Contents

STRUCTURES

A structure in assembly language is defined using the STRUCT and ENDS directives.

Inside the structure, fields are defined using the same syntax as for ordinary variables. Structures can contain virtually any number of fields:

```
0890 name STRUCT
0891 field-declarations
0892 name ENDS
```

For example, the following structure defines an employee structure with fields for ID number, last name, years of service, and an array of salary history values:

```
0900 Employee STRUCT
0901     IdNum BYTE "000000000"
0902     LastName BYTE 30 DUP(0)
0903     Years WORD 0
0904     SalaryHistory DWORD 0,0,0,0
0905 Employee ENDS
```

The first field, IdNum, is a byte-sized field initialized to the string "000000000".

The second field, LastName, is a 30-byte array of bytes initialized to all zeros.

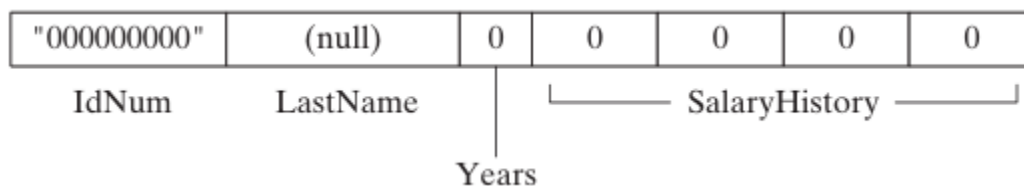The third field, Years, is a 2-byte word initialized to zero.

The fourth field, SalaryHistory, is a 4-byte double word initialized to all zeros.

Note: The DUP operator is used to initialize the SalaryHistory array to all zeros. The syntax 30 DUP(0) creates an array of 30 bytes, each of which is initialized to zero.

Structure Memory Layout

In memory, structures are laid out contiguously, with each field starting at the next available address.

The following diagram shows the memory layout of the Employee structure:

Memory Layout:

```
+--------------------------------------------------+
|  IdNum                                           |  (1 byte)
+--------------------------------------------------+
|  LastName                                        |  (30 bytes)
+--------------------------------------------------+
|  Years                                           |  (2 bytes)
+--------------------------------------------------+
|  SalaryHistory                                   |  (4 bytes)
+--------------------------------------------------+
```

Using Structures

Declaring structures:

```
identifier structureType < initializer-list >
```

The **identifier** is the name of the structure variable.

The **structureType** is the name of the structure type that the variable will be.

The **initializer-list** is a list of values that will be used to initialize the structure fields. The values in the initializer-list must be in the same order as the fields in the structure definition.

If you do not specify an initializer-list, the structure fields will be initialized to their default values.

Example:

```
Employee worker <>
```

This code declares a structure variable named worker of type Employee. The <> (angle brackets) indicate that the structure fields should be initialized to their default values.

Referencing Structure Variables:

To reference a structure variable, you simply use its name. For example, the following code references the Years field of the worker structure variable:

Once a structure is defined, you can declare variables of that type using the same syntax as for ordinary variables. For example, the following code declares two employee variables:

```
mov dx, worker.Years
```

Referencing Structure Members:

To reference a structure member, you use the following syntax:

```
structureVariable.memberName
```

The structureVariable is the name of the structure variable. The memberName is the name of the structure member.

Example:

```
mov dx, worker.SalaryHistory
```

This code references the SalaryHistory field of the worker structure variable.

Using the OFFSET Operator

The OFFSET operator can be used to obtain the address of a structure member. The syntax for the OFFSET operator is as follows:

```
OFFSET structureVariable.memberName
```

The structureVariable is the name of the structure variable. The memberName is the name of the structure member.

Example:

```
mov edx, OFFSET worker.LastName
```

This code obtains the address of the LastName field of the worker structure variable.

==

Declaring another structure:

```
0926 Employee1 Employee
0927 Employee2 Employee
```

You can then access the fields of these variables using the dot operator (.). For example, the following code sets the IdNum field of Employee1 to 123456789:

```
mov Employee1.IdNum, 123456789
```

You can also access the fields of a structure using the displacement operator ([]). For example, the following code sets the Years field of Employee2 to 5:

```
mov Employee2[2], 5
```

Note: The displacement operator is calculated relative to the beginning of the structure. In the previous example, the displacement of the Years field is 2, because it is the second field in the structure.

Aligning Structure Fields

To achieve the best memory I/O performance, structure members should be aligned to addresses matching their data types.

This means that a byte member should be aligned on a byte boundary, a word member should be aligned on a word boundary, and so on.

If structure members are not aligned correctly, the CPU will require more time to access them.

The following table lists the alignments used by the Microsoft C and C++ compilers and by Win32 API functions:

Member Type	Alignment
BYTE, SBYTE	8-bit (byte) boundary
WORD, SWORD	16-bit (word) boundary
DWORD, SDWORD	32-bit (doubleword) boundary
QWORD	64-bit (quadword) boundary
REAL4	32-bit (doubleword) boundary
REAL8	64-bit (quadword) boundary
Structure	Largest alignment requirement of any member
Union	Alignment requirement of the first member

The key points are:

- Basic data types like BYTE, WORD, DWORD etc have natural alignment to their size boundary.

- 32-bit and 64-bit floating point values align to doubleword and quadword respectively.

- Structures align to the largest alignment needed by any member.

- Unions align to the first member.

Properly aligning data prevents performance issues from unaligned memory accesses. The assembler takes care of alignment automatically based on the data type.

To align structure fields in assembly language, you can use the ALIGN directive.

The ALIGN directive sets the address alignment of the next field or variable.

For example, the following code aligns the myVar variable to a doubleword boundary:

```
0940 .data
0941     ALIGN DWORD
0942     myVar DWORD ?
```

You can also use the ALIGN directive to align structure fields.

For example, the following code correctly defines the Employee structure, using ALIGN to put Years on a word boundary and SalaryHistory on a doubleword boundary:

```
0947 Employee STRUCT
0948     IdNum BYTE "000000000" ; 9
0949     LastName BYTE 30 DUP(0) ; 30
0950     ALIGN WORD ; 1 byte added
0951     Years WORD 0 ; 2
0952     ALIGN DWORD ; 2 bytes added
0953     SalaryHistory DWORD 0,0,0,0 ; 16
0954 Employee ENDS ; 60 total
```

Why is Aligning Structure Fields Important?

Aligning structure fields is important for performance because the CPU can access aligned data more efficiently.

When structure fields are not aligned, the CPU may have to perform additional instructions to access the data.

This can lead to a significant performance decrease, especially in applications that manipulate large amounts of data.

INDIRECT AND INDEXED OPERANDS STRUCTS

Indirect Operands

Indirect operands are used to address memory locations through the contents of a register. This can be useful for accessing structure members, since it allows you to use a register to store the address of the structure.

To use an indirect operand, you need to use the **PTR operator.** The PTR operator tells the assembler that the operand is indirect.

For example, the following code uses an indirect operand to access the Years field of the worker structure variable:

```
0992 mov esi, OFFSET worker
0993 mov ax, (Employee PTR [esi]).Years
```

This code first moves the address of the worker structure variable into the esi register.

Then, it uses the PTR operator to tell the assembler that the [esi] operand is indirect.

This means that the assembler will load the contents of the esi register and use that as the address of the Years field.

Indexed Operands

Indexed operands are used to address memory locations by adding an offset to the contents of a register.

This can be useful for accessing arrays of structures, since it allows you to use a register to store the index of the element in the array.

To use an indexed operand, you need to use the [register + offset] syntax.

The register is the register that contains the index of the element in the array.

The offset is the offset of the structure field from the beginning of the structure.

For example, the following code uses an indexed operand to access the Years field of the employee in index position 1 of the department array:

```
1000 .data
1001     department Employee 5 DUP(<>)
1002 .code
1003     mov esi, TYPE Employee
1004     ; index = 1
1005     mov department[esi].Years, 4
```

This code first moves the size of the Employee structure into the esi register.

Then, it uses the [esi] syntax to access the element in index position 1 of the department array. Finally, it sets the Years field of that element to 4.

Looping through an Array

Indirect and indexed operands can be used to loop through an array of structures.

The following code shows how to loop through the AllPoints array and assign coordinates to each element:

```
1010 ; Loop Through Array
1011 (AllPoints.asm)
1012 INCLUDE Irvine32.inc
1013 NumPoints = 3
1014 .data
1015     ALIGN WORD
1016     AllPoints COORD NumPoints DUP(<0,0>)
1017 .code
1018     main PROC
1019     mov edi, 0
1020     ; array index
1021     mov ecx, NumPoints
1022     ; loop counter
1023     mov ax, 1
1024     ; starting X, Y values
1025     L1:
1026     mov (COORD PTR AllPoints[edi]).X, ax
1027     mov (COORD PTR AllPoints[edi]).Y, ax
1028     add edi, TYPE COORD
1029     inc ax
1030     loop L1
1031     exit
1032     main ENDP
1033     END main
```

This code first moves the value 0 into the edi register. This will be the array index. Then, it moves the value of the NumPoints variable into the ecx register. This will be the loop counter.

Next, the code moves the value 1 into the ax register. This will be the starting X and Y values for the coordinates.

Then, the code enters a loop. Inside the loop, it first uses an indirect operand to access the X field of the element in index position edi of the AllPoints array. It then sets the X field to the value in the ax register.

Next, the code uses an indirect operand to access the Y field of the element in index position edi of the AllPoints array. It then sets the Y field to the value in the ax register.

After that, the code increments the edi register by the size of the COORD structure. This will move the array index to the next element in the array.

Finally, the code increments the ax register by 1. This will increment the X and Y values for the next coordinates.

The loop will continue until the ecx register reaches zero. At that point, the loop will exit and the program will terminate.

Conclusion

Indirect and indexed operands are powerful tools that can be used to access structure members and arrays of structures.

PERFORMANCE OF ALIGNED AND MISALIGNED STRUCTS

Aligned structure members can be accessed by the CPU more efficiently than misaligned structure members. This is because the CPU can perform fewer instructions to access aligned data.

The performance impact of misaligned structure members depends on the CPU architecture and the specific structure layout. However, in general, misaligned structure members can lead to a decrease in performance.

The following code shows a simple test to compare the performance of aligned and misaligned structure members:

```
1035 .data
1036     ALIGN DWORD
1037     startTime DWORD ?
1038     ; align startTime
1039     emp Employee <>
1040     ; or: emp EmployeeBad <>
1041 .code
1042     call
1043     GetMSeconds
1044     ; get starting time
1045     mov
1046     startTime,eax
1047     mov
1048     ecx,0FFFFFFFFh
1049     ; loop counter
1050     L1:
1051     mov
1052     emp.Years,5
1053     mov
1054     emp.SalaryHistory,35000
1055     loop
1056     L1
1057     call
1058     GetMSeconds
1059     ; get starting time
1060     sub
1061     eax,startTime
1062     call
1063     WriteDec
1064     ; display elapsed time
```

This code gets the system time, executes a loop that accesses structure fields, and calculates the elapsed time. The variable emp can be declared as an Employee or EmployeeBad object.

The Employee structure is aligned, while the EmployeeBad structure is not aligned.

When the code is executed using the Employee structure, the elapsed time is 6141 milliseconds.

When the code is executed using the EmployeeBad structure, the elapsed time is 6203 milliseconds.

The difference in elapsed time is small (62 milliseconds), but it is still measurable.

This suggests that even a small amount of misalignment can have a negative impact on performance.

It is important to note that the performance impact of misaligned structure members can vary depending on the CPU architecture and the specific structure layout.

In some cases, the performance impact may be more significant than the 62 milliseconds observed in the previous example.

Conclusion

It is generally a good practice to align structure members. This will help to ensure that the CPU can access the structure members efficiently, which can lead to improved performance.

Example: Displaying the System Time

The following is a step-by-step explanation of the program ShowTime.asm in depth:

```
1067 ; Structures (ShowTime.ASM)
1068 INCLUDE Irvine32.inc
1069 .data
1070     sysTime SYSTEMTIME <>
1071     XYPos COORD <10,5>
1072     consoleHandle DWORD ?
1073     colonStr BYTE ":",0
1074 .code
1075     main PROC
1076     ; Get the standard output handle for the Win32 Console.
1077     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
1078     mov consoleHandle,eax
```

The first part of the program defines the data structures that will be used.

The sysTime variable is a SYSTEMTIME structure, which will be used to store the system time.

The XYPos variable is a COORD structure, which will be used to store the cursor position.

The consoleHandle variable is a DWORD variable, which will be used to store the handle to the standard output handle.

The GetStdHandle function is used to retrieve the handle to the standard output handle.

The handle to the standard output handle is used by the SetConsoleCursorPosition and WriteString functions to write data to the console.

```
1083 ; Set the cursor position and get the system time.
1084 INVOKE SetConsoleCursorPosition, consoleHandle, XYPos
1085 INVOKE GetLocalTime, ADDR sysTime
```

The next part of the program sets the cursor position to the specified coordinates and retrieves the system time.

The SetConsoleCursorPosition function is used to set the cursor position to the specified coordinates. The GetLocalTime function is used to retrieve the system time.

```
1090 ; Display the system time (hh:mm:ss).
1091 movzx eax,sysTime.wHour
1092 ; hours
1093 call WriteDec
1094 mov edx,OFFSET colonStr
1095 ; ":"
1096 call WriteString
1097 movzx eax,sysTime.wMinute
1098 ; minutes
1099 call WriteDec
1100 call WriteString
1101 movzx eax,sysTime.wSecond
1102 ; seconds
1103 call WriteDec
1104 call Crlf
```

The next part of the program displays the system time to the console.

The WriteDec function is used to write a decimal number to the console.

The WriteString function is used to write a string to the console.

The Crlf function is used to write a carriage return and line feed to the console.

```
1108 call WaitMsg
1109 ; "Press any key..."
1110 exit
1111 main ENDP
1112 END main
```

The last part of the program displays a message to the console and waits for the user to press a key.

The WaitMsg function is used to display a message to the console and wait for the user to press a key.

Conclusion

The ShowTime.asm program is a simple example of how to use structures in assembly language. By using structures, you can group related data together and make your code more readable and maintainable.

STRUCTURES CONTAINING STRUCTURES

Structures can contain instances of other structures. This is called a nested structure.

For example, a Rectangle structure can be defined in terms of its upper-left and lower-right corners, both COORD structures:

```
1116 Rectangle STRUCT
1117     UpperLeft COORD <>
1118     LowerRight COORD <>
1119 Rectangle ENDS
```

This means that a **Rectangle structure** will contain **two COORD structures**, one for the upper-left corner and one for the lower-right corner.

Rectangle variables can be declared without overrides or by overriding individual COORD fields. The following examples show how to declare Rectangle variables:

```
1123 rect1 Rectangle < >
1124 rect2 Rectangle { }
1125 rect3 Rectangle { {10,10}, {50,20} }
1126 rect4 Rectangle < <10,10>, <50,20> >
```

The first two declarations, rect1 and rect2, will create Rectangle variables with the default values for the UpperLeft and LowerRight fields.

The third declaration, rect3, will create a Rectangle variable with the specified values for the UpperLeft and LowerRight fields.

The fourth declaration, rect4, is an alternative way to declare a Rectangle variable with the pecified values for the UpperLeft and LowerRight fields.

Once a Rectangle variable has been declared, you can access its fields using the dot notation.

For example, the following code moves the value 10 to the X coordinate of the upper-left corner of the rect1 variable:

```
mov rect1.UpperLeft.X, 10
```

You can also access a structure field using an indirect operand. For example, the following code moves the value 10 to the Y coordinate of the upper-left corner of the structure pointed to by the esi register:

```
1137 mov esi,OFFSET rect1
1138 mov (Rectangle PTR [esi]).UpperLeft.Y, 10
```

The OFFSET operator can be used to return pointers to individual structure fields, including nested fields. For example, the following code moves the value 50 to the X coordinate of the lower-right corner of the rect2 variable:

```
1141 mov edi,OFFSET rect2.LowerRight
1142 mov (COORD PTR [edi]).X, 50
```

Example program:

```
1149 ; Drunkard's Walk
1150 ; Drunkard's walk program simulates a professor's random path in an imaginary grid.
1151 ; The professor starts at coordinates 25, 25 and wanders around the immediate area.
1152
1153 INCLUDE Irvine32.inc
1154
1155 ; Constants
1156 WalkMax = 50     ; Maximum number of steps
1157 StartX = 25      ; Starting X-coordinate
1158 StartY = 25      ; Starting Y-coordinate
1159
1160 ; Define a structure to store the path and number of steps
1161 DrunkardWalk STRUCT
1162     path COORD WalkMax DUP(<0,0>) ; Array of COORD objects for the path
1163     pathsUsed WORD 0                ; Number of steps taken
1164 DrunkardWalk ENDS
1165
1166 ; Prototypes
1167 DisplayPosition PROTO currX:WORD, currY:WORD
1168
1169 .data
1170 aWalk DrunkardWalk <> ; Create an instance of the DrunkardWalk structure
1171
1172 .code
1173 main PROC
1174     mov esi,OFFSET aWalk ; Get the address of the aWalk structure
1175     call TakeDrunkenWalk ; Simulate the professor's walk
1176     exit
1177 main ENDP
```

```
1179 ;-------------------------------------------------------
1180 TakeDrunkenWalk PROC
1181 LOCAL currX:WORD, currY:WORD
1182
1183 ; Takes a walk in random directions (north, south, east, west).
1184 ; Receives: ESI points to a DrunkardWalk structure
1185 ; Returns: the structure is initialized with random values
1186 ;-------------------------------------------------------
1187
1188 pushad ; Preserve registers
1189
1190 ; Initialize EDI with the address of the path array
1191 mov edi,esi
1192 add edi,OFFSET DrunkardWalk.path
1193
1194 mov ecx,WalkMax  ; Set loop counter
1195 mov currX,StartX ; Initialize current X-location
1196 mov currY,StartY ; Initialize current Y-location
1197
1198 Again:
1199 ; Insert the current location in the array.
1200 mov ax,currX
1201 mov (COORD PTR [edi]).X,ax ; Store X-coordinate
1202 mov ax,currY
1203 mov (COORD PTR [edi]).Y,ax ; Store Y-coordinate
1204
1205 INVOKE DisplayPosition, currX, currY ; Display the current position
1206
1207 mov eax,4 ; Choose a random direction (0-3)
1208 call RandomRange
```

```
1210 .IF eax == 0
1211     ; North
1212     dec currY
1213 .ELSEIF eax == 1
1214     ; South
1215     inc currY
1216 .ELSEIF eax == 2
1217     ; West
1218     dec currX
1219 .ELSE
1220     ; East (EAX = 3)
1221     inc currX
1222 .ENDIF
1223
1224 add edi,TYPE COORD ; Move to the next COORD
1225 loop Again
1226
1227 Finish:
1228 mov (DrunkardWalk PTR [esi]).pathsUsed, WalkMax ; Set pathsUsed to WalkMax
1229 popad ; Restore registers
1230 ret
1231 TakeDrunkenWalk ENDP
1232
```

```
1233 ;------------------------------------------------------
1234 DisplayPosition PROC currX:WORD, currY:WORD
1235 ; Display the current X and Y positions.
1236 ;------------------------------------------------------
1237
1238 .data
1239 commaStr BYTE ",",0 ; Comma string
1240
1241 .code
1242 pushad ; Preserve registers
1243
1244 ; Display the current X position
1245 movzx eax,currX
1246 call WriteDec
1247
1248 mov edx,OFFSET commaStr ; Load the comma string
1249 call WriteString
1250
1251 ; Display the current Y position
1252 movzx eax,currY
1253 call WriteDec
1254
1255 call Crlf ; Move to the next line
1256
1257 popad ; Restore registers
1258 ret
1259 DisplayPosition ENDP
1260
1261 END main
```

I'll provide an explanation of the "Drunkard's Walk" program, which simulates a professor's random path in an imaginary grid:

Include Directives:

The program begins with an INCLUDE directive to include the Irvine32.inc library, which provides functions for console I/O and other useful features. Constants:

Constants like WalkMax, StartX, and StartY are defined at the beginning. These constants determine the maximum number of steps in the walk and the starting coordinates of the professor.

Structure Definition:

The program defines a structure called DrunkardWalk using the STRUCT and ENDS directives. This structure contains two components: path: An array of COORD objects, which will store the professor's path. pathsUsed: A word to keep track of the number of steps taken.

Prototypes:

The DisplayPosition function's prototype is declared. It's used to display the current position (X and Y coordinates) of the professor. Data Section:

The .data section is used to define data, including a variable aWalk, which is an instance of the DrunkardWalk structure.

Code Section:

The .code section contains the program's main logic. main Procedure:

The main procedure is the program's entry point. It initializes esi with the address of the aWalk structure. Then, it calls the TakeDrunkenWalk procedure to simulate the professor's walk.

TakeDrunkenWalk Procedure:

This procedure simulates the professor's walk. It initializes local variables currX and currY to the starting coordinates and uses a loop to simulate the walk. Inside the loop: The current coordinates are inserted into the path array.

The DisplayPosition function is called to display the current position. A random direction (north, south, east, or west) is chosen for the next step. The loop continues until the maximum number of steps (WalkMax) is reached.

DisplayPosition Procedure:

This procedure displays the current X and Y positions with proper formatting. It uses WriteDec to display the coordinates and WriteString to add a comma between X and Y. It also adds a line break with Crlf to format the output.

Program Termination:

After the professor's walk is completed, the program sets the pathsUsed field of the aWalk structure to WalkMax to indicate the number of steps taken.

End of Program:

The END main statement marks the end of the program. In summary, this program uses a structure to simulate a random walk for a professor. It keeps track of the professor's path and displays the X and Y coordinates at each step.

The professor starts at a specified position, and the program randomly selects the next step's direction until the maximum number of steps is reached. The structure helps organize and manage the data related to the walk.

UNIONS IN ASSEMBLY

In assembly language, unions are used to group fields together at the same offset.

The size of a union is determined by its largest field.

To declare a union, you use the **UNION and ENDS directives**, like this:

```
1265  myUnion UNION
1266       field1 DWORD 0
1267       field2 WORD 0
1268       field3 BYTE 0
1269  myUnion ENDS
```

You don't have to indent.

Fields in a union follow the same rules as structures, but each field can have only one initializer.

Nesting Unions in Structures

You can nest a union inside a structure in two ways:

Declare the union inside the structure:

```
1277  MyStructure STRUCT
1278       UNION MyUnion
1279            field1 DWORD ?
1280            field2 WORD ?
1281            field3 BYTE ?
1282       ENDS
1283  MyField DWORD ?
1284  MyStructure ENDS
```

Use the union's name in a declaration inside the structure:

```
1290 MyStructure STRUCT
1291     MyUnion MyField
1292        field1 DWORD ?
1293        field2 WORD ?
1294        field3 BYTE ?
1295 MyStructure ENDS
```

Declaring and Using Union Variables:

Declaring a union variable is similar to declaring a structure variable.

However, you can only have one initializer:

```
myVar MyUnion <1234h>
```

To use a specific field of a union variable, you supply the field's name. For example:

```
mov myVar.field1, eax
mov myVar.field2, bx
```

Unions Can Contain Structures

Unions can also contain structures.

For instance, you can have a union that can hold different types of structures based on the situation. Example - KEY_EVENT_RECORD:

In the Windows API, there's a structure called KEY_EVENT_RECORD that contains a union named uChar. It's used to handle keyboard input events.

Depending on the situation, it can contain either a Unicode character or an ASCII character.

The union lets you use the appropriate character based on the event type.

Unions and structures are helpful for organizing data in assembly language programs.

Unions allow you to store data of different sizes in the same memory space, which can be useful in various situations.

MACROS IN ASSEMBLY

Certainly, I'll simplify and clarify the information about macros in assembly language.

Overview of Macros:

A macro is a named block of assembly language statements. It can be called multiple times in a program.

When you invoke a macro, a copy of its code is inserted directly into the program where it was invoked.

This is known as inline expansion, and there's no actual CALL instruction involved. Placement of Macros:

Macro definitions are usually placed at the beginning of the program's source code or in a separate file included with an INCLUDE directive.

Macros are expanded during the assembler's preprocessing step.

The assembler replaces every invocation of the macro with a copy of the macro's source code. If a program defines a macro but never calls it, the macro code does not appear in the compiled program. Defining Macros:

A macro is defined using the **MACRO and ENDM directives.**

The syntax is as follows:

```
1311 macroname MACRO parameter-1, parameter-2...
1312 statement-list
1313 ENDM
```

Parameters are named placeholders for arguments passed to the macro.

Parameters can be any text, integers, variable names, or other values. The preprocessor treats them as text.

Parameters don't have type information; type checking occurs during assembly.

Example - mPutchar Macro:

The mPutchar macro takes a single parameter named char. It pushes eax, moves the character char into al, and calls the WriteChar procedure.

Finally, it pops eax to restore the original value.

When you use this macro, it replaces the macro invocation with these statements, and the character is displayed on the console.

In essence, macros are a way to create reusable blocks of code that can be inserted directly into your program. They make your code more modular and easier to maintain.

```
1316 mPutchar MACRO char
1317     push eax        ; Push the value in the eax register
1318     mov al, char    ; Move the character 'char' into the al register
1319     call WriteChar  ; Call the WriteChar procedure to display the character
1320     pop eax         ; Pop the original value back into eax
1321 ENDM
1322
1323
1324 ;You'd use it like this:
1325
1326
1327 .code
1328 ; ... Your code here ...
1329 mPutchar 'A' ; Display the character 'A' on the console
1330 ; ... Your code here ...
```

Invoking Macros:

To use a macro, insert its name in your program, and you can provide arguments.

The syntax for invoking a macro is: **macroname argument-1, argument-2, ...** macroname must be a previously defined macro name in your source code.

Each argument replaces a parameter in the macro. The order of arguments must match the order of parameters.

You can pass a different number of arguments than the number of parameters in the macro. If you pass too many arguments, the assembler warns you.

If you pass too few, the unfilled parameters remain empty.

For example, if you have a macro called mPutchar that displays characters on the console, you can invoke it like this: mPutchar 'A'.

The macro call is expanded to the code that displays 'A' on the console.

Debugging Macros:

Debugging programs with macros can be challenging.

Check the listing file (.LST) after assembling to ensure that each macro is expanded as intended.

In the Visual Studio debugger, you can view the disassembly to see how each macro call is expanded into actual code. This can help with debugging.

Additional Macro Features:

You can use the REQ qualifier to specify that a macro parameter is required.

If the macro is invoked without an argument for a required parameter, the assembler shows an error message.

For example, in the mPutchar macro, you can specify that the char parameter is required:

```
mPutchar MACRO char:REQ
```

To exclude comments from appearing in macro expansions, use double semicolons (;;).

This way, comments within a macro definition won't show up when the macro is expanded. For example, you can add comments like this: **;; reminder: char must contain 8 bits** within your macro, and they won't appear in the expanded code.

In summary, invoking macros is like using predefined functions in your program. You provide arguments to the macro, which then replaces them with its predefined instructions.

Debugging macros can be done by checking the listing file and using the debugger. Additionally, you can specify required parameters and control which comments appear in macro expansions.

```
1340 mPutchar MACRO char:REQ
1341     push eax            ; Save the value in the eax register
1342     mov al, char        ; Move the character from 'char' to the al register
1343     call WriteChar      ; Call the WriteChar procedure to display the character
1344     pop eax             ; Restore the original value in eax
1345 ENDM
```

This macro essentially saves the value in eax, displays the character specified by the char argument, and then restores eax to its original value. Remember that the REQ qualifier indicates that the char parameter is required when invoking the macro.

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:

```
1363 makeString MACRO text
1364 LOCAL string
1365 .data
1366     string BYTE text,0
1367 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:

```
1375 mWrite MACRO text
1376     LOCAL string    ; Define a local label
1377     .data           ; Switch to the data section
1378     string BYTE text, 0   ; Define the string
1379
1380     .code           ; Switch back to the code section
1381     push edx        ; Push edx onto the stack
1382     mov edx, OFFSET string   ; Load the address of the string into edx
1383     call WriteString        ; Call the WriteString procedure with the address of the string
1384     pop edx         ; Restore the value of edx
1385 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 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:

```
1420  ; Using the Book's Macro Library (32-Bit Mode Only)
1421  ; To enable the library, include the following line after your existing INCLUDE:
1422  INCLUDE Macros.inc
1423
1424  ; Macros in the Macros.inc Library:
1425  ; mDump - Displays a variable using its name and default attributes.
1426  ; mDumpMem - Displays a range of memory.
1427  ; mGotoxy - Sets the cursor position in the console window buffer.
1428  ; mReadString - Reads a string from the keyboard.
1429  ; mShow - Displays a variable or register in various formats.
1430  ; mShowRegister - Displays a 32-bit register's name and contents in hexadecimal.
1431  ; mWrite - Writes a string literal to the console window.
1432  ; mWriteSpace - Writes one or more spaces to the console window.
1433  ; mWriteString - Writes a string variable's contents to the console window.
1434
1435  ; Example usage of the mDumpMem macro:
1436  ; mDumpMem OFFSET array, LENGTHOF array, TYPE array
1437  ; 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.

===

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:

```
1583 .data
1584     str1 BYTE "Please enter your name: ",0
1585 .code
1586     mWriteString str1
```

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

```
1590 push edx
1591 mov edx,OFFSET str1
1592 call WriteString
1593 pop edx
```

This code will then be expanded to the following instructions:

```
1595 push edx
1596 mov edx,OFFSET str1
1597 call WriteString
1598 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

```
1610 INCLUDE Irvine32.inc
1611 INCLUDE Macros.inc
1612 .data
1613     array DWORD 1,2,3,4,5,6,7,8
1614     firstName BYTE 31 DUP(?)
1615     lastName BYTE 31 DUP(?)
1616 .code
1617     main PROC
1618        mGotoxy 0,0          ; Position cursor
1619        ; Display heading
1620        mWrite <"Sample Macro Program",0dh,0ah>
1621        ; Input first name
1622        mGotoxy 0,5
1623        mWrite "Please enter your first name: "
1624        mReadString firstName
1625        ; Input last name
1626        call Crlf
1627        mWrite "Please enter your last name: "
1628        mReadString lastName
1629        ; Display name
1630        call Crlf
1631        mWrite "Your name is "
1632        mWriteString firstName
1633        mWriteSpace
1634        mWriteString lastName
1635        ; Display array
1636        call Crlf
1637        mDumpMem OFFSET array, LENGTHOF array, TYPE array
1638        exit
1639     main ENDP
1640     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.

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:

```
1697 ; mWriteln Macro
1698 ; This macro writes a string to the console followed by a carriage return and line feed (CRLF).
1699
1700 mWriteln MACRO text:=<" ">
1701     ; Check if a text argument is provided. If not, use a space as the default.
1702     IFB <text>
1703         mWrite " " ; Display a space if no text is provided
1704     ELSE
1705         mWrite text ; Display the provided text
1706     ENDIF
1707
1708     ; Call the Crlf procedure to move to the next line.
1709     call Crlf
1710 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.

E

XITM:

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.

```
1815 ; Define the mCalc_row_sum macro
1816 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1817      LOCAL L1
1818
1819      ; Save registers
1820      push ebx
1821      push ecx
1822      push esi
1823
1824      ; Set up required registers
1825      mov eax, index
1826      mov ebx, arrayOffset
1827      mov ecx, rowSize
1828
1829      ; Calculate the row offset
1830      mul ecx                  ; row index * row size
1831      add ebx, eax             ; row offset
1832
1833      ; Prepare the loop counter
1834      shr ecx, (TYPE eltType / 2)  ; Calculate the number of elements in a row
1835
1836      ; Initialize accumulator and column index
1837      mov eax, 0               ; Accumulator
1838      mov esi, 0               ; Column index
1839
```

```
1840      L1:
1841      ; Check if eltType is DWORD
1842      IFIDNI <eltType>, <DWORD>
1843          mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
1844      ELSE
1845          ; Assuming eltType is BYTE or WORD
1846          movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
1847      ENDIF
1848
1849      add eax, edx             ; Add to accumulator
1850      inc esi                  ; Move to the next column
1851
1852      loop L1
1853
1854      ; Restore registers
1855      pop esi
1856      pop ecx
1857      pop ebx
1858
1859 ENDM
1860
```

```
1861 .data
1862 ; Define data arrays
1863 tableB BYTE 10h, 20h, 30h, 40h, 50h
1864 RowSizeB = ($ - tableB)
1865 tableW WORD 10h, 20h, 30h, 40h, 50h
1866 RowSizeW = ($ - tableW)
1867 tableD DWORD 10h, 20h, 30h, 40h, 50h
1868 RowSizeD = ($ - tableD)
1869 index DWORD ?
1870
1871 .code
1872 ; Call the mCalc_row_sum macro for different arrays and data types
1873 mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
1874 mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
1875 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.

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.

```
1880  .code
1881  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:

```
1885  mShowRegister MACRO regName
1886  .data
1887  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:

```
1895 count = 10
1896 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:

```
1905 TempStr TEXTEQU %(SIZEOF array)
1906 %
1907 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:

```
1930 ShowWarning MACRO message
1931 mWrite "&message"
1932 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:

```
1940 IsDefined MACRO symbol
1941     IFDEF symbol
1942         EXITM <-1> ;; True
1943     ELSE
1944         EXITM <0> ;; False
1945     ENDIF
1946 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:

```
1950 IF IsDefined(RealMode)
1951     mov ax, @data
1952     mov ds, ax
1953 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:

```
1970 Startup MACRO
1971     IF IsDefined(RealMode)
1972         mov ax, @data
1973         mov ds, ax
1974     ENDIF
1975 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.

REPEAT BLOCKS

MASM provides several looping directives for generating repeated blocks of statements: WHILE, REPEAT, FOR, and FORC.

These directives operate at assembly time and use constant values for loop conditions and counters.

WHILE Directive:

The WHILE directive repeats a block of statements as long as a specific constant expression remains true. It has the following syntax:

```
1987 WHILE constExpression
1988    statements
1989 ENDM
```

For example, you can use the WHILE directive to generate Fibonacci numbers within a specific range, like so:

```
1992 .data
1993     val1 = 1
1994     val2 = 1
1995     DWORD val1 ; First two values
1996     DWORD val2
1997     val3 = val1 + val2
1998
1999 WHILE val3 LT 0F0000000h
2000    DWORD val3
2001    val1 = val2
2002    val2 = val3
2003    val3 = val1 + val2
2004 ENDM
```

This code generates Fibonacci numbers and stores them as assembly-time constants until the value exceeds 0F0000000h.

REPEAT Directive:

The REPEAT directive repeats a statement block a fixed number of times at assembly time, based on an unsigned constant integer expression.

```
2008 REPEAT constExpression
2009    statements
2010 ENDM
```

It's used when you need to repeat a block of code a predetermined number of times, similar to the DUP directive.

REPEAT Directive Example: Creating an Array

In this example, we use the REPEAT directive to create an array of WeatherReadings. Each WeatherReadings struct contains a location string and arrays for rainfall and humidity readings. The loop repeats for a total of WEEKS_PER_YEAR times:

```
2050 WEEKS_PER_YEAR = 52
2051
2052 WeatherReadings STRUCT
2053     location BYTE 50 DUP(0)
2054     REPEAT WEEKS_PER_YEAR
2055         LOCAL rainfall, humidity
2056         rainfall DWORD ?
2057         humidity DWORD ?
2058     ENDM
2059 WeatherReadings ENDS
```

This code defines a structured array for recording weather readings over the course of a year.

FOR Directive:

The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols. Each symbol in the list represents one iteration of the loop.

```
2015 FOR parameter, <arg1, arg2, arg3, ...>
2016    statements
2017 ENDM
```

It's useful when you want to perform a set of operations for each item in a list of symbols.

FOR Directive Example: Student Enrollment

In this example, the FOR directive is used to create multiple SEMESTER objects for student enrollment in different semesters. The loop iterates over a list of semester names and generates corresponding SEMESTER objects:

```
2065 .data
2066 FOR semName, <Fall2013, Spring2014, Summer2014, Fall2014>
2067     semName SEMESTER <>
2068 ENDM
```

This code generates SEMESTER objects with different names for each semester.

FORC Directive:

The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string represents one iteration of the loop.

```
2020 FORC parameter, <string>
2021    statements
2022 ENDM
```

It's handy when you need to process a block of code for each character in a string.

Student Enrollment:

You can use the FOR directive to create multiple SEMESTER objects, each with a different name from a list of symbols. This can be useful for managing student enrollments over multiple semesters.

Character Lookup Table:

The FORC directive can be used to generate a character lookup table. In this example, a table of non-alphabetic characters is created by iterating through a string of special characters. These looping directives offer flexibility and structure for generating repetitive code in assembly language programs, making it easier to manage and control complex operations.

FORC Directive Example: Character Lookup Table

In this example, the FORC directive is used to create a character lookup table for non-alphabetic characters. Each character in the string is processed to generate a corresponding entry in the lookup table:

```
2070 Delimiters LABEL BYTE
2071 FORC code, <@#$%^&*!<!>>
2072     BYTE "&code"
2073 ENDM
```

This code generates a lookup table containing the ASCII values and corresponding characters for various special symbols.

In this example, we create a linked list data structure using the ListNode structure, which contains a data area (NodeData) and a pointer to the next node (NextPtr).

The program defines and populates multiple instances of ListNode objects within a loop to create a linked list.

Here's the revised and expanded code with added explanations:

```
2077  INCLUDE Irvine32.inc
2078
2079  ; Define the ListNode structure
2080  ListNode STRUCT
2081      NodeData DWORD ?
2082      NextPtr DWORD ?
2083  ListNode ENDS
2084
2085  TotalNodeCount = 15
2086  NULL = 0
2087  Counter = 0
2088
2089  .data
2090  LinkedList LABEL PTR ListNode
2091
2092  ; Use the REPEAT directive to create a linked list
2093  REPEAT TotalNodeCount
2094      Counter = Counter + 1
2095      ; Create a new ListNode with data and link to the next node
2096      ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
2097  ENDM
2098
2099  ; Create a tail node to mark the end of the list
2100  ListNode <0, 0>
2101
2102  .code
2103  main PROC
2104      mov esi, OFFSET LinkedList  ; Initialize the pointer to the start of the list
```

```
2105
2106 NextNode:
2107      ; Check for the tail node (end of the list).
2108      mov eax, (ListNode PTR [esi]).NextPtr
2109      cmp eax, NULL
2110      je quit  ; If NextPtr is NULL, exit the loop
2111
2112      ; Display the node data.
2113      mov eax, (ListNode PTR [esi]).NodeData
2114      call WriteDec  ; Display the integer value
2115      call Crlf      ; Move to the next line for the next value
2116
2117      ; Get the pointer to the next node.
2118      mov esi, (ListNode PTR [esi]).NextPtr
2119      jmp NextNode
2120
2121 quit:
2122      exit
2123
2124 main ENDP
2125
2126 END main
```

Program to illustrate what we have learnt above:

```asm
INCLUDE Irvine32.inc

; Define a macro that generates bytes based on a list of values
mGenerateBytes MACRO values
    LOCAL L1
    FOR val, <values>
        BYTE val
    ENDM
ENDM

.data
byteArray BYTE 0, 0, 0, 0

.code
main PROC
    ; Question 7 - Macro Expansion
    ; a
    mGenerateBytes <100, 20, 30>
    mov eax, OFFSET byteArray
    call DisplayByteArray

    ; b
    mGenerateBytes <AL, 20>
    mov eax, OFFSET byteArray
    call DisplayByteArray
```

```asm
2158        ; c
2159        byteVal = 42
2160        countVal = 5
2161        mGenerateBytes <byteVal, countVal>
2162        mov eax, OFFSET byteArray
2163        call DisplayByteArray
2164
2165        ; Question 8 - Linked List Scenario
2166        TotalNodeCount = 15
2167        NULL = 0
2168        Counter = 0
2169        LinkedList LABEL PTR ListNode
2170
2171        REPEAT TotalNodeCount
2172            Counter = Counter + 1
2173            ListNode <Counter, ($ + SIZEOF ListNode)>
2174        ENDM
2175
2176        ; Traverse and display the linked list
2177        mov esi, OFFSET LinkedList
2178
```

```
2179 NextNode:
2180     mov eax, (ListNode PTR [esi]).NextPtr
2181     cmp eax, NULL
2182     je quit
2183
2184     mov eax, (ListNode PTR [esi]).NodeData
2185     call WriteDec
2186     call Crlf
2187
2188     mov esi, (ListNode PTR [esi]).NextPtr
2189     jmp NextNode
2190
2191 quit:
2192     exit
2193
2194 main ENDP
2195
2196 ; Helper function to display the contents of the byteArray
2197 DisplayByteArray PROC
2198     mov ecx, LENGTHOF byteArray
2199     mov esi, 0
2200     mov edx, OFFSET byteArray
2201
2202 DisplayLoop:
2203     mov al, [edx + esi]
2204     call WriteHex
2205     call Crlf
2206     inc esi
2207     loop DisplayLoop
2208     ret
2209 DisplayByteArray ENDP
```

2210 It's gonna be **END main** you know...👊