembly view of the generated code with source code interspersed at appropriate points. Below is the above code and its implementation in MIPS assembly language:

```
420:                    LCD_CS_LAT_BIT = 1;
9D008958 3C03BF88   LUI V1, -16504
9D00895C 8C626160   LW V0, 24928(V1)
9D008960 24040001   ADDIU A0, ZERO, 1
9D008964 7C826304   INS V0, A0, 12, 1
9D008968 AC626160   SW V0, 24928(V1)
```

It's not necessary to be an expert in MIPS assembler, the key points here are that the value of Port F is read using an LW instruction, modified in the next two instructions, and the modified value is written back to Port F using an SW instruction. How can this code be harmful? The fault is that it is not an "Atomic Operation"!

Consider that Port F, having many bits, might be connected to more than one device besides the LCD. Further consider that this other device might be controlled by code in another task or perhaps an interrupt service routine. So now, contemplate the following scenario with an LCD and a Grunge Master 6000 (The 6000 is less costly than the 9000 but has more features than the 3000) both on Port F:

1. The LW instruction reads the current value of Port F.
2. The code modifies this value, setting bit number 12.
3. An interrupt occurs. While the interrupt is being processed, some other part of Port F is modified to send a command to the Grunge Master 6000.
4. The interrupt completes and normal execution resumes.
5. The SW instruction writes it's modified value to Port F without the changes made during the interrupt! A BUG!

This nasty little bug would take the form of the Grunge Master 6000 command being "randomly" truncated. The odds of catching it with a break-point are extremely low. This is the sort of bug that has people saying things like "HELP! My Grunge Master 6000 code works perfectly until I add my LCD driver code."

So how can this problem be circumvented? The answer is that the operations on Port F need to be indivisible or atomic. There cannot be a middle part where an interrupt could cause trouble. Some common answers include:

- Disable all interrupts while modifying the Port bits. This way the sensitive area of code is protected. The downside is that means turning interrupts off a lot and adding complexity to the code and increases interrupt latency. Further, it is easy to miss a spot and leave a vulnerability in the code.
- The CPU could have built in, atomic, instructions for setting, clearing and toggling bits. This is actually the case of 8 and 16 bit PICs. By performing the bit manipulation in a single, uninterruptable instruction, there is not a middle section that needs to be protected. Alas, the 32 bit PICs do not have these instructions.
- Microchip gave the 32 the PICs something else. Most peripheral registers have 3 "shadow" registers called the CLR, SET and INV registers. Writes to these registers perform Boolean operations on the register that they shadow. These are shown below:

```
CLR:  Register = Register & ~(Value)
SET:  Register = Register | Value
INV:  Register = Register ^ Value
```

These Boolean operations are atomic making them safe for use on shared resources without disabling interrupts.

To the left is a snippet from the PIC32 datasheet regarding these shadow registers. Note that reads from the shadow registers yield undefined values. Further note that the atomic nature of the operations applies to all three operations, not just INV or toggle as seems to be implied in the datasheet.

So, how does using these registers change the code? For starters, we must stop using the pre-packaged bit definition and switch to a new model. This means new definitions:

```
// Definitions for the LCD Chip Selct pin.
#define LCD_CS_BIT_MASK   (1 << 12)
#define LCD_CS_TRIS_SET   TRISFSET
#define LCD_CS_TRIS_CLR   TRISFCLR
#define LCD_CS_LAT_SET    LATFSET
#define LCD_CS_LAT_CLR    LATFCLR
```

And the code changes as well:

```
419:                    // Disable LCD
420:                    LCD_CS_LAT_SET = LCD_CS_BIT_MASK;
9D008958 3C02BF88    LUI V0, -16504
9D00895C 24031000    ADDIU V1, ZERO, 4096
9D008960 AC436168    SW V1, 24936(V0)
```

The code is now safe. The update to Port F is accomplished atomically by the SW instruction and the sequence is eight bytes shorter than before as a bonus. The real benefit here though is that Port F is now safe to use for other devices without fear of mind busting interaction bugs ruining your day!

Sometimes, it really does pay to check carefully under the covers. It’s what good engineers do!

As always, comments and suggestions are welcomed and invited.

Peter Camilleri (aka Squidly Jones)

PS: Astute readers will have recognized the Grunge Master 6000 as being somewhat similar to the Grunt Master 6000 introduced in Scott Adams' Dilbert universe. While similar in a lot of respects, only the Grunge Master is suitable for microprocessor interfacing.

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=880]** .