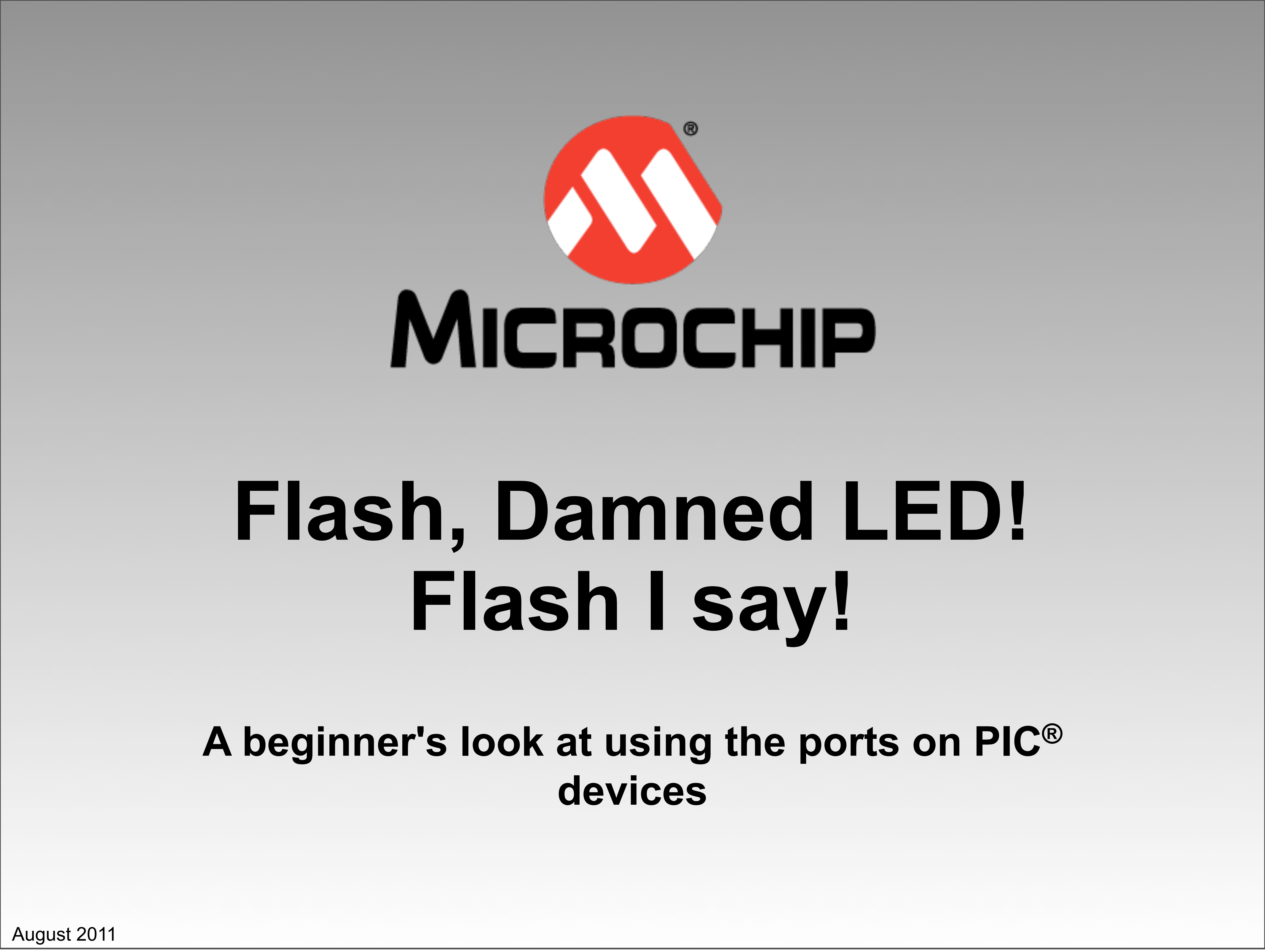


MICROCHIP

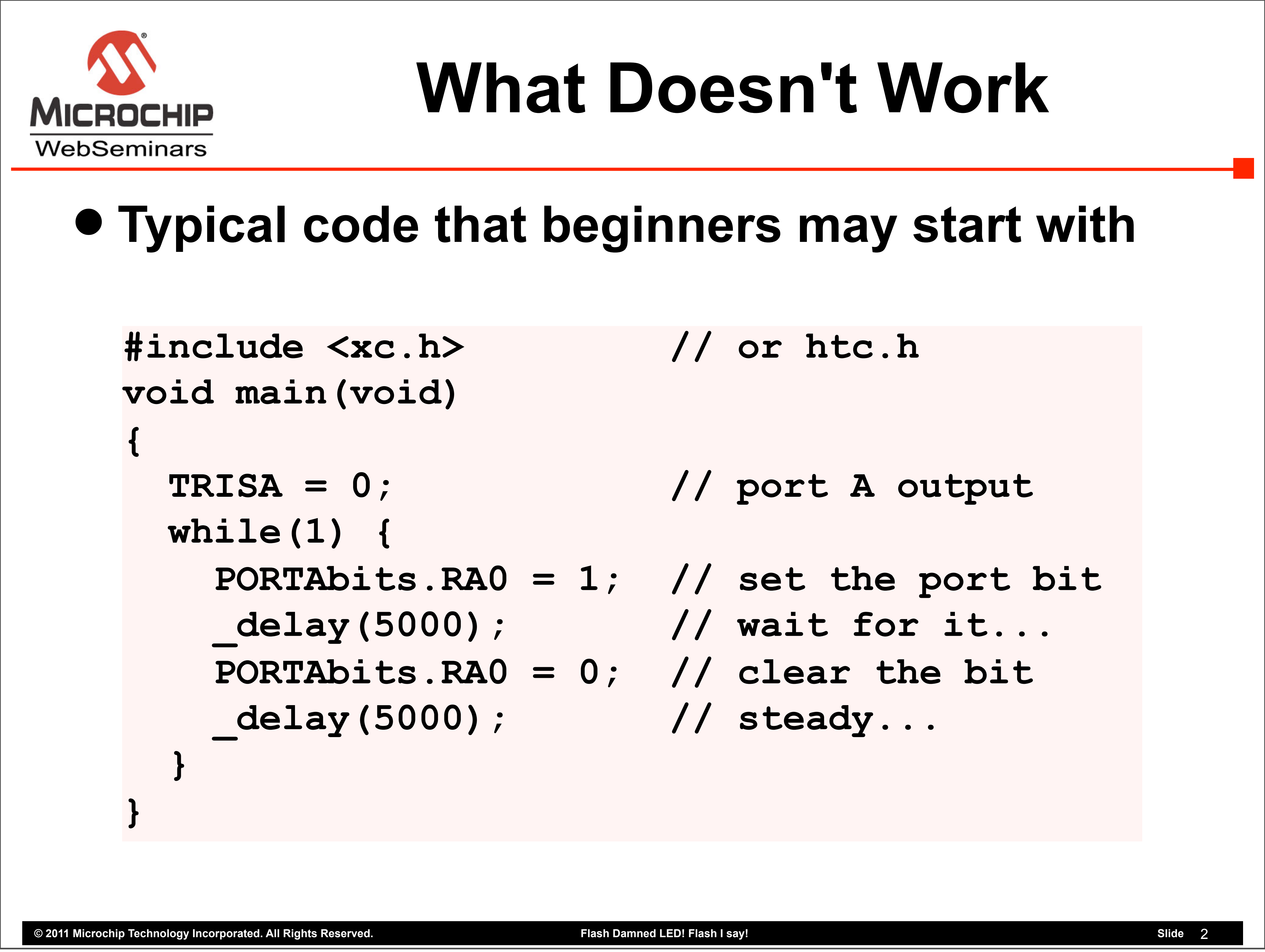
Flash, Damned LED! Flash I say!

A beginner's look at using the ports on PIC® devices

August 2011

This webinar is for those just starting out with PIC MCUs. Maybe the following situation applies to you: You have a development board, you have correctly connected an L-E-D to one pin of a port, you've written some C code that tries to toggle the L-E-D, but nothing happens. Or maybe you are in a situation where you have code that runs okay in the simulator, but when programmed into the actual device, that darned L-E-D still refused to light.

In this webinar, we will look at a number of common mistakes made by programmers in code to flash an L-E-D which is often use as a 'hello world' program. The code examples we will show are applicable to most 8-bit devices, but the principles are valid for all Microchip PIC MCUs.



What Doesn't Work

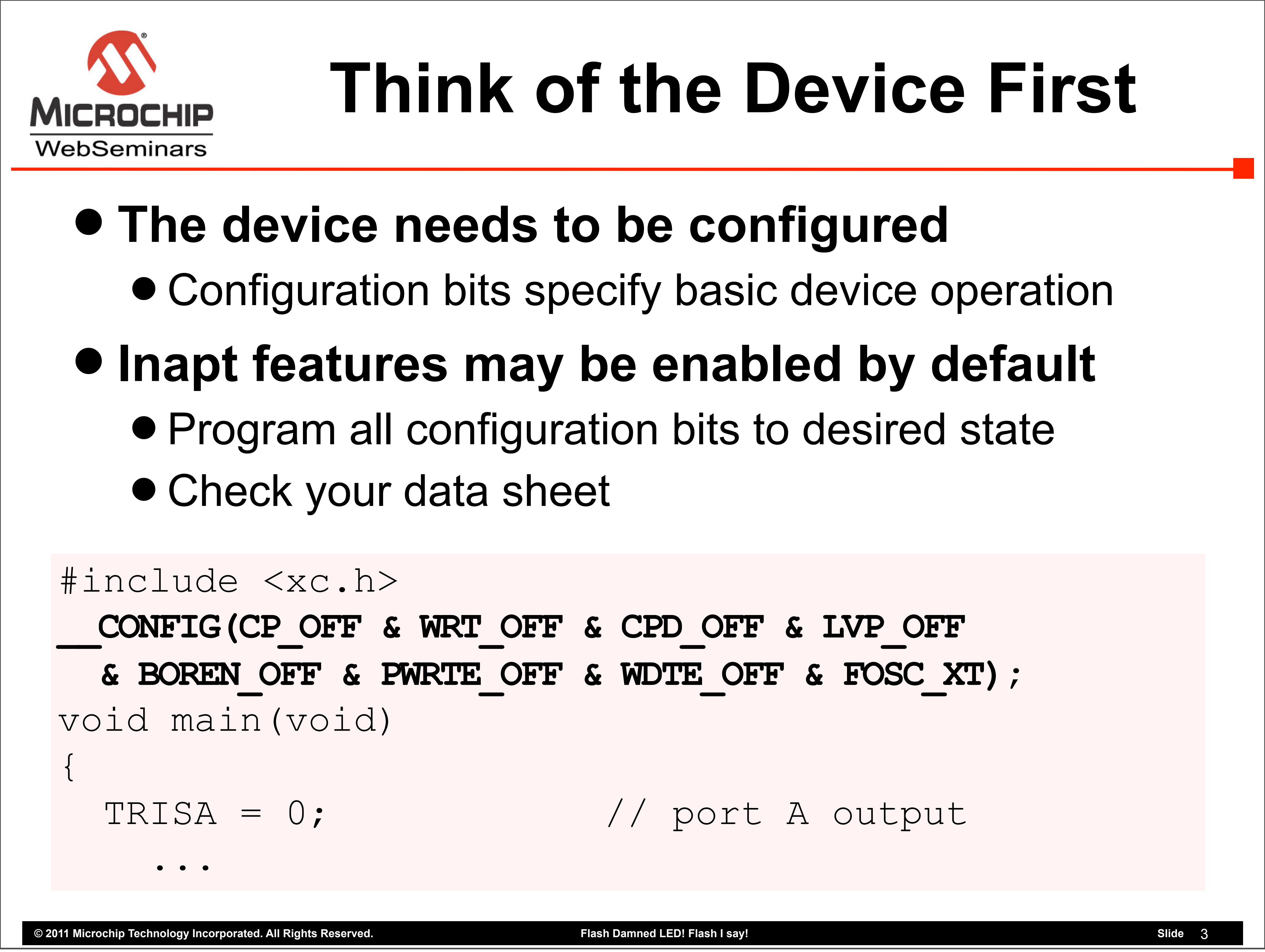
- Typical code that beginners may start with

```
#include <xc.h>             // or htc.h
void main(void)
{
  TRISA = 0;                // port A output
  while(1) {
    PORTAbits.RA0 = 1;      // set the port bit
    _delay(5000);           // wait for it...
    PORTAbits.RA0 = 0;      // clear the bit
    _delay(5000);           // steady...
  }
}
```

© 2011 Microchip Technology Incorporated. All Rights Reserved.

Here is some code that looks like it might toggle an L-E-D attached to bit 0 of port A. We are using the RA0 member of the PORTAbits structure that is mapped over the port register. The header file included, xc.h, ensures that this special variable is defined. If you are not using the XC8 compiler, you will need to use htc.h as the generic header file name. This code correctly sets up the port direction register, TRISA, so that the port pins are in an output state. In-line delays have been added so that we'll be able to see the L-E-D flash, and the code which performs the toggle has been placed in an endless loop so that our main function will never end and the program will continue to run.

Look's good, doesn't it? And yet, this code is unlikely to work on any PIC device. Let's focus on a number points that have been overlooked in this code.



Think of the Device First

- **The device needs to be configured**
  - Configuration bits specify basic device operation
- **Inapt features may be enabled by default**
  - Program all configuration bits to desired state
  - Check your data sheet

```
#include <xc.h>
__CONFIG(CP_OFF & WRT_OFF & CPD_OFF & LVP_OFF
  & BOREN_OFF & PWRTE_OFF & WDTE_OFF & FOSC_XT);
void main(void)
{
  TRISA = 0;              // port A output
     ...
```

It doesn't matter what code you have, if the device is not configured correctly, then nothing is going to run.

PIC devices use what are called 'configuration bits' to set up the basic operating mode of the device. A lot of these bits are not relevant to our code to flash an L-E-D, but some are critical.

Each device has it own settings which are controlled by these bits, so check your device data sheet for more information, but here are some of the more important settings that you might need to investigate.

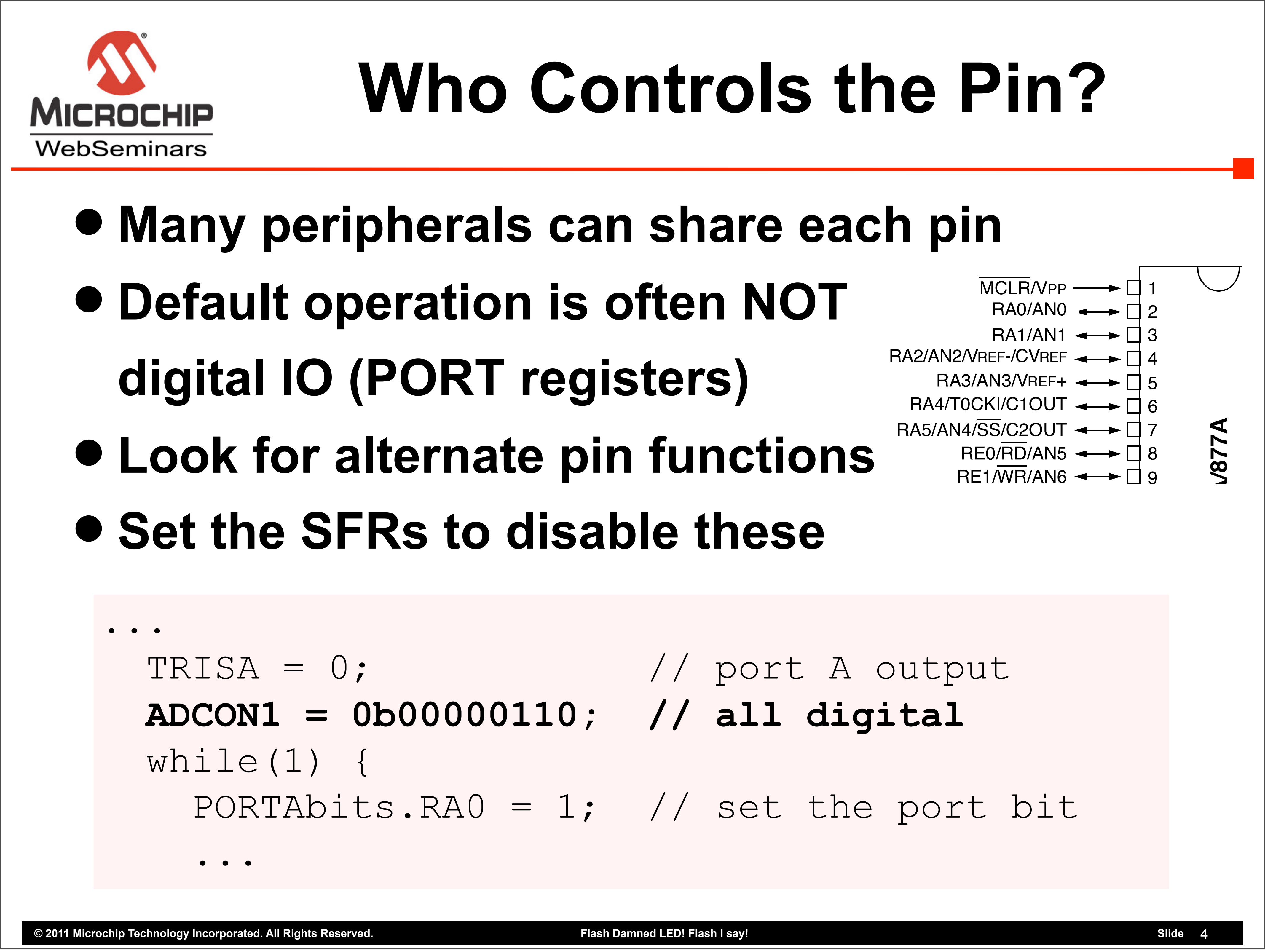
The oscillator type is one important setting specified via these configuration bits. You can typically choose from an internal oscillator, or an external crystal oscillator of various types. If this setting does not match your hardware, your device instruction clock may not even be running.

Other settings that might be relevant include the watchdog timer, the master clear pin, and brown-out reset. In fact, it's a good idea to check *every* setting in your device data sheet and specify the configuration setting that is relevant. A lot of these features, like the watch dog timer, are enabled by default and can alter the way your code will behave.

The added line of code to set the configuration bits is shown in bold type on this slide. In this example, we are targeting a PIC16F877A that is using a crystal oscillator. The watchdog timer has been disabled, as has all the other features. The device debug mode is set automatically by MPLAB IDE. The configuration bit settings *you* need will almost certainly be different to this. Check your device data sheet to see what setting are available on your device and how they work. Check your compiler's user's guide for information on how you set the configuration bits and the names of the settings.

If your code runs on the simulator, but not on an actual part, then the configuration bit settings are the first thing you should check.

If you have used the appropriate code to set the configuration bits and you are using MPLAB IDE version 8, make sure the configuration dialog option is set to read the configuration bit settings in your code, not those specified in the dialog.



Who Controls the Pin?

- Many peripherals can share each pin
- Default operation is often NOT digital IO (PORT registers)
- Look for alternate pin functions
- Set the SFRs to disable these

```
...
  TRISA = 0;              // port A output
  ADCON1 = 0b00000110;    // all digital
  while(1) {
    PORTAbits.RA0 = 1;    // set the port bit
    ...
```

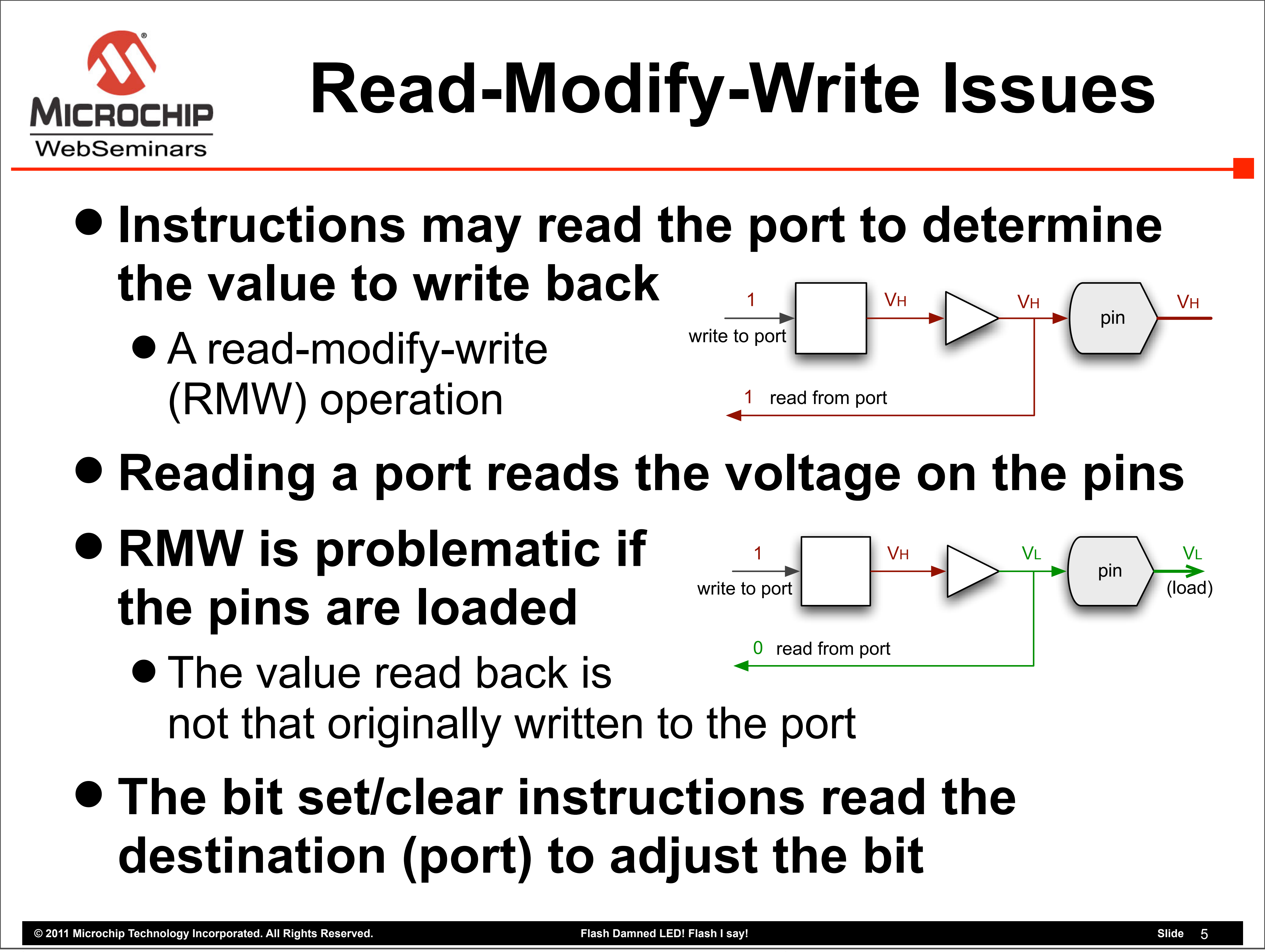
© 2011 Microchip Technology Incorporated. All Rights Reserved.

The PIC devices have many different peripherals on board. It's amazing how they can fit them all into such a small device. But this does lead to one of the most common beginner problems. Since there are many peripherals, but only a small number of physical pins in the package, the pins are multiplexed and three or even four peripheral IO lines might share the same pin.

The device data sheet will typically tag each pin with several labels that correspond to each possible function that pin may be assigned. Some device data sheets also have a 'pinout description table' which explains the possible functions of most pins. In the device pinout shown, pin 2 can be either bit 0 of digital IO port A (RA0) or an analog input (AN0). In many instances, the digital IO, or port peripheral, is *not* assigned the pin by default. You need to check and disable any peripheral that is using the pin before you can use the digital IO port.

For pin 2 on this device, the PIC16F877A, we are promoted to look up the settings for the analog-to-digital converter. By doing so, we discover that, by default, pin 2 is an analog input. Well, that wont be helpful if I want to flash an L-E-D, so in our example we have added code that will configure the port pins to be digital IO. This is shown in bold type.

The changes we have made to our example code so far will solve most problems, but there are several other issues that affect simple test programs such as we have.



Read-Modify-Write Issues

- Instructions may read the port to determine the value to write back
  - A read-modify-write (RMW) operation
- Reading a port reads the voltage on the pins
- RMW is problematic if the pins are loaded
  - The value read back is not that originally written to the port
- The bit set/clear instructions read the destination (port) to adjust the bit

1
write to port
V_H
V_H
pin
V_H
1 read from port
1
write to port
V_H
V_L
pin
V_L
(load)
0 read from port

© 2011 Microchip Technology Incorporated. All Rights Reserved.

One problem that can occur relates to how the port register is physically accessed. Most PIC instructions that specify a file register destination read that destination address first before writing the new modified value back. This is called a 'read-modify-write operation'. The increment instruction makes a good example of such an operation: it reads the value stored at an address, adds 1 to it, then writes this value back to the same address.

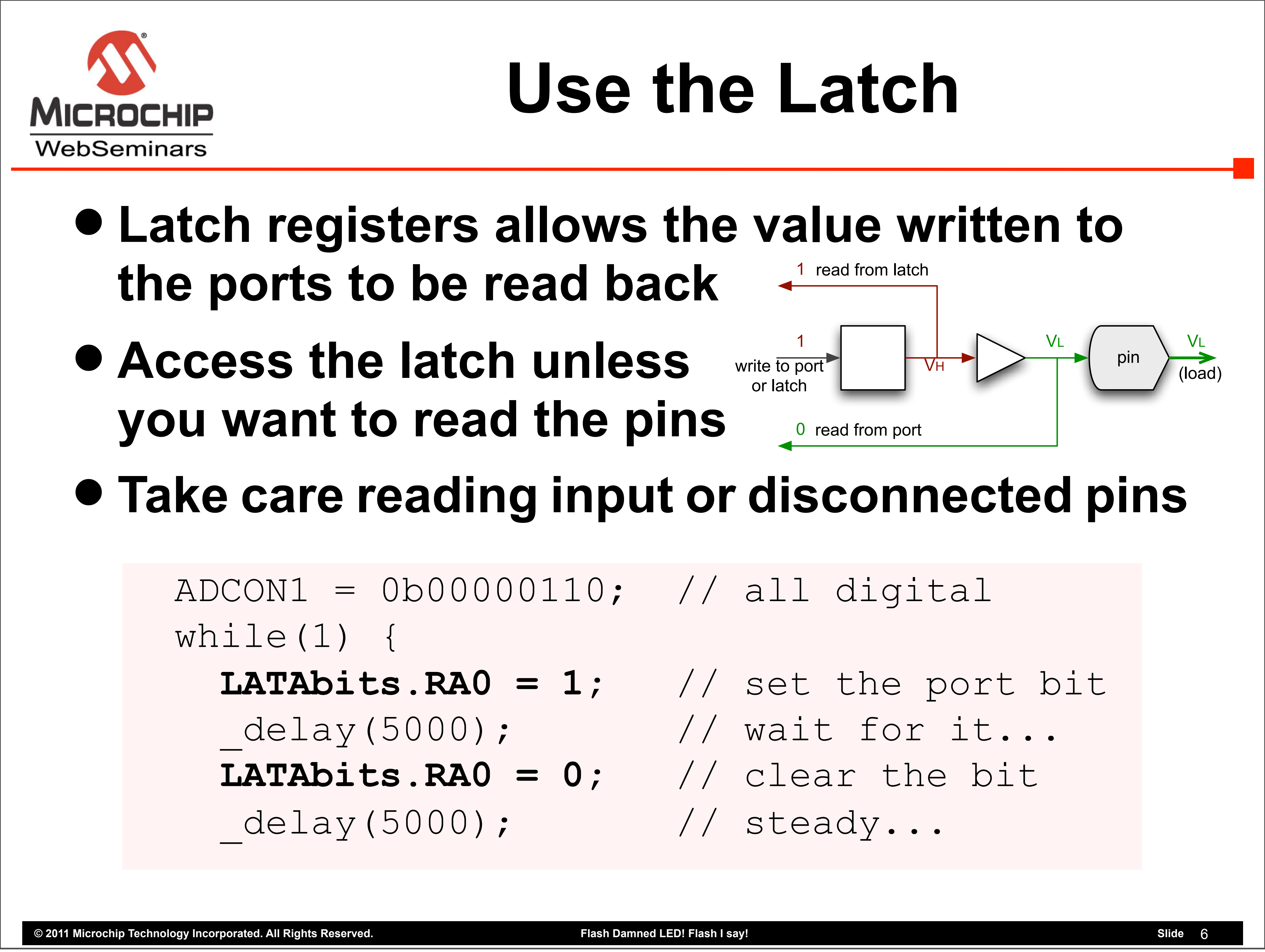
This is not usually a problem, but when acting on the port registers, things are little different. When you read a port register, you are reading the voltage on the actual pin, not a value stored in a memory location.

In the simplified diagram of the digital IO associated with a port shown towards the top of this slide, a logic 1 is written to the port register. This value is held by the register -- the square box in the diagram -- and a buffer drives the port pin voltage.

If the pin is being loaded, then there is no guarantee that the voltage on the pin is logically the same as what was written to the register. In the lower diagram, although a logic 1 was written to the port register, the buffer cannot maintain the high voltage on the pin and this voltage is interpreted as a logic 0 when it is read back.

If the load is highly capacitive, then the voltage on the pin may stabilise at the correct logic level, but only after a relatively long period of time -- time in which you could access the port in subsequent code.

Any instruction which reads its destination to determine the value to write back may fail due to this read-modify-write operation acting on a port register address. A lot of programmers do not realise that the bit set and bit clear instructions first read the entire enclosing byte before writing back that byte with the specified bit adjusted, so these instructions can fail in the same way. These instructions are commonly used when accessing ports and all sorts of odd behaviour can result if you don't take this situation into account.



MICROCHIP WebSeminars

Use the Latch

- Latch registers allows the value written to the ports to be read back
- Access the latch unless you want to read the pins
- Take care reading input or disconnected pins

1 read from latch
1
write to port
or latch
VH
VL
pin
VL
(load)
0 read from port

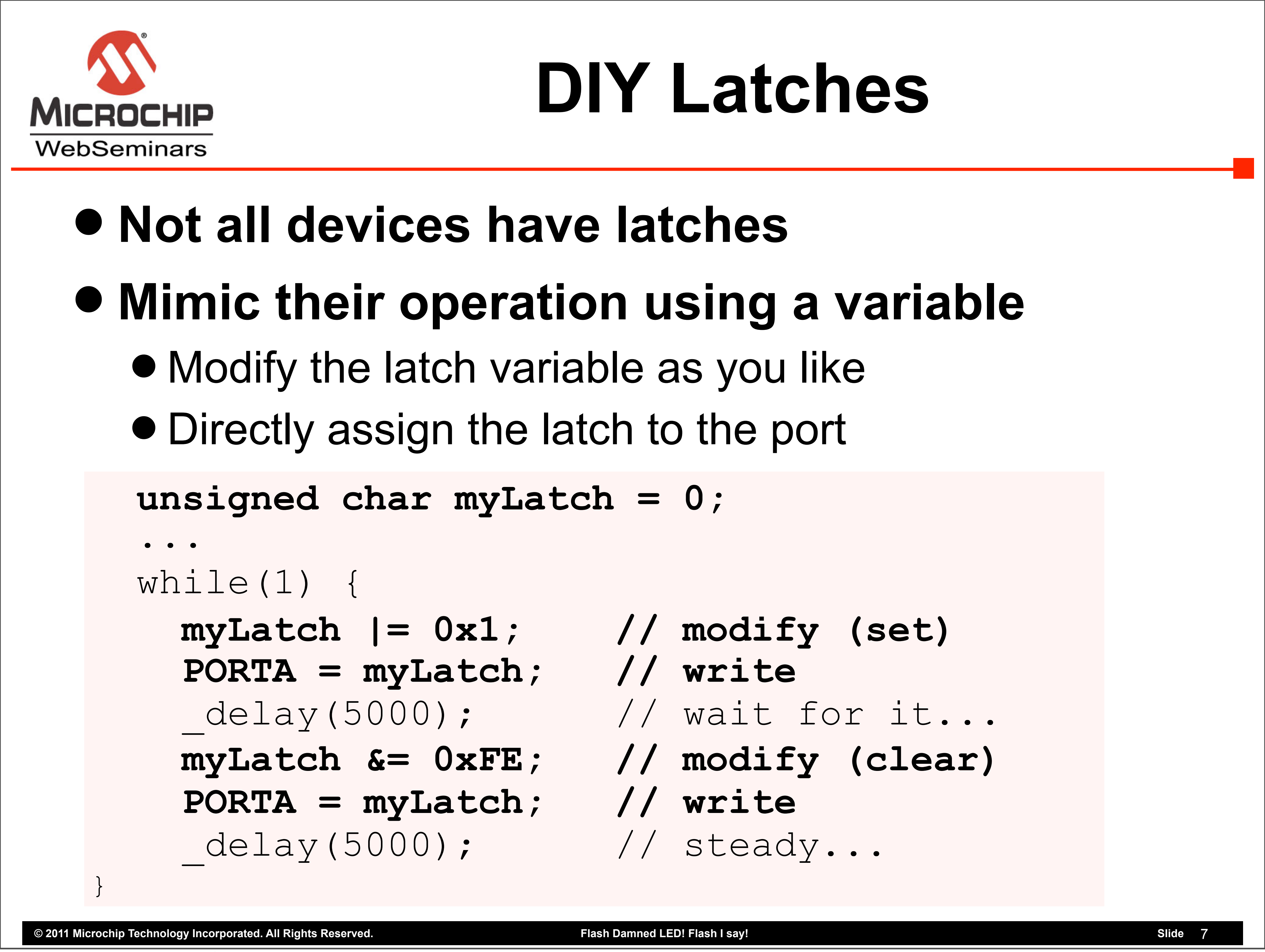
```
ADCON1 = 0b00000110;  // all digital
while(1) {
  LATAbits.RA0 = 1;   // set the port bit
  _delay(5000);       // wait for it...
  LATAbits.RA0 = 0;   // clear the bit
  _delay(5000);       // steady...
```

© 2011 Microchip Technology Incorporated. All Rights Reserved.

Many PIC devices have an addition register associated with each digital IO port, called a 'latch register'. This allows you to read back the value written to the corresponding port register. Writing to the port register performs the same action as writing to the corresponding latch register i.e. the value is held and will drive the pin. But, you have a choice of reading from the latch register or from the port itself, i.e. the actual pins.

Always access the latch register instead of the port register, unless you specifically want to read the values on the pins. Our example program has been updated so it accesses the latches associated with port A and this is shown in bold type. This example is C code, but the compiler used will output bit set and bit clear instructions to implement this program in machine code. By using the latch register we avoid any possible read-modify-write issues associated with these instructions.

Also remember that when a read-modify-write instruction reads a port, those pins that are analog inputs or that are physically disconnected will be read as logic 0. This 0 may then be written back to the port by the instruction. If these input or disconnected pins are subsequently set to be outputs, the value on the port may not be what you are expecting, even though you did not explicitly modify the port value.



DIY Latches

- **Not all devices have latches**
- **Mimic their operation using a variable**
  - Modify the latch variable as you like
  - Directly assign the latch to the port

```
  unsigned char myLatch = 0;
  ...
  while(1) {
    myLatch |= 0x1;      // modify (set)
    PORTA = myLatch;     // write
    _delay(5000);        // wait for it...
    myLatch &= 0xFE;     // modify (clear)
    PORTA = myLatch;     // write
    _delay(5000);        // steady...
}
```

© 2011 Microchip Technology Incorporated. All Rights Reserved.

Some of the smaller PIC devices do not have latches, but this does not mean that you need suffer unexpected results when using ports. You can easily mimic the operation of a latch with an ordinary variable.

Use the variable to hold the value you want to write to the port. Perform any modification of the value in the latch variable. Once you have finished, copy the latch variable to the port register, using only the assignment operator. Our code example has been modified to show how you might use this variable if your target device does not have a latch register.

--

I hope by now your L-E-D is flashing away like a Christmas tree. Many code faults are caused by simple oversights and I hope that this webinar has alerted you to potential issues that you can keep in mind for all your future projects.