Special Operators for Macros

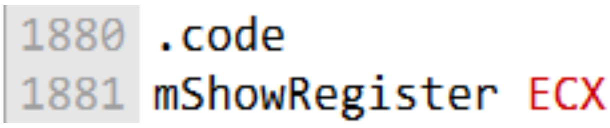
In assembly language macros, there are four special operators that enhance their flexibility and usability:

Substitution Operator (&):

The substitution operator (&) is a valuable tool in macros. It helps resolve ambiguous references to parameter names within a macro.

For example, consider the mShowRegister macro, which displays the name and hexadecimal contents of a 32-bit register.

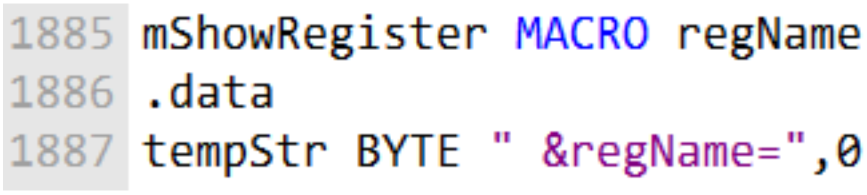
When calling this macro with a register name, like ECX, the macro produces output that includes the register name and its value.



```
.code
mShowRegister ECX
```

Creating a String Variable: If you want to create a string variable inside a macro that includes the register name, using just regName within the string won't work as expected. The preprocessor might treat it as a regular string and not replace it with the argument passed to the macro.

Using the Substitution Operator (&): To force the preprocessor to insert the macro argument (e.g., ECX) into the string literal, you can use the substitution operator &. This operator ensures that the macro argument is correctly incorporated into the string. Here's an example of how you can define the tempStr variable with the & operator:



```
mShowRegister MACRO regName
.data
tempStr BYTE " &regName=",0
```

In summary, the substitution operator & is a powerful tool for resolving parameter references within macros, making them more versatile and efficient in handling various inputs.

Expansion Operator (%) in Macros

In assembly language, the expansion operator (%) plays a vital role in macros. It can be used in several ways to evaluate expressions and expand text macros:

Evaluating Expressions: When used with TEXTEQU, the % operator evaluates a constant expression and converts the result to an integer.

For example, if you have a variable count = 10, you can use % to calculate (5 + count) and get the integer result, which is then represented as text:

```
count = 10
sumVal TEXTEQU %(5 + count) ; This results in "15"
```

Flexibility for Passing Arguments: The % operator offers flexibility in passing arguments to macros. If a macro expects a constant integer argument, you can use the % operator to pass an integer expression. For example:

```
mGotoxyConst %(5 * 10), %(3 + 4)
```

In this case, the expressions within %(...) are evaluated to their integer values, which are then passed to the macro.

Expanding Macros on a Line: When the expansion operator (%) is the first character on a source code line, it instructs the preprocessor to expand all text macros and macro functions found on that line.

This can be useful for creating dynamic text during assembly. For example, to display the size of an array, you can use TEXTEQU to create a text macro, and then expand it on the next line:

```
TempStr TEXTEQU %(SIZEOF array)
%
ECHO The array contains TempStr bytes
```

This approach allows for dynamic text generation during assembly.

Displaying Line Numbers: In some cases, macros can display the line number from which they were called to help with debugging.

For instance, the LINENUM text macro references **@LINE**, a predefined assembler operator that returns the current source code line number.

When an error condition is detected in the macro, the **line number can be displayed in an error message,** making it easier to identify and fix issues in the source code.

In summary, the expansion operator (%) is a versatile tool in macros, enabling the evaluation of expressions, dynamic text generation, and enhanced debugging by displaying line numbers in error messages.

Literal-Text Operator (< >)

The literal-text operator (< >) is a tool that allows you to group characters and symbols into a single text literal.

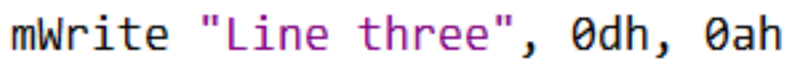
Its main purpose is to prevent the preprocessor from treating these characters as separate arguments or operators.

This is particularly useful when you have a string that contains special characters like commas, percent signs, ampersands, or semicolons.

These special characters could otherwise be misinterpreted by the preprocessor.

For example, consider the mWrite macro, which expects a string literal as its argument.

If you pass it the following string without using the literal-text operator:



```
mWrite "Line three", 0dh, 0ah
```

The preprocessor would consider this as three separate arguments. In this case, text after the first comma would be discarded because the macro expects only one argument. To prevent this, you can surround the string with the literal-text operator:

```
mWrite <"Line three", 0dh, 0ah>
```

By doing this, the preprocessor treats all text enclosed within the brackets as a single macro argument.

Literal-Character Operator (!)

The literal-character operator (!) serves a similar purpose to the literal-text operator.

It's used to instruct the preprocessor to treat a predefined operator as a regular character.

This is useful when you need to include special characters within a text literal without them being misinterpreted by the preprocessor.

For example, consider the definition of the BadYValue symbol:

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Here, the ! operator is used to prevent the > symbol from being treated as a text delimiter. It ensures that the entire text within the < > brackets is preserved as a single text literal.

Example: Using %, &, and ! Operators

To illustrate these operators, let's consider an example. Suppose you have a symbol called BadYValue, and you want to create a macro called ShowWarning.

This macro takes a text argument, encloses it in quotes, and then passes it to the mWrite macro. You can achieve this using the substitution operator (&) as follows:

```
ShowWarning MACRO message
mWrite "&message"
ENDM
```

Now, when you invoke ShowWarning and pass it the expression %BadYValue, the % operator evaluates (dereferences) BadYValue, turning it into its string representation. The program then displays the warning message correctly:

```
ShowWarning %BadYValue
```

As a result, the program runs and displays the warning message as intended: "Warning: Y-coordinate is > 24."

In summary, the literal-text operator and the literal-character operator are tools to control how the preprocessor interprets and handles special characters within your assembly code, allowing you to maintain the desired structure and functionality of your macros and text literals.

Macro Functions

A macro function is similar to a regular macro procedure, but with a key difference: it always returns a constant value, either an integer or a string, using the EXITM directive. Let's look at an example to understand this better:

```
IsDefined MACRO symbol
    IFDEF symbol
        EXITM <-1> ;; True
    ELSE
        EXITM <0> ;; False
    ENDIF
ENDM
```

In this example, the IsDefined macro function checks whether a given symbol has been defined. If the symbol is defined, it returns true (represented by -1); otherwise, it returns false (0).

Invoking a Macro Function:

When you want to use a macro function, you need to enclose its argument list in parentheses.

For instance, let's call the IsDefined macro and pass it the symbol RealMode, which may or may not have been defined:

```
IF IsDefined(RealMode)
    mov ax, @data
    mov ds, ax
ENDIF
```

If the assembler has already encountered a definition of RealMode before this point in the assembly process, it will assemble the two instructions as shown.

You can also use the macro function within other macros, like this Startup macro:

```
Startup MACRO
    IF IsDefined(RealMode)
        mov ax, @data
        mov ds, ax
    ENDIF
ENDM
```

The IsDefined macro can be valuable when you're designing programs for different memory models. For example, you can use it to determine which include file to use based on whether Real Mode is defined.

Defining the RealMode Symbol

To use the IsDefined macro effectively, you need to define the RealMode symbol appropriately. There are a couple of ways to do this. One is to add the following line at the beginning of your program:

```
RealMode = 1
```

Alternatively, you can define symbols using the assembler's command-line options. For example, this command defines the RealMode symbol with a value of 1:

```
ML -c -DRealMode=1 myProg.asm
```

This allows you to control whether the RealMode symbol is defined or not when assembling your program.

HelloNew Program:

The HelloNew.asm program provided is an example that demonstrates the usage of the macros described.

It displays a message on the screen and chooses the appropriate include file based on whether RealMode is defined.

The program is adaptable to both 16-bit Real Mode and 32-bit Protected Mode.

In summary, macro functions return constant values, either integers or strings, and are useful for conditional assembly based on the existence of defined symbols, making your assembly code more versatile and flexible.

Questions

What is the purpose of the IFB directive?
The purpose of the IFB directive is to check if a macro argument is blank (empty). It returns true if the argument is empty and false if it contains any content. It's often used to handle cases where a macro expects specific arguments and needs to respond differently when arguments are missing.

What is the purpose of the IFIDN directive?

The purpose of the IFIDN directive is to perform a case-sensitive match between two symbols (or macro parameter names) and determine if they are equal. It returns true if the symbols are the same and false if they are different. This directive is helpful for ensuring that certain conditions are met within a macro.

Which directive stops all further expansion of a macro?

The directive that stops all further expansion of a macro is the EXITM directive. When EXITM is encountered within a macro, it halts the macro's execution, preventing any more macro expansion or code generation.

How is IFIDNI different from IFIDN?

The key difference between IFIDNI and IFIDN is their case sensitivity. IFIDNI performs a case-insensitive match, meaning it treats symbols or names as equal regardless of letter case. In contrast, IFIDN is case-sensitive and only returns true if the symbols or names match exactly, including their letter case.

What is the purpose of the IFDEF directive?

The purpose of the IFDEF directive is to check whether a particular symbol (usually a macro or a variable) has been defined earlier in the code. It returns true if the symbol is defined and false if it is not. IFDEF is commonly used to conditionally include or exclude code blocks based on the existence of specific symbols in the assembly program.