ASM286 Assembly Language Reference

November 18, 2019

Contents

Chapter 1

Introduction

About This Manual

ASM286 supports the 80286 and 8086 microprocessors, and 80287 and 8087 floating-point coprocessors. Throughout this manual, the word "processor" refers to any of the above microprocessors and the words "floating-point coprocessor" refer to any of the above coprocessors.

This manual is a reference for the ASM286 assembly language. It assumes that you are familiar with assembly language programming and 8086/80286 processor architecture. If you aren't, see the Intel 80286 Programmer's Reference Manual.

About This Chapter

This chapter introduces the assembly language. It has three major sections:

- Lexical Elements

  This section describes the assembler character set, tokens, separators, identifiers, comments, and the difference between source file lines and logical statement lines.

- Statements

  This section introduces the assembler directives, processor instruction set, and floating-point instruction set.

- Program Structure

This section provides a template for assembler programs together with a simple example program (see Appendix B for another example program). It summarizes the essential parts of every ASM286 program.

Lexical Elements

This section describes the lexical elements of the assembly language, except for its keywords and reserved words. See also: Keywords and reserved words, Appendix C

Character Set

The assembler character set is a subset of the ASCII character set. Each character in a source file should be one of the following:

Alphanumerics:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
0123456789
```

SpecialCharacters:

```
+-*/()[]<>;'."_:?@$&
```

Logical Delimiters: space, tab, and newline

If a program contains any character that is not in the preceding set, the assembler treats the character as a logical space.

Uppercase and lowercase letters are not distinguished from each other except in character strings. For example, xyz and XYZ are interchangeable, but 'xyz' and 'XYZ' are not equivalent character strings.

The special characters and combinations of special characters have particular meanings in a program, as described throughout this manual.

See also: ASCII character set, Appendix D

Tokens and Separators

A token is the smallest meaningful unit of a source program, much as words are the smallest meaningful units of a sentence. A token is one of the following:

- An end of statement

- A delimiter
- An identifier
- A constant
- An assembler keyword or reserved word

A separator that is a logical space or a delimiter must be specified between two adjacent tokens that are identifiers, constants, keywords, and/or reserved words. The most commonly used separator is the space character.

The end of statement token must be specified between two adjacent statements. The most commonly used statement terminator is the newline character.

See also: Constants, Chapter 4 keywords and reserved words, Appendix C

Logical Spaces

Any unbroken sequence of spaces can be used wherever a single space character is valid. Horizontal tabs are also used as token separators. The assembler interprets horizontal tabs as a single logical space. However, tabs are reproduced as multiple space characters in the print (listing) file to maintain the appearance of the source file.

See also: Print file, ASM386 Macro Assembler Operating Instructions

Logical spaces may not be specified within tokens such as identifiers, constants, keywords, or reserved words. The assembler treats any invalid character(s) in the context of a source file as a separator.

Delimiters

Like logical spaces, delimiters mark the end of a token, but each delimiter has a different special meaning. Some examples are commas and colons.

When a delimiter is present, a logical space between two tokens need not be specified. However, extra space or tab characters often make programs easier to read.

Delimiters are described in context throughout this manual.

Identifiers

An identifier is a name for a programmer-defined entity such as a segment, variable, label, or constant. Valid identifiers conform to the following rules:

- The initial character must be a letter (A...Z or a...z) or one of the following special characters: ? A question mark (ASCII value: 3FH) @ An at sign (ASCII value: 40H) _ An underscore (ASCII value: 5FH)
- The remaining characters may be letters, digits (0..9), and the preceding special characters. Separators may not be specified within identifiers.
- An identifier may be up to 255 characters in length; it is considered unique only up to 31 characters.
- Every identifier within a program module represents one and only one entity. A named entity is accessible from anywhere in the module when it is referenced by name. The assembler does not have identifier scope rules that allow you to specify the same name for two distinct entities in different contexts.

Continued Statements and Comments

An assembler statement usually occupies a single source file line. A source file line is a sequence of characters ended by a newline character.

However, the end of line in a source file is not necessarily the logical end of a statement. Assembler statements do terminate with a newline character, but logical statements can extend over several lines by using the continuation character (&).

The end of line in a source file always terminates a comment. The semicolon (;) is the initial character of a comment.

Valid comments and statements conform to the following rules:

- A comment begins with a semicolon (;) and ends when the line that contains it is terminated. The assembler ignores comments.
- A statement or comment may be continued on subsequent continuation lines. The first character following the line terminator that is not a logical space must be an ampersand (&).
- Statements and comments may extend over many source file lines if they conform to the following:
  - Symbols (such as identifiers, keywords, and reserved words) cannot be broken across continuation lines.
  - Character strings must be closed with an apostrophe on one line and reopened with an apostrophe on a subsequent continuation line, with an intervening comma (,) after the ampersand. Space and tab characters within a character string are significant; they are not treated as logical spaces.

– If a comment is continued, the first character following the ampersand that is not a logical space must be a semicolon (;).

Examples

The following examples illustrate the difference between the end of a source file line and the logical end of an assembler statement. The notation <newline> represents the newline character. Both examples are equivalent.

1. This example has a single statement on a single source file line. The end of the source file line and the logical end of the statement are the same.

```
;         1         2         3         4<newline>
; 234567890123456789012345678901234567890<newline>
<newline>                 ; interpreted as logical space
MOV AX, FOO<newline>
```

2. This example has many ends of lines in the source file, but it has only one logical end of statement.

```
;         1         2         3         4<newline>
; 234567890123456789012345678901234567890<newline>
<newline>                 ; interpreted as logical space
MOV                       ; this ASM286<newline>
& AX,                     ; statement extends<newline>
&                         ; <newline>
&                         ; <newline>
&                         ; over<newline>
&                         ; several lines<newline>
& FOO                     ; statement ends here<newline>
<newline>
```

Chapter 2

Defining And Initializing Data

This chapter has four major sections:

- An overview of assembler labels, variables, and data This section explains:
  - Assembler label and variable types
  - The relationship between assembler variable types and the values associated with variables: the processor or floating-point coprocessor data types
  - How to specify data values in assembler programs
- Assembler variables This section explains:
  - Storage allocations for variables
  - V ariable attributes
  - Defining and initializing simple-type variables with the DBIT, DB, DW, DD, DP, DQ, and DT directives
  - Defining compound types with the RECORD and STRUC directives; defining and initializing variables of these types (records and structures)
  - Defining and initializing variables with DUP clause(s)
- Assembler labels This section explains:
  - Label attributes
  - The location counter and the ORG and EVEN directives
  - The LABEL directive

  - Defining implicit NEAR labels

  - The PROC directive

- Using symbolic data, including named variables and labels, with the EQU and PURGE directives

Specifying Assembler Data Values

Assembler data can be expressed in binary, hexadecimal, octal, decimal, or ASCII form. Decimal values that represent integers or reals can be specified with a minus sign; a plus sign is redundant but accepted. Real numbers can also be expressed in floating-point decimal or in hexadecimal notations. Table 4-2 summarizes the valid ways of specifying data values in assembler programs.

Table 4-2. Assembler Data Value Specification Rules

Value in	Examples		Rules of Formulation
Binary	1100011B	110B	A sequence of 0's and 1's followed by the letter B.
Octal	7777O	4567Q	A sequence of digits in the range 0..7 followed by the letter O or the letter Q.
Decimal	3309	3309D	A sequence of digits in the range 0..9 followed by an optional letter D.
Hexadecimal	55H	4BEACH	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter H. A digit must begin the sequence.
ASCII	'AB'	'UPDATE.EXT'	Any ASCII string enclosed in single quotes.
Decimal	-1.	1E-32 3.14159	A rational number that may be preceded by a sign and followed by an optional exponent. A decimal point is required if no exponent is present but is optional otherwise. The exponent begins with the letter E followed by an optional sign and a sequence of digits in the range 0..9.
Hexadecimal	40490FR	0C0000R	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter R. The sequence must begin with a digit, and the total number of digits and letters must be (8/16/20) or (9/17/21 with the first digit 0).

A real hexadecimal specification must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of the floating-point number. For this reason, such values must have exactly 8, 16, or 20 hexadecimal digits, corresponding to the single, double, and extended precision reals that the floating- point coprocessor and the floating-point instructions handle. Such values can have 9, 17, or 21 hexadecimal digits only if the initial digit must be a zero because the value begins with a letter.

Data values can be specified in an assembler program in a variety of formats, as shown

in Table 4-2. The way the processor or floating-point coprocessor represents such data internally is called its storage format.

See also: Processor storage formats, Appendix A floating-point coprocessor storage formats, Chapter 7

Initializing Variables

Assembler variables can be initialized by:

- Variable or segment names that represent logical addresses
- Constants (see Table 4-2)
- Constant expressions

A series of operands and operators is called an expression. An expression that yields a constant value is called a constant expression.

See also: Assembler expressions, Chapter 5

The assembler evaluates constant expressions in programs.

How the Assembler Evaluates Constant Expressions

The assembler can perform arithmetic operations on 8-, and 16-bit numbers. The assembler interprets these numbers as integer or ordinal data types.

An integer value specified with a sign is a constant expression. The assembler evaluates integer or ordinal operands and expressions using 32-bit two's complement integer arithmetic.

By using this arithmetic, the assembler can evaluate expressions whose operands' sizes might extend beyond the storage type of the result. As long as the expression's value fits in the storage type of the destination, the assembler does not generate an error when intermediate results are too large. The assembler does generate an error if the final result is too large to fit in the destination.

Variables

A variable defines a logical address for the storage of value(s). An assembler variable is not required to have a name as long as its associated value(s) are accessible. But, every variable has a type; records and structures have a compound type.

Assembler variables must be defined with a storage allocation statement. A storage allocation specifies a type (storage size in bytes) and defines a logical address for a variable that gives access to the variable's value(s). A storage allocation statement may also specify initial value(s) for a variable.

Use the DB, DW, DD, DP, DQ, or DT directive to allocate storage for simple-type variables of the following sizes:

DB	8-bits (byte)
DW	16-bits (word)
DD	32-bits (dword)
DP	48-bits (pword)
DQ	64-bits (qword)
DT	80-bits (10 bytes)

Use a DUP clause within any assembler data allocation statement to allocate and optionally initialize a sequence of storage units of a single variable type. DUP defines an array-like variable whose element values are accessed by an offset from the variable name or from the initially specified storage unit.

Syntax

```
[name] dtyp init [,...]
```

Where:

name — is the name of the variable. Within the module, it must be a unique identifier.

dtyp — is DB, DW, DD, DP, DQ, or DT.

init — is the initial value to be stored in the allocated space. init can be a numeric constant (expressed in binary, hexadecimal, decimal, or octal), an ASCII string, or the question mark character (?), which specifies storage with undefined value(s). dtyp restricts the values that may be specified for init.

Defining and Initializing Variables of a Simple Type

All assembler variable definitions use the DB, DW, DD, DQ, DP, or DT directives. The template components of compound variable types are simple types defined with these directives.

DB Directive

Syntax

```
[name] DB init [,...]
```

Where:

name	is the name of the variable. Within the module, it must be a unique identifier.
init	is a question mark (?), a constant expression, or a string of up to 255 ASCII characters enclosed in single quotes (').

Discussion

DB reserves storage for and optionally initializes a variable of type BYTE. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range 0..255 (processor ordinal) or -128..127 (processor integer).

The components of character string values must be ASCII characters and the whole string must be enclosed in single quotes. To include a single quote character within such a string, specify two single quotes (").

Each ASCII character requires a byte of storage. In BYTE strings, successive characters occupy successive bytes. The name of the variable represents the logical address of the first character in such a string.

Examples

1. This example initializes the variable ABYTE to the constant value 100 (decimal). It reserves storage for another byte with an undefined value.

   ```
   ABYTE DB 100
   DB ?
   ```

2. This example initializes three successive bytes to the values 4, 10, and 200, respectively.

```
BYTES3 DB 4,10,200
```

3. This example initializes seven bytes containing the ASCII values of the characters A, B, C, ' , D, E, and F, respectively.

```
STRGWQUOT DB 'ABC''DEF'
```