ECHO and LOCAL Directives

ECHO directive

The ECHO directive writes a string to standard output as the program is assembled. This can be useful for debugging or for printing informative messages to the user.

Example:

```
1349 mPutchar MACRO char:REQ
1350 ECHO Expanding the mPutchar macro
1351 push
1352 eax
1353 mov
1354 al,char
1355 call
1356 WriteChar
1357 pop
1358 eax
1359 ENDM
```

When this macro is expanded, the assembler will print the message "Expanding the mPutchar macro" to standard output.

LOCAL Directive

Macros can often contain labels. If a macro is invoked multiple times in a program, the labels in the macro will be redefined each time. This can lead to errors if the same label is used in multiple places in the program.

The LOCAL directive can be used to avoid this problem. When a label is declared as LOCAL, the assembler will generate a unique name for the label each time the macro is expanded.

Example:

```
makeString MACRO text
LOCAL string
.data
    string BYTE text,0
ENDM
```

In this example, the makeString macro declares a variable named string. If the macro is invoked multiple times, the assembler will generate a unique name for the string variable each time.

To use the LOCAL directive, simply add the LOCAL keyword before the label declaration. For example:

```
LOCAL label
```

The LOCAL keyword can be used for any type of label, including code labels and data labels.

The ECHO and LOCAL directives can be useful for debugging macros and avoiding label redefinition errors.

Macros Containing Code and Data

Macros can contain both code and data. This means that a macro can define both instructions and variables. Example:

```
mWrite MACRO text
    LOCAL string   ; Define a local label
    .data          ; Switch to the data section
    string BYTE text, 0   ; Define the string

    .code          ; Switch back to the code section
    push edx       ; Push edx onto the stack
    mov edx, OFFSET string   ; Load the address of the string into edx
    call WriteString        ; Call the WriteString procedure with the address of the string
    pop edx        ; Restore the value of edx
ENDM
```

This macro defines a function called mWrite() that displays a string on the console. The macro takes a single parameter, text, which is the string to display.

the string to display.

The macro first defines a local label named string. This label is used to identify the string in the .data section of the program.

The macro then defines the code for the mWrite() function. This code pushes the address of the string onto the stack, calls the WriteString() function to display the string, and then pops the address of the stack.

Nested Macros

A macro invoked from another macro is called a nested macro. When the assembler's preprocessor encounters a call to a nested macro, it expands the macro in place.

This means that the parameters passed to the enclosing macro are passed directly to its nested macros. Example:

```
1390 mWriteln MACRO text
1391     mWrite text    ; Invoke the mWrite macro with the given text
1392     call Crlf      ; Call the Crlf procedure to add a new line
1393 ENDM
```

This mWriteln macro simplifies the process of writing a line of text. It calls the mWrite macro to display the provided text and then adds a new line by calling the Crlf procedure.

This macro defines a function called mWriteln() that writes a string to the console and appends an end of line. The macro invokes the mWrite() macro to display the string, and then calls the Crlf() procedure to append an end of line.

The text parameter is passed directly to the mWrite() macro. Example:

```
mWriteln "My Sample Macro Program"
```

When the assembler expands this statement, it will first expand the mWriteln() macro. This will result in the following code:

```
1406 mWrite "My Sample Macro Program"
1407 call Crlf
```

This code will then be expanded to the following instructions:

```
1412 push edx           ; Push the value in the edx register onto the stack
1413 mov edx, OFFSET ??0002   ; Load the address of label ??0002 into edx
1414 call WriteString  ; Call the WriteString procedure with the address in edx
1415 pop edx           ; Pop the value from the stack back into edx
1416 call Crlf         ; Call the Crlf procedure to add a new line
```

Tips for Creating Macros:

Keep macros short and simple. This will make them easier to understand and maintain.

Use a modular approach when creating macros. This means breaking down complex macros into smaller, simpler macros. This will make the macros more reusable and flexible.

Use the LOCAL directive to avoid label redefinition errors.

Document your macros clearly. This will help other programmers understand how to use them.

Macros can be a powerful tool for simplifying and improving code. However, it is important to use them carefully and to follow the tips above to avoid problems.

Irvine32/64 library and Macros:

```
; Using the Book's Macro Library (32-Bit Mode Only)
; To enable the library, include the following line after your existing INCLUDE:
INCLUDE Macros.inc

; Macros in the Macros.inc Library:
; mDump - Displays a variable using its name and default attributes.
; mDumpMem - Displays a range of memory.
; mGotoxy - Sets the cursor position in the console window buffer.
; mReadString - Reads a string from the keyboard.
; mShow - Displays a variable or register in various formats.
; mShowRegister - Displays a 32-bit register's name and contents in hexadecimal.
; mWrite - Writes a string literal to the console window.
; mWriteSpace - Writes one or more spaces to the console window.
; mWriteString - Writes a string variable's contents to the console window.

; Example usage of the mDumpMem macro:
; mDumpMem OFFSET array, LENGTHOF array, TYPE array
; Displays the 'array' variable using its default attributes.
```

In summary, the "Macros.inc" library contains various macros that simplify common tasks in assembly programming, such as displaying variables, manipulating cursor positions, reading from the keyboard, and more.

You can enable this library by including it in your program, and then you can use the provided macros to streamline your code.

==========

Let's expand on them a bit:

mDumpMem macro

The **mDumpMem macro** is used to display a block of memory in the console window. It requires three arguments: the memory offset, the number of items to display, and the size of each memory component. The macro internally invokes the DumpMem library procedure, passing the three arguments to it (ESI, ECX, and EBX, respectively). Here's a cleaned-up version of the explanation:

```
1440 mDumpMem MACRO address:REQ, itemCount:REQ, componentSize:REQ
1441
1442   push ebx
1443   push ecx
1444   push esi
1445
1446   mov esi, address
1447   mov ecx, itemCount
1448   mov ebx, componentSize
1449
1450   call DumpMem
1451
1452   pop esi
1453   pop ecx
1454   pop ebx
1455
1456 ENDM
```

The mDumpMem macro displays a memory dump using the DumpMem procedure.

It takes 3 parameters:
- address - Offset of memory block to dump
- itemCount - Number of components to display
- componentSize - Size in bytes of each component

It avoids passing EBX, ECX, ESI to prevent corrupting registers used internally.

Example usage:

```
mDumpMem OFFSET array, LENGTHOF array, TYPE array
```

This would display the contents of the array memory block, with number of items and size based on the array's declared size and type.

It shows both the memory offset and hex dump of the values, with the componentSize determining the format.

So mDumpMem provides a convenient way to dump memory blocks for debugging.

==========

mDump macro

The mDump macro displays the address and hexadecimal contents of a variable.

It takes two parameters:

- varName - Name of the variable to dump
- useLabel - Optional label to display

The size and format match the variable's declared type.

If useLabel is passed a non-blank value, it will print the variable name.

```
1463 mDump MACRO varName:REQ, useLabel
1464
1465 call Crlf
1466
1467 IFNB <useLabel>
1468   mWrite "Variable name: &varName"
1469 ENDIF
1470
1471 mDumpMem OFFSET varName, LENGTHOF varName, TYPE varName
1472
1473 ENDM
```

The &varName substitution operator inserts the actual variable name into the string.

IFNB checks if useLabel was passed a non-blank value.

mDumpMem is called to print the hex dump using the variable's attributes.

So mDump provides a convenient way to quickly dump variables for debugging.

debugging.

==========

mGotoxy macro

The mGotoxy macro positions the cursor in the console window.

It takes two BYTE parameters for the X and Y coordinates.

Avoid passing DH and DL to prevent register conflicts.

```
1477 mGotoxy MACRO X:REQ, Y:REQ
1478
1479   push edx
1480
1481   mov dh,Y
1482   mov dl,X
1483
1484   call Gotoxy
1485
1486   pop edx
1487
1488 ENDM
```

Register conflict example:

If DH and DL are passed as arguments, the expanded code would be:

```
1492 push edx
1493 mov dh,dl ; Y value overwritten
1494 mov dl,dh ; X value now incorrect
1495 call Gotoxy
1496 pop edx
```

DH is overwritten by DL before it can be copied to DL.

So the macro documentation warns not to pass DH/DL to avoid this problem.

In summary, mGotoxy sets the cursor position, but care must be taken with register arguments to prevent conflicts. The macro code/docs make this clear.

==========

mReadString macro

The mReadString macro reads keyboard input into a buffer.

It takes one parameter:

- **varName -** Name of the buffer to store the input string

Avoids using ECX and EDX internally to prevent register conflicts.

```
1500 mReadString MACRO varName:REQ
1501
1502   push ecx
1503   push edx
1504
1505   mov edx,OFFSET varName
1506   mov ecx,SIZEOF varName
1507
1508   call ReadString
1509
1510   pop edx
1511   pop ecx
1512
1513 ENDM
```

```
1517 firstName BYTE 30 DUP(?)
1518
1519 mReadString firstName
```

This would call ReadString to input a string from the keyboard into the firstName buffer.

So mReadString encapsulates the details of calling ReadString to simplify inputting strings.

==========

mShow macro

The mShow macro displays a register or variable's name and contents in different formats. It is useful for debugging.

mShow takes a register/variable name followed by format specifiers:

- **H - Hexadecimal**
- **D - Decimal (unsigned by default)**
- **I - Signed decimal**
- **B - Binary**
- **N - Newline**

You can combine multiple formats like "HDB" and add multiple newlines. The default is "HIN".

Display AX in multiple formats:

```
1524 mov ax,4096
1525 mShow AX          ; HIN (hex, signed decimal, newline)
1526 mShow AX,DBN      ; Decimal, binary, newline
```

```
1529 ;Output
1530 AX = 1000h = 4096d
1531 AX = 4096d = 00010000 00000000b
```

Display multiple registers on one line:

```
1540 mov ax,1
1541 mov bx,2
1542 mov cx,3
1543 mov dx,4
1544
1545 mShow AX,D
1546 mShow BX,D
1547 mShow CX,D
1548 mShow DX,DN
```

```
1552 ;Output
1553 AX = 1d BX = 2d CX = 3d DX = 4d
```

Display variable in decimal with newline:

```
1555 mydword DWORD ?
1556
1557 mShow mydword,DN
```

==========

mShowRegister macro

mShowRegister displays a 32-bit register's name and hexadecimal contents. It is useful for debugging.

It takes two parameters:
- **regName -** String to display as the register name
- **regValue -** The 32-bit register value

```
mShowRegister EBX, ebx
```

Displays:
EBX=7FFD9000

Implementation
The macro does the following:

- Declares a local string variable tempStr to hold the label
- Pushes EAX and EDX to preserve registers used internally
- Builds the label string with the name and '='
- Calls WriteString to display the label
- Pops EDX to restore it
- Moves the register value into EAX
- Calls WriteHex to display the hex value
- Pops EAX to restore it

Some key points:
- Local string variable avoids modifying caller's code
- Register pushing/popping prevents corruption
- WriteString and WriteHex handle the output
- Substitution inserts regName and regValue from caller

So mShowRegister encapsulates the details of displaying a register's name and value for debugging. The caller simply specifies the name and register.

==========

The mWriteSpace macro writes one or more spaces to the console window. It takes an optional integer parameter specifying the number of spaces to write. The default value is one.

The mWriteString macro writes the contents of a string variable to the console window. It takes a single parameter, which is the name of the string variable to write.

Here is a more detailed explanation of how these macros work:

mWriteSpace

The mWriteSpace macro defines a local label named spaces. This label is used to identify a string of spaces in the .data section of the program.

The macro then defines the code for the mWriteSpace() function. This code pushes the address of the spaces string onto the stack, calls the WriteString() function to display the string, and then pops the address of the stack.

Example:

```
mWriteSpace 5
```

When the assembler expands this statement, it will first expand the mWriteSpace() macro. This will result in the following code:

```
1571 push edx
1572 mov edx,OFFSET spaces
1573 call WriteString
1574 pop edx
```

This code will then be expanded to the following instructions:

```
1577 push edx
1578 mov edx,5
1579 rep movsb
1580 pop edx
```

The rep movsb instruction will copy 5 bytes from the source (the EDX register) to the destination (the console window). The source will be incremented by one after each byte is copied, and the destination will be incremented by one after each byte is copied.

==========

mWriteString

The mWriteString macro defines a function called mWriteString() that writes the contents of a string variable to the console window. The macro takes a single parameter, which is the name of the string variable to write.

The macro first saves the EDX register on the stack. This is because the WriteString() function uses the EDX register to store the address of the string to write.

The macro then loads the address of the string variable into the EDX register. Finally, the macro calls the WriteString() function and then pops the EDX register from the stack.
Example:

```
.data
    str1 BYTE "Please enter your name: ",0
.code
    mWriteString str1
```

When the assembler expands this statement, it will first expand the mWriteString() macro. This will result in the following code:

```
push edx
mov edx,OFFSET str1
call WriteString
pop edx
```

This code will then be expanded to the following instructions:

```
push edx
mov edx,OFFSET str1
call WriteString
pop edx
```

The WriteString() function will then write the contents of the str1 variable to the console window.

Conclusion

The mWriteSpace and mWriteString macros can be useful for simplifying code and making it more readable. For example, the following code:

```
WriteString("Please enter your name: ");
```

can be rewritten using the macros as follows:

```
mWriteString "Please enter your name: "
```

Here is the Wraps.asm program explained with the code rewritten for clarity:

Purpose:
Demonstrates use of wrapper macros for common procedures.

Macros used:
- mGotoxy - Set cursor position
- mWrite - Display formatted output
- mWriteString - Display a string
- mReadString - Input a string
- mDumpMem - Hex dump memory

```
INCLUDE Irvine32.inc
INCLUDE Macros.inc
.data
    array DWORD 1,2,3,4,5,6,7,8
    firstName BYTE 31 DUP(?)
    lastName BYTE 31 DUP(?)
.code
    main PROC
      mGotoxy 0,0       ; Position cursor
      ; Display heading
      mWrite <"Sample Macro Program",0dh,0ah>
      ; Input first name
      mGotoxy 0,5
      mWrite "Please enter your first name: "
      mReadString firstName
      ; Input last name
      call Crlf
      mWrite "Please enter your last name: "
      mReadString lastName
      ; Display name
      call Crlf
      mWrite "Your name is "
      mWriteString firstName
      mWriteSpace
      mWriteString lastName
      ; Display array
      call Crlf
      mDumpMem OFFSET array, LENGTHOF array, TYPE array
      exit
    main ENDP
    END main
```

Here are the review questions rewritten:

1. When a macro is invoked, does the assembler automatically insert CALL and RET instructions in the generated code? Explain.
2. Where is macro expansion handled - at assembly time or runtime?
3. What is the main advantage of using macros with parameters compared to macros without parameters?
4. Can a macro definition appear before or after the code that invokes it, as long as it is in the code segment? Explain.

5. How does replacing a procedure call with a macro invocation typically affect code size if the macro is called multiple times? Explain.
6. Can a macro contain data definitions like DW and DB? Explain.

Here are answers to the review questions:

1. False - CALL and RET instructions are not automatically inserted when a macro is invoked. The macro expansion is inserted directly into the code.
2. True - Macro expansion is handled by the assembler's preprocessor before the code is assembled.
3. Macros with parameters are more flexible since they can accept arguments from the caller. This avoids having to modify the macro definition for different uses.
4. True - As long as it is in the code segment, a macro can appear anywhere, before or after its usage. The assembler handles macros separately during preprocessing.
5. True - If a macro is invoked multiple times, the code will be expanded/duplicated each time, increasing the overall code size compared to calling a single procedure.
6. False - Macros can contain data definitions, though the data is inserted wherever the macro is expanded.