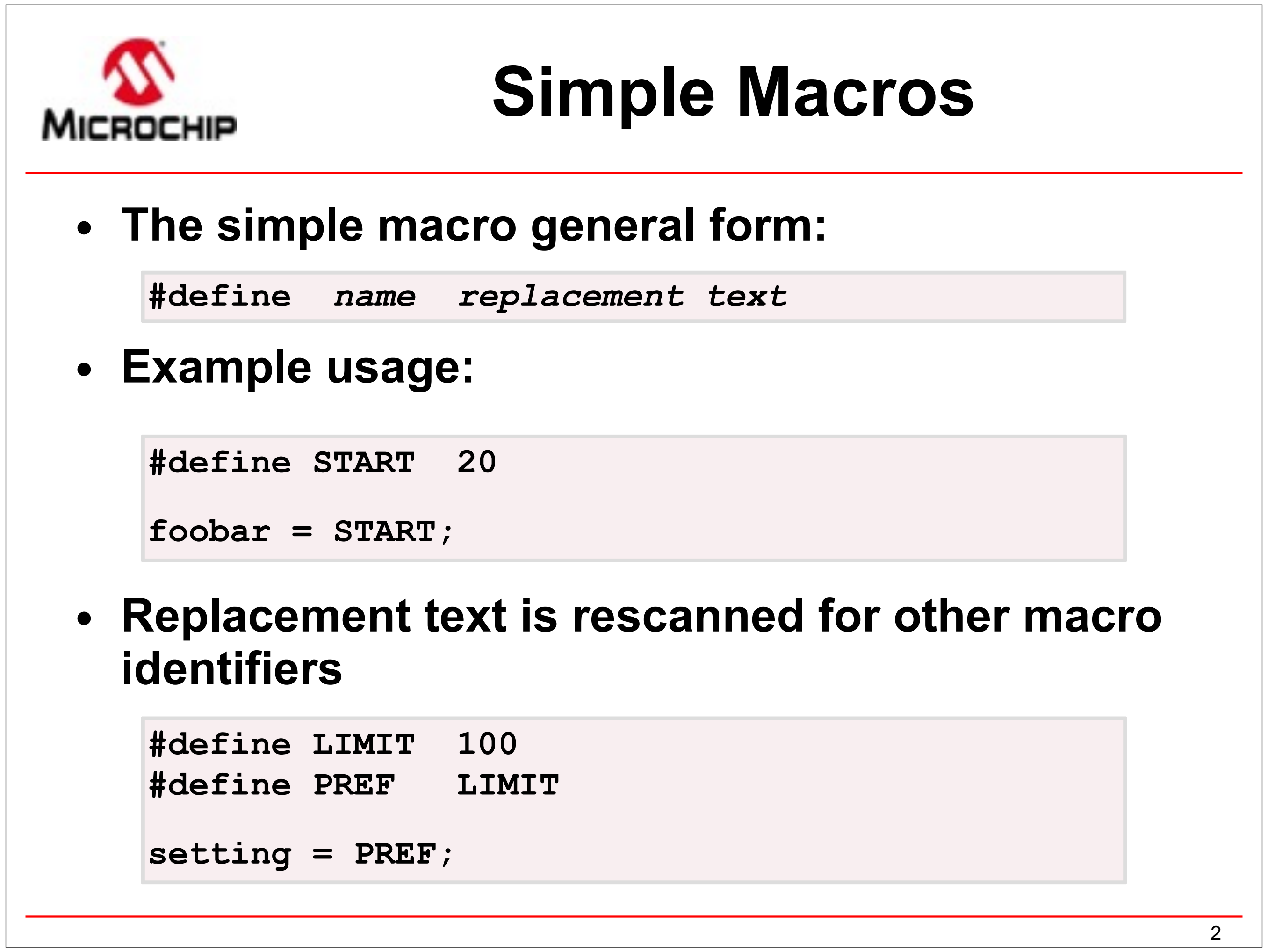
MICROCHIP

Macro Exposure

An in-depth look at common uses and mis-uses of preprocessor macros

This webinar looks at the C preprocessor and its handling of preprocessor macros. We'll take a look at some more advanced examples of macros, and also look at some misconceptions associated with macros and how the preprocessor expands and evaluates them. This information is relevant for any ANSI C compiler.



Simple Macros

- **The simple macro general form:**

```
#define  name  replacement text
```

- **Example usage:**

```
#define START  20

foobar = START;
```

- **Replacement text is rescanned for other macro identifiers**

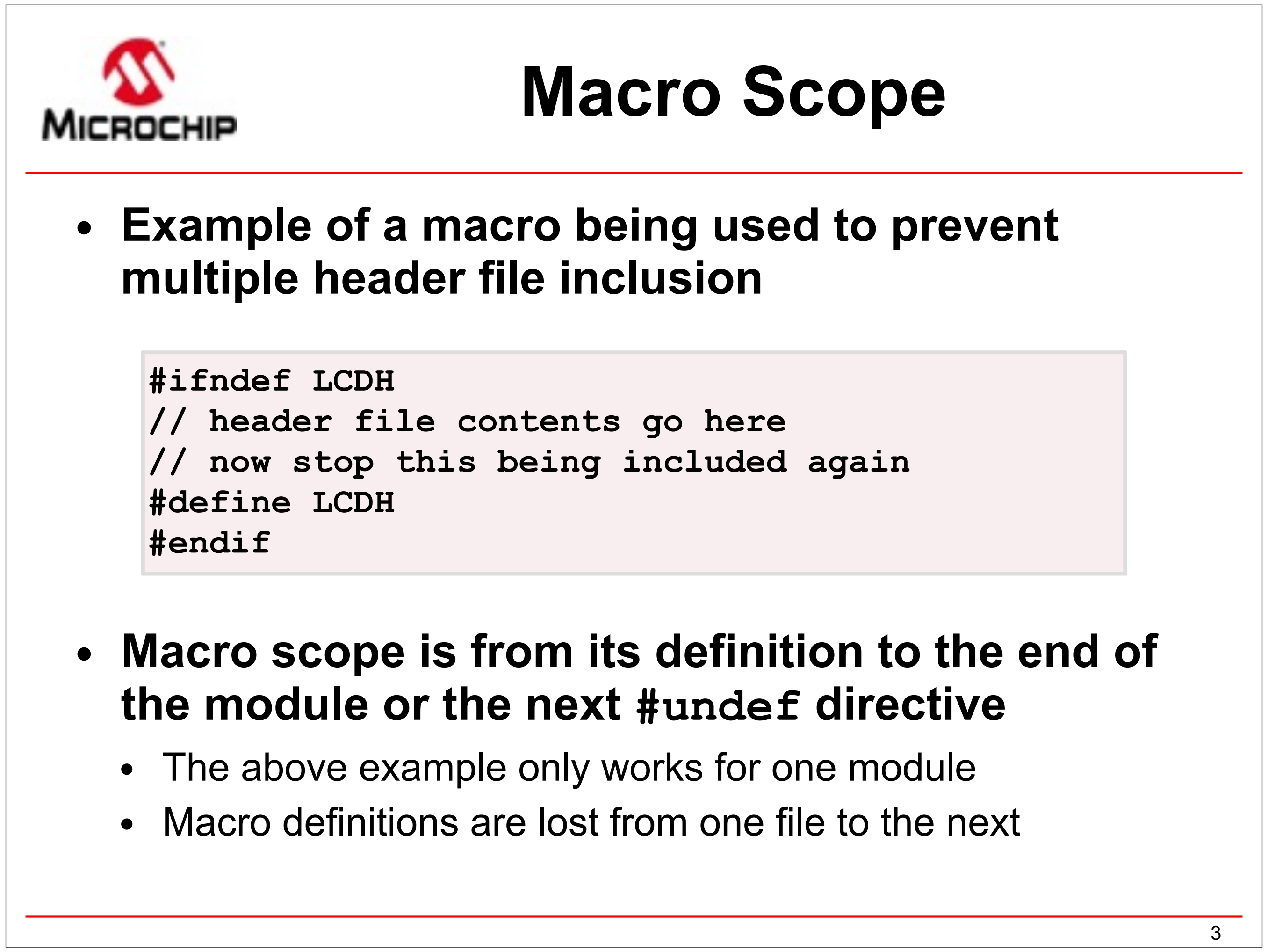
```
#define LIMIT  100
#define PREF   LIMIT

setting = PREF;
```

First, a quick revision of basic macro expansion.

A preprocessor macro definition in its simplest form is shown at the top of this slide. It replaces every instance of the token name with the replacement text. So in the example immediately below, the variable `foobar` is assigned the value 20.

One of the complications of macro expansion is that after substitution, the replacement text is rescanned to see if it forms other preprocessor macro identifiers. This expansion process continues until no more macro identifiers are found.

In the last example on this slide, the variable `setting` is assigned the value 100.



Macro Scope

- **Example of a macro being used to prevent multiple header file inclusion**

```
#ifndef LCDH
// header file contents go here
// now stop this being included again
#define LCDH
#endif
```

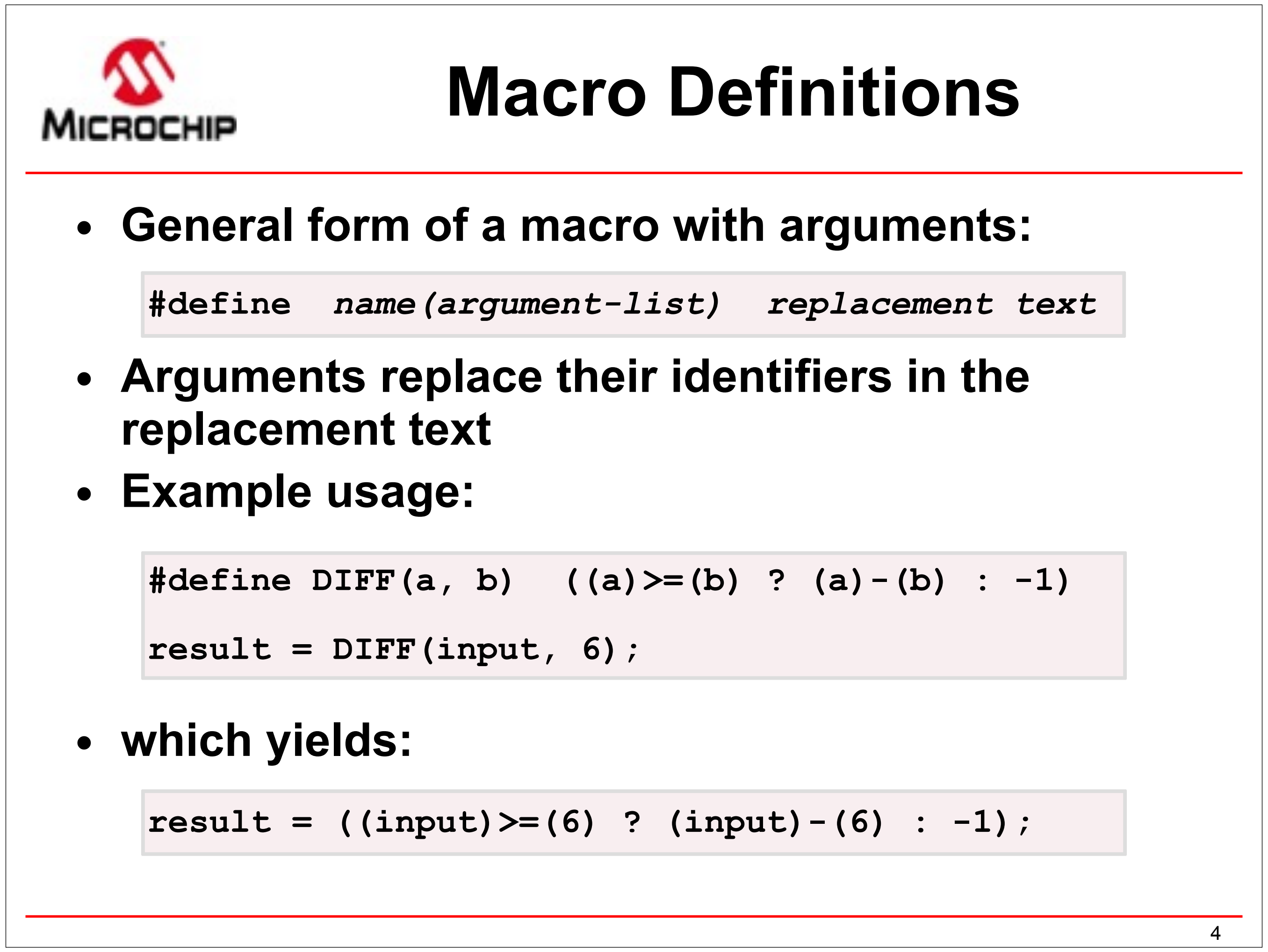
- **Macro scope is from its definition to the end of the module or the next `#undef` directive**
  - The above example only works for one module
  - Macro definitions are lost from one file to the next

A common use for a plain macro is to prevent the contents of header files being compiled more than once, as shown in the header file stub in the example box. Only if a particular macro symbol, LCDH in this example, has not already been defined will the contents of the header file be included. The header file contents are enclosed in `#ifndef` and `#endif` directives. The macro symbol which controls this inclusion is defined along with the header file contents, thus preventing further inclusion of the same code. The macro `LCDH` has no replacement text in this example, but it is still considered as being defined by the preprocessor.

Most compiler-supplied header files use such a mechanism, and this is useful for your header files as well. However, there is a common misconception that this mechanism prevents the header file contents from being compiled more than once, regardless of where the header file was included. The reality is that this only prevents multiple inclusion in the one module, not in other modules. To understand this, you have to consider a macro's scope. And if you are not sure of a the difference between a source file and and a module, you might want to consult your favourite C text.

Just like with C identifiers, preprocessor macros have only a limited scope, which is the part of your code where they are actually defined. A macro's scope simply starts where the macro is defined and ends at the end of that module, unless a `#undef` directive is encountered first. The preprocessor works on each source file independently and macros defined in one file are forgotten once it moves to the next.

If you were to place variables definitions inside a header file, even those that used this technique, and included it into more than one source file, you may find yourself receiving multiply defined symbol errors from your compiler.



Macro Definitions

- General form of a macro with arguments:

```
#define  name(argument-list)  replacement text
```

- Arguments replace their identifiers in the replacement text
- Example usage:

```
#define DIFF(a, b)  ((a)>=(b) ? (a)-(b) : -1)

result = DIFF(input, 6);
```

- which yields:

```
result = ((input)>=(6) ? (input)-(6) : -1);
```

Preprocessor macros can also be defined to have one or more arguments. The general form of such a macro is shown here in the top example.

Such macros allow you to use the arguments in the replacement text and perform more complex substitution. Whenever the argument's identifier is encountered in the replacement text, it is replaced with the corresponding argument specified when the macro was invoked.

The example shows a macro, DIFF, which has two arguments, called a and b, and which evaluates the difference between these unless that would be negative, in which case it evaluates as negative 1. To prevent any nasty side effects in the expansion, each argument in the replacement text is enclosed in brackets. This is highly recommended at all times.

The lower box shows the result after preprocessing. In this case, input is a C variable that will be evaluated in the statement.

Arguments as Strings

- The # operator turns arguments into strings

```
#define PRODUCT(year)   "Matrix" #year

const char * productName = PRODUCT(2000);
```

- This will yield:

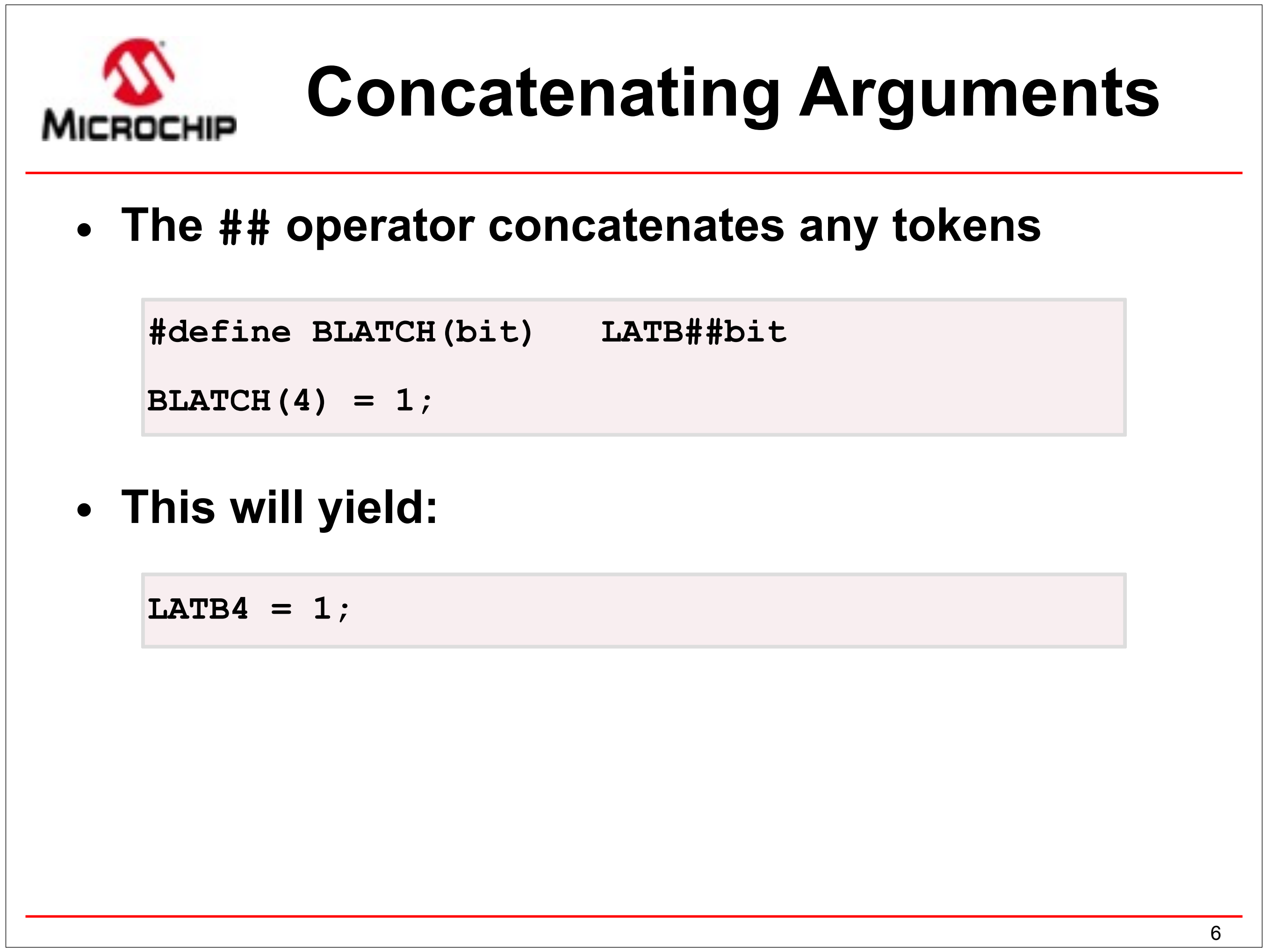
```
const char * productName = "Matrix" "2000";
```

which will ultimately become:

```
const char * productName = "Matrix2000";
```

There are some special operators you can use when a macro has arguments. One is the # operator, which turns arguments into C strings.

In the example in the top box, the variable productName will point to a string "Matrix2000" after preprocessing. During expansion, double quotes are placed around the #year tokens, then these are replaced with the argument value, in this case 2000. The usual C string concatenation joins the string "Matrix" with the expanded year string.



Concatenating Arguments

- The `##` operator concatenates any tokens

```
#define BLATCH(bit)   LATB##bit

BLATCH(4) = 1;
```

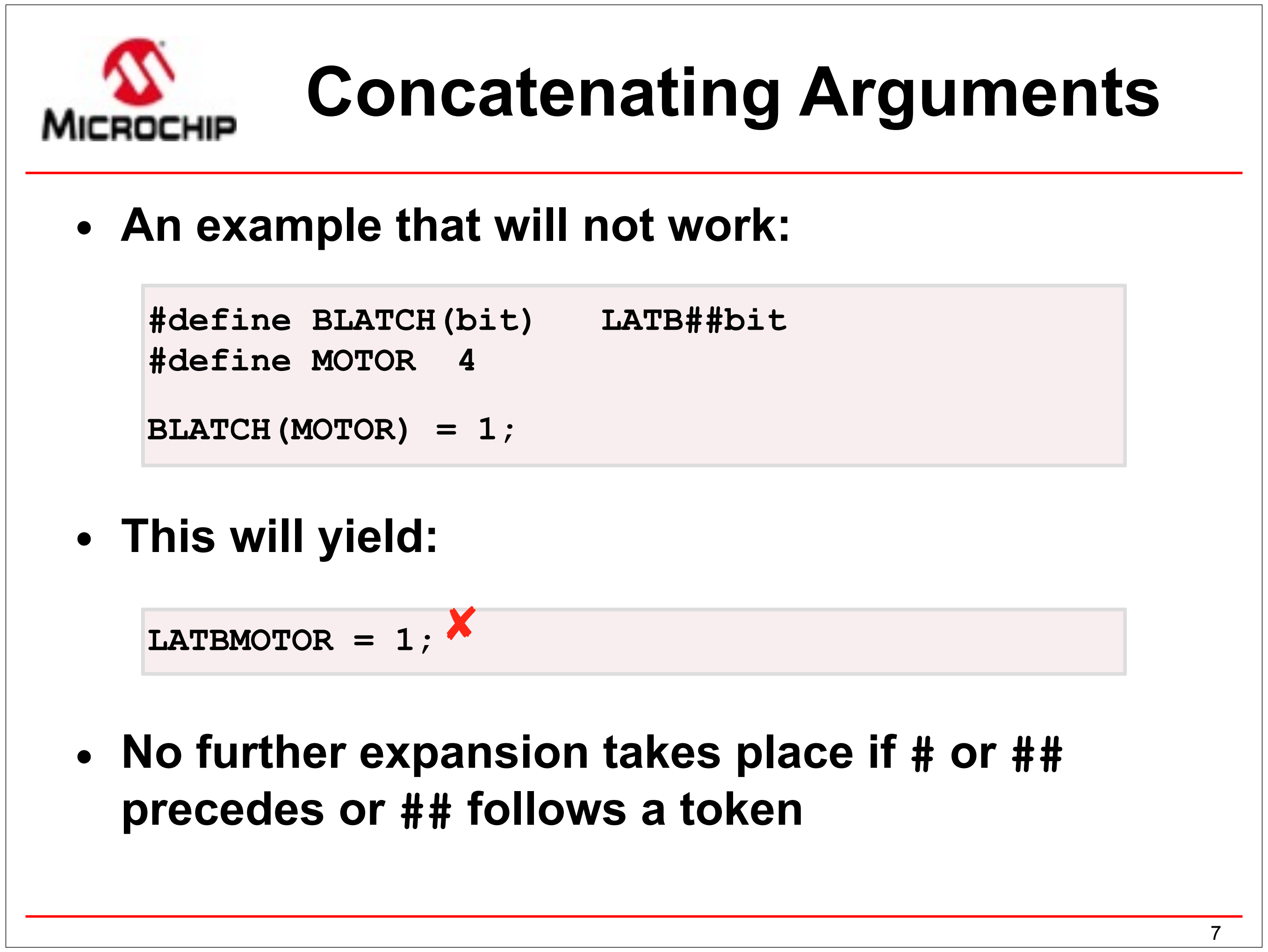
- This will yield:

```
LATB4 = 1;
```

The other macro operator is a double `#`, `##`, which allows you to concatenate any tokens, not just strings.

In the example shown in the top box, this operator is used to joint the token LATB with the bit argument token. When invoked with the argument 4, it expands to the single token LATB4, which is the register bit variable that corresponds to bit 4 in latch B on some PIC devices.

This is very useful, but there is a potential problem if the arguments themselves are other preprocessor macros.



Concatenating Arguments

- An example that will not work:

```
#define BLATCH(bit)    LATB##bit
#define MOTOR   4

BLATCH(MOTOR) = 1;
```

- This will yield:

```
LATBMOTOR = 1; ✘
```

- No further expansion takes place if # or ## precedes or ## follows a token

Let's say that the programmer has adjusted their code in the previous example to make it a little more readable. A new macro has been added that defines MOTOR to be the port pin 4. However, you can see in the lower example box, the expansion of this macro did not take place when it was used and we are left with a non-existent identifier, LATBMOTOR.

The problem here is one of ordering: macro arguments are not expanded prior to being inserted into the replacement text. A preprocessor rule states that tokens in the replacement text of a macro will only be replaced themselves if they are not preceded by a `#` or `##` operator, or followed by a `##`. So in our example, after MOTOR is inserted into the replacement text for BLATCH, it will not be further expanded since it is next to the `##` operator.

Fortunately we can find a way that allows us to do what we want.

Concatenating Arguments

- **A two-step process can fix this:**

```
#define PASTE(a, b)    a##b
#define BLATCH(bit)    PASTE(LATB, bit)

#define MOTOR   4

BLATCH(MOTOR) = 1;
```

- **This will yield:**

```
LATB4 = 1;
```

- **`BLATCH()` performs the argument expansion; `PASTE()` performs the concatenation**

So that the expansion of MOTOR is not impeded, we use two macros as shown: one that takes care of the argument substitution and subsequent expansion; and another which takes care of the concatenation. This two-step process means we can concatenate arguments, and the arguments can be macros themselves.

Nesting Replacement Text

- **Nice try!**

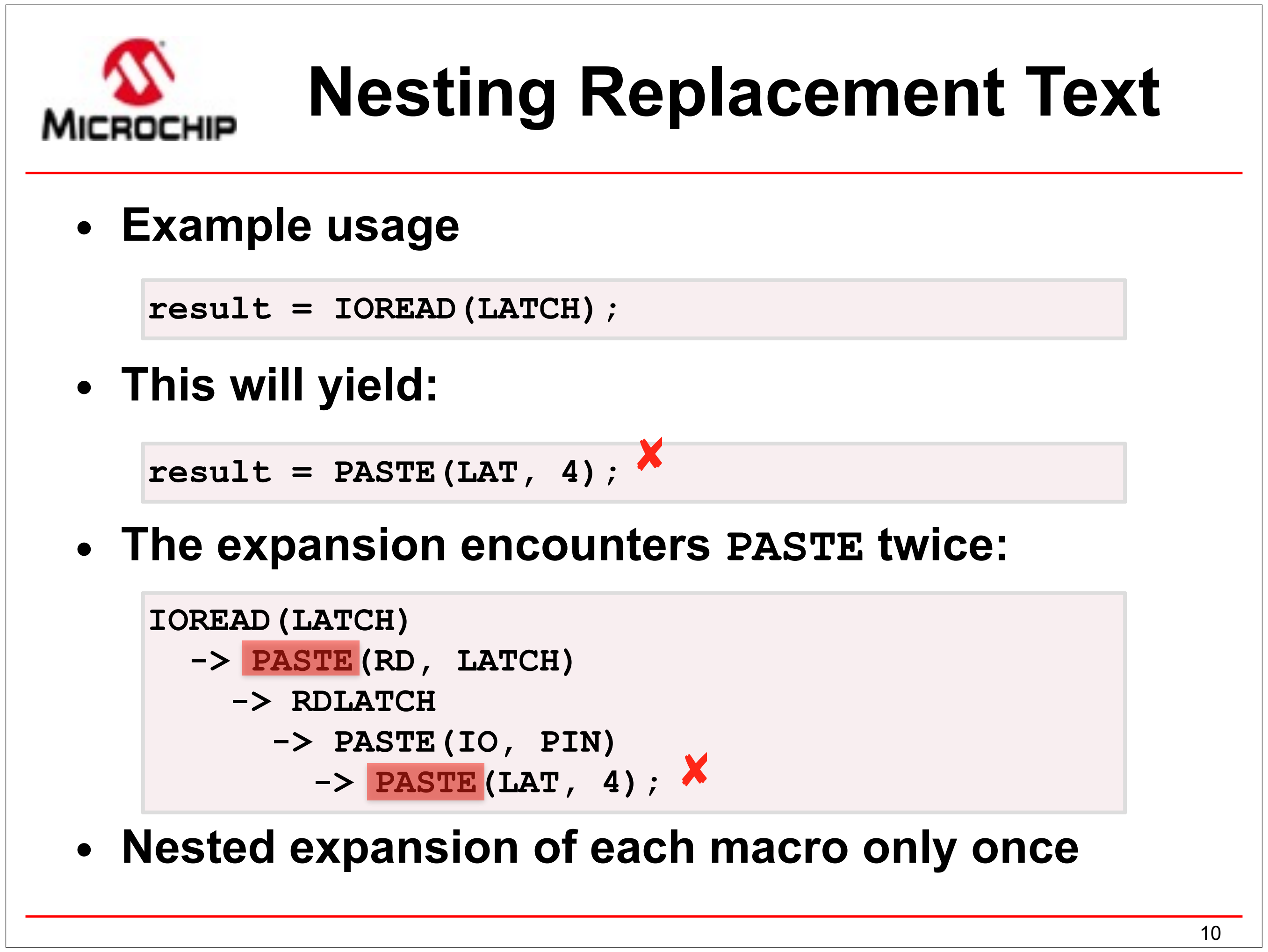
```
#define PASTE2(a,b)    a##b
#define PASTE(a,b)     PASTE2(a,b)

#ifdef  PIC18  // set for PIC18 devices
#define IO             LAT
#else
#define IO             PORT
#endif

#define PIN            4

#define RDPORT         PASTE(PORT, PIN)
#define RDLATCH        PASTE(IO, PIN)

#define IOREAD(loc)    PASTE(RD, loc)
```

By now, you might be ready to conquer the programming world and become the macro concatenation king, but before you set out on this endeavour, here's a somewhat contrived example that might look like it will work okay, but will only generate a compiler error for our efforts.

The first two lines of code define our concatenating macros we used in the previous slide. The two-step concatenation method has been used so this would appear to handle any expansion issues correctly. Reading down, we define IO to be LAT for PIC18 devices, or PORT otherwise; and we define a PIN macro to be 4.

The last line is the macro, IOREAD, that we are ultimately trying to create. It takes one argument which can be either be LATCH or PORT and this macro will prefix the letters RD to the argument. Above this definition, the two remaining lines of code define macros: RDPORT, which concatenates PORT and the PIN definition, and the macro RDLATCH, which will either concatenate LAT or PORT to the PIN definition, based on the selected device.

You might want to stop this presentation to ensure you see how everything is defined.



Nesting Replacement Text

- **Example usage**

```
result = IOREAD(LATCH);
```

- **This will yield:**

```
result = PASTE(LAT, 4); ✘
```

- **The expansion encounters `PASTE` twice:**

```
IOREAD(LATCH)
  -> PASTE(RD, LATCH)
    -> RDLATCH
      -> PASTE(IO, PIN)
        -> PASTE(LAT, 4); ✘
```

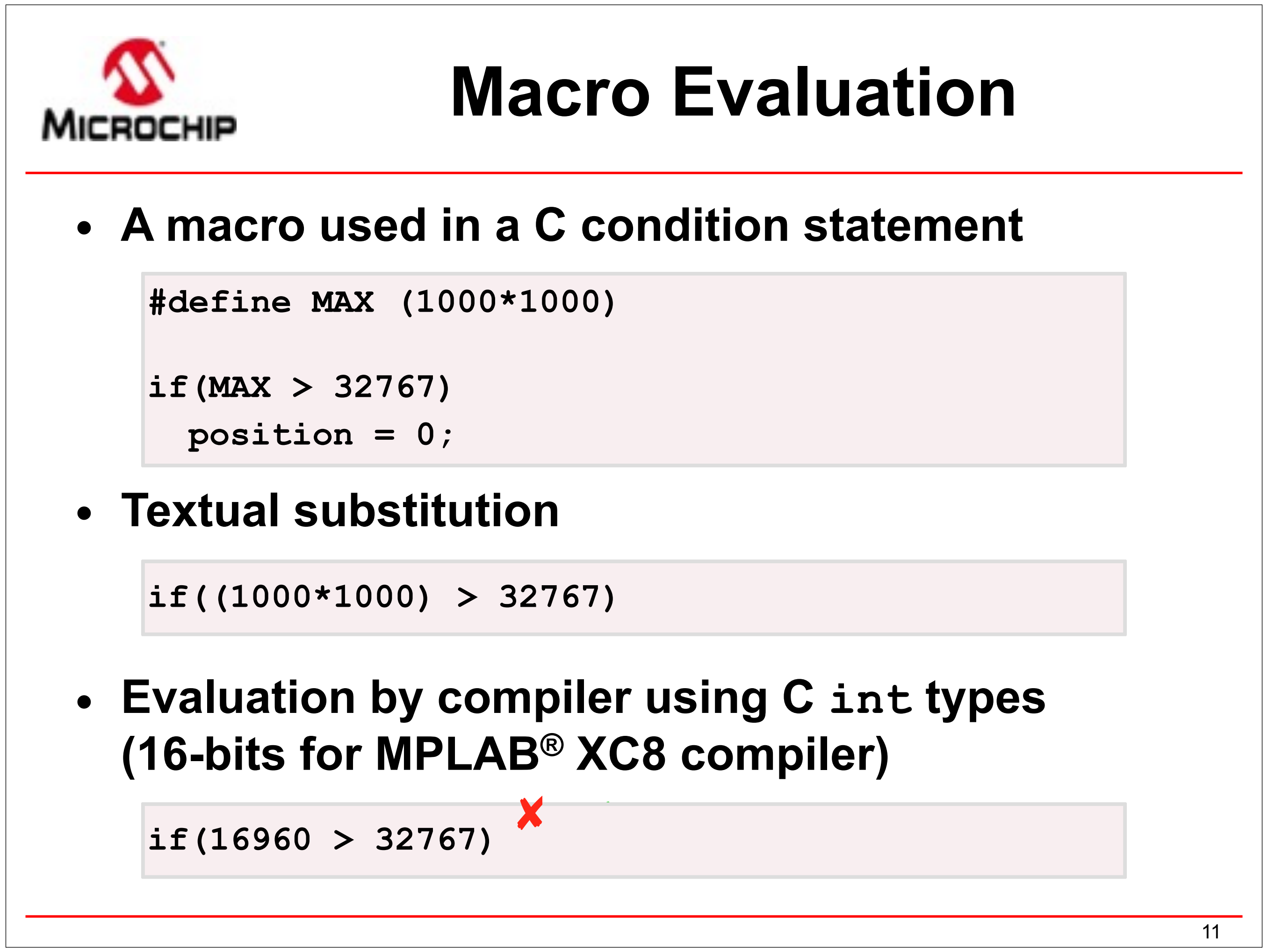
- **Nested expansion of each macro only once**

So, if we were to use this macro in code, as shown in the top box, we might expect it to read from pin 4 of either the latch or port, but instead, if you check the preprocessed output or assembly list file, you will notice that, as shown in the next code box, our PASTE macro for some reason did not get expanded and the compiler unsuccessfully looks for a C function with that name.

So, what went wrong? The problem is the last three lines of the example code shown on the previous slide. These violate another preprocessor rule, which states that if any nested replacements encounter the name of a macro currently being expanded, it is not considered for further replacement.

In our case here, the expansion of IOREAD involves expanding the macro PASTE, that forms RDLATCH, but the expansion of this macro requires expanding PASTE for a second time. The second expansion is not performed due to this rule, and so we see it remain unexpanded in the final output.

We could work around this issue by creating an identical version of PASTE, but which has a different name. We could use PASTE in the IOREAD definition, and the new paste macro in the RDPORT and RDLATCH definitions.



Macro Evaluation

- **A macro used in a C condition statement**

```
#define MAX (1000*1000)

if(MAX > 32767)
  position = 0;
```

- **Textual substitution**

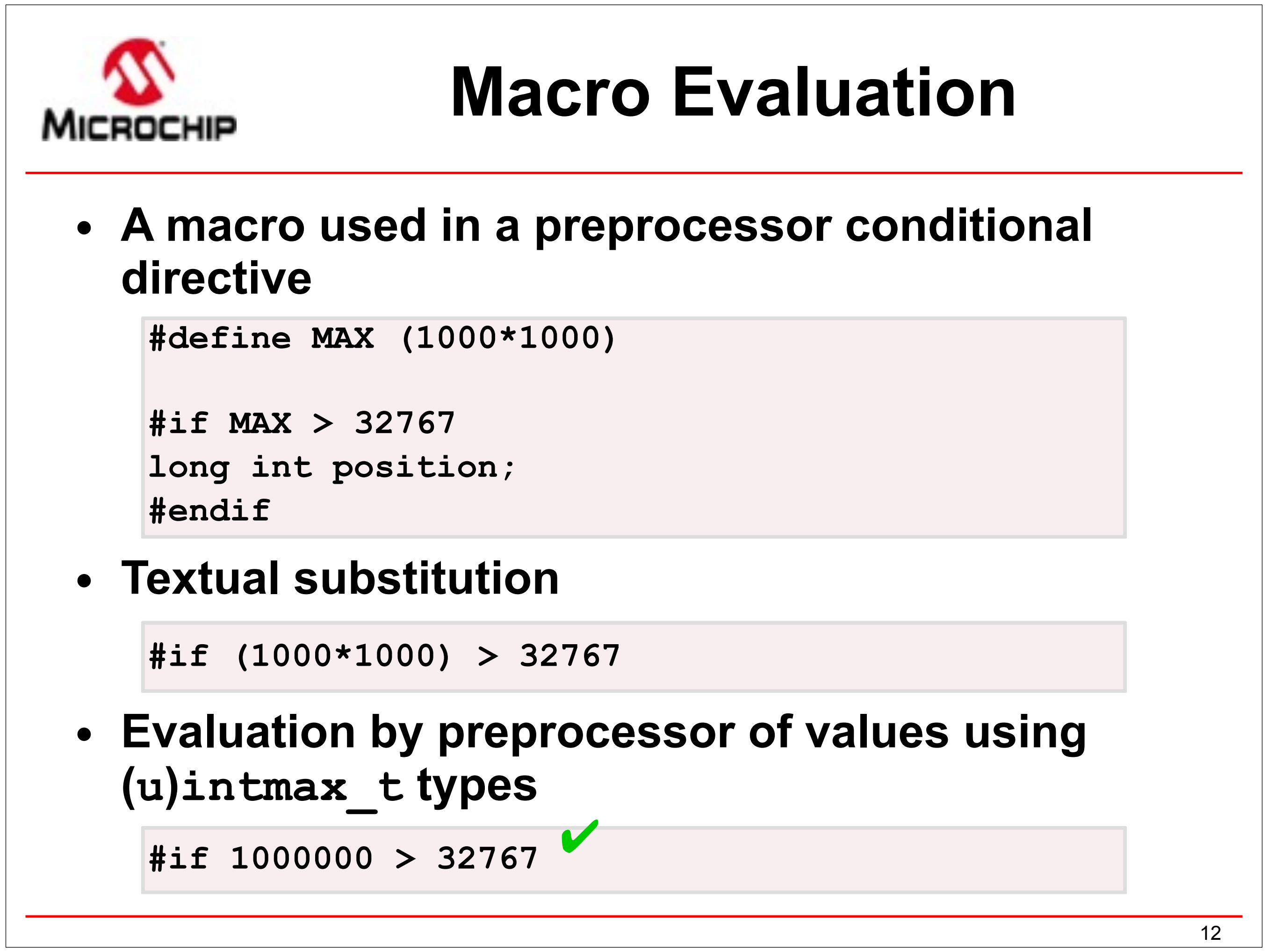
```
if((1000*1000) > 32767)
```

- **Evaluation by compiler using C `int` types (16-bits for MPLAB® XC8 compiler)**

```
if(16960 > 32767)
```
✗

When the preprocessor expands a macro, the replacement is purely a string of characters. So in this example, `MAX`, in the `if()` statement's controlling expression, is replaced with the characters: open bracket, one, zero, zero, zero, star, one, zero, zero, zero, close bracket. The preprocessor does not *evaluate* this expression, nor is there any concept of types at this point; this is just a string. Later in the compilation process, the compiler will see the expanded expression, and only then will the characters be tokenised, associated with numerical values and operators, and take on C types, as dictated by the usual C language rules.

In this example, the *compiler* will evaluate the controlling expression. The constant 1000 will have an `int` type. If this code was being compiled using the MPLAB® XC8 compiler, which has an int size of 16 bits, the multiplication will also be performed as a 16-bit operation. The multiplication will overflow, and produce the result 16960, which is *less than* 32767. This conditional controlling expression in this instance is false.



Macro Evaluation

- **A macro used in a preprocessor conditional directive**

```
#define MAX (1000*1000)

#if MAX > 32767
long int position;
#endif
```

- **Textual substitution**

```
#if (1000*1000) > 32767
```

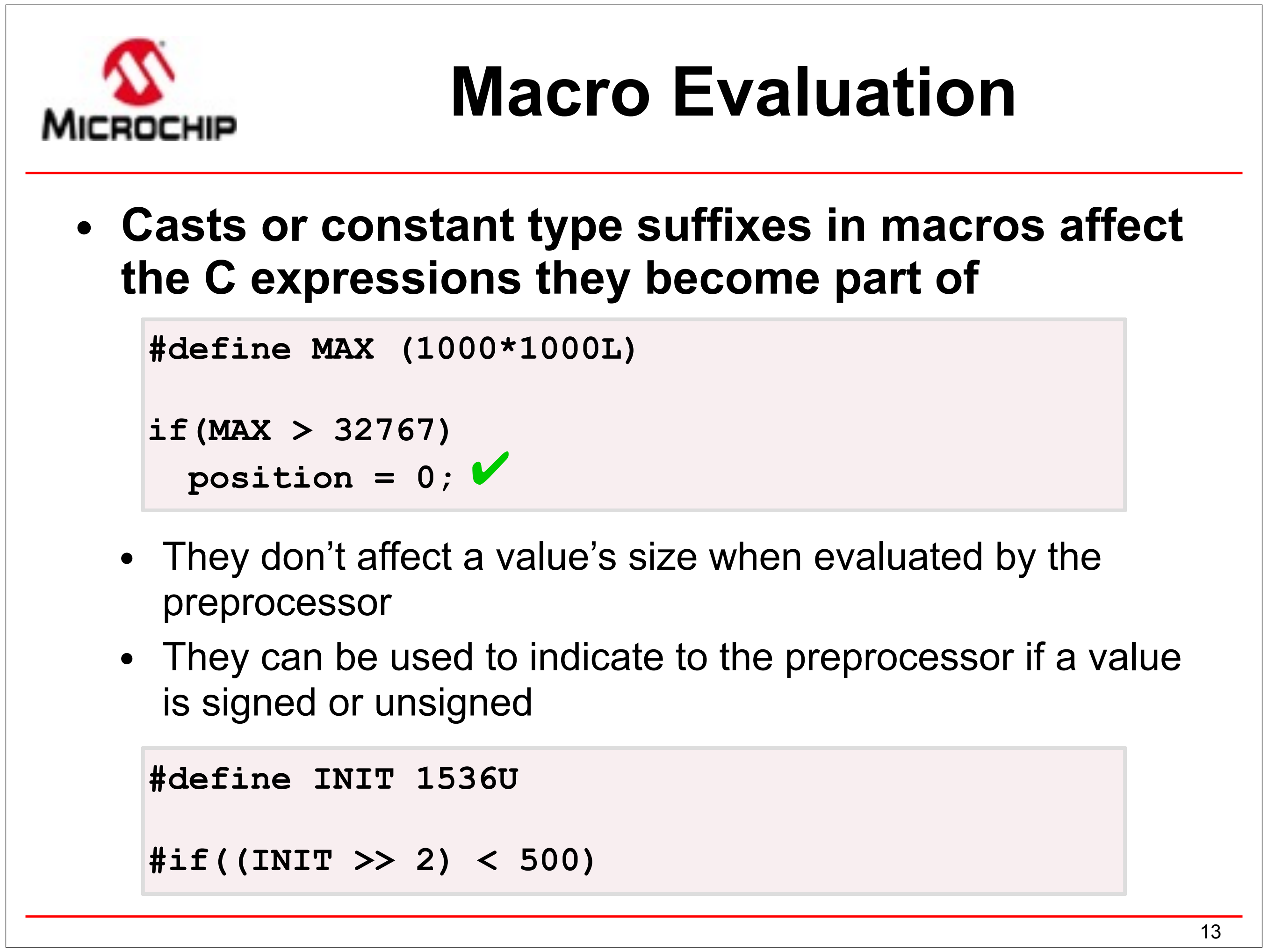
- **Evaluation by preprocessor of values using (u)`intmax_t` types**

```
#if 1000000 > 32767
```
✔

But there is one instance when the *preprocessor* has to expand *and* evaluate the replacement text of a macro, and that is when it is used in the controlling expression of a `#if` or `#elif` conditional directive. The following example shows the macro `MAX` now being used in a `#if` directive to determine if the C variable `position` should be defined in the code. The controlling expression appears identical to that in the previous example, where it was used in an `if()` statement. After substitution, the preprocessor has to determine if the expression following the `#if` is true or false. The only way it can do this is to *evaluate* the expression.

The preprocessor's evaluation of the constant tokens in this controlling expression is similar to that performed by the compiler, except that signed or unsigned integer types act as if they have the type `intmax_t` or `uintmax_t`, respectively, as defined by the header `<stdint.h>`. For the MPLAB® XC8 compiler, this will be a *32-bit* representation. This means that the preprocessor multiplication of 1000 by itself will be performed as a 32-bit operation. The result will be 1 million, which is *larger* than 32767. This conditional controlling expression in this case is true.

The important thing to note from this is that the preprocessor can calculate a different value to that calculated by the compiler for the same expression. MPLAB® XC16's and XC32's `int` size and the size used by the preprocessor for literal constants are different to those used by MPLAB® XC8, but these compilers exhibit a similar behaviour.



Macro Evaluation

- **Casts or constant type suffixes in macros affect the C expressions they become part of**

```
#define MAX (1000*1000L)

if(MAX > 32767)
  position = 0; ✔
```

- They don't affect a value's size when evaluated by the preprocessor
- They can be used to indicate to the preprocessor if a value is signed or unsigned

```
#define INIT 1536U

#if((INIT >> 2) < 500)
```

If you need to change the types that are used by the *compiler* when evaluating a macro, you can use any appropriate feature in the C language, for example, by adding a constant type suffix. Use of the `L` suffix in the first example would not affect the size of the values used by the preprocessor, if it were to evaluate this expression, but it does change the types used by the compiler. The preprocessor always uses the same size to evaluate expressions, but you can force expressions to be treated as signed or unsigned. The last example shows code using the `U` type suffix, which will require the preprocessor to evaluate an unsigned right shift.

--

Who'd have thought macro expansion could get this complicated! Hopefully this presentation has shown you that some caution is needed when using macros, particularly when concatenating macro arguments and when macros are evaluated by the preprocessor.

Although the preprocessor is rarely considered when writing code, it conforms to its own set of rules. But it's these rules that make preprocessor macros a consistent tool that can greatly assist you writing flexible code.