Macros

- Introducing Macros
- Defining Macros
- Invoking Macros
- Macro Examples
- Nested Macros
- Example Program: Wrappers

Introducing Macros

- A macro[1] is a named block of assembly language statements.
- Once defined, it can be invoked (called) one or more times.
- During the assembler's preprocessing step, each macro call is expanded into a copy of the macro.
- The expanded code is passed to the assembly step, where it is checked for correctness.

[1]Also called a macro procedure.

Defining Macros

- A macro must be defined before it can be used.
- Parameters are optional.
- Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Syntax:

```
macroname MACRO [parameter-1, parameter-2,...]
    statement-list
ENDM
```

mNewLine Macro Example

This is how you define and invoke a simple macro.

```
mNewLine MACRO                          ; define the macro
   call Crlf
ENDM
.data

.code
mNewLine                                ; invoke the macro
```

The assembler will substitute "call crlf" for "mNewLine".

mPutChar Macro

Writes a single character to standard output.

Definition:

```
mPutchar MACRO char
    push eax
    mov al,char
    call WriteChar
    pop eax
ENDM
```

Invocation:

```
.code
mPutchar 'A'
```

Expansion:

```
1      push eax
1      mov al,'A'
1      call WriteChar
1      pop eax
```

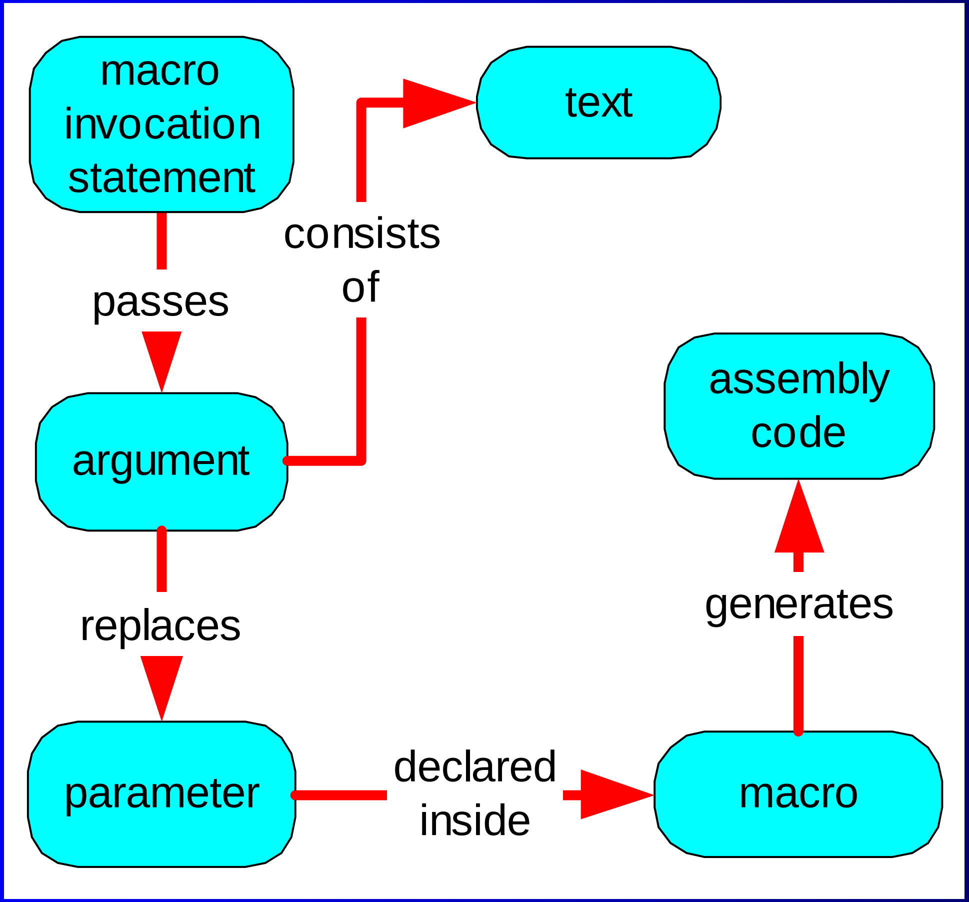
viewed in the listing file

Invoking Macros (1 of 2)

- When you invoke a macro, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code.
- Arguments are treated as simple text by the preprocessor.

Invoking Macros (2 of 2)

Relationships between macros, arguments, and parameters:



mWriteStr Macro (1 of 2)

Provides a convenient way to display a string, by passing the string name as an argument.

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
.data
str1 BYTE "Welcome!",0
.code
mWriteStr str1
```

mWriteStr Macro (2 of 2)

The expanded code shows how the str1 argument replaced the parameter named buffer:

```
mWriteStr MACRO buffer
     push edx
     mov  edx,OFFSET buffer
     call WriteString
     pop  edx
ENDM
```

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

Invalid Argument

- If you pass an invalid argument, the error is caught when the expanded code is assembled.
- Example:

```
.code
mPutchar 1234h
```

```
1       push eax
1       mov al,1234h            ; error!
1       call WriteChar
1       pop eax
```

Blank Argument

- If you pass a blank argument, the error is also caught when the expanded code is assembled.
- Example:

```
.code
mPutchar
```

```
1       push eax
1       mov al,
1       call WriteChar
1       pop eax
```

Macro Examples

- mReadStr - reads string from standard input
- mGotoXY - locates the cursor on screen
- mDumpMem - dumps a range of memory

mReadStr

The mReadStr macro provides a convenient wrapper around ReadString procedure calls.

```
mReadStr MACRO varName
    push ecx
    push edx
    mov edx,OFFSET varName
    mov ecx,(SIZEOF varName) - 1
    call ReadString
    pop edx
    pop ecx
ENDM
.data
firstName BYTE 30 DUP(?)
.code
mReadStr firstName
```

mGotoXY

The mGotoXY macro ets the console cursor position by calling the Gotoxy library procedure.

```
mGotoxy MACRO X:REQ, Y:REQ
   push edx
   mov  dh,Y
   mov  dl,X
   call Gotoxy
   pop  edx
ENDM
```

The REQ next to X and Y identifies them as required parameters.

mDumpMem

The mDumpMem macro streamlines calls to the link library's DumpMem procedure.

```
mDumpMem MACRO address, itemCount, componentSize
    push ebx
    push ecx
    push esi
    mov  esi,address
    mov  ecx,itemCount
    mov  ebx,componentSize
    call DumpMem
    pop  esi
    pop  ecx
    pop  ebx
ENDM
```

mWrite

The mWrite macro writes a string literal to standard output. It is a good example of a macro that contains both code and data.

```
mWrite MACRO text
    LOCAL string
    .data                          ;; data segment
    string BYTE text,0             ;; define local string
    .code                          ;; code segment
    push edx
    mov  edx,OFFSET string
    call Writestring
    pop  edx
ENDM
```

The LOCAL directive prevents string from becoming a global label.

Nested Macros

- The mWriteLn macro contains a nested macro (a macro invoked by another macro).

```
mWriteLn MACRO text
    mWrite text
    call Crlf
ENDM
```

```
mWriteLn "My Sample Macro Program"
```

```
2   .data
2   ??0002 BYTE "My Sample Macro Program",0
2   .code
2   push edx
2   mov  edx,OFFSET ??0002
2   call Writestring
2   pop  edx
1   call Crlf
```

nesting level

Your turn . . .

- Write a nested macro that clears the screen, locates the cursor at a given row and column, asks the user to enter an account number, and inputs the account number. Use any macros shown so far.
- Use the following data to test your macro:

```
.data
acctNum BYTE 30 DUP(?)
.code
main proc
   mAskForString 5,10,"Input Account Number: ", \
        acctNum
```

Solution . . .

. . . Solution

```
mAskForString MACRO row,col,prompt,inbuf
    call Clrscr
    mGotoXY col,row
    mWrite prompt
    mReadStr inbuf
ENDM
```

Example Program: Wrappers

- Demonstrates various macros from this chapter
- Shows how macros can simplify argument passing
- View the source code

Conditional-Assembly Directives

- Checking for Missing Arguments
- Default Argument Initializers
- Boolean Expressions
- IF, ELSE, and ENDIF Directives
- The IFIDN and IFIDNI Directives
- Special Operators
- Macro Functions

Checking for Missing Arguments

- The IFB directive returns true if its argument is blank. For example:

```
IFB <row>                 ;; if row is blank,
  EXITM                   ;; exit the macro
ENDIF
```

mWriteString Example

Display a message during assembly if the string parameter is empty:

```
mWriteStr MACRO string
    IFB <string>
        ECHO -----------------------------------------
        ECHO * Error: parameter missing in mWriteStr
        ECHO * (no code generated)
        ECHO -----------------------------------------
        EXITM
    ENDIF
    push edx
    mov edx,OFFSET string
    call WriteString
    pop edx
ENDM
```

Default Argument Initializers

- A default argument initializer automatically assigns a value to a parameter when a macro argument is left blank. For example, mWriteln can be invoked either with or without a string argument:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call Crlf
ENDM
.code
mWriteln "Line one"
mWriteln
mWriteln "Line three"
```

Sample output:

```
Line one

Line three
```

Boolean Expressions

A boolean expression can be formed using the following operators:

- LT - Less than
- GT - Greater than
- EQ - Equal to
- NE - Not equal to
- LE - Less than or equal to
- GE - Greater than or equal to

Only assembly-time constants may be compared using these operators.

IF, ELSE, and ENDIF Directives

A block of statements is assembled if the boolean expression evaluates to true. An alternate block of statements can be assembled if the expression is false.

```
IF boolean-expression
     statements
[ELSE
     statements]
ENDIF
```

Simple Example

The following IF directive permits two MOV instructions to be assembled if a constant named RealMode is equal to 1:

```
IF RealMode EQ 1
  mov ax,@data
  mov ds,ax
ENDIF
```

RealMode can be defined in the source code any of the following ways:

```
RealMode = 1

RealMode EQU 1

RealMode TEXTEQU 1
```

The IFIDN and IFIDNI Directives

- IFIDN compares two symbols and returns true if they are equal (case-sensitive)
- IFIDNI also compares two symbols, using a case-insensitive comparison
- Syntax:

```
IFIDNI <symbol>, <symbol>
      statements
ENDIF
```

Can be used to prevent the caller of a macro from passing an argument that would conflict with register usage inside the macro.

IFIDNI Example

Prevents the user from passing EDX as the second argument to the mReadBuf macro:

```
mReadBuf MACRO bufferPtr, maxChars
  IFIDNI <maxChars>,<EDX>
    ECHO Warning: Second argument cannot be EDX
    ECHO **************************************
    EXITM
  ENDIF
  .
  .
ENDM
```

Special Operators

- The substitution (&) operator resolves ambiguous references to parameter names within a macro.
- The expansion operator (%) expands text macros or converts constant expressions into their text representations.
- The literal-text operator (<>) groups one or more characters and symbols into a single text literal. It prevents the preprocessor from interpreting members of the list as separate arguments.
- The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.

Substitution (&)

Text passed as regName is substituted into the literal string definition:

```
ShowRegister MACRO regName
.data
tempStr BYTE " &regName=",0
.
.
.code
ShowRegister EDX           ; invoke the macro
```

Macro expansion:

```
tempStr BYTE " EDX=",0
```

Expansion (%)

Forces the evaluation of an integer expression. After the expression has been evaluated, its value is passed as a macro argument:

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

The preprocessor generates the following code:

1 push edx
1 mov dl,50
1 mov dh,7
1 call Gotoxy
1 pop edx
```

Literal-Text (<>)

The first macro call passes three arguments. The second call passes a single argument:

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

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

Literal-Character (!)

The following declaration prematurely ends the text definition when the first > character is reached.

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

The following declaration continues the text definition until the final > character is reached.

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

Macro Functions (1 of 2)

- A macro function returns an integer or string constant
- The value is returned by the EXITM directive
- Example: The IsDefined macro acts as a wrapper for the IFDEF directive.

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

Notice how the assembler defines True and False.

Macro Functions (2 of 2)

- When calling a macro function, the argument(s) must be enclosed in parentheses
- The following code permits the two MOV statements to be assembled only if the RealMode symbol has been defined:

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

Defining Repeat Blocks

- WHILE Directive
- REPEAT Directive
- FOR Directive
- FORC Directive
- Example: Linked List

WHILE Directive

- The WHILE directive repeats a statement block as long as a particular constant expression is true.
- Syntax:

```
WHILE constExpression
    statements
ENDM
```

WHILE Example

Generates Fibonacci integers between 1 and F0000000h at assembly time:

```
.data
val1 = 1
val2 = 1
DWORD val1                          ; first two values
DWORD val2
val3 = val1 + val2

WHILE val3 LT 0F0000000h
   DWORD val3
   val1 = val2
   val2 = val3
   val3 = val1 + val2
ENDM
```

REPEAT Directive

- The REPEAT directive repeats a statement block a fixed number of times.
- Syntax:

```
REPEAT constExpression
    statements
ENDM
```

ConstExpression, an unsigned constant integer expression, determines the number of repetitions.

REPEAT Example

The following code generates 100 integer data definitions in the sequence 10, 20, 30, . . .

```
iVal = 10
REPEAT 100
    DWORD iVal
    iVal = iVal + 10
ENDM
```

How might we assign a data name to this list of integers?

Your turn . . .

What will be the last integer to be generated by the following loop? 500

```
rows = 10
columns = 5
.data
iVal = 10
REPEAT rows * columns
    DWORD iVal
    iVal = iVal + 10
ENDM
```

FOR Directive

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

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

FOR Example

The following Window structure contains frame, title bar, background, and foreground colors. The field definitions are created using a FOR directive:

```
Window STRUCT
  FOR color,<frame,titlebar,background,foreground>
     color DWORD ?
  ENDM
Window ENDS
```

Generated code:

```
Window STRUCT
   frame DWORD ?
   titlebar DWORD ?
    background DWORD ?
    foreground DWORD ?
Window ENDS
```

FORC Directive

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

```
FORC parameter, <string>
    statements
ENDM
```

FORC Example

Suppose we need to accumulate seven sets of integer data for an experiment. Their label names are to be Group_A, Group_B, Group_C, and so on. The FORC directive creates the variables:

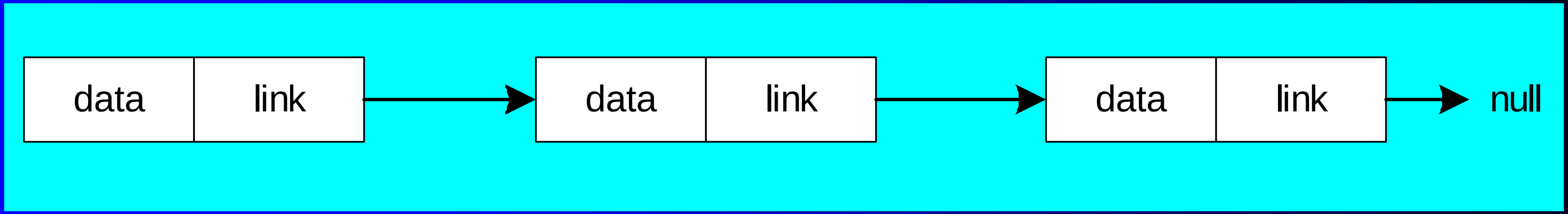
```
FORC code,<ABCDEFG>
   Group_&code WORD ?
ENDM
```

Generated code:

```
  Group_A WORD ?
  Group_B WORD ?
  Group_C WORD ?
  Group_D WORD ?
  Group_E WORD ?
  Group_F WORD ?
  Group_G WORD ?
```

Example: Linked List (1 of 5)

- We can use the REPT directive to create a singly linked list at assembly time.
- Each node contains a pointer to the next node.
- A null pointer in the last node marks the end of the list



Linked List (2 of 5)

- Each node in the list is defined by a ListNode structure:

```
ListNode STRUCT
    NodeData DWORD ?          ; the node's data
    NextPtr  DWORD ?          ; pointer to next node
ListNode ENDS

TotalNodeCount = 15
NULL = 0
Counter = 0
```

Linked List (3 of 5)

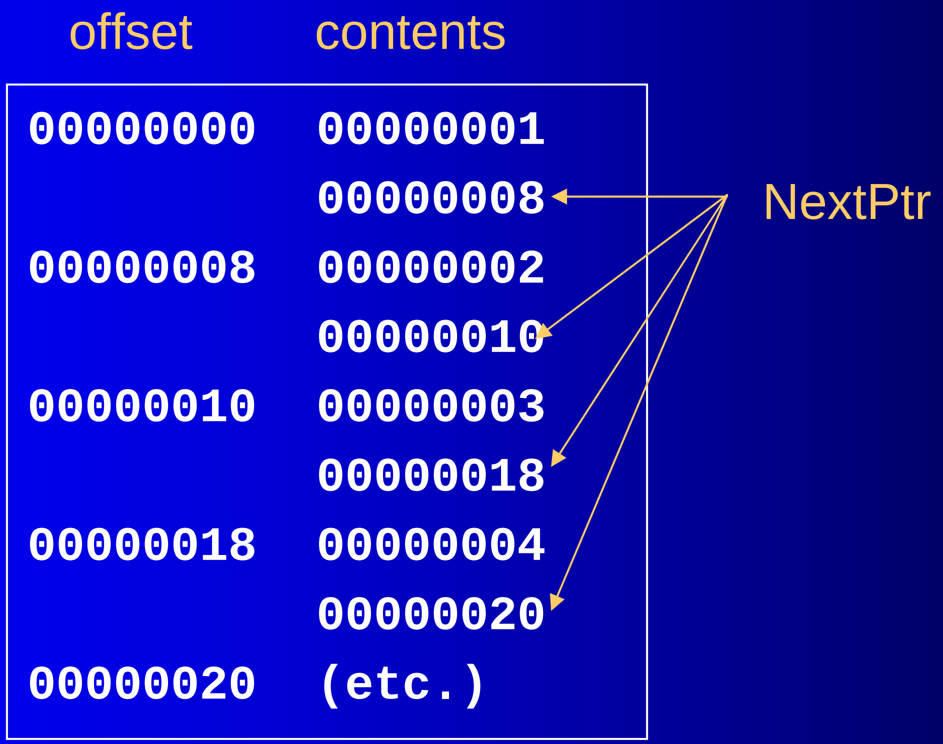
- The REPEAT directive generates the nodes.
- Each ListNode is initialized with a counter and an address that points 8 bytes beyond the current node's location:

```
.data
LinkedList LABEL DWORD
REPEAT TotalNodeCount
   Counter = Counter + 1
   ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
ENDM
```

The value of $ does not change—it remains fixed at the location of the LinkedList label.

Linked List (4 of 5)

The following hexadecimal values in each node show how each NextPtr field contains the address of its following node.



offset	contents
00000000	00000001
	00000008
00000008	00000002
	00000010
00000010	00000003
	00000018
00000018	00000004
	00000020
00000020	(etc.)

Linked List (5 of 4)

View the program's source code

Sample output:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```