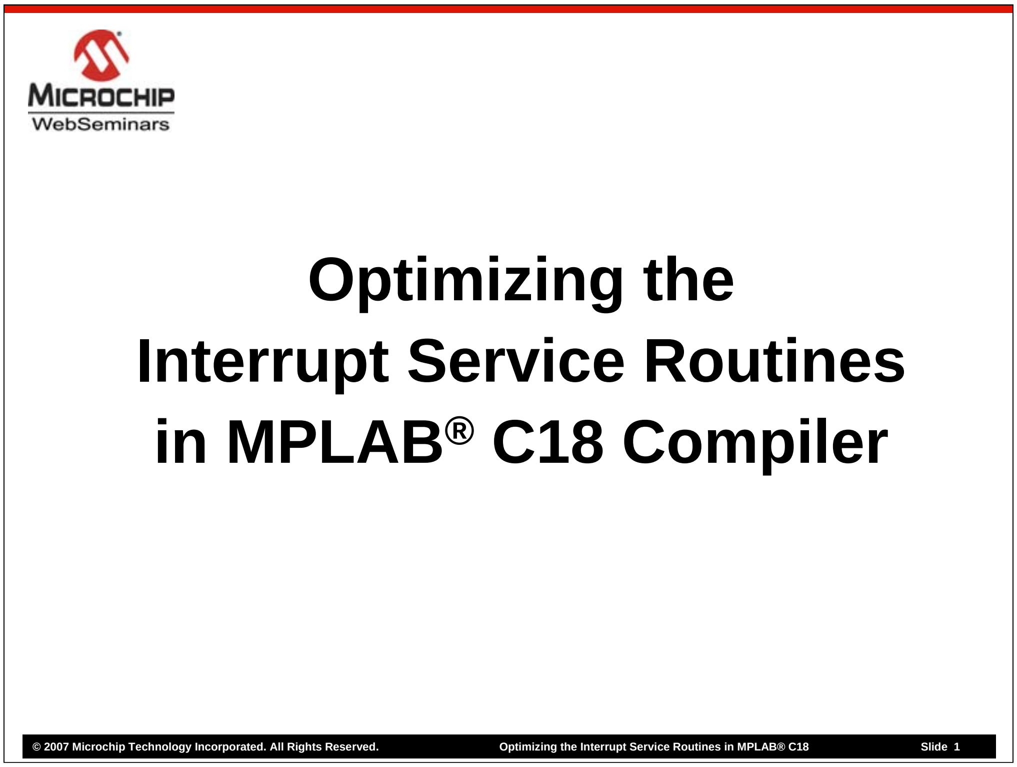
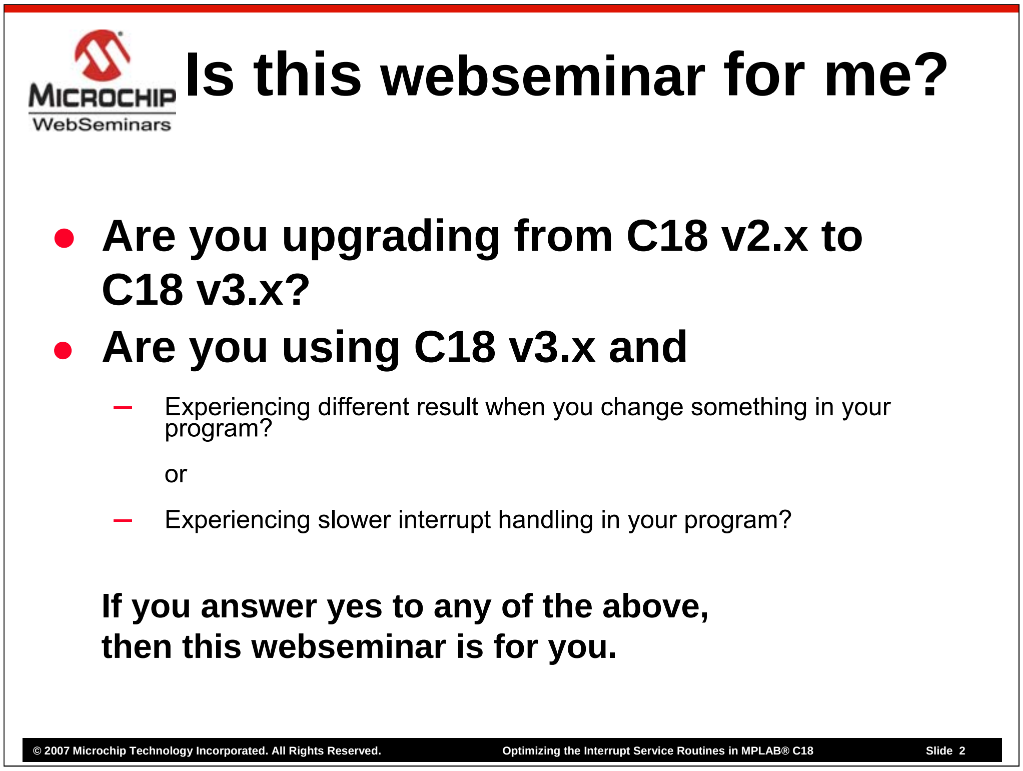


Optimizing the Interrupt Service Routines in MPLAB® C18 Compiler

Hi and thank you for choosing this webseminar. This presentation is about optimizing the interrupt service routines in MPLAB® C18 compiler.

My name is Alireza Moshtaghi and I am a software engineer in the language tools team at Microchip Technology.

Just by looking at the topic of this webseminar one can tell that it is touching on a very specific corner of software development on the C18 programming environment. And obviously you may ask “is this webseminar for me?”



Is this webseminar for me?

- **Are you upgrading from C18 v2.x to C18 v3.x?**
- **Are you using C18 v3.x and**
  - Experiencing different result when you change something in your program?

    or
  - Experiencing slower interrupt handling in your program?

If you answer yes to any of the above, then this webseminar is for you.

Well, if you are upgrading from version 2.x of C18 compiler to version 3.x

Or

If you are already using version 3.0 and above of MPLAB® C18 compiler but experiencing certain problems for example:

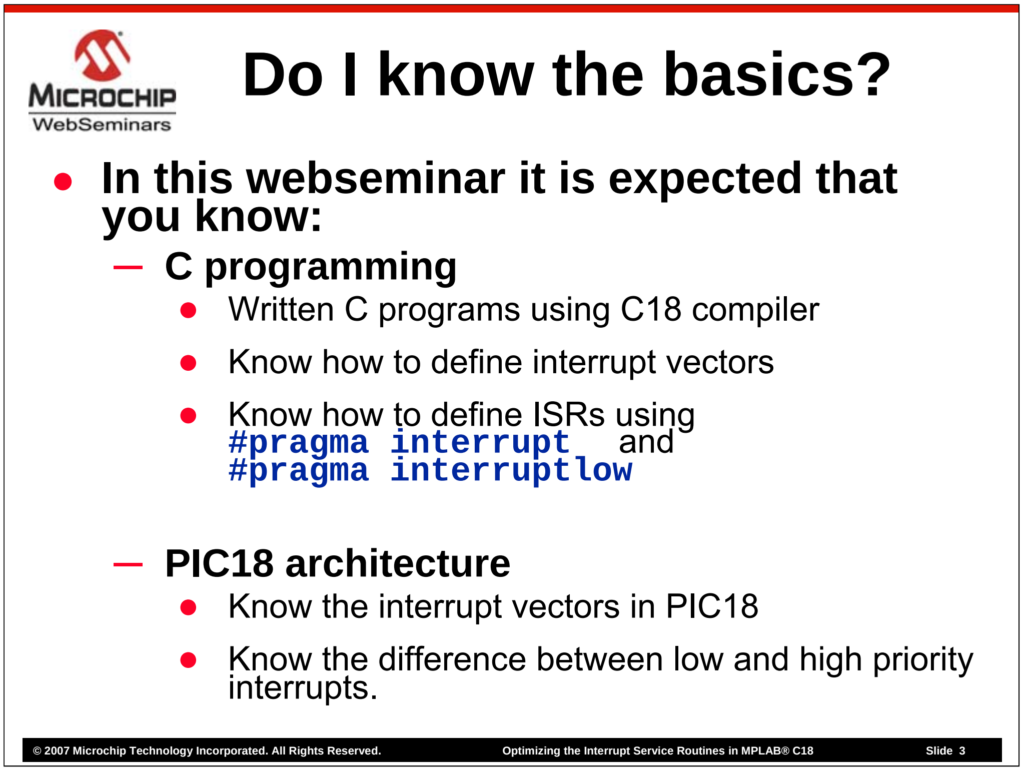
When you modify something in your code, and notice a different result in another part of your program,

Or

If in general you are experiencing slower interrupt handling performance than expected,

Or if simply you want to learn more about C18 compiler,

Then you can benefit from this webseminar, given that you know the basics.



Do I know the basics?

- In this webseminar it is expected that you know:
  - C programming
    - Written C programs using C18 compiler
    - Know how to define interrupt vectors
    - Know how to define ISRs using `#pragma interrupt` and `#pragma interruptlow`
  - PIC18 architecture
    - Know the interrupt vectors in PIC18
    - Know the difference between low and high priority interrupts.

This webseminar is not comprehensive, it is assumed that you know the basics.

As far as C programming, we would like you to have had experienced programming using the MPLAB® C18 compiler.

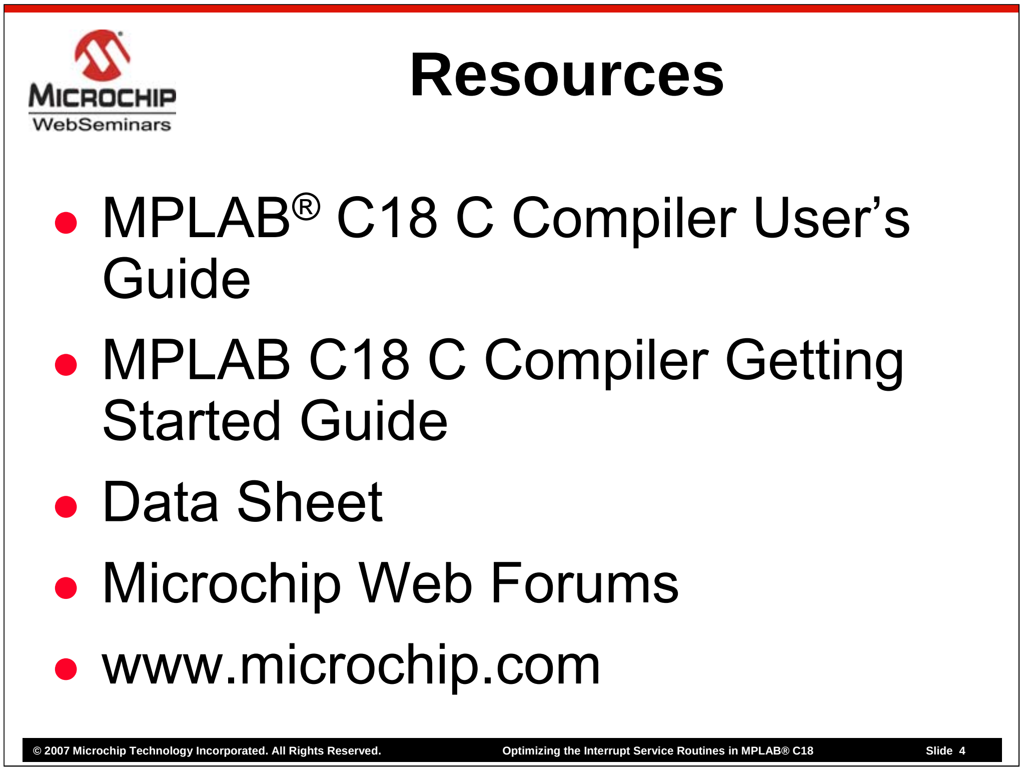
It is crucial that you know how to define interrupt vectors and to know how to define the Interrupt Service Routine using the #pragma interrupt and #pragma interruptlow.

Also, it is important that you know about the PIC18 architecture:

Know about the interrupt vectors;

And know about the difference between low and high priority interrupts.

If you think that you need to know about these topics…

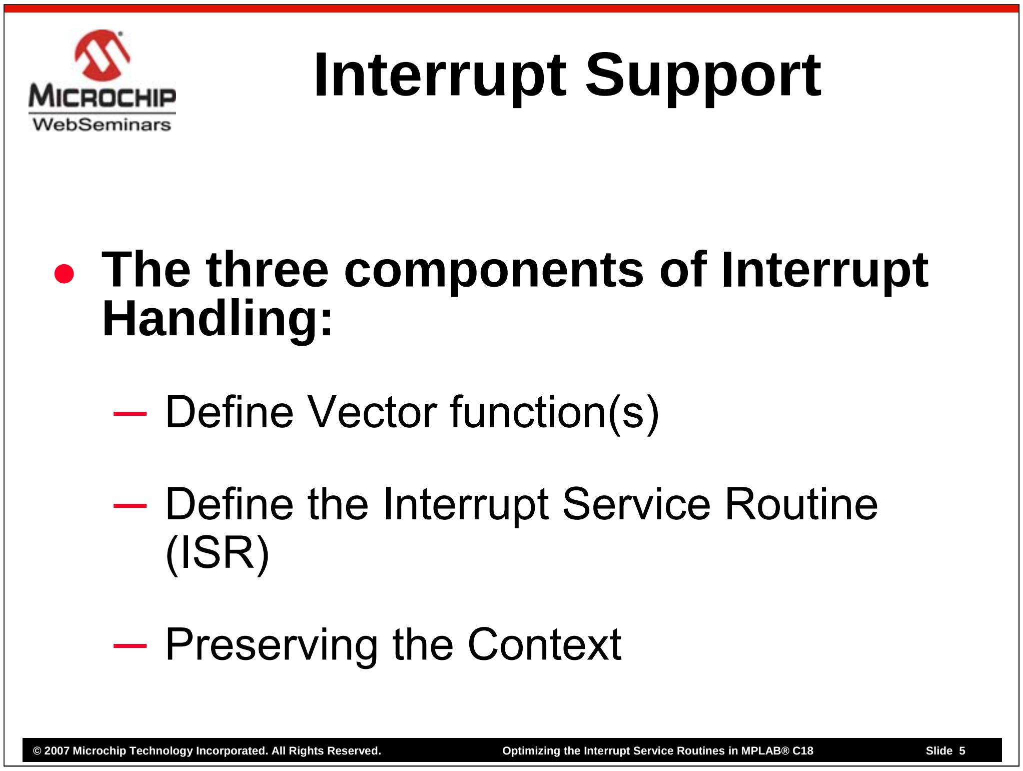


MICROCHIP WebSeminars

Resources

- MPLAB® C18 C Compiler User’s Guide
- MPLAB C18 C Compiler Getting Started Guide
- Data Sheet
- Microchip Web Forums
- www.microchip.com

You can refer to these resources. I will repeat this slide at the end of this presentation.



Interrupt Support

- **The three components of Interrupt Handling:**
  - Define Vector function(s)
  - Define the Interrupt Service Routine (ISR)
  - Preserving the Context

© 2007 Microchip Technology Incorporated. All Rights Reserved. Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 5

If you have come this far, then you really are interested. So lets directly go to the main topic.

In the discussions about Interrupt Service routines in general, three topics are important:

Defining the vector function(s),

defining the interrupt service routine itself,

And preserving the context, or as some people call it, context switch.

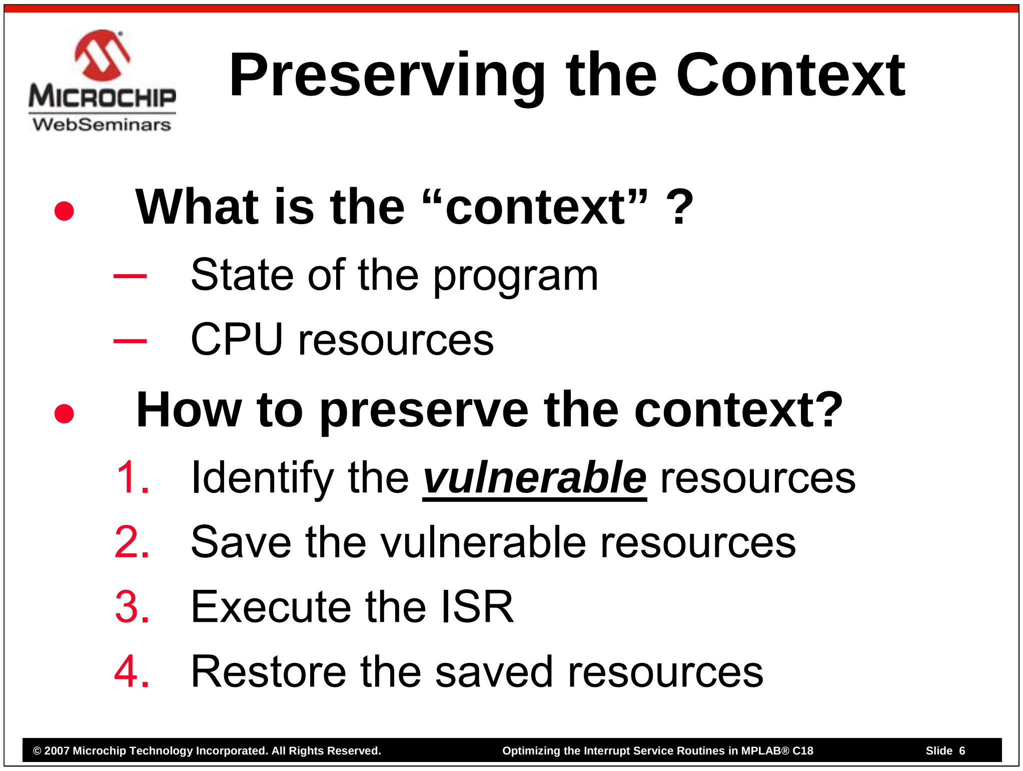
As I mentioned before, it is assumed that you are already familiar with the topics in first two bullets

(pause)

The focus of this webseminar is on the third bullet, preserving the context and how to make it more efficient.

(pause)

But what does context preservation exactly mean?



MICROCHIP WebSeminars

Preserving the Context

- **What is the "context" ?**
  - State of the program
  - CPU resources
- **How to preserve the context?**
  1. Identify the ***vulnerable*** resources
  2. Save the vulnerable resources
  3. Execute the ISR
  4. Restore the saved resources

In the computer science jargon, context is synonymous to the state of the program. Some people may also look at it as CPU resources that the program is using at a given time. These resources may be anything from registers to memory and peripherals.

(pause)

In a few minutes you will see the list of resources that MPLAB® C18 manages.

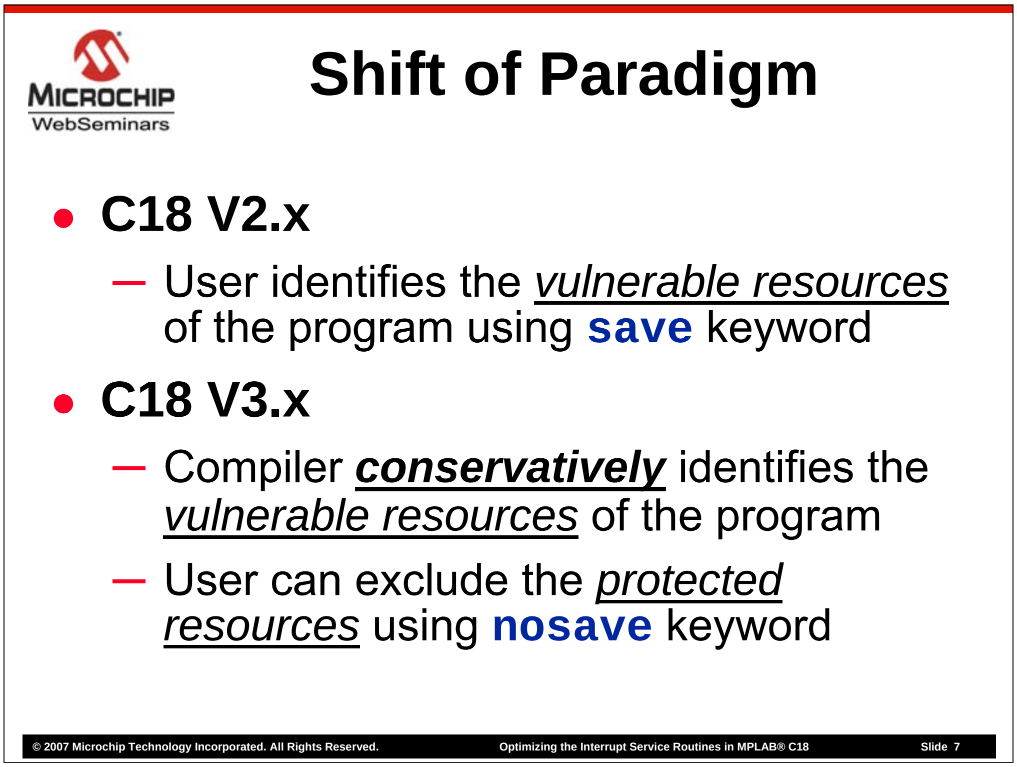
But before that, I would like you to know what it takes to preserve the context.

The first step is to identify the vulnerable resources. Vulnerable resources are usually the resources that are shared by the ISR and the main line code.

Once we know which resources are vulnerable, we must arrange for them to be saved prior to the execution of ISR and be restored after the ISR is finished

usually the process of copying the vulnerable resources onto the stack and later restoring them is very time consuming. So careful identification of vulnerable resources is the key to implementing a fast Interrupt Service Routine.

Now the question is who is responsible for identifying the vulnerable resources?



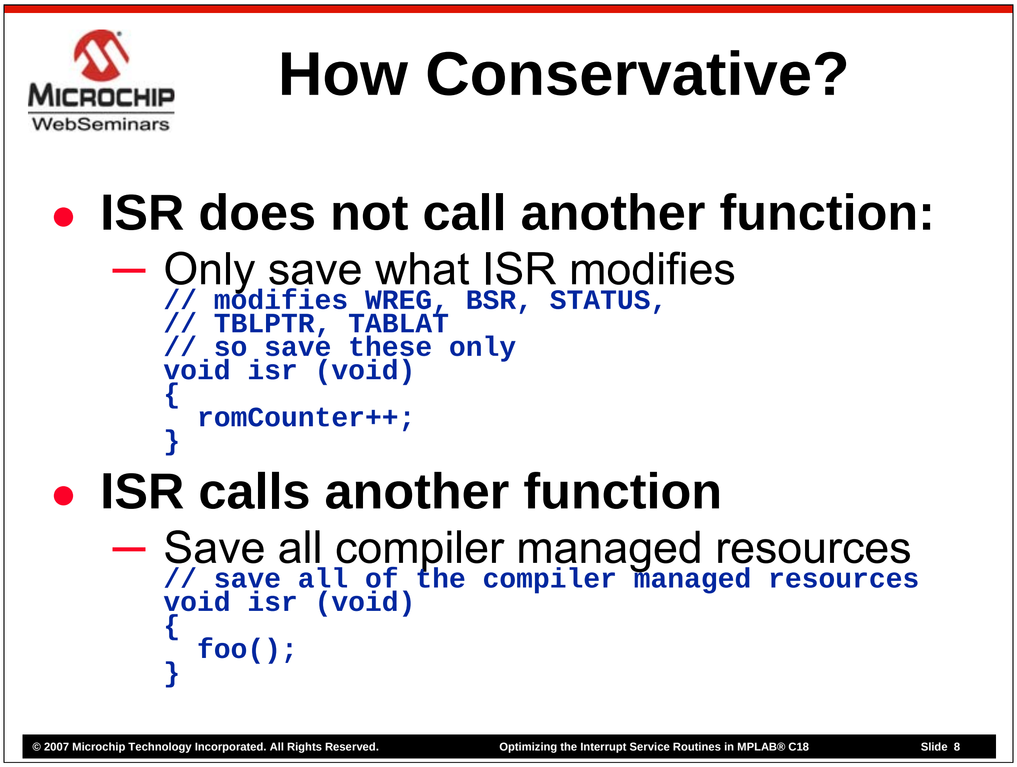
MICROCHIP WebSeminars

Shift of Paradigm

- **C18 V2.x**
  - User identifies the *vulnerable resources* of the program using **save** keyword
- **C18 V3.x**
  - Compiler ***conservatively*** identifies the *vulnerable resources* of the program
  - User can exclude the *protected resources* using **nosave** keyword

In the C18 versions before 3.0, the user is responsible for identifying the vulnerable resources. Pretty much the compiler would not even try to identify what needs to be saved. This turned out to be an error prone and frustrating burden.

So in versions 3.0 and higher of MPLAB C18, the compiler conservatively identifies the vulnerable resources. It means that, if the compiler can not prove whether a resource is vulnerable or not, it plays it safe and saves the resource anyway. However, the user can use the nosave keyword to keep the compiler from saving the resources that are not vulnerable. Later we will see examples about this feature.



MICROCHIP
WebSeminars

How Conservative?

- **ISR does not call another function:**
  - Only save what ISR modifies

```
// modifies WREG, BSR, STATUS,
// TBLPTR, TABLAT
// so save these only
void isr (void)
{
  romCounter++;
}
```

- **ISR calls another function**
  - Save all compiler managed resources

```
// save all of the compiler managed resources
void isr (void)
{
  foo();
}
```

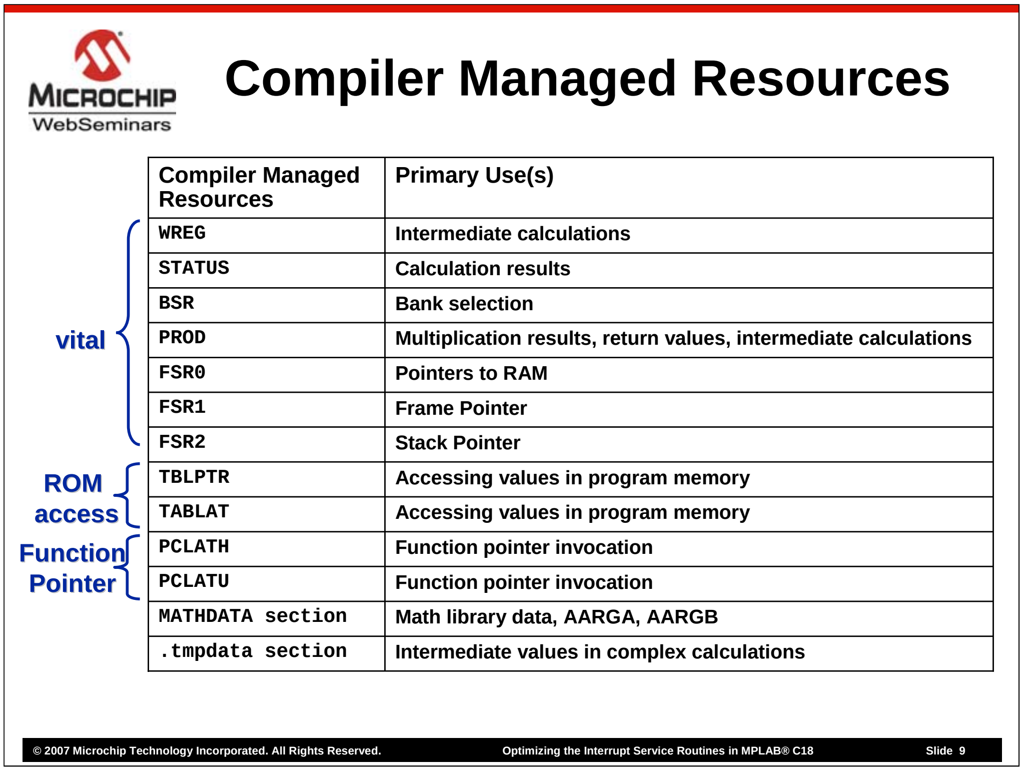
Here is an example about the conservative approach of C18

The Interrupt Service Routine in the first bullet does not call any functions. So the compiler can analyze the isr function and identify only the resources that are modified. It turns out that this function modifies WREG, BSR, STATUS, TBLPTR and TABLAT registers. So the compiler will automatically save these registers and only these registers.

In the second example, function foo is called from the Interrupt Service Routine. In this case, the compiler does not know what is being modified in function foo in fact function foo may very well be in a separate compilation module. So the compiler plays conservatively and saves all of the compiler managed resources.

(pause)

This takes us to the question “what are the compiler managed resources?”



Compiler Managed Resources

	Compiler Managed Resources	Primary Use(s)
vital	`WREG`	Intermediate calculations
	`STATUS`	Calculation results
	`BSR`	Bank selection
	`PROD`	Multiplication results, return values, intermediate calculations
	`FSR0`	Pointers to RAM
	`FSR1`	Frame Pointer
	`FSR2`	Stack Pointer
ROM access	`TBLPTR`	Accessing values in program memory
	`TABLAT`	Accessing values in program memory
Function Pointer	`PCLATH`	Function pointer invocation
	`PCLATU`	Function pointer invocation
	`MATHDATA section`	Math library data, AARGA, AARGB
	`.tmpdata section`	Intermediate values in complex calculations

This table captures the resources that the MPLAB C18 compiler manages. Of course the compiler analyzes the Interrupt Service Routine, only for these resources, to find if any of them need to be preserved.

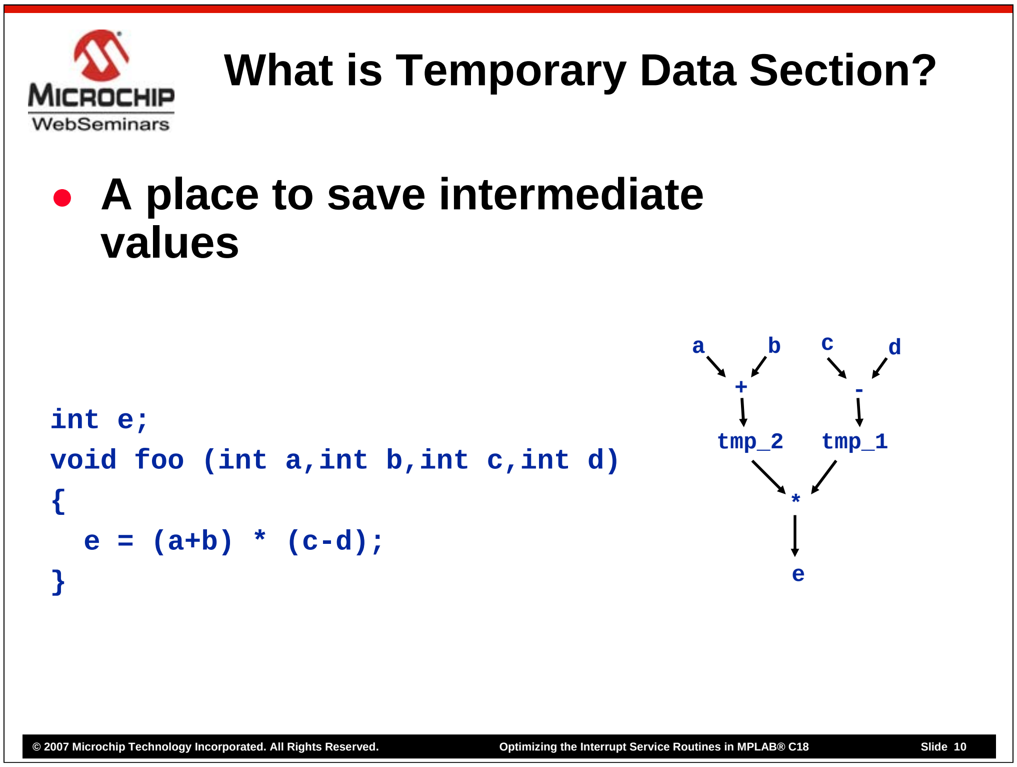
The first 7 registers in this table are what I call vital; meaning that most of the time you would leave the compiler free to preserve these resources across the ISR if needed.

Next are the TBLPTR and TABLAT registers which are used for accessing the program memory, if you know that your ISR thread does not access any rom data, you can keep the compiler from unnecessarily saving TBLPTR and TABLAT registers.

Likewise, if you know that your ISR thread does not use function pointers, you can keep the compiler from saving PCLATH and PCLATU registers.

MATHDATA is a special section that is used by the math libraries, it is also used in the traditional mode to implement some of the function interfaces. This section is small and usually you do not need to worry about it.

The last item in the table is .tmpdata section which contains the intermediate values in complex calculations. In the next slides I explain this section.



What is Temporary Data Section?

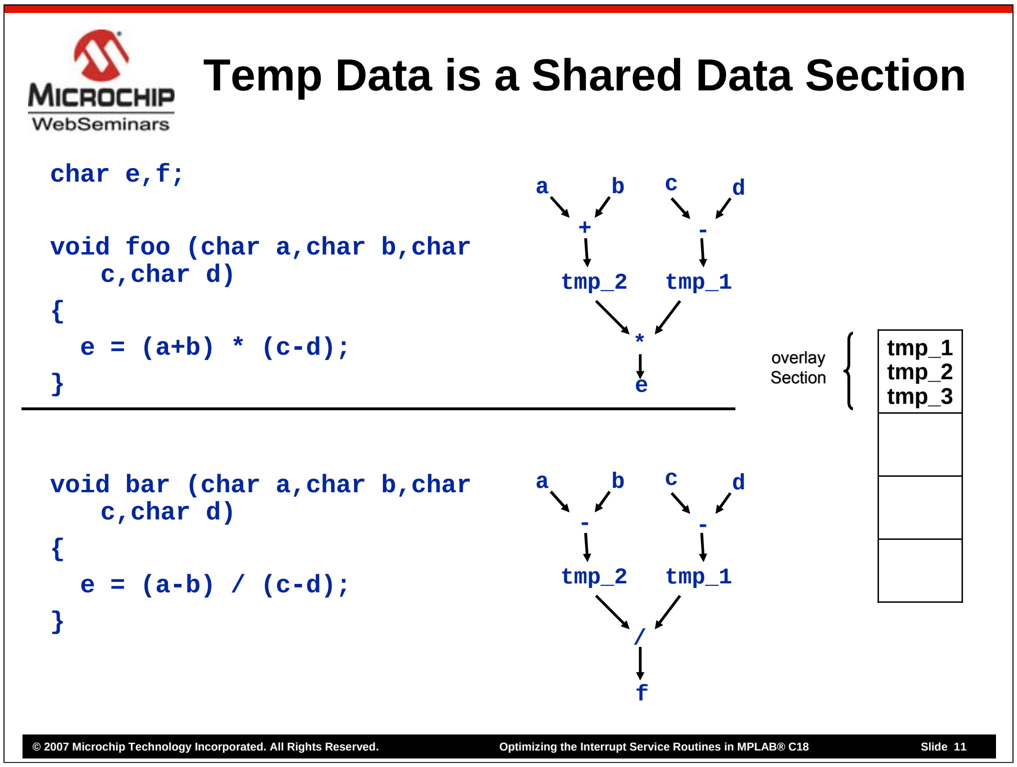
- A place to save intermediate values

```
int e;
void foo (int a,int b,int c,int d)
{
  e = (a+b) * (c-d);
}
```

Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 10

Look at this simple example. On an 8-bit processor, it is unavoidable that some temporary variables are needed when calculating an expression such as (a+b) * (c-d).

The dataflow graph on the right hand side of this slide shows a possible way of calculating this expression. I am not saying that C18 performs the exact same thing, however, in general you can consider this close to reality, where the result of a+b is placed in a temporary variable then the result of c-d is placed in another temporary variable and then the multiply is performed on the two temporary variables to produce the value in e



MICROCHIP WebSeminars

Temp Data is a Shared Data Section

```
char e,f;

void foo (char a,char b,char
   c,char d)
{
  e = (a+b) * (c-d);
}
```

```
void bar (char a,char b,char
   c,char d)
{
  e = (a-b) / (c-d);
}
```

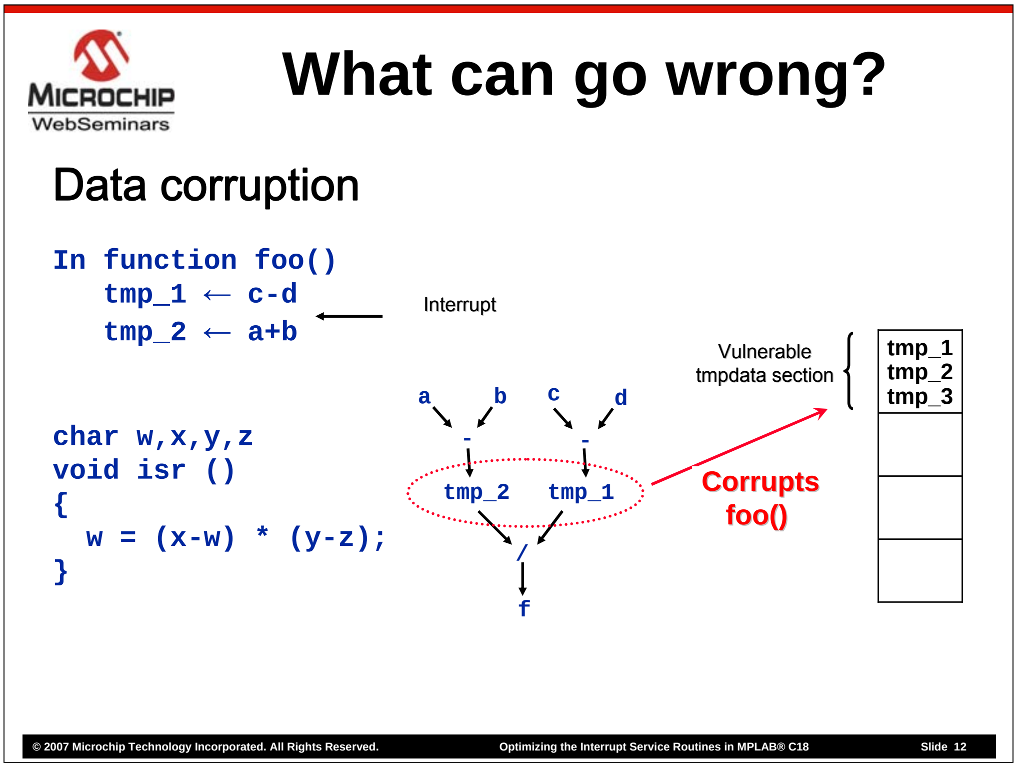
a
b
c
d
+
-
tmp_2
tmp_1
*
e
a
b
c
d
-
-
tmp_2
tmp_1
/
f
overlay Section
tmp_1
tmp_2
tmp_3

© 2007 Microchip Technology Incorporated. All Rights Reserved. Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 11

Furthermore, in C18, in order to save valuable memory, the .tmpdata section is shared among functions, or in C18 terminology, it is overlaid. It means that in the case of foo() and bar() functions, the same tmp_1 and tmp_2 variables are used to save the intermediate calculations.

Now this impose no problem because foo() and bar() are part of the same execution thread. In other words, foo() and bar() will never be active at the same time.

So what can go wrong?



What can go wrong?

Data corruption

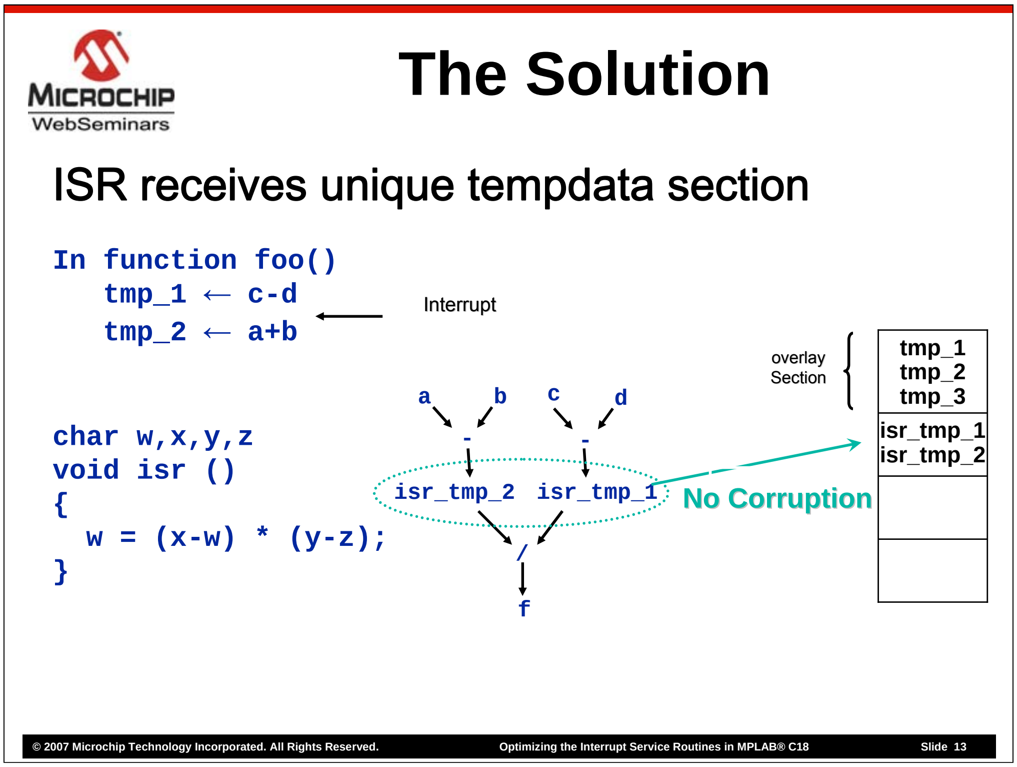
```
In function foo()
   tmp_1 ← c-d
   tmp_2 ← a+b


char w,x,y,z
void isr ()
{
  w = (x-w) * (y-z);
}
```

Presence of an interrupt can change things. In this hypothetical example, function foo() is using the .tmpdata section for intermediate calculations. Consider the scenario that foo has calculated c-d and has placed its result in tmp_1. Before it can calculate a+b, an interrupt occurs and isr() is activated.

If the isr() function were to use the same temporary variables to do its own calculations on w, x, y and z, the corruption of data in tmp_1 was emminent.

This is an example of vulnerable resource.

Now how does C18 address this possible problem?



MICROCHIP WebSeminars

The Solution

ISR receives unique tempdata section

```
In function foo()
   tmp_1 ← c-d
   tmp_2 ← a+b
```

Interrupt

```
char w,x,y,z
void isr ()
{
  w = (x-w) * (y-z);
}
```

a b c d

- -

isr_tmp_2 isr_tmp_1

/

f

overlay Section

tmp_1 tmp_2 tmp_3
isr_tmp_1 isr_tmp_2

No Corruption

© 2007 Microchip Technology Incorporated. All Rights Reserved. Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 13

In C18, the ISR receives a unique .tmpdata section. In effect, in this method, we are removing the vulnerability of the .tmpdata section.

So no matter what function foo() does on the original .tmpdata section, the isr() function will place its temporary variables in a separate section, eliminating the chance of data corruption.

Now in this example, the Interrupt Service Routine is not calling any other function, so we are safe.

(pause)

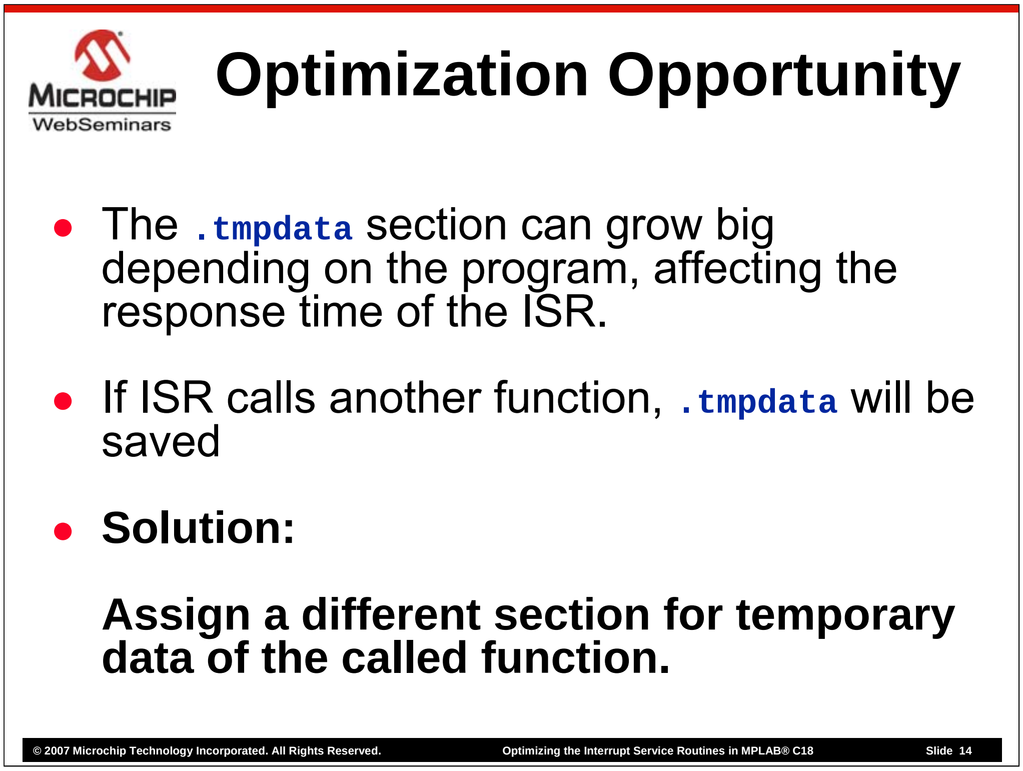
But what happens if it were to call another function? what if the called function modifies the original .tmpdata?

(pause)

As I mentioned before, in this case, the compiler will play conservatively and save all resources.

Of course it is very likely that in this approach, compiler saves more than needed, imposing unwanted overhead that slow down the Interrupt Service Routine.

So we have an optimization opportunity. But the compiler can’t handle it on its own, and requires some clue from the user.



Optimization Opportunity

- The `.tmpdata` section can grow big depending on the program, affecting the response time of the ISR.
- If ISR calls another function, `.tmpdata` will be saved
- **Solution:**

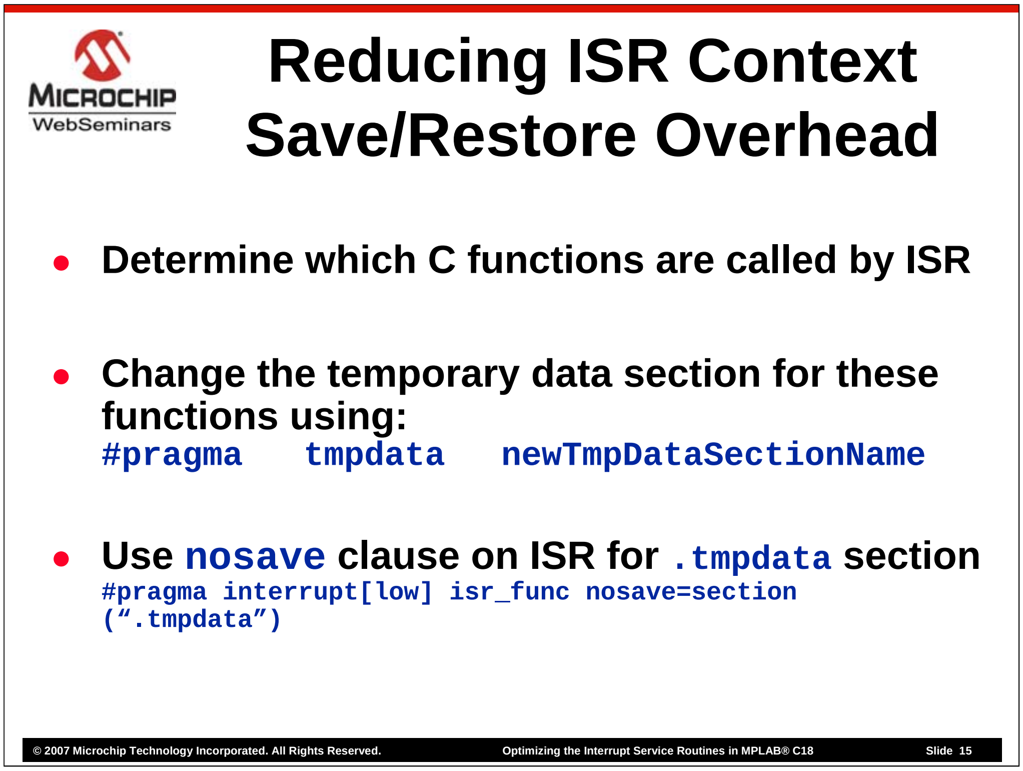
  Assign a different section for temporary data of the called function.

Usually, .tmpdata section grows large depending on how complex the calculations in the program are, so if the ISR is calling another function, the majority of penalty that the program pays is on preserving this section.

(pause)

So how can we avoid the need for saving the .tmpdata section? The solution is to remove the vulnerability of the .tmpdata section by assigning the ISR thread to a different temporary data section.

How?



MICROCHIP WebSeminars

Reducing ISR Context Save/Restore Overhead

- Determine which C functions are called by ISR
- Change the temporary data section for these functions using:

```
#pragma   tmpdata   newTmpDataSectionName
```

- Use **nosave** clause on ISR for **.tmpdata** section

```
#pragma interrupt[low] isr_func nosave=section
(“.tmpdata”)
```

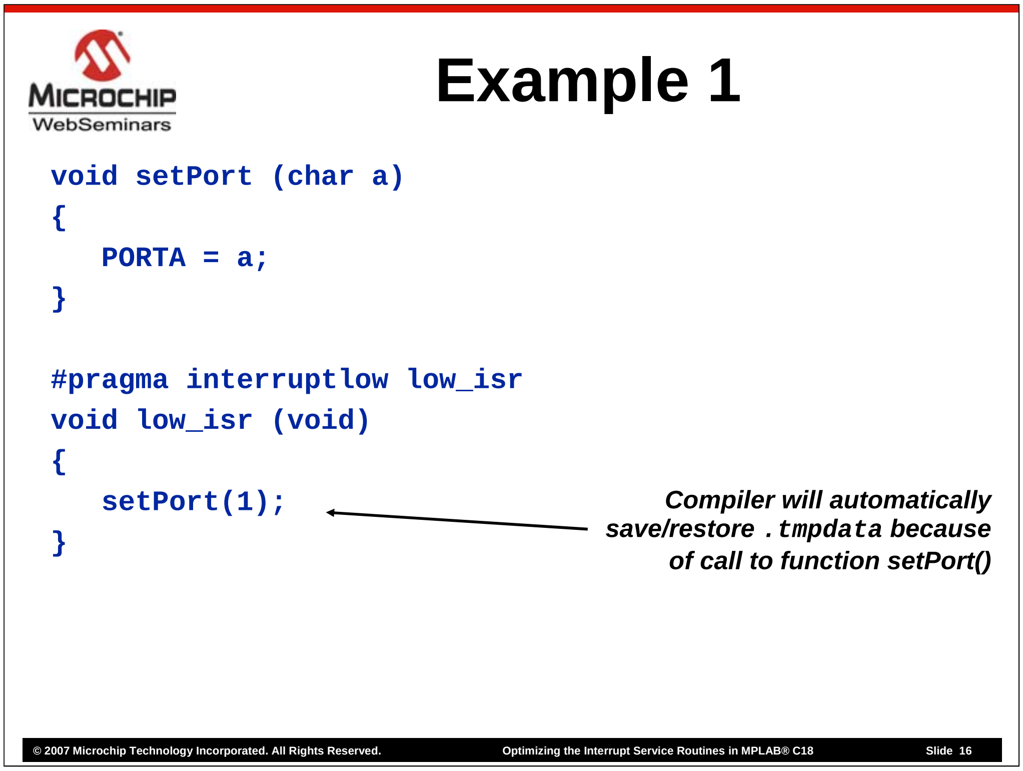
Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 15

First determine which C functions are called by the ISR.

Then Change the temporary data section for these functions using #pragma tmpdata with the new tmporary data section name.

Now, it is safe to tell the compiler to not save the .tmpdata section for this Interrupt Service Routine

You can do this using the nosave keyword in the #pragma interrupt or the #pragma interruptlow



Example 1

```
void setPort (char a)
{
   PORTA = a;
}

#pragma interruptlow low_isr
void low_isr (void)
{
   setPort(1);
}
```

Compiler will automatically save/restore `.tmpdata` because of call to function setPort()

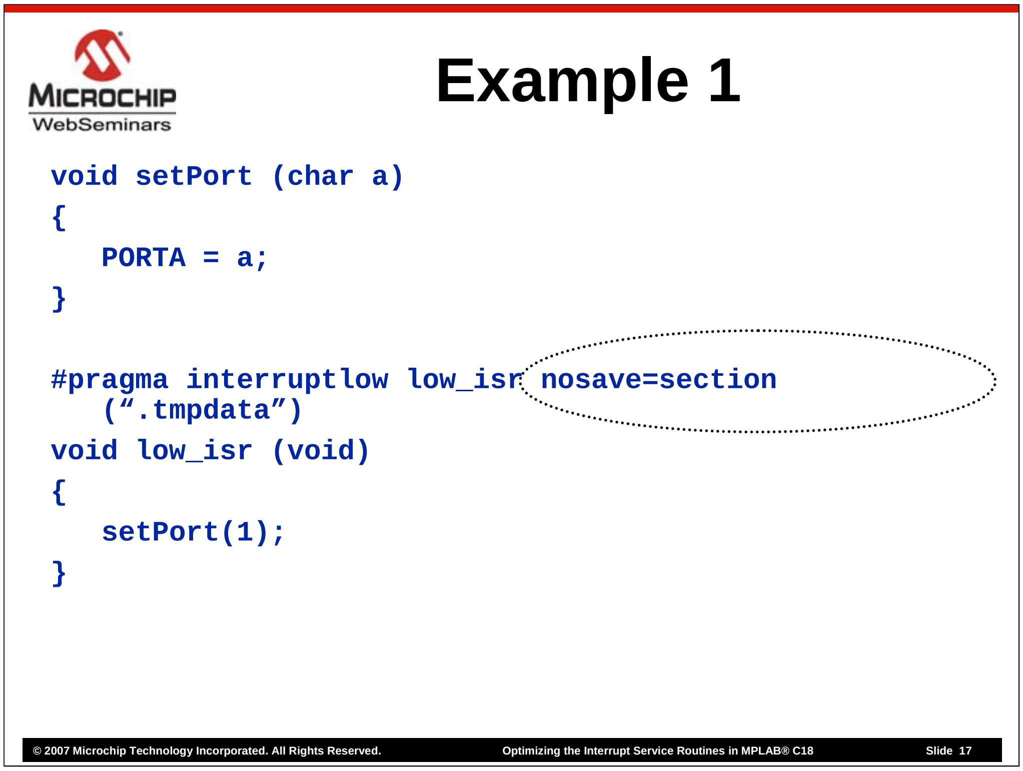
Now to wrap things up, lets look at an example:

And see how can we apply what we have learned.

Before I show you the solution, I would like to ask that you pause the presentation for a moment and give yourself a chance to remember what we discussed so far

(pause)

Here, function setPort() is only doing a write to port A, by doing code analysis, a savvy programmer may be able to conclude that setPort() function makes no access to .tmpdata. But the compiler does not perform any cross function analysis, it just knows that a function is called from the low_isr() interrupt service routine, so it saves and restores all compiler managed resources including the .tmpdata section, resulting in a poor performance for the Interrupt Service Routine.



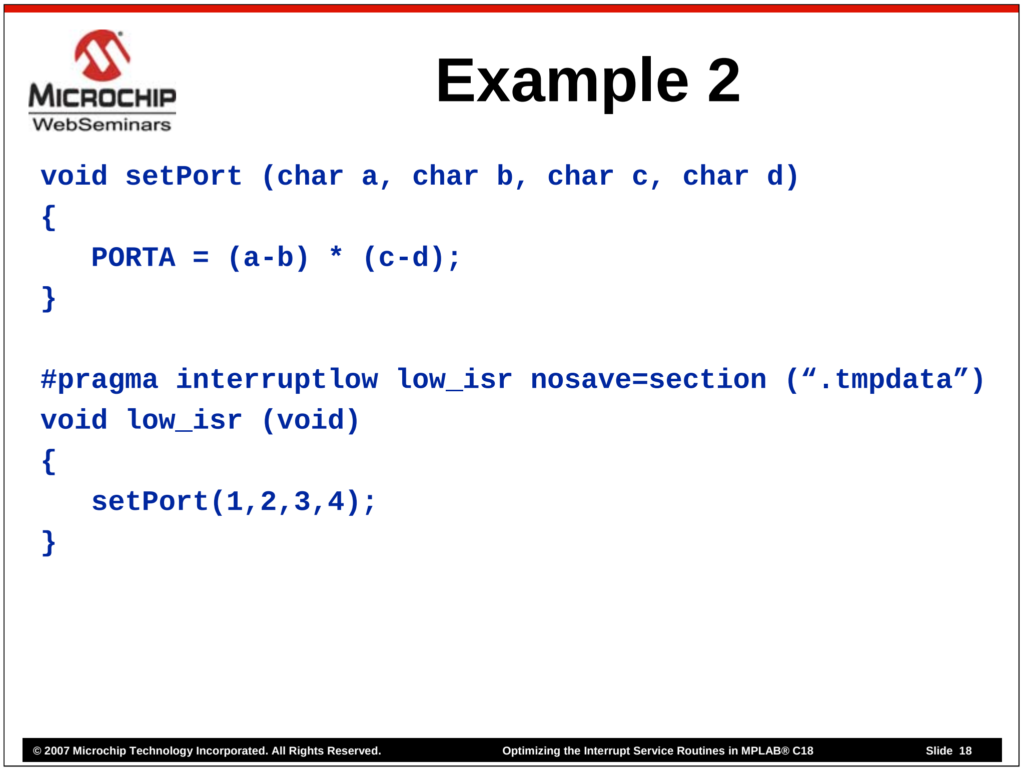
Example 1

```
void setPort (char a)
{
   PORTA = a;
}

#pragma interruptlow low_isr nosave=section
   (“.tmpdata”)
void low_isr (void)
{
   setPort(1);
}
```

Now that we know that tmpdata section is not used, we can use the nosave keyword in the pragma interruptlow to keep the compiler from saving it.

You may ask do I always have to make assembly code analysis in order to know if I can use the nosave keyword on .tmpdata section?

The answer is no and I will show you why in the next example.



Example 2

```
void setPort (char a, char b, char c, char d)
{
   PORTA = (a-b) * (c-d);
}

#pragma interruptlow low_isr nosave=section (“.tmpdata”)
void low_isr (void)
{
   setPort(1,2,3,4);
}
```

Optimizing the Interrupt Service Routines in MPLAB® C18 Slide 18

In this example, I have modified the setPort() function to also do some complex calculation.

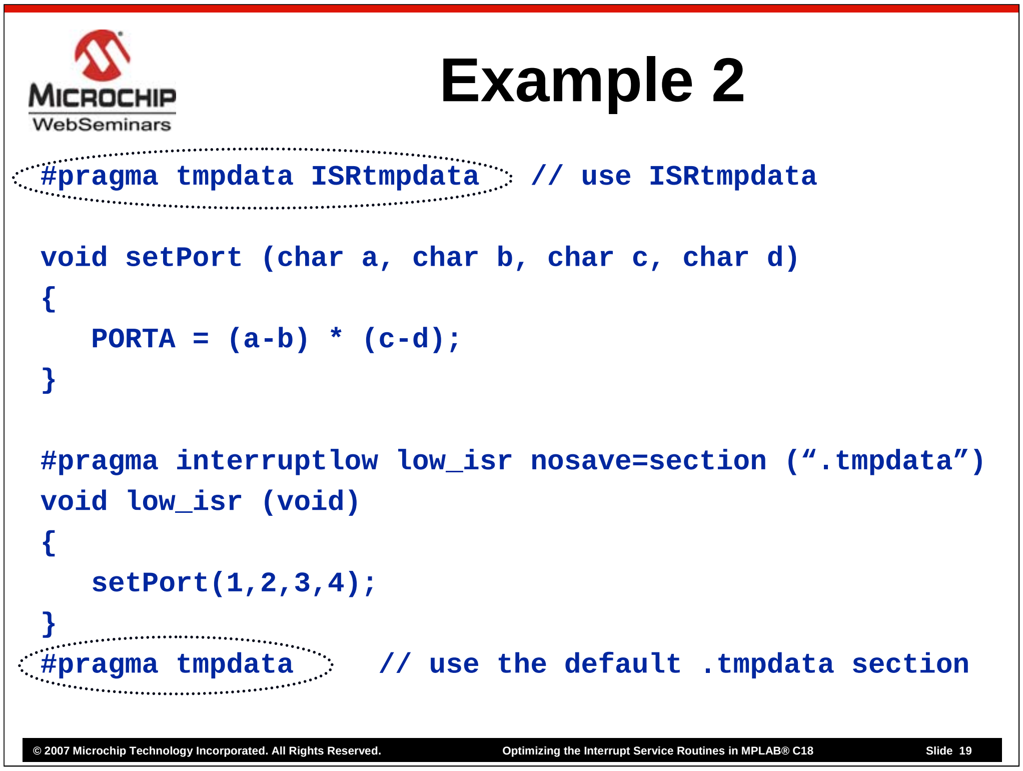
Here, setPort() is, in fact, modifying the .tmpdata section, but, my Interrupt Service routine is not saving it…

As result of this modification, user will experience malfunction elsewhere in the program.

This is a common mistake, and usually difficult to catch, because the symptoms occur in the main thread, while the problem is in the ISR thread.

(pause)

In order to avoid this problem and, also, have the performance that we want in the ISR without need to analyze the assembly code generated by the compiler …



Example 2

```
#pragma tmpdata ISRtmpdata   // use ISRtmpdata

void setPort (char a, char b, char c, char d)
{
   PORTA = (a-b) * (c-d);
}

#pragma interruptlow low_isr nosave=section (“.tmpdata”)
void low_isr (void)
{
   setPort(1,2,3,4);
}
#pragma tmpdata    // use the default .tmpdata section
```

We can direct the compiler to place the temporary data for our Interrupt Service thread in a section, separate from the default .tmpdata section. By doing so, we don’t need to analyze the code to see if the tmpdata section is used or not.

In this example, in the top oval, I am defining a new temporary data section using the #pragma tmpdata and calling it ISRtmpdata.

The effect of this pragma is to make all functions that follow it to use ISRtmpdata section as their temporary data section.

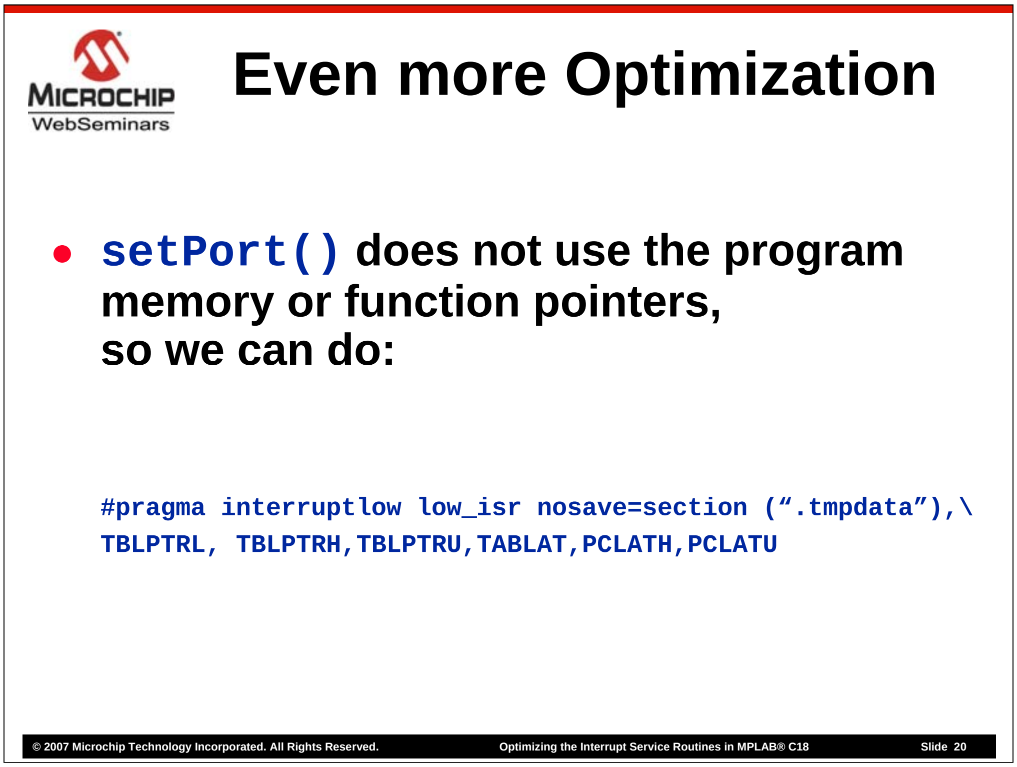
Using a #pragma tmpdata without any name makes the compiler to reset the temporary data section to the default for the functions that follow.

And this is what I am using in the bottom oval in order to return to the default .tmpdata for the rest of my program.

(pause)

The combination of these two #pragma statements takes care of my objective, which is to assign a unique temporary data section to my Interrupt Service Routine thread, and remove the vulnerability of .tmpdata section, so I can safely avoid preserving this potentially large section across the ISR.

But is there any more optimization possible for this ISR?



Even more Optimization

- `setPort()` **does not use the program memory or function pointers,
so we can do:**

```
#pragma interruptlow low_isr nosave=section (“.tmpdata”),\
TBLPTRL, TBLPTRH,TBLPTRU,TABLAT,PCLATH,PCLATU
```

Optimizing the Interrupt Service Routines in MPLAB® C18

Slide 20

As a matter of fact, yes:

Note that neither of the two functions low_isr() and setPort() use rom variables nor they use function pointers. So I can add the register:

“table pointer low”,

“table pointer high”,

“table pointer upper”,

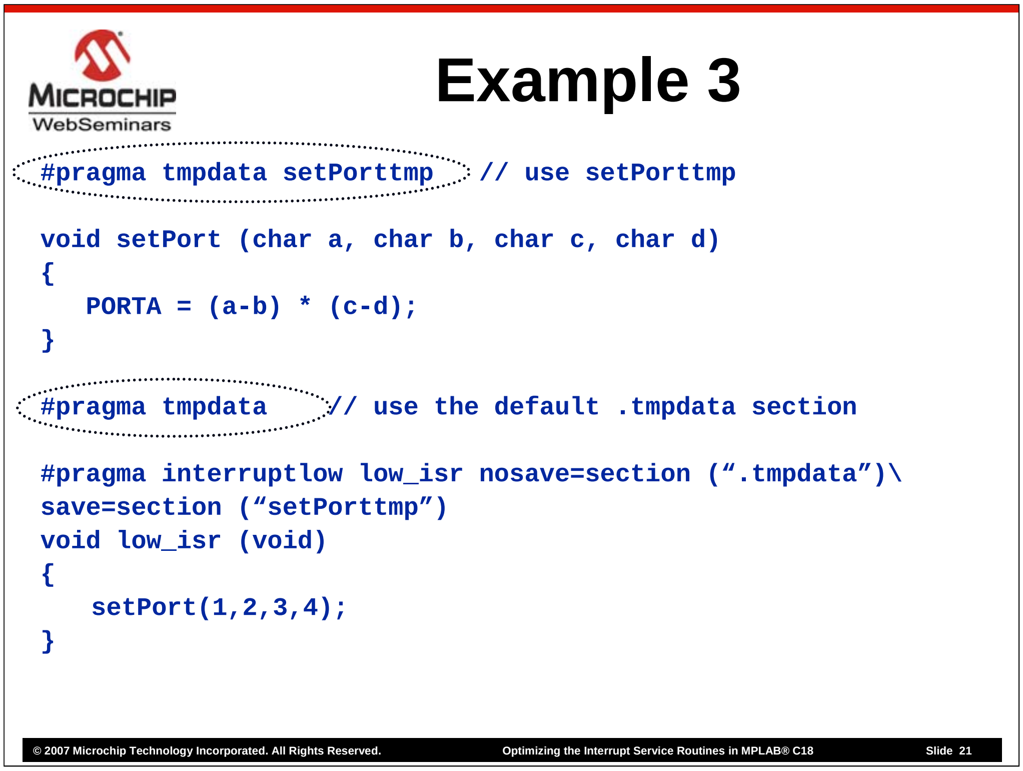
“table latch”,

“pc latch high” and

“pc latch upper”

to the list of nosave items and be confident about the correct execution of my program.

In this example, I am assuming that setPort() function is only called from the ISR. If your main line code also calls the setPort() function, then we can’t share the same temporary data section for the isr_low() and setPort() functions



Example 3

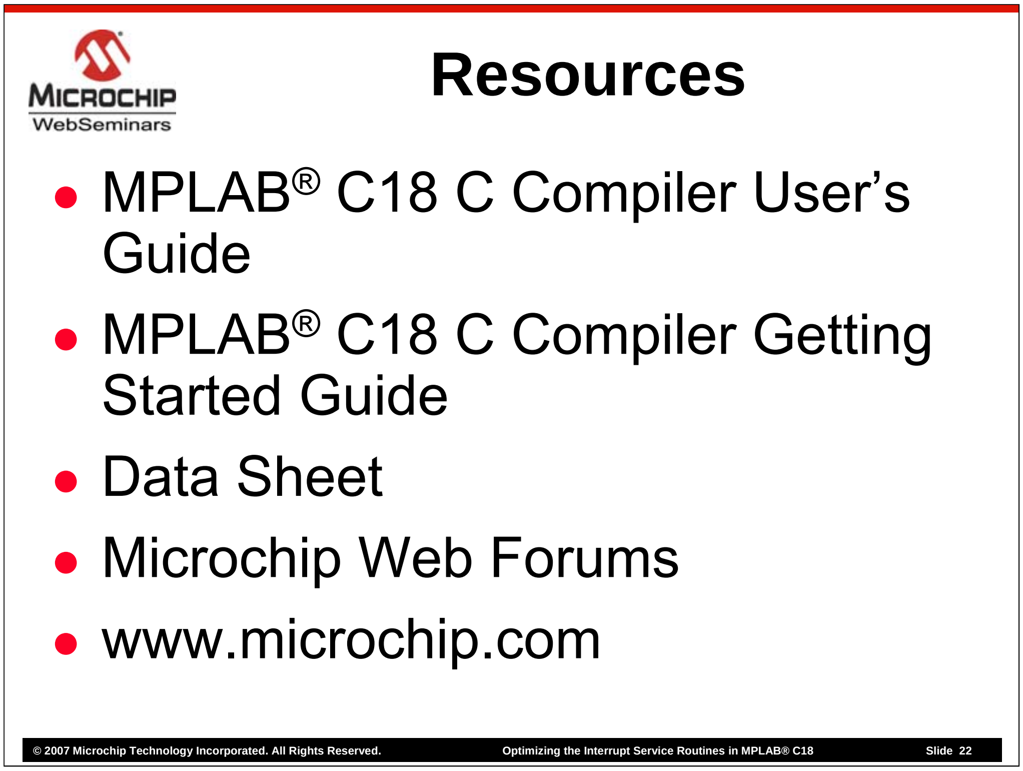
```
#pragma tmpdata setPorttmp  // use setPorttmp

void setPort (char a, char b, char c, char d)
{
   PORTA = (a-b) * (c-d);
}

#pragma tmpdata     // use the default .tmpdata section

#pragma interruptlow low_isr nosave=section (“.tmpdata”)\
save=section (“setPorttmp”)
void low_isr (void)
{
    setPort(1,2,3,4);
}
```

In that case, I must use a separate temporary data section for setPort() function.

Here is the same example, except that setPort() function is called not only from low_isr, but also from the main line code.

What I am doing here is to isolate the temporary data section of setPort() from the default .tmpdata section as well as the temporary data section of my ISR.

Note that I have added the the save keyword in the #pragma interruptlow statement, because the temporary data section for setPort() function must be saved.



Resources

- MPLAB® C18 C Compiler User’s Guide
- MPLAB® C18 C Compiler Getting Started Guide
- Data Sheet
- Microchip Web Forums
- www.microchip.com

Well, this is all I had for this webseminar.

I hope it helps you in achieving the objectives of your projects. Here I am repeating the list of resources that you can refer to in case of any question.

The users guide of C18 provides a detailed description of everything that you need to know.

The Getting started guide gives you a big picture through many examples. There is a section of Frequently asked question in this guide that every one can benefit from.

The data sheets contain comprehensive architectural information about the part that you want to use.

And finally there is a forum for C18 that you can always go and ask your questions from gurus.

You can access all of these from www.microchip.com

Thank you.