CHAPTER 5

Defining and Using Complex Data Types

With the complex data types available in MASM 6.1—arrays, strings, records, structures, and unions—you can access data as a unit or as individual elements that make up a unit. The individual elements of complex data types are often the integer types discussed in Chapter 4, "Defining and Using Simple Data Types."

"Arrays and Strings" reviews how to declare, reference, and initialize arrays and strings. This section summarizes the general steps needed to process arrays and strings and describes the MASM instructions for moving, comparing, searching, loading, and storing.

"Structures and Unions" covers similar information for structures and unions: how to declare structure and union types, how to define structure and union variables, and how to reference structures and unions and their fields.

"Records" explains how to declare record types, define record variables, and use record operators.

Arrays and Strings

An array is a sequential collection of variables, all of the same size and type, called "elements." A string is an array of characters. For example, in the string "ABC," each letter is an element. You can access the elements in an array or string relative to the first element. This section explains how to handle arrays and strings in your programs.

Declaring and Referencing Arrays

Array elements occupy memory contiguously, so a program references each element relative to the start of the array. To declare an array, supply a label name, the element type, and a series of initializing values or **?** placeholders. The following examples declare the arrays `warray` and `xarray`:

```
warray  WORD    1, 2, 3, 4
xarray  DWORD   0FFFFFFFFh, 789ABCDEh
```

Initializer lists of array declarations can span multiple lines. The first initializer must appear on the same line as the data type, all entries must be initialized, and, if you want the array to continue to the new line, the line must end with a comma. These examples show legal multiple-line array declarations:

```
big             BYTE    21, 22, 23, 24, 25,
                        26, 27, 28

somelist        WORD    10,
                        20,
                        30
```

If you do not use the **LENGTHOF** and **SIZEOF** operators discussed later in this section, an array may span more than one logical line, although a separate type declaration is needed on each logical line:

```
var1    BYTE    10, 20, 30
        BYTE    40, 50, 60
        BYTE    70, 80, 90
```

The DUP Operator

You can also declare an array with the **DUP** operator. This operator works with any of the data allocation directives described in "Allocating Memory for Integer Variables" in Chapter 4. In the syntax

count **DUP (***initialvalue* [[**,** *initialvalue*]]...**)**

the *count* value sets the number of times to repeat all values within the parentheses. The *initialvalue* can be an integer, character constant, or another **DUP** operator, and must always appear within parentheses. For example, the statement

```
barray  BYTE    5 DUP (1)
```

allocates the integer 1 five times for a total of 5 bytes.

The following examples show various ways to allocate data elements with the **DUP** operator:

```
array   DWORD   10 DUP (1)                    ; 10 doublewords
                                              ;   initialized to 1
buffer  BYTE    256 DUP (?)                   ; 256-byte buffer

masks   BYTE    20 DUP (040h, 020h, 04h, 02h) ; 80-byte buffer
                                              ;   with bit masks
three_d DWORD   5 DUP (5 DUP (5 DUP (0)))     ; 125 doublewords
                                              ;   initialized to 0
```

Referencing Arrays

Each element in an array is referenced with an index number, beginning with zero. The array index appears in brackets after the array name, as in

```
array[9]
```

Assembly-language indexes differ from indexes in high-level languages, where the index number always corresponds to the element's position. In C, for example, `array[9]` references the array's tenth element, regardless of whether each element is 1 byte or 8 bytes in size.

In assembly language, an element's index refers to the number of bytes between the element and the start of the array. This distinction can be ignored for arrays of byte-sized elements, since an element's position number matches its index. For example, defining the array

```
prime  BYTE 1, 3, 5, 7, 11, 13, 17
```

gives a value of 1 to `prime[0]`, a value of 3 to `prime[1]`, and so forth.

However, in arrays with elements larger than 1 byte, index numbers (except zero) do not correspond to an element's position. You must multiply an element's position by its size to determine the element's index. Thus, for the array

```
wprime  WORD 1, 3, 5, 7, 11, 13, 17
```

`wprime[4]` represents the third element (5), which is 4 bytes from the beginning of the array. Similarly, the expression `wprime[6]` represents the fourth element (7) and `wprime[10]` represents the sixth element (13).

The following example determines an index at run time. It multiplies the position by two (the size of a word element) by shifting it left:

```
mov     si, cx          ; CX holds position number
shl     si, 1           ; Scale for word referencing
mov     ax, wprime[si]  ; Move element into AX
```

The offset required to access an array element can be calculated with the following formula:

*n*th element of array = array[(*n*-1) * size of element]

Referencing an array element by distance rather than position is not difficult to master, and is actually very consistent with how assembly language works. Recall that a variable name is a symbol that represents the contents of a particular address in memory. Thus, if the array `wprime` begins at address DS:2400h, the reference `wprime[6]` means to the processor "the word value contained in the DS segment at offset 2400h-plus-6-bytes."

As described in "Direct Memory Operands," Chapter 3, you can substitute the plus operator (+) for brackets, as in:

```
wprime[9]
wprime+9
```

Since brackets simply add a number to an address, you don't need them when referencing the first element. Thus, `wprime` and `wprime[0]` both refer to the first element of the array `wprime`.

If your program runs only on an 80186 processor or higher, you can use the **BOUND** instruction to verify that an index value is within the bounds of an array. For a description of **BOUND**, see the *Reference*.

LENGTHOF, SIZEOF, and TYPE for Arrays

When applied to arrays, the **LENGTHOF**, **SIZEOF**, and **TYPE** operators return information about the length and size of the array and about the type of the
initializers.

The **LENGTHOF** operator returns the number of elements in the array. The **SIZEOF** operator returns the number of bytes used by the initializers in the array definition. **TYPE** returns the size of the elements of the array. The following examples illustrate these operators:

```
array   WORD    40 DUP (5)

larray  EQU     LENGTHOF array    ; 40 elements
sarray  EQU     SIZEOF   array    ; 80 bytes
tarray  EQU     TYPE     array    ;  2 bytes per element

num     DWORD   4, 5, 6, 7, 8, 9, 10, 11

lnum    EQU     LENGTHOF num      ;  8 elements
snum    EQU     SIZEOF   num      ; 32 bytes
tnum    EQU     TYPE     num      ;  4 bytes per element

warray  WORD    40 DUP (40 DUP (5))

len     EQU     LENGTHOF warray   ; 1600 elements
siz     EQU     SIZEOF   warray   ; 3200 bytes
typ     EQU     TYPE     warray   ;    2 bytes per element
```

Declaring and Initializing Strings

A string is an array of characters. Initializing a string like `"Hello, there"` allocates and initializes 1 byte for each character in the string. An initialized string can be no longer than 255 characters.

For data directives other than **BYTE**, a string may initialize only the first element. The initializer value must fit into the specified size and conform to the expression word size in effect (see “Integer Constants and Constant Expressions” in Chapter 1), as shown in these examples:

```
wstr    WORD    "OK"
dstr    DWORD   "DATA"  ; Legal under EXPR32 only
```

As with arrays, string initializers can span multiple lines. The line must end with a comma if you want the string to continue to the next line.

```
str1    BYTE    "This is a long string that does not ",
                "fit on one line."
```

You can also have an array of pointers to strings.

```
PBYTE   TYPEDEF PTR BYTE
        .DATA
msg1    BYTE    "Operation completed successfully."
msg2    BYTE    "Unknown command"
msg3    BYTE    "File not found"
pmsg    PBYTE   msg1            ; pmsg is an array
        PBBYTE  msg2            ;   of pointers to
        PBYTE   msg3            ;   above messages
```

Strings must be enclosed in single (') or double (") quotation marks. To put a single quotation mark inside a string enclosed by single quotation marks, use two single quotation marks. Likewise, if you need quotation marks inside a string enclosed by double quotation marks, use two sets. These examples show the various uses of quotation marks:

```
char    BYTE    'a'
message BYTE    "That's the message."       ; That's the message.
warn    BYTE    'Can''t find file.'         ; Can't find file.
string  BYTE    "This ""value"" not found." ; This "value" not found.
```

You can always use single quotation marks inside a string enclosed by double quotation marks, as the initialization for `message` shows, and vice versa.

The ? Initializer

You do not have to initialize an array. The **?** operator lets you allocate space for the array without placing specific values in it. Object files contain records for initialized data. Unspecified space left in the object file means that no records contain initialized data for that address. The actual values stored in arrays allocated with **?** depend on certain conditions. The **?** initializer is treated as a zero in a **DUP** statement that contains initializers in addition to the **?** initializer. If the **?** initializer does not appear in a **DUP** statement, or if the **DUP** statement contains only **?** initializers, the assembler leaves the allocated space unspecified.

LENGTHOF, SIZEOF, and TYPE for Strings

Because strings are simply arrays of byte elements, the **LENGTHOF**, **SIZEOF**, and **TYPE** operators behave as you would expect, as illustrated in this example:

```
msg     BYTE    "This string extends ",
                "over three ",
                "lines."

lmsg    EQU     LENGTHOF msg    ; 37 elements
smsg    EQU     SIZEOF   msg    ; 37 bytes
tmsg    EQU     TYPE     msg    ;  1 byte per element
```

Processing Strings

The 8086-family instruction set has seven string instructions for fast and efficient processing of entire strings and arrays. The term "string" in "string instructions" refers to a sequence of elements, not just character strings. These instructions work directly only on arrays of bytes and words on the 8086–80486 processors, and on arrays of bytes, words, and doublewords on the 80386/486 processors. Processing larger elements must be done indirectly with loops.

The following list gives capsule descriptions of the five instructions discussed in this section.

Instruction	Description
MOVS	Copies a string from one location to another
STOS	Stores contents of the accumulator register to a string
CMPS	Compares one string with another
LODS	Loads values from a string to the accumulator register
SCAS	Scans a string for a specified value

All of these instructions use registers in a similar way and have a similar syntax. Most are used with the repeat instruction prefixes **REP**, **REPE** (or **REPZ**), and **REPNE** (or **REPNZ**). **REPZ** is a synonym for **REPE** (Repeat While Equal) and **REPNZ** is a synonym for **REPNE** (Repeat While Not Equal).

This section first explains the general procedures for using all string instructions. It then illustrates each instruction with an example.

Overview of String Instructions

The string instructions have specific requirements for the location of strings and the use of registers. To operate on any string, follow these three steps:

1. Set the direction flag to indicate the direction in which you want to process the string. The **STD** instruction sets the flag, while **CLD** clears it.

If the direction flag is clear, the string is processed upward (from low addresses to high addresses, which is from left to right through the string). If the direction flag is set, the string is processed downward (from high addresses to low addresses, or from right to left). Under MS-DOS, the direction flag is normally clear if your program has not changed it.

2. Load the number of iterations for the string instruction into the CX register.

   If you want to process 100 elements in a string, move 100 into CX. If you wish the string instruction to terminate conditionally (for example, during a search when a match is found), load the maximum number of iterations that can be performed without an error.

3. Load the starting offset address of the source string into DS:SI and the starting address of the destination string into ES:DI. Some string instructions take only a destination or source, not both (see Table 5.1).

   Normally, the segment address of the source string should be DS, but you can use a segment override to specify a different segment for the source operand. You cannot override the segment address for the destination string. Therefore, you may need to change the value of ES. For information on changing segment registers, see "Programming Segmented Addresses" in Chapter 3.

Note Although you can use a segment override on the source operand, a segment override combined with a repeat prefix can cause problems in certain situations on all processors except the 80386/486. If an interrupt occurs during the string operation, the segment override is lost and the rest of the string operation processes incorrectly. Segment overrides can be used safely when interrupts are turned off or with the 80386/486 processors.

You can adapt these steps to the requirements of any particular string operation. The syntax for the string instructions is:

[[*prefix*]] **CMPS** [[*segmentregister***:**]] *source*, [[**ES:**]] *destination*
    LODS [[*segmentregister***:**]] *source*
[[*prefix*]] **MOVS** [[**ES:**]] *destination*, [[*segmentregister***:**]] *source*
[[*prefix*]] **SCAS** [[**ES:**]] *destination*
[[*prefix*]] **STOS** [[**ES:**]] *destination*

Some instructions have special forms for byte, word, or doubleword operands. If you use the form of the instruction that ends in **B** (**BYTE**), **W** (**WORD**), or **D** (**DWORD**) with **LODS**, **SCAS**, and **STOS**, the assembler knows whether the element is in the AL, AX, or EAX register. Therefore, these instruction forms do not require operands.

Table 5.1 lists each string instruction with the type of repeat prefix it uses and indicates whether the instruction works on a source, a destination, or both.

Table 5.1 Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	DS:SI, ES:DI
SCAS	**REPE/REPNE**	Destination	ES:DI
CMPS	**REPE/REPNE**	Both	DS:SI, ES:DI
LODS	None	Source	DS:SI
STOS	**REP**	Destination	ES:DI
INS	**REP**	Destination	ES:DI
OUTS	**REP**	Source	DS:SI

The repeat prefix causes the instruction that follows it to repeat for the number of times specified in the count register or until a condition becomes true. After each iteration, the instruction increments or decrements SI and DI so that it points to the next array element. The direction flag determines whether SI and DI are incremented (flag clear) or decremented (flag set). The size of the instruction determines whether SI and DI are altered by 1, 2, or 4 bytes each time.

Each prefix governs the number of repetitions as follows:

Prefix	Description
REP	Repeats instruction CX times
REPE, **REPZ**	Repeats instruction maximum CX times while values are equal
REPNE, **REPNZ**	Repeats instruction maximum CX times while values are not equal

The prefixes apply to only one string instruction at a time. To repeat a block of instructions, use a loop construction. (See "Loops" in Chapter 7.)

At run time, if a string instruction is preceded by a repeat sequence, the processor:

1. Checks the CX register and exits if CX is 0.
2. Performs the string operation once.
3. Increases SI and/or DI if the direction flag is clear. Decreases SI and/or DI if the direction flag is set. The amount of increase or decrease is 1 for byte operations, 2 for word operations, and 4 for doubleword operations.
4. Decrements CX without modifying the flags.

5. Checks the zero flag (for **SCAS** or **CMPS**) if the **REPE** or **REPNE** prefix is used. If the repeat condition holds, loops back to step 1. Otherwise, the loop ends and execution proceeds to the next instruction.

When the repeat loop ends, SI (or DI) points to the position following a match (when using **SCAS** or **CMPS**), so you need to decrement or increment DI or SI to point to the element where the last match occurred.

Although string instructions (except **LODS**) are used most often with repeat prefixes, they can also be used by themselves. In these cases, the SI and/or DI registers are adjusted as specified by the direction flag and the size of operands.

Using String Instructions

To use the 8086-family string instructions, follow the steps outlined in the previous section. Examples in this section illustrate each instruction.

You can also use the techniques in this section with structures and unions, since arrays and strings can be fields in structures and unions. (See the section "Structures and Unions," following.)

Moving Array Data

The **MOVS** instruction copies data from one area of memory to another. To move data, first load the count, source and destination addresses into the appropriate registers. Then use **REP** with the **MOVS** instruction.

```
        .MODEL  small
        .DATA
source  BYTE    10 DUP ('0123456789')
destin  BYTE    100 DUP (?)
        .CODE
        mov     ax, @data           ; Load same segment
        mov     ds, ax              ;   to both DS
        mov     es, ax              ;   and ES
        .
        .
        .
        cld                         ; Work upward
        mov     cx, LENGTHOF source ; Set iteration count to 100
        mov     si, OFFSET source   ; Load address of source
        mov     di, OFFSET destin   ; Load address of destination
        rep     movsb               ; Move 100 bytes
```

Filling Arrays

The **STOS** instruction stores a specified value in each position of a string. The string is the destination, so it must be pointed to by ES:DI. The value to store must be in the accumulator.

The next example stores the character `'a'` in each byte of a 100-byte string, filling the entire string with "aaaa...." Notice how the code stores 50 words rather than

100 bytes. This makes the fill operation faster by reducing the number of iterations. To fill an odd number of bytes, you need to adjust for the last byte.

```
        .MODEL  small, C
        .DATA
destin  BYTE    100 DUP (?)
ldestin EQU     (LENGTHOF destin) / 2
        .CODE
        .                           ; Assume ES = DS
        .
        .
        cld                         ; Work upward
        mov     ax, 'aa'            ; Load character to fill
        mov     cx, ldestin         ; Load length of string
        mov     di, OFFSET destin   ; Load address of destination
        rep     stosw               ; Store 'aa' into array
```

Comparing Arrays

The **CMPS** instruction compares two strings and points to the address after which a match or nonmatch occurs. If the values are the same, the zero flag is set. Either string can be considered the destination or the source unless a segment override is used. This example using **CMPSB** assumes that the strings are in different segments. Both segments must be initialized to the appropriate segment register.

```
        .MODEL  large, C
        .DATA
string1 BYTE    "The quick brown fox jumps over the lazy dog"
        .FARDATA
string2 BYTE    "The quick brown dog jumps over the lazy fox"
lstring EQU     LENGTHOF string2
        .CODE
        mov     ax, @data           ; Load data segment
        mov     ds, ax              ;   into DS
        mov     ax, @fardata        ; Load far data segment
        mov     es, ax              ;   into ES
        .
        .
        .
        cld                         ; Work upward
        mov     cx, lstring         ; Load length of string
        mov     si, OFFSET string1  ; Load offset of string1
        mov     di, OFFSET string2  ; Load offset of string2
        repe    cmpsb               ; Compare
        je      allmatch            ; Jump if all match
        .
        .
        .
allmatch:                           ; Special case for all match
```

Loading Data from Arrays

The **LODS** instruction loads a value from a string into the accumulator register. This instruction is not used with a repeat instruction prefix, since continually reloading the accumulator serves no purpose.

The code in this example loads, processes, and displays each byte in a string.

```
        .DATA
info    BYTE    0, 1, 2, 3, 4, 5, 6, 7, 8, 9
linfo   WORD    LENGTHOF info
        .CODE
        .
        .
        .
        cld                     ; Work upward
        mov     cx, linfo       ; Load length
        mov     si, OFFSET info ; Load offset of source
        mov     ah, 2           ; Display character function

get:
        lodsb                   ; Get a character
        add     al, '0'         ; Convert to ASCII
        mov     dl, al          ; Move to DL
        int     21h             ; Call DOS to display character
        loop    get             ; Repeat
```

Searching Arrays

The **SCAS** instruction compares the value pointed to by ES:DI with the value in the accumulator. If both values are the same, it sets the zero flag.

A repeat prefix lets **SCAS** work on an entire string, scanning (from which **SCAS** gets its name) for a particular value called the target. **REPNE SCAS** sets the zero flag if it finds the target value in the array. **REPE SCAS** sets the zero flag if the scanned array contains nothing but the target value.

This example assumes that ES is not the same as DS and that the address of the string is stored in a pointer variable. The **LES** instruction loads the far address of the string into ES:DI.

```
        .DATA
string  BYTE    "The quick brown fox jumps over the lazy dog"
pstring PBYTE   string              ; Far pointer to string
lstring EQU     LENGTHOF string     ; Length of string
        .CODE
        .
        .
        .
        cld                         ; Work upward
        mov     cx, lstring         ; Load length of string
        les     di, pstring         ; Load address of string
        mov     al, 'z'             ; Load character to find
        repne   scasb               ; Search
        jne     notfound            ; Jump if not found
        .                           ; ES:DI points to character
        .                           ;   after first 'z'
        .
notfound:                           ; Special case for not found
```

Translating Data in Byte Arrays

The **XLAT** (Translate) instruction copies a byte from an array of bytes into the AL register. The instruction takes its name from its ability to translate an element's number into the element itself. For example, given the number 7, **XLAT** returns byte #7 from the array. The array may hold byte-sized integers or, very often, a table or list of characters. The syntax for **XLAT** is:

XLAT[**B**] [[[*segment*:]]*memory*]

The optional **B** suffix (for "byte") reflects the size of data the instruction handles. Both **XLAT** and **XLATB** assemble to exactly the same machine code.

To use **XLAT**, place the offset of the start of the array in the BX register and the desired index value in AL. Array indexes always begin with 0 in assembly language. To retrieve the first byte of the array, set AL to 0; to retrieve the second byte, set AL to 1, and so forth. **XLAT** returns the byte element in AL, overwriting the index number.

By default, the DS register contains the segment of the table, but you can use a segment override to specify a different segment. You need not give an operand except when specifying a segment override. (For information about the segment override operator, see "Direct Memory Operands" in Chapter 3.)

This example illustrates **XLAT** by looking up hexadecimal characters in a list. The code converts an eight-bit binary number to a string representing a hexadecimal number.

```
; Table of hexadecimal digits
hex     BYTE    "0123456789ABCDEF"
convert BYTE    "You pressed the key with ASCII code "
key     BYTE    ?,?,"h",13,10,"$"
        .CODE
        .
        .
        .
        mov     ah, 8               ; Get a key in AL
        int     21h                 ; Call DOS
        mov     bx, OFFSET hex      ; Load table address
        mov     ah, al              ; Save a copy in high byte
        and     al, 00001111y       ; Mask out top character
        xlat                        ; Translate
        mov     key[1], al          ; Store the character
        mov     cl, 12              ; Load shift count
        shr     ax, cl              ; Shift high char into position
        xlat                        ; Translate
        mov     key, al             ; Store the character
        mov     dx, OFFSET convert  ; Load message
        mov     ah, 9               ; Display character
        int     21h                 ; Call DOS
```

Although AL cannot contain an index value greater than 255, you can use **XLAT** with arrays containing more than 256 elements. Simply treat each 256-byte block of the array as a smaller sub-array. For example, to retrieve the 260th element of an array, add 256 to BX and set AL=3 (260-256-1).

Structures and Unions

A structure is a group of possibly dissimilar data types and variables that can be accessed as a unit or by any of its components. The fields within the structure can have different sizes and data types.

Unions are identical to structures, except that the fields of a union overlap in memory, which allows you to define different data formats for the same memory space. Unions can store different types of data depending on the situation. They also can store data as one data type and retrieve it as another data type.

Whereas each field in a structure has an offset relative to the first byte of the structure, all the fields in a union start at the same offset. The size of a structure is the sum of its components; the size of a union is the length of the longest field.

A MASM structure is similar to a **struct** in the C language, a **STRUCTURE** in FORTRAN, and a **RECORD** in Pascal. Unions in MASM are similar to unions in C and FORTRAN, and to variant records in Pascal.

Follow these steps when using structures and unions:

1. Declare a structure (or union) type.
2. Define one or more variables having that type.
3. Reference the fields directly or indirectly with the field (dot) operator.

You can use the entire structure or union variable or just the individual fields as operands in assembler statements. This section explains the allocating, initializing, and nesting of structures and unions.

MASM 6.1 extends the functionality of structures and also makes some changes to MASM 5.1 behavior. If you prefer, you can retain MASM 5.1 behavior by specifying **OPTION OLDSTRUCTS** in your program.

Declaring Structure and Union Types

When you declare a structure or union type, you create a template for data. The template states the sizes and, optionally, the initial values in the structure or union, but allocates no memory.

The **STRUCT** keyword marks the beginning of a type declaration for a structure. (**STRUCT** and **STRUC** are synonyms.) The format for **STRUCT** and **UNION** type declarations is:

name {**STRUCT** | **UNION**} [[*alignment*]] [[**,NONUNIQUE**]]
fielddeclarations
name **ENDS**

The *fielddeclarations* is a series of one or more variable declarations. You can declare default initial values individually or with the **DUP** operator. (See "Defining Structure and Union Variables," following.) "Referencing Structures, Unions, and Fields," later in this chapter, explains the **NONUNIQUE** keyword. You can nest structures and unions, as explained in "Nested Structures and Unions," also later in this chapter.

Initializing Fields

If you provide initializers for the fields of a structure or union when you declare the type, these initializers become the default value for the fields when you define a variable of that type. "Defining Structure and Union Variables," following, explains default initializers.

When you initialize the fields of a union type, the type and value of the first field become the default value and type for the union. In this example of an initialized union declaration, the default type for the union is **DWORD**:

```
DWB     UNION
  d     DWORD   00FFh
  w     WORD    ?
  b     BYTE    ?
DWB     ENDS
```

If the size of the first member is less than the size of the union, the assembler initializes the rest of the union to zeros. When initializing strings in a type, make sure the initial values are long enough to accommodate the largest possible string.

Field Names

Structure and union field names must be unique within a nesting level because they represent the offset from the beginning of the structure to the corresponding field.

A label elsewhere in the code may have the same name as a structure field, but a text macro cannot. Also, field names between structures need not be unique. Field names must be unique if you place **OPTION M510** or **OPTION OLDSTRUCTS** in your code or use the /Zm option from the command line, since versions of MASM prior to 6.0 require unique field names. (See Appendix A.)

Alignment Value and Offsets for Structures

Data access to structures is faster on aligned fields than on unaligned fields. Therefore, alignment gains speed at the cost of space. Alignment improves access on 16-bit and 32-bit processors but makes no difference in programs executing on an 8-bit 8088 processor.

The way the assembler aligns structure fields determines the amount of space required to store a variable of that type. Each field in a structure has an offset relative to 0. If you specify an *alignment* in the structure declaration (or with the /Zp*n* command-line option), the offset for each field may be modified by the *alignment* (or *n*).

The only values accepted for *alignment* are 1, 2, and 4. The default is 1. If the type declaration includes an *alignment*, each field is aligned to either the field's size or the *alignment* value, whichever is less. If the field size in bytes is greater than the alignment value, the field is padded so that its offset is evenly divisible by the alignment value. Otherwise, the field is padded so that its offset is evenly divisible by the field size.

Any padding required to reach the correct offset for the field is added prior to allocating the field. The padding consists of zeros and always precedes the aligned field. The size of the structure must also be evenly divisible by the structure alignment value, so zeros may be added at the end of the structure.

If neither the *alignment* nor the /Zp command-line option is used, the offset is incremented by the size of each data directive. This is the same as a default *alignment* equal to 1. The *alignment* specified in the type declaration overrides the /Zp command-line option.

These examples show how the assembler determines offsets:

```
STUDENT2    STRUCT  2   ; Alignment value is 2
  score     WORD    1   ; Offset = 0
  id        BYTE    2   ; Offset = 2 (1 byte padding added)
  year      DWORD   3   ; Offset = 4
  sname     BYTE    4   ; Offset = 8 (1 byte padding added)
STUDENT2    ENDS
```

One byte of padding is added at the end of the first byte-sized field. Otherwise, the offset of the `year` field would be 3, which is not divisible by the alignment value of 2. The size of this structure is now 9 bytes. Since 9 is not evenly divisible by 2, 1 byte of padding is added at the end of `student2`.

```
STUDENT4    STRUCT  4            ; Alignment value is 4
  sname     BYTE    1            ; Offset =  0 (1 byte padding added)
  score     WORD    10 DUP (100) ; Offset =  2
  year      BYTE    2            ; Offset = 22 (1 byte padding
                                 ;   added so offset of next field
                                 ;   is divisible by 4)
  id        DWORD   3            ; Offset = 24
STUDENT4    ENDS
```

The alignment value affects the alignment of structure variables, so adding an alignment value affects memory usage. This feature provides compatibility with structures in Microsoft C. MASM 6.1 provides an improved H2INC utility, which C programmers can use to translate C structures to assembly. (See *Environment and Tools,* Chapter 20.)

The **ALIGN**, **EVEN**, and **ORG** directives can modify how field offsets are placed during structure definition. The **EVEN** and **ALIGN** directives insert padding bytes to round the field offset up to the specified alignment boundary. The **ORG** directive changes the offset of the next field to a given value, either positive or negative. If you use **ORG** when declaring a structure, you cannot define a structure of that type. **ORG** is useful when accessing existing data structures, such as a stack frame created by a high-level language.

Defining Structure and Union Variables

Once you have declared a structure or union type, you can define variables of that type. For each variable defined, memory is allocated in the current segment in the format declared by the type. The syntax for defining a structure or union variable is:

[[*name*]] *typename* < [[*initializer* [[,*initializer*]]...]] >

[[*name*]] *typename* **{** [[*initializer* [[,*initializer*]]...]] **}**

[[*name*]] *typename constant* **DUP ({** [[*initializer* [[,*initializer*]]...]] **})**

The *name* is the label assigned to the variable. If you do not provide a name, the assembler allocates space for the variable but does not give it a symbolic name. The *typename* is the name of a previously declared structure or union type.

You can give an *initializer* for each field. Each initializer must correspond in type with the field defined in the type declaration. For unions, the type of the initializer must be the same as the type for the first field. An initialization list can also use the **DUP** operator.

The list of initializers can be broken only after a comma unless you end the line with a continuation character (\). The last curly brace or angle bracket must appear on the same line as the last initializer. You can also use the line continuation character to extend a line as shown in the `Item4` declaration that follows. Angle brackets and curly braces can be intermixed in an initialization as long as they match. This example illustrates the options for initializing lists in structures of type `ITEMS`:

```
ITEMS       STRUCT
  Iname     BYTE      'Item Name'
  Inum      WORD      ?
  UNION     ITYPE               ; UNION keyword appears first
    oldtype BYTE      0         ;   when nested in structure.
    newtype WORD      ?         ;   (See "Nested Structures
  ENDS                          ;   and Unions," following ).
ITEMS       ENDS
        .
        .
        .
        .DATA
Item1   ITEMS   < >             ; Accepts default initializers
Item2   ITEMS   { }             ; Accepts default initializers
Item3   ITEMS   <'Bolts', 126>  ; Overrides default value of first
                                ;   2 fields; use default of
                                ;   the third field
Item4   ITEMS   { \
                 'Bolts',       ; Item name
                 126 \          ; Part number
                }
```

The example defines—that is, allocates space for—four structures of the `ITEMS` type. The structures are named `Item1` through `Item4`. Each definition requires the angle brackets or curly braces even when not initialized. If you initialize more than one field, separate the values with commas, as shown in `Item3` and `Item4`.

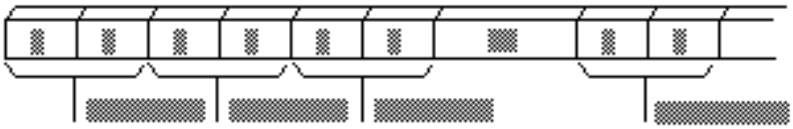
You need not initialize all fields in a structure. If a field is blank, the assembler uses the structure's initial value given for that field in the declaration. If there is no default value, the field value is left unspecified.

For nested structures or unions, however, these are equivalent:

```
Item5   ITEMS   {'Bolts', ,     }
Item6   ITEMS   {'Bolts', , { } }
```

A variable and an array of union type `WB` look like this:

```
WB      UNION
  w     WORD    ?
  b     BYTE    ?
WB      ENDS

num     WB      {0Fh}                       ; Store 0Fh
array   WB      (40 / SIZEOF WB) DUP ({2})  ; Allocates and
                                            ;   initializes 20 unions
```



Arrays as Field Initializers

The size of the initializer determines the length of the array that can override the contents of a field in a variable definition. The override cannot contain more elements than the default. Specifying fewer override array elements changes the first *n* values of the default where *n* is the number of values in the override. The rest of the array elements take their default values from the initializer.

Strings as Field Initializers

If the override is shorter, the assembler pads the override with spaces to equal the length of the initializer. If the initializer is a string and the override value is not a string, the override value must be enclosed in angle brackets or curly braces.

A string can override any member of type **BYTE** (or **SBYTE**). You need not enclose the string in angle brackets or curly braces unless mixed with other override methods.

If a structure has an initialized string field or an array of bytes, any new string assigned to a variable of the field that is smaller than the default is padded with spaces. The assembler adds four spaces at the end of `'Bolts'` in the variables of type `ITEMS` previously shown. The `Iname` field in the `ITEMS` structure cannot contain a field initializer longer than `'Item Name'`.

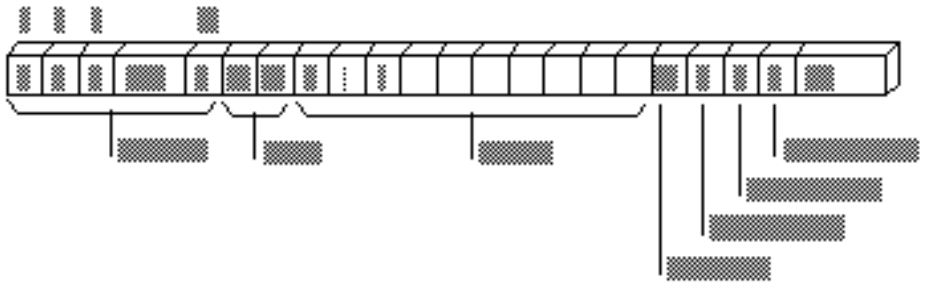
Structures as Field Initializers

Initializers for structure variables must be enclosed in curly braces or angle brackets, but you can specify overrides with fewer elements than the defaults.

This example illustrates the use of default values with structures as field initializers:

```
DISKDRIVES      STRUCT
  a1            BYTE ?
  b1            BYTE ?
  c1            BYTE ?
DISKDRIVES      ENDS

INFO            STRUCT
  buffer        BYTE    100 DUP (?)
  crlf          BYTE    13, 10
  query         BYTE    'Filename: ' ; String <= can override
  endmark       BYTE    36
  drives        DISKDRIVES <0, 1, 1>
INFO            ENDS

info1   INFO    { , , 'Dir' }

; Next line illegal since name in query field is too long:
; info2  INFO    {"TESTFILE", , "DirectoryName"}

lotsof  INFO    { , , 'file1', , {0,0,0} },
                { , , 'file2', , {0,0,1} },
                { , , 'file3', , {0,0,2} }
```

The following diagram shows how the assembler stores `info1`.



The initialization for `drives` gives default values for all three fields of the structure. The fields left blank in `info1` use the default values for those fields. The `info2` declaration is illegal because `"DirectoryName"` is longer than the initial string for that field.

Arrays of Structures and Unions

You can define an array of structures using the **DUP** operator (see "Declaring and Referencing Arrays," earlier in this chapter) or by creating a list of structures. For example, you can define an array of structure variables like this:

```
Item7   ITEMS    30 DUP ({,,{10}})
```

The `Item7` array defined here has 30 elements of type `ITEMS`, with the third field of each element (the union) initialized to 10.

You can also list array elements as shown in the following example.

```
Item8   ITEMS    {'Bolts', 126, 10},
                 {'Pliers',139, 10},
                 {'Saws',  414, 10}
```

Redeclaring a Structure

The assembler generates an error when you declare a structure more than once unless the following are the same:

- Field names
- Offsets of named fields
- Initialization lists
- Field alignment value

LENGTHOF, SIZEOF, and TYPE for Structures

The size of a structure determined by **SIZEOF** is the offset of the last field, plus the size of the last field, plus any padding required for proper alignment. (For information about alignment, see “Declaring Structure and Union Types,” earlier in this chapter.)

This example, using the preceding data declarations, shows how to use the **LENGTHOF**, **SIZEOF**, and **TYPE** operators with structures.

```
INFO            STRUCT
  buffer        BYTE    100 DUP (?)
  crlf          BYTE    13, 10
  query         BYTE    'Filename: '
  endmark       BYTE    36
  drives        DISKDRIVES <0, 1, 1>
INFO            ENDS

info1   INFO    { , , 'Dir' }
lotsof  INFO    { , , 'file1', , {0,0,0} },
                { , , 'file2', , {0,0,1} },
                { , , 'file3', , {0,0,2} }

sinfo1  EQU     SIZEOF    info1  ; 116 = number of bytes in
                                 ;   initializers
linfo1  EQU     LENGTHOF  info1  ; 1 = number of items
tinfo1  EQU     TYPE      info1  ; 116 = same as size

slotsof EQU     SIZEOF    lotsof ; 116 * 3 = number of bytes in
                                 ;   initializers
llotsof EQU     LENGTHOF  lotsof ; 3 = number of items
tlotsof EQU     TYPE      lotsof ; 116 = same as size for structure
                                 ;   of type INFO
```

LENGTHOF, SIZEOF, and TYPE for Unions

The size of a union determined by **SIZEOF** is the size of the longest field plus any padding required. The length of a union variable determined by **LENGTHOF** equals the number of initializers defined inside angle brackets or curly braces. **TYPE** returns a value indicating the type of the longest field.

```
DWB     UNION
  d     DWORD   ?
  w     WORD    ?
  b     BYTE    ?
DWB     ENDS

num     DWB     {0FFFFh}
array   DWB     (100 / SIZEOF DWB) DUP ({0})

snum    EQU     SIZEOF   num      ; = 4
lnum    EQU     LENGTHOF num      ; = 1
tnum    EQU     TYPE     num      ; = 4
sarray  EQU     SIZEOF   array    ; = 100 (4*25)
larray  EQU     LENGTHOF array    ; = 25
tarray  EQU     TYPE     array    ; = 4
```

Referencing Structures, Unions, and Fields

Like other variables, structure variables can be accessed by name. You can access fields within structure variables with this syntax:

variable. field

References to fields must always be fully qualified, with the structure or union names and the dot operator preceding the field name. The assembler requires that you use the dot operator only with structure fields, not as an alternative to the plus operator; nor can you use the plus operator as an alternative to the dot operator.

The following example shows several ways to reference the fields of a structure of type DATE.

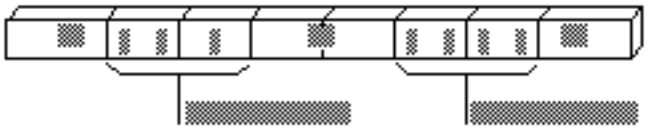
```
DATE    STRUCT                          ; Defines structure type
  month BYTE    ?
  day   BYTE    ?
  year  WORD    ?
DATE    ENDS

yesterday       DATE    {1, 20, 1993}     ; Declare structure
                                          ;   variable
        .
        .
        .
        mov     al, yesterday.day         ; Use structure variables
        mov     bx, OFFSET yesterday      ; Load structure address
        mov     al, (DATE PTR [bx]).month ; Use as indirect operand
        mov     al, [bx].date.month       ; This is necessary only if
                                          ;   month is already a
                                          ;   field in a different
                                          ;   structure
```

Under **OPTION M510** or **OPTION OLDSTRUCTS**, unique structure names do not need to be qualified. However, if the **NONUNIQUE** keyword appears in a structure definition, all fields of the structure must be fully qualified when referenced, even if the **OPTION OLDSTRUCTS** directive appears in the code. Also, you must qualify all references to a field. (For information on the **OPTION** directive, see Chapter 1.)

Even if the initialized union is the size of a **WORD** or **DWORD**, members of structures or unions are accessible only through the field's names.

In the following example, the two **MOV** statements show how you can access the elements of an array of unions.

```
WB      UNION
  w     WORD    ?
  b     BYTE    ?
WB      ENDS

array   WB      (100 / SIZEOF WB) DUP ({0})

        mov     array[12].w, 40h
        mov     array[32].b,  2
```



As the preceding code illustrates, you can use unions to access the same data in more than one form. One application of structures and unions is to simplify the task of reinitializing a far pointer. For a far pointer declared as

```
FPWORD  TYPEDEF FAR PTR WORD

        .DATA
WordPtr FPWORD ?
```

you must follow these steps to point `WordPtr` to a word value named `ThisWord` in the current data segment.

```
        mov     WORD PTR WordPtr[2], ds
        mov     WORD PTR WordPtr, OFFSET ThisWord
```

The preceding method requires that you remember whether the segment or the offset is stored first. However, if your program declares a union like this:

```
uptr      UNION
  dwptr   FPWORD   0
  STRUCT
    offs  WORD     0
    segm  WORD     0
  ENDS
uptr      ENDS
```

You can initialize a far pointer with these steps:

```
        .DATA
WrdPtr2 uptr    <>
        .
        .
        .
        mov     WrdPtr2.segm, ds
        mov     WrdPtr2.offs, OFFSET ThisWord
```

This code moves the segment and the offset into the pointer and then moves the pointer into a register with the other field of the union. Although this technique does not reduce the code size, it avoids confusion about the order for loading the segment and offset.

Nested Structures and Unions

You can nest structures and unions in several ways. This section explains how to refer to the fields in a nested structure or union. The following example illustrates the four techniques for nesting, and how to reference the fields. Note the syntax for nested structures. The techniques are reviewed following the example.

```
ITEMS           STRUCT
  Inum          WORD    ?
  Iname         BYTE    'Item Name'
ITEMS           ENDS

INVENTORY       STRUCT
  UpDate        WORD    ?
  oldItem       ITEMS   { \
                        100,
                        'AF8' \         ; Named variable of
                        }               ;   existing structure
                ITEMS   { ?, '94C' }    ; Unnamed variable of
                                        ;   existing type
  STRUCT ups                            ; Named nested structure
    source      WORD    ?
    shipmode    BYTE    ?
  ENDS
  STRUCT                                ; Unnamed nested structure
    f1          WORD    ?
    f2          WORD    ?
  ENDS
INVENTORY       ENDS

        .DATA

yearly  INVENTORY       { }
```

```
; Referencing each type of data in the yearly structure:

        mov     ax, yearly.oldItem.Inum
        mov     yearly.ups.shipmode, 'A'
        mov     yearly.Inum, 'C'
        mov     ax, yearly.f1
```

To nest structures and unions, you can use any of these techniques:

- The field of a structure or union can be a named variable of an existing structure or union type, as in the `oldItem` field. Because `INVENTORY` contains two structures of type `ITEMS` , the field names in `oldItem` are not unique. Therefore, you must use the full field names when referencing those fields, as in the statement

  ```
  mov     ax, yearly.oldItem.Inum
  ```

- To declare a named structure or union inside another structure or union, give the **STRUCT** or **UNION** keyword first and then define a label for it. Fields of the nested structure or union must always be qualified:

  ```
  mov     yearly.ups.shipmode, 'A'
  ```

- As shown in the `Items` field of `Inventory`, you also can use unnamed variables of existing structures or unions inside another structure or union. In these cases, you can reference fields directly:

  ```
  mov     yearly.Inum, 'C'
  mov     ax, yearly.f1
  ```

Records

Records are similar to structures, except that fields in records are bit strings. Each bit field in a record variable can be used separately in constant operands or expressions. The processor cannot access bits individually at run time, but it can access bit fields with instructions that manipulate bits.

Records are bytes, words, or doublewords in which the individual bits or groups of bits are considered fields. In general, the three steps for using record variables are the same as those for using other complex data types:

1. Declare a record type.
2. Define one or more variables having the record type.
3. Reference record variables using shifts and masks.

Once it is defined, you can use the record variable as an operand in assembler statements.

This section explains the record declaration syntax and the use of the **MASK** and **WIDTH** operators. It also shows some applications of record variables and constants.

Declaring Record Types

A record type creates a template for data with the sizes and, optionally, the initial values for bit fields in the record. It does not allocate memory space for the
record.

The **RECORD** directive declares a record type for an 8-bit, 16-bit, or 32-bit record that contains one or more bit fields. The maximum size is based on the expression word size. See **OPTION EXPR16** and **OPTION EXPR32** in Chapter 1. The syntax is:

recordname **RECORD** *field* [[**,** *field*]]...

The *field* declares the name, width, and initial value for the field. The syntax for each *field* is:

fieldname:*width*[[=*expression*]]

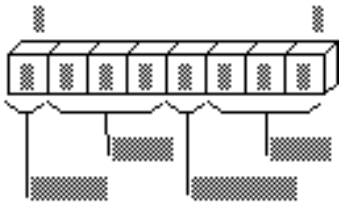
Global labels, macro names, and record field names must all be unique, but record field names can have the same names as structure field names. *Width* is the number of bits in the field, and *expression* is a constant giving the initial (or default) value for the field. Record definitions can span more than one line if the continued lines end with commas.

If *expression* is given, it declares the initial value for the field. The assembler generates an error message if an initial value is too large for the width of its field.

The first field in the declaration always goes into the most significant bits of the record. Subsequent fields are placed to the right in the succeeding bits. If the fields do not total exactly 8, 16, or 32 bits as appropriate, the entire record is shifted right, so the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record are initialized to 0.

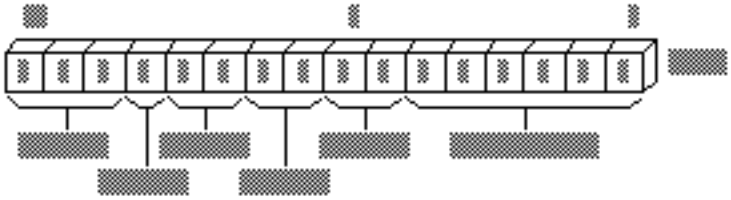
The following example creates a byte record type `COLOR` having four fields: `blink`, `back`, `intense`, and `fore`. The contents of the record type are shown after the example. Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. It allocates no space in the data segment.

```
COLOR   RECORD  blink:1, back:3, intense:1, fore:3
```



The next example creates a record type `CW` that has six fields. Each record declared with this type occupies 16 bits of memory. Initial (default) values are given for each field. You can use them when declaring data for the record. The bit diagram after the example shows the contents of the record type.

```
CW      RECORD  r1:3=0, ic:1=0, rc:2=0, pc:2=3, r2:2=1, masks:6=63
```



Defining Record Variables

Once you have declared a record type, you can define record variables of that type. For each variable, the assembler allocates memory in the format declared by the type. The syntax is:

[[*name*]] *recordname* <[[*initializer* [[,*initializer*]]...]] >

[[*name*]] *recordname* **{** [[*initializer* [[,*initializer*]]...]] **}**

[[*name*]] *recordname constant* ***DUP*** **(** [[*initializer* [[,*initializer*]]...]] **)**

The *recordname* is the name of a record type previously declared with the **RECORD** directive.

A *fieldlist* for each field in the record can be a list of integers, character constants, or expressions that correspond to a value compatible with the size of the field. You must include curly braces or angle brackets even when you do not specify an initial value.

If you use the **DUP** operator (see "Declaring and Referencing Arrays," earlier in this chapter) to initialize multiple record variables, only the angle brackets and

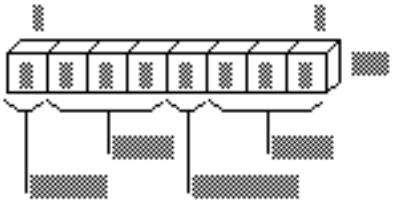
any initial values need to be enclosed in parentheses. For example, you can define an array of record variables with

```
xmas    COLOR   50 DUP ( <1, 2, 0, 4> )
```

You do not have to initialize all fields in a record. If an initial value is blank, the assembler automatically stores the default initial value of the field. If there is no default value, the assembler clears each bit in the field.

The definition in the following example creates a variable named `warning` whose type is given by the record type `COLOR`. The initial values of the fields in the variable are set to the values given in the record definition. The initial values override any default record values given in the declaration.

```
COLOR   RECORD  blink:1,back:3,intense:1,fore:3 ; Record
                                               ;   declaration
warning COLOR   <1, 0, 1, 4>                   ; Record
                                               ;   definition
```



LENGTHOF, SIZEOF, and TYPE with Records

The **SIZEOF** and **TYPE** operators applied to a record name return the number of bytes used by the record. **SIZEOF** returns the number of bytes a record variable occupies. You cannot use **LENGTHOF** with a record declaration, but you can use it with defined record variables. **LENGTHOF** returns the number of records in an array of records, or 1 for a single record variable. The following example illustrates these points.

```
; Record definition
; 9 bits stored in 2 bytes
RGBCOLOR        RECORD  red:3,  green:3,  blue:3

        mov     ax, RGBCOLOR          ; Equivalent to "mov ax, 01FFh"
;       mov     ax, LENGTHOF RGBCOLOR ; Illegal since LENGTHOF can
                                      ;   apply only to data label
        mov     ax, SIZEOF   RGBCOLOR ; Equivalent to "mov ax, 2"
        mov     ax, TYPE     RGBCOLOR ; Equivalent to "mov ax, 2"
```

```
; Record instance
; 8 bits stored in 1 byte
RGBCOLOR2       RECORD red:3, green:3, blue:2
rgb             RGBCOLOR2 <1, 1, 1>   ; Initialize to 00100101y

        mov     ax, RGBCOLOR2         ; Equivalent to
                                          ;   "mov ax, 00FFh"
        mov     ax, LENGTHOF rgb      ; Equivalent to "mov ax, 1"
        mov     ax, SIZEOF   rgb      ; Equivalent to "mov ax, 1"
        mov     ax, TYPE     rgb      ; Equivalent to "mov ax, 1"
```

Record Operators

The **WIDTH** operator (used only with records) returns the width in bits of a record or record field. The **MASK** operator returns a bit mask for the bit positions occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a bit field. The following example shows how to use **MASK** and **WIDTH**.

```
        .DATA
COLOR           RECORD  blink:1, back:3, intense:1, fore:3
message         COLOR   <1, 5, 1, 1>
wblink  EQU     WIDTH blink           ; "wblink"     = 1
wback   EQU     WIDTH back            ; "wback"      = 3
wintens EQU     WIDTH intense         ; "wintens"    = 1
wfore   EQU     WIDTH fore            ; "wfore"      = 3
wcolor  EQU     WIDTH COLOR           ; "wcolor"     = 8
        .CODE
        .
        .
        .
        mov     ah, message           ; Load initial   1101 1001
        and     ah, NOT MASK back     ; Turn off   AND 1000 1111
                                      ; "back"         ---------
                                      ;                1000 1001
        or      ah, MASK blink        ; Turn on     OR 1000 0000
                                      ; "blink"        ---------
                                      ;                1000 1001
        xor     ah, MASK intense      ; Toggle     XOR 0000 1000
                                      ; "intense"      ---------
                                      ;                1000 0001

        IF      (WIDTH COLOR) GT 8    ; If color is 16 bit, load
        mov     ax, message           ;   into 16-bit register
        ELSE                          ; else
        mov     al, message           ;   load into low 8-bit register
        xor     ah, ah                ;   and clear high 8-bits
        ENDIF
```

The example continues by illustrating several ways in which record fields can serve as operands and expressions:

```
; Rotate "back" of "message" without changing other values

mov     al, message      ; Load value from memory
mov     ah, al           ; Save a copy for work      1101 1001=ah/al
and     al, NOT MASK back; Mask out old bits     AND 1000 1111=mask
                         ;   to save old message     ---------
                         ;                           1000 1001=al
mov     cl, back         ; Load bit position
shr     ah, cl           ; Shift to right            0000 1101=ah
inc     ah               ; Increment                 0000 1110=ah

shl     ah, cl           ; Shift left again          1110 0000=ah
and     ah, MASK back    ; Mask off extra bits   AND 0111 0000=mask
                         ;   to get new message      ---------
                         ;                           0110 0000 ah
or      ah, al           ; Combine old and new    OR 1000 1001 al
                         ;                           ---------
mov     message, ah      ; Write back to memory      1110 1001 ah
```

Record variables are often used with the logical operators to perform logical operations on the bit fields of the record, as in the previous example using the **MASK** operator.