Conditional Assembly Directives

Here is an organized summary of the key conditional assembly directives:

- **IF, ELSE, ENDIF -** Conditionally assemble code blocks based on a condition
- **IRP, IRPC -** Repeat a code block for each parameter value
- **REPT -** Repeat a block a specified number of times
- **WHILE, ENDW -** Repeat a block while a condition is true
- **EXITM -** Exit macro expansion early
- **LOCAL -** Declare local macro symbols

These allow macros to contain conditional logic, repetition, local symbols, early exit, and other advanced logic.

The important thing is that they work at assembly time, not runtime. The assembler evaluates them to determine what code to include or exclude in the final program. This makes macros much more powerful than simple text substitution.

Conditional-assembly directives

Conditional-assembly directives can be used to control the assembly of code based on certain conditions.

This can be useful for creating macros that are more flexible and adaptable. The general syntax for conditional-assembly directives is as follows:

```
1645 IF condition
1646     statements
1647
1648 [ELSE
1649     statements]
1650
1651 ENDIF
```

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are

assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

Here is a table of the most common conditional-assembly directives:

Directive	Description
IF *expression*	Permits assembly if the value of *expression* is true (nonzero). Possible relational operators are LT, GT, EQ, NE, LE, and GE.
IFB <*argument*>	Permits assembly if *argument* is blank. The argument name must be enclosed in angle brackets (<>).
IFNB <*argument*>	Permits assembly if *argument* is not blank. The argument name must be enclosed in angle brackets (<>).
IFIDN <*arg1*>,<*arg2*>	Permits assembly if the two arguments are equal (identical). Uses a case-sensitive comparison.
IFIDNI <*arg1*>,<*arg2*>	Permits assembly if the two arguments are equal. Uses a case-insensitive comparison.
IFDIF <*arg1*>,<*arg2*>	Permits assembly if the two arguments are unequal. Uses a case-sensitive comparison.
IFDIFI <*arg1*>,<*arg2*>	Permits assembly if the two arguments are unequal. Uses a case-insensitive comparison.
IFDEF *name*	Permits assembly if *name* has been defined.
IFNDEF *name*	Permits assembly if *name* has not been defined.
ENDIF	Ends a block that was begun using one of the conditional-assembly directives.
ELSE	Terminates assembly of the previous statements if the condition is true. If the condition is false, ELSE assembles statements up to the next ENDIF.
ELSEIF *expression*	Assembles all statements up to ENDIF if the condition specified by a previous conditional directive is false and the value of the current expression is true.
EXITM	Exits a macro immediately, preventing any following macro statements from being expanded.

Table 2:

Directive	Description
`.IF`	Assembles the following statements if the condition is true.
`.ELSE`	Assembles the following statements if the condition is false (and no previous `.ELSE` directive was true).
`.ENDIF`	Marks the end of a conditional-assembly block.

The following example shows how to use conditional-assembly directives to create a macro that can be used to print a message to the console:

```
1655 ; PrintMessage Macro
1656 ; Displays a message using the mWrite macro if the message is provided.
1657 ; Parameters:
1658 ;   - message: The message to be displayed
1659 ;
1660 ; Usage example:
1661 ; PrintMessage "Hello, world!"
1662
1663 PrintMessage MACRO message
1664     .IF message
1665         mWrite message
1666     .ENDIF
1667 ENDM
```

This macro takes a single parameter, message, which is the message to be printed to the console.

The IF directive checks to see if the message parameter is empty. If it is not empty, the mWrite macro is used to print the message to the console.

Here is an example of how to use the PrintMessage macro:

```
PrintMessage "Hello, world!"
```

This will print the message "Hello, world!" to the console.

Conditional-assembly directives can be used to create macros that are more flexible and adaptable.

For example, the PrintMessage macro could be modified to support different types of messages, such as error messages, warning messages, and informational messages.

It is important to note that conditional-assembly directives are evaluated at assembly time, not at runtime.

This means that the condition specified in a conditional-assembly directive must be constant and cannot be evaluated based on runtime values.

Checking for Missing Arguments

To prevent errors caused by missing arguments in a macro, you can use conditional directives like IFB (if blank) or IFNB (if not blank) to check if an argument is provided. Here's a simplified explanation and code example:

```
1675 mWriteString MACRO string
1676     IFB <string>
1677         ECHO ----------------------------------------
1678         ECHO * Error: Parameter missing in mWriteString
1679         ECHO * (No code generated)
1680         ECHO ----------------------------------------
1681         EXITM
1682     ENDIF
1683
1684     push edx
1685     mov edx, OFFSET string
1686     call WriteString
1687     pop edx
1688 ENDM
```

In assembly language macros, it's important to handle missing arguments to avoid errors during macro expansion. A missing argument can lead to invalid instructions when the macro is expanded.

To address this issue, you can use conditional directives:

IFB (if blank) returns true if a macro argument is blank, meaning it's not provided. IFNB (if not blank) returns true if a macro argument is not blank, indicating it's provided.

Let's take an example where we have a macro called mWriteString. This macro displays a string using the WriteString procedure, but it needs a string argument to work correctly.

In this example, if the string argument is missing, the macro will display an error message during assembly and won't generate any code. This helps ensure that your macros are used correctly with the required arguments.

Default Argument Initializers

Macros can have default argument initializers.

This means that if a macro argument is missing when the macro is called, the default argument will be used instead.

The syntax for a default argument initializer is as follows:

```
paramname := <argument>
```

Spaces before and after the operators are optional.

For example, the following macro has a default argument initializer for the text parameter:

```
; mWriteln Macro
; This macro writes a string to the console followed by a carriage return and line feed (CRLF).

mWriteln MACRO text:=<" ">
    ; Check if a text argument is provided. If not, use a space as the default.
    IFB <text>
        mWrite " " ; Display a space if no text is provided
    ELSE
        mWrite text ; Display the provided text
    ENDIF

    ; Call the Crlf procedure to move to the next line.
    call Crlf
ENDM
```

This macro first checks if a text argument is provided. If it's missing, it uses a space as the default value.

Then, it calls the mWrite macro to display the text and adds a line break by invoking the Crlf procedure.

If the mWriteln macro is called with no arguments, the default argument initializer (" ") will be used for the text parameter.

This will cause the macro to print a single space followed by an end of line to the console.

Boolean Expressions

The assembler allows the following relational operators to be used in constant Boolean expressions:

```
1715 <:      ;Less than
1716 >:      ;Greater than
1717 ==:     ;Equal to
1718 !=:     ;Not equal to
1719 <=:     ;Less than or equal to
1720 >=:     ;Greater than or equal to
```

These operators can be used in conjunction with the IF and other conditional directives to control the assembly of code.

For example, the following code uses the IF directive to check if the value of the x variable is greater than 10:

```
1724 IF x > 10
1725     ; Assemble the following code if x is greater than 10
1726     ; ...
1727 ENDIF
```

In this code snippet, the instructions within the IF block will be assembled only if the condition x > 10 is true.

If x is indeed greater than 10, the code within the IF block will be processed; otherwise, it will be skipped.

IF, ELSE, and ENDIF directives

The **IF, ELSE, and ENDIF directives** are used to control the assembly of code based on certain conditions.

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

The ENDIF directive marks the end of a conditional-assembly block.

The following is an example of how to use the IF, ELSE, and ENDIF directives:

```
1732 IF userAge > 18
1733     ; Assemble this code if the user's age is greater than 18
1734     mov edx, OFFSET adultMessage
1735 ELSE
1736     ; Assemble this code if the user's age is 18 or younger
1737     mov edx, OFFSET minorMessage
1738 ENDIF
```

In this example, the code chooses different messages to display based on whether the userAge is greater than 18 or not. If it's greater than 18, it displays the "adultMessage," otherwise, it displays the "minorMessage."

mGotoxyConst macro

The mGotoxyConst macro is an example of a macro that uses the IF, ELSE, and ENDIF directives to control the assembly of code.

The macro takes two parameters, X and Y, which must be constant

expressions.

The macro checks to see if the values of X and Y are within the valid ranges of 0 to 79 and 0 to 24, respectively.

If either of the values is outside of the valid range, the macro displays a warning message and sets a flag.

If the flag is set, the macro exits. Otherwise, the macro assembles the code to move the cursor to the specified coordinates.

The following is an example of how to use the mGotoxyConst macro:

```
mGotoxyConst 10, 20
```

This will move the cursor to column 10, row 20.

The IF, ELSE, and ENDIF directives can be used to create more complex and flexible macros.

The IFIDN and IFIDNI Directives

```
1748 mReadBuf MACRO bufferPtr, maxChars
1749 ;
1750 ; Reads from the keyboard into a buffer.
1751 ; Receives: offset of the buffer, count of the maximum
1752 ; number of characters that can be entered. The
1753 ; second argument cannot be edx or EDX.
1754 ;-----------------------------------------------------
1755 IFIDNI <maxChars>,<EDX> ; Check if the second argument is EDX (case-insensitive)
1756     ECHO Warning: Second argument to mReadBuf cannot be EDX
1757     ECHO **************************************************
1758     EXITM ; Exit the macro
1759 ENDIF
1760
1761 push ecx
1762 push edx
1763 mov edx, bufferPtr ; Move the buffer offset to EDX
1764 mov ecx, maxChars ; Move the maximum character count to ECX
1765 call ReadString
1766 pop edx
1767 pop ecx
1768 ENDM
```

This code defines a macro mReadBuf that reads from the keyboard into a buffer. It checks if the second argument, maxChars, is equal to EDX in a case-insensitive manner using IFIDNI.

If it is, it displays a warning message and exits the macro. If not, it proceeds to read from the keyboard into the specified buffer.

IFIDNI:

IFIDNI is a conditional assembly directive.

It is used to check if two symbols (or macro parameter names) are equal in a case-insensitive manner.

In the code, it's used to compare maxChars with EDX to ensure that the second argument is not equal to EDX.

If the comparison is true, it means the second argument is EDX, and a warning message is displayed.

If the comparison is false, the code proceeds to read input into the buffer.

ECHO:

ECHO is used to write a message to the console during assembly. In the code, it's used to display a warning message when the second argument is EDX.

EXITM:

EXITM is used to exit a macro.

In the code, if the second argument is EDX, the macro execution is halted using EXITM. push and pop:

These instructions are used to push and pop values onto and from the stack, respectively.

In the code, push ecx and push edx are used to save the values of registers ECX and EDX on the stack. This is done to protect the

original values of these registers.

Later, pop edx and pop ecx are used to restore the original values of these registers before exiting the macro.

mov:

The mov instruction is used to move data between registers and memory locations.

In the code, mov edx, bufferPtr is used to load the address of the buffer into the EDX register, and mov ecx, maxChars is used to load the value of maxChars into ECX before calling ReadString.

Overall, the code ensures that the second argument passed to the mReadBuf macro is not EDX and issues a warning if it is.

If the argument is not EDX, the code continues to read input into the specified buffer. The use of conditional assembly directives and stack manipulation ensures proper execution of the macro.

Summing a matrix row

```
1783 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1784     push ebx
1785     push ecx
1786     push esi
1787
1788     mov eax, index
1789     mov ebx, arrayOffset
1790     mov ecx, rowSize
1791     shr ecx, (TYPE eltType / 2) ; Adjust rowSize for the element type
1792
1793     add ebx, eax ; Calculate the row offset
1794     mov eax, 0    ; Initialize accumulator
1795     mov esi, 0    ; Initialize column index
1796
1797 L1:
1798     IFIDNI <eltType>, <DWORD>
1799         mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
1800     ELSE
1801         movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
1802     ENDIF
1803
1804     add eax, edx
1805     inc esi
1806     loop L1
1807
1808     pop esi
1809     pop ecx
1810     pop ebx
1811 ENDM
```

Here's what the macro does:

It takes four parameters: index, arrayOffset, rowSize, and eltType.

The rowSize is adjusted based on the eltType parameter, ensuring it represents the number of elements in each row.

The macro initializes the registers and sets up the required variables.

It calculates the row offset and initializes the accumulator.

Inside the loop labeled L1, it uses an IFIDNI conditional to check if the eltType is DWORD.

Depending on this condition, it reads the elements from memory

correctly.

It accumulates the elements into eax and increments the column index.

The loop continues until all elements are processed.

Finally, the macro cleans up by popping the registers from the stack.

This simplified version of the macro calculates the sum of a row in an array of different element types, taking into account the size of each element type.

```
; Define the mCalc_row_sum macro
mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
    LOCAL L1

    ; Save registers
    push ebx
    push ecx
    push esi

    ; Set up required registers
    mov eax, index
    mov ebx, arrayOffset
    mov ecx, rowSize

    ; Calculate the row offset
    mul ecx              ; row index * row size
    add ebx, eax         ; row offset

    ; Prepare the loop counter
    shr ecx, (TYPE eltType / 2)  ; Calculate the number of elements in a row

    ; Initialize accumulator and column index
    mov eax, 0           ; Accumulator
    mov esi, 0           ; Column index

```

```
    L1:
    ; Check if eltType is DWORD
    IFIDNI <eltType>, <DWORD>
        mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
    ELSE
        ; Assuming eltType is BYTE or WORD
        movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
    ENDIF

    add eax, edx          ; Add to accumulator
    inc esi               ; Move to the next column

    loop L1

    ; Restore registers
    pop esi
    pop ecx
    pop ebx

ENDM

```

```
.data
; Define data arrays
tableB BYTE 10h, 20h, 30h, 40h, 50h
RowSizeB = ($ - tableB)
tableW WORD 10h, 20h, 30h, 40h, 50h
RowSizeW = ($ - tableW)
tableD DWORD 10h, 20h, 30h, 40h, 50h
RowSizeD = ($ - tableD)
index DWORD ?

.code
; Call the mCalc_row_sum macro for different arrays and data types
mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
mCalc_row_sum index, OFFSET tableD, RowSizeD, DWORD
```

Certainly, here's an explanation of the provided code:

The code introduces a macro called mCalc_row_sum, which is designed to calculate the sum of a row in a two-dimensional array. This macro takes four parameters: index, arrayOffset, rowSize, and eltType.

Inside the macro, registers ebx, ecx, and esi are pushed onto the stack to ensure they are preserved and not affected by the macro's

operations.

The index parameter represents the row index, arrayOffset is the offset of the array, rowSize indicates the number of bytes in each table row, and eltType specifies the array type, which can be BYTE, WORD, or DWORD.

The row offset is calculated by multiplying the rowSize with the index and adding it to arrayOffset. This is done to find the starting address of the row within the array.

To determine the number of elements in a row (whether they are bytes, words, or double words), the macro uses the eltType.

If it's DWORD, no scaling is required. If it's BYTE or WORD, the ecx register is shifted to the right by 1 or 2 bits, respectively, to adjust it to the number of elements in a row.

The accumulator (eax) and the column index (esi) are initialized to 0, as the macro iterates over the row.

Within a loop labeled L1, the macro loads an element from memory based on the element type (eltType). If the element type is DWORD, it uses a simple mov instruction.

If the element type is BYTE or WORD, it uses movzx to zero-extend the value.

The element value is added to the accumulator (eax), and the column index is incremented (esi) to move to the next element in the row.

The loop continues until all elements in the row have been processed.

After the loop, the registers are popped to restore their original values.

The .data section defines three different arrays (tableB, tableW, and tableD) with different data types (BYTE, WORD, and DWORD) and calculates the size of a row for each array.

The .code section demonstrates how to use the mCalc_row_sum macro with these arrays, specifying the appropriate data type for each call.

Overall, this macro allows you to easily calculate the sum of a row in a 2D array with different data types, making your code more versatile and readable.