900 Series HP 3000 Computer Systems

DATA TYPES CONVERSION
Programmer's Guide

HP Part No. 32650-90015
Printed in U.S.A. 1989

Second Edition
E1089

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright © 1989 by Hewlett-Packard Company

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	November 1987	A.01.00
Update 1	July 1988	A.10.00
Second Edition	October 1989	A.30.00

Preface

The Data Types Conversion Programmer's Manual is intended for MPE XL programmers who are experienced in one or more high-level programming languages. The purpose of the manual is to help the programmer who needs to receive and pass data across languages or programming environments.

This guide is part of the Programmer Series. Other manuals in the series are depicted in the documentation map at the front of the manual. The manuals for specific programming languages may be useful as well; the most common are listed below:

- *HP Business BASIC/XL Reference Manual* (32715-90001).
- *HP C Programmer's Guide* (92434-90002), *HP C Reference Manual* (92434-90001), *HP C/XL Reference Manual Supplement* (31506-90001), and *HP C/XL Library Reference Manual Supplement* (30026-90001).
- *HP COBOL II/XL Programmer's Guide* (31500-90002) and *COBOL II Reference Manual* (31500-90001) and *COBOL II/XL Reference Manual Supplement* (31500-90005).
- *HP FORTRAN 77/XL Reference Manual* (31501-90010) and *HP FORTRAN 77/XL Programmer's Guide* (31501-90011).
- *HP Pascal Reference Manual* (31502-90001) and *HP Pascal Programmer's Guide* (31502-90002).
- *HP RPG/XL Programmer's Guide* (30318-90001) and *HP RPG Reference Manual* (30318-90003).

Compiler Library/XL Reference Manual (32650-90029) may also be useful.

Chapter 1, Introduction, gives an overview of the manual and of the topic of data types, their format, storage, and conversion. It explains what primitive data types are recognized by MPE XL and its subsystems, why data conversion may be necessary, and the differences in data representation between the Native Mode and Compatability Mode programming environments in MPE XL.

Chapter 2, Formatting Data Types, presents the formats of the various data types supported on MPE XL and its subsystems. Bit formats are pictured, field boundaries given, and formatting conventions explained. A table compares the correspondence of primitive data types across system intrinsics and programming languages.

Chapter 3, Converting Data Types, takes each of the primitive data types, one by one, and gives some suggestions for converting to each of the other data types.

Appendix A, ASCII and EBCDIC Code Values, shows the character code values with their decimal, octal, and hexadecimal equivalents.

It would be most useful to skim the entire manual once, then look up specific topics as needed. The Table of Figures lists the various bit format maps. Look at the Table of Contents, Table of Tables, and index for other specific topics.

Contents

Figures

Tables

1

Introduction

This chapter gives you background on creating and receiving data in forms that your program and the operating system understand. It presents data types defined by the designers of 900 Series HP 3000 Computer Systems, and the proper format for those types.

How Do the Programmer and the Computer Communicate Data?

The computer can receive information, manipulate it, and store it. It can access stored information, read it, and send it out. All information is represented in the computer by combinations of ones and zeros, each one called a binary bit.

Text characters and numeric values are passed in and out of the computer as a sequence of bits in fixed-sized chunks called words. Hewlett-Packard Precision Architecture (HP-PA) design is based on a 32-bit (4-byte) word.

Registers are designed to hold one 32-bit word of data. Because they are fast but costly, registers typically hold only the data being currently processed and the most frequently used simple machine instructions.

The designers also define what types of data the system will recognize and how each type is to be formatted. This way, the system and the programmer can access and pass data in complete and meaningful blocks. The programmer often uses a high-level language compiler to translate between the system and user.

Defining Data Types

The designers of the computer define certain primitive system data types in order to receive input, store or manipulate data, and return information in a predictable way.

How a process will use data input depends on the context. If a process requires a certain data type, it will attempt to interpret input as that type. For example, if you pass 32 binary bits to an intrinsic parameter that requires a file address, it may attempt to access the cell at that location in memory. If you pass the same 32 bits to a parameter that requires a character array, it may print a four-letter word. If you pass the same 32 bits to a parameter that requires a 64-bit floating-point real value, you may cause an error or program abort.

Primitive Data Types

The HP-PA instruction set is designed to operate on certain fundamental data types. The following data types are recognized by MPE XL and its subsystems:

- Characters.
- The following numeric types:
  - Integers.
  - Real numbers (in floating-point notation).
  - Decimal.

Note

Although decimal is not really a system primitive type, it is included in this manual because it is so widely used on MPE XL. Floating-point decimals are used by BASIC; packed and unpacked decimals are used by COBOL and RPG.

Intrinsic Data Types

The compilers of high-level languages running on MPE XL have mechanisms to access the system-defined procedures called intrinsics. MPE XL intrinsic parameters recognize the following data types:

Address (`@`).
Array (`A`).
Boolean (`B`).
Character (`C`).
Integers: Signed (`I`) and Unsigned (`U`).
Real (`R`).
Record (`Rec`).

Character, integer, and real numbers are system primitive types. Address and Boolean types are numbers with special uses. Array and record are structures that group data.

Language Data Types

Some high-level programming languages running on MPE XL define their own data types based on the primitive types. The language compiler makes any necessary conversions between the primitive data types and the language-dependent data types. This conversion is transparent to the programmer. These types are described in the appropriate language manuals in the Language Manual Series.

Languages may simply rename the primitive type, like the integer, a system type common to all languages. Languages may use the system types as building blocks to create a more complex data structure. For example, the array is not a primitive type, but is defined by programming languages as a connected group of data, all of the same type.

Formatting Data Types

The designers of the computer specify specific formats for each data type so the computer can access or output a sequence of bits in a predictable way.

The format specifies alignment and size. The alignment predicts the (starting) boundary; it tells where a meaningful unit of information begins. The size tells the length of the unit of information; it predicts the end.

The proper format for a data type depends on two factors:

- Programming environment.
- Programming language.

Programming Environment

MPE XL supports two programming environments: Native Mode (NM) and Compatibilty Mode (CM). A program can be designed to run in NM or in CM, or to switch back and forth between subroutines in each of them.

NM takes full advantage of Hewlett-Packard Precision Architecture (HP-PA), which is based on a 32-bit word. CM emulates the MPE V/E operating system, which is based on a 16-bit word.

Note

In this manual, assume that data types are MPE XL NM data types, unless CM is specifically mentioned.

Programming Languages

Each environment supports its own high-level languages and compilers. Most are shown in Table 1-1.

Table 1-1. Languages Supported on MPE XL

Native Mode	Compatibility Mode
HP Business BASIC/XL	HP Business BASIC/V
HP C/XL	
HP COBOL II/XL	COBOL II/V
	HP FORTRAN 66/V
HP FORTRAN 77/XL	HP FORTRAN 77/V
HP Pascal/XL	HP Pascal/V
HP RPG/XL	RPG/V
	SPL/V

For language-specific data types and formatting conventions, consult the appropriate language manual.

- *HP Business BASIC/XL Reference Manual* (32715-90001)
- *HP C Reference Manual* (92434-90001) and *HP C/XL Reference Manual Supplement* (31506-90001)
- *HP COBOL II/XL Programmer's Guide* (31500-90002)
- *COBOL II Reference Manual* (31500-90001) and *COBOL II/XL Reference Manual Supplement* (31500-90005)
- *HP FORTRAN 77/XL Reference Manual* (31501-90010)
- *HP FORTRAN 77/XL Programmer's Guide* (31501-90002)
- *HP Pascal Reference Manual* (31502-90001)
- *HP Pascal Programmer's Guide* (31502-90002)

Converting Data Types

You may want to change the form of information. Data output can be created by one MPE process that cannot be used in another without translation or conversion. Plan for conversion if you pass data to be used in the following situations:

- with data of another type
- between programming environments
- across programming languages

Language commands, system intrinsics, and compiler library routines help you convert between types and formats.

Using Different Types Together

You may need to make different types of data together compatible to use them in a program. For example, to calculate the total cost of a product, you may need to multiply a price by the number sold. If the price is stored as ASCII data type and the number sold is stored as integer, one of them will have to be converted to the same data type as the other.

Subroutines are already available for many common conversions. There are also intrinsics at the system level, and commands within programming languages to convert.

Chapter 2 defines the NM primitive types and provides their bit maps. Chapter 3 gives some conversion methods.

Passing Between Programming Environments

MPE V/E and MPE XL in Compatibility Mode (MPE XL CM) are based on a 16-bit word; MPE XL in Native Mode (MPE XL NM) is based on a 32-bit word. Some data types are represented differently. For example, a real number in a CM-compiled program will, by default, be in HP3000 format. The same real value in an NM-compiled program will, by default, be in IEEE format.

If conversion is necessary, consider re-compiling routines, writing subroutines to reformat, or using system intrinsics.

Passing Across Programming Languages

The high-level languages do not all recognize the same primitive data types. COBOL uses the decimal data type, which is not recognized by Pascal; however, the floating-point real number type is mutually understood by Pascal and MPE XL, but is not recognized by COBOL.

Languages may define their own complex data types that cannot be interpreted by other languages. If you pass data between routines that do not use the same types or formats, you lose integrity and meaning. The receiving routine may not be able to read the data at all. It may divide the bits it reads into the wrong size chunks. It may interpret the arrangement of bits by its own formatting conventions. The result could be completely different information than you intended.

You must plan for conversion if a program uses a subroutine written in a language with incompatible types. Some languages have commands that translate directly as data is read in and written out.

You may need to write a routine to transform the data indirectly. Remember that all the data used in any MPE language is a primitive data type or is based on a primitive type. You could write one routine to translate data from the first language types into primitive system types, and then another routine to translate those system types into a form the second language can use.

2

Formatting Data Types

This chapter helps you understand the data types supported on MPE XL. The first part of the chapter defines and describes the primitive data types recognized by MPE XL Native Mode systems and subsystems, including bit formats and alignments. The second part describes some formatting considerations in MPE XL supported programming languages and environments.

Recognizing Primitive Data Types

Data is an abstraction of information. Data must be structured in a form that the computer is designed to process; data conversion is the translation of information to a form acceptable to the computer.

The 900 Series HP 3000 Computer Systems instruction set is designed to operate on certain fundamental data types. The following data types are recognized by MPE XL and its subsystems:

- Characters.
- The following numeric types:

  Integers.
  Real numbers (in floating point notation).
  Decimals: packed, unpacked, and floating-point.

Note

Although decimal is not really a system primitive type, it is included in this manual because it is so widely used on MPE XL. Floating-point decimals are used by BASIC; packed and unpacked decimals are used by COBOL and RPG.

Each data type requires a specific bit format. In this manual, bit fields are described as (*bit*:*length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (13:3)" refers to bits 13, 14, and 15. Bit 0 is the most significant bit.

Character

Character code formats are primitive data types. Characters are the letters, numbers, and symbols on your keyboard. The computer relates each alphanumeric character to an 8-bit (one byte) binary number, according to a correspondence code. Some of the characters are easily displayable, like `+`, `?`, `8`, and `z`; some are not, like a blank space or the carriage return.

MPE supports the two common American English character codes: ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). Several natural language types are also supported. See Appendix A for ASCII and EBCDIC codes and equivalents.

Character data types are useful for storing strings of symbols like names, addresses, or identification numbers, and for reading the keyboard or writing to the screen. Remember, variables saved as data type character are recognized by the computer as symbols, not as numeric values.

ASCII

MPE and its subsystems use ASCII data type to represent character data. ASCII is the format adopted by ANSI, the American National Standards Institute. Most MPE interfaces use ASCII to accept or return character data.

Appendix A shows the ASCII and EBCDIC character code values, along with their decimal, octal, and hexadecimal equivalents.

ASCII is used in this guide as the name of a data type. ASCII data type corresponds to the ASCII character code format. The codes for byte values in the range 0 to 127 conform to the ASCII standard format. Byte values in the range 128 to 255 are interpreted using Hewlett-Packard's extended ROMAN8 character set. MPE XL and its subsystems use values in this range to support extended (8-bit) character sets.

Figure 2-1 shows the ASCII data type bit format.

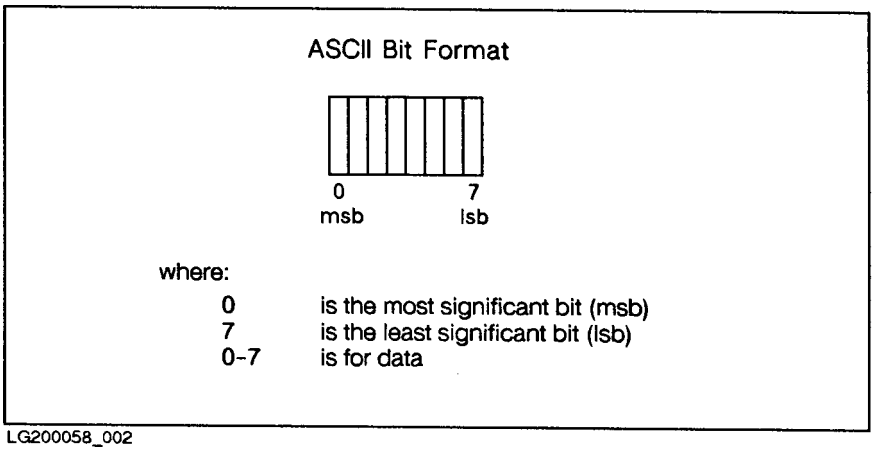

Figure 2-1. Bit Format: ASCII Character

EBCDIC

EBCDIC is another coding format widely used in the computer industry for character data. Like ASCII, it is based on the byte.

EBCDIC is used in this guide as the name of a data type. EBCDIC data type corresponds to EBCDIC character code format for byte values in the range 0 to 255.

Appendix A shows the ASCII and EBCDIC character code values, along with their decimal, octal, and hexadecimal equivalents.

Figure 2-2 shows the bit format for EBCDIC data type.

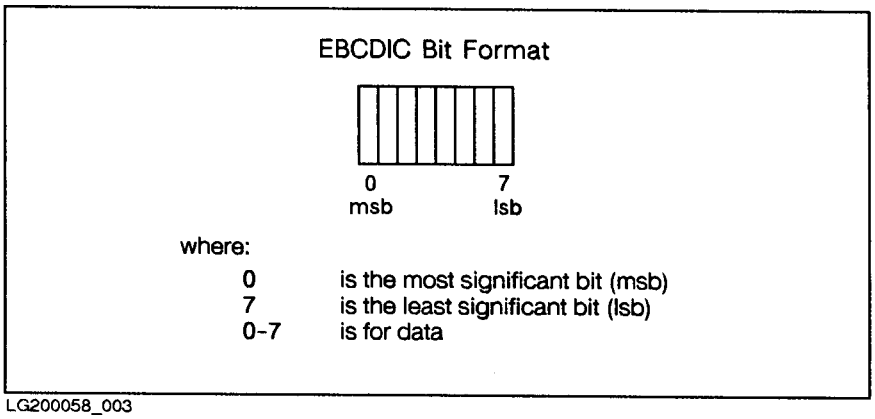

Figure 2-2. Bit Format: EBCDIC Character

Numeric

MPE XL subsystems support three primitive data types for numbers:

- Integer.
- Real.
- Decimal.

Integer

An integer is any positive or negative whole number, including zero. Integers are useful for counting and for incrementing in loops. Signed integers are a useful form for exchanging numeric data between languages.

MPE XL integers can be 8, 16, 32, or 64 bits long. They can be unsigned or signed (+ or −). Signed integers are represented in twos complement form.

Table 2-1. MPE XL Integer Types

Size	Type	Range	Stored At:
8-bit:	unsigned	0 to 255	byte addresses
16-bit:	signed	−32,768 to 32,767	half-word addresses
	unsigned	0 to 65,535	half-word addresses
32-bit:	signed	−2,147,483,648 to 2,147,483,647	word addresses
	unsigned	0 to 4,294,967,295	word addresses

The chart below shows the representation of the whole number (base-ten) 73 as an unsigned integer, a signed positive number, and a signed negative number.

Unsigned	Signed	
	Positive	Negative
(73)	(+73)	(−73)
01001001	01001001	10110111

Unsigned Integer. Unsigned integers are stored in the computer in their base-two form. If you are reading or writing unsigned integers in a language, the compiler converts for you, according to the formatting conventions of the individual language.

An unsigned n-bit number can represent any value from 0 to 2^n-1.

Reading an Unsigned Integer: One method of reading an unsigned integer as a base-ten value is to consider the bits as columns whose values are powers of two. The rightmost (least significant) bit is the units column and has a weight of 2^0, or 1. Going toward the left (the most significant bit), the columns have progressively greater weight: 2^0, 2^1, 2^2, ... 2^{n-1}. The decimal-based value of unsigned binary numbers is computed by multiplying the value in each column by the weight of the column, and then adding all the results. An unsigned integer represented with ones in the 2^0, 2^3 , and 2^6 columns and zeros in all the other columns would be computed as follows:

$1*(2^0) + 1*(2^3) + 1*(2^6) = 73.$

Writing an Unsigned Integer: One method of manually determining the unsigned integer representation of a base-ten value is to use successive subtraction. For example, the largest power of 2 that is less than or equal to the value of decimal-base 73 is 2^6, or 64. Subtracting 64 from 73 leaves a remainder of 9. The largest power of 2 that is less or equal than 9 is 2^3, or 8. Subtracting 8 from 9 leaves a remainder of 1. The only power of 2 that is less than or equal to 1 is 2^0, or 1. This leaves a remainder of 0, so the computation is finished. Thus, 73 is represented in binary with a 1 in the 2^0, the 2^3, and the 2^6 columns and a zero in all the others.

Signed Integer. Signed integers are stored in the computer in twos complement form. If you are reading or writing signed integers in a language, the compiler converts for you, according to the formatting conventions of the individual language.

A signed n-bit integer in twos complement form can represent any value from $-(2^{n-1})$ to $+2^{n-1}-1$.

When the n-bit positive integer i is added to its n-bit integer negative (complement), $-i$, and both are in twos complement form, the result is always an n-bit zero.

Reading a Signed Integer: The computer represents both positive and negative numbers in twos complement form much the same way that it would represent an unsigned integer: beginning at the rightmost (least significant bit) and going toward the left, the columns have progressively greater weight: 2^0, 2^1, 2^2, ... 2^{n-1}. The only difference is that the most significant bit of a twos complement number is *negative*. That is, it has a weight of $-(2^{n-1})$.

To manually convert a signed integer in twos complement form to a base-ten integer, you can use the column method explained in Unsigned Integers, above. However, you give the leftmost column of a twos complement number a weight of $-(2^{n-1})$.

In the example below, this method is used to interpret the signed binary integers 01010101 and 10101010, written in twos complement form, as decimal-based integers:

$(01010101)_{base\ 2}$ = the sum of:				$(10101010)_{base\ 2}$ = the sum of:		
(1×2^0)	=	1		(0×2^0)	=	0
(0×2^1)	=	0		(1×2^1)	=	2
(1×2^2)	=	4		(0×2^2)	=	0
(0×2^3)	=	0		(1×2^3)	=	8
(1×2^4)	=	16		(0×2^4)	=	0
(0×2^5)	=	0		(1×2^5)	=	32
(1×2^6)	=	64		(0×2^6)	=	0
$(0 \times -(2^7))$	=	0		$(1 \times -(2^7))$	=	-128
$(01010101)_{base\ 2}$	=	$85_{\text{base 10}}$	and	$(10101010)_{base\ 2}$	=	$-86_{\text{base 10}}$

Writing a Signed Integer: Converting a signed base-ten number to twos complement form is not difficult.

You can represent the positive signed integers just as explained in Unsigned Integers, above.

You can represent a negative integer quickly and easily using the following technique, which takes advantage of the properties of binary numbers: First, ignoring the sign, represent the value as an unsigned binary integer. Next, reverse all the 0s and 1s. Finally, add 1 to the result. Thus, the twos complement of 10101010 is (01010101 + 1), or 01010110.

You can check your conversion by adding the positive and negative numbers (in twos complement form) to see if they total zero. From the example above, notice that adding the 8-bit integer 10101010 to its twos complement, 01010110, yields a 9-bit result, 100000000. However, the system defines the result type to be 8-bit integer and recognizes only the 8 zeros, so the result is zero.

Figure 2-3 shows bit formats for the 32-bit integer type.

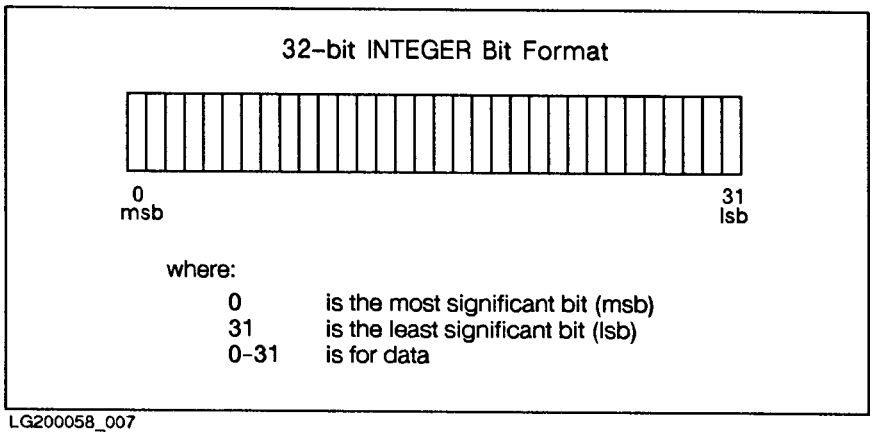

Figure 2-3. Bit Format: 32-Bit Integer

Real

A real number is a value in the set of zero and the positive or negative rational numbers. Signed integers and fractions are included, although fractions may be approximated. Imaginary and complex numbers are not included in the set of real numbers, although high-level languages may have constructs for storing and working with them.

The real data type is a useful form for representing very large or small values. Special formats are reserved to represent zero, infinity, and NaN (not a number).

Real data type represents real numbers by using a type of floating-point, or scientific, notation. In this notation, you generally express a very large or very small number as a fraction multiplied by a power of the number base. For example, the base-ten number .000025 could be expressed as $+.25 * 10^{-4}$ The general floating-point, or scientific notation, form is:

S_f F * (B ** S_e E)

where:	S_f	is the sign (+ or −) of the number.
	F	is the fraction or mantissa.
	*	is the symbol for multiplication.
	B	the base is represented as an integer.
	**	is the symbol for exponentiation.
	S_e	is the sign (+ or −) of the exponent.
	E	the exponent or characteristic is represented as an integer.

Note

In this manual, assume all representations of floating-point real numbers use an integer base of 10 (decimal-based, or base-ten) unless otherwise indicated. Internally, the computer uses a base of two (is binary-based), and the conversion is approximate.

You can represent real numbers four ways. You can choose either in IEEE or HP3000 format and use either single-precision or double-precision size.

IEEE or HP3000 Format. MPE XL recognizes two formats for storing floating-point real numbers: IEEE and HP3000. Programs compiled in NM use IEEE as the default. Programs compiled in CM use HP3000, the MPE XL emulation of the MPE V/E system floating-point format. NM programs accessing HP3000 data must either specify a special compiler option or convert CM data to NM before operations.

Single or Double Precision. You can represent single-precision (32-bit) or double-precision (64-bit) real numbers in both IEEE and HP3000 notation. Table 2-2 shows a summary of the range and accuracy of each.

Table 2-2. Ranges and Accuracies for Floating-Point Real Numbers

	IEEE	HP3000
Single precision:		
Accuracy (in decimal digits)	7.2	6.9
Range	−3.4E38 to −1.4E−45 0 +1.4E−45 to +3.4E38	−1.27E77 to −8.6E−78 0 +8.6E−78 to +1.2E77
Double precision:		
Accuracy (in decimal digits)	15.9	16.5
Range	−1.8E308 to −4.9E−324 0 +4.9E−324 to +1.8E308	−1.2E77 to −8.6E−78 0 +8.6E−78 to +1.2E77
Note: Values in this table are rounded.		

Fields of a Real Number. In MPE XL format, real numbers have three fields:

Sign.
Mantissa.
Exponent.

Different representations of real numbers have the three fields aligned on different boundaries. In all formats, the sign field is the first bit, the mantissa is in normalized form, and the exponent is biased.

The sign field, bit (0:1), is 0 if number is positive, 1 if negative.

Mantissas are represented in normalized form. That is, the leading one is stripped and binary point is not explicitly expressed. Each expressed mantissa, then, has an implied leading one and binary point. For example, a mantissa represented by 10101010101010101010101 is interpreted as the value 1.10101010101010101010101.

The exponents of real numbers are biased. This means that both positive and negative true exponents are represented using only unsigned binary integers. The bias amount, or excess, is the difference between the true exponent and the represented exponent. The negative true exponents correspond to the lower range of the represented exponents. The positive true exponents correspond to the upper range of the represented exponents. The true exponent zero corresponds to the midpoint in the range of the represented exponents. For example, consider an exponent field n bits long where the true exponent is T, the represented exponent is E, and the bias is b. For any real number x, then, $x^T = x^{E-b}$, and $x^E = x^{T+b}$.

Exponent fields of all zeros or all ones are reserved. If the exponent of a floating-point number is all zeros and the mantissa is zero, the number is regarded as zero. If the exponent of a floating-point number is all zeros and the mantissa is not all zero, the number is regarded as denormalized. If the exponent of a floating-point number is all ones and the mantissa is zero, the number is regarded as a signed infinity. If the exponent is all ones and the mantissa is not zero, the interpretation is NaN (Not-a-Number, undefined).

If any process attempts to operate on an infinity or a NaN, a system trap may occur and data may be corrupted. Invalid operation is signaled when the source is a signaling or a quiet NaN. The result is the destination format's largest finite number with the sign of the source.

Any operation that involves a signaling NaN or invalid operation returns a quiet NaN as the result when no trap occurs and a floating-point result is to be delivered. If an operation is using one or two quiet NaNs as input, it signals no exception; however, if a floating-point result is to be delivered, a quiet NaN is returned that is the same as one of the input NaNs.

IEEE Real Number Format. IEEE numbers conform to the format set up by the Institute of Electrical and Electronics Engineers and the American National Standards Institute (std 754-1985). Single-precision numbers are one NM word, aligned on 32-bit boundaries. Double precision numbers are two NM words, aligned on 64-bit boundaries.

Note

In this manual, bit fields are described as (*bit*:*length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (11:3)" refers to bits 11, 12, and 13. Bit 0 is the most significant bit.

IEEE numbers in MPE floating-point notation contain three fields:

Sign: The sign field is bit (0:1), the first bit of the first word. A value of 0 indicates the number is positive, and a value of 1 indicates the number is negative. The sign bit is the only difference between a real number value and its negative.

Exponent: The single-precision exponent field is bits (1:8) of the first NM word, and is biased by 127. The double-precision exponent field is bits (1:11) of the first NM word, and is biased by 1023.

Mantissa: The single-precision mantissa field is bits (9:23). The double-precision mantissa field is bits (12:52). MPE stores the mantissa as normalized data represented as a binary number of 23 bits for the single-precision format, and 52 bits, with an assumed 1. leading the field.

A previous section, "Fields of a Real Number", explains biased exponent and normalized mantissa.

IEEE Conversion Example. Consider converting an IEEE single-precision floating-point number into a base-ten number using this formula:

$$(-1)^{\text{sign}} * 2^{\text{Exponent}-127} * (1.0 + \text{Mantissa} + 2^{-23})$$

where:

Sign	Bit (0:1), the sign field, is 0 if number is positive, 1 if negative.
*	is the symbol for multiplication.
Exponent	Bits (1:8), the exponent field, is the biased representation of the true exponent.
+	is the symbol for addition.
Mantissa	Bits (9:23) is the normalized form of the mantissa, or fraction.
2^{-23}	is added for rounding.

The (base-ten) floating-point number 100.00 (hexadecimal $42c80000) is represented as 0 10000101 10010000000000000000000. Using the formula, we obtain the correct result as follows:

Table 2-3.
Determining the Base-Ten Equivalent of an IEEE Real Number

	S(ign)	E(xponent)	M(antissa)
=	0	10000101	10010000000000000000000

$$(-1)^{S} \quad *2^{E-127} \quad *(1.0+M+2^{-23})$$

$$= \quad -1^{0} \quad * 2^{133-127} \quad * 1.0+9/16 +2^{-23}$$

$$= \quad 1 \quad * 64 \quad * 1.0 + 0.5625 + .00000011920929$$

$$= \quad 64 \quad * \quad 1.56250011020929$$

$$= \quad 100$$

Figure 2-4 shows the bit format for floating-point real numbers in IEEE single-precision format.

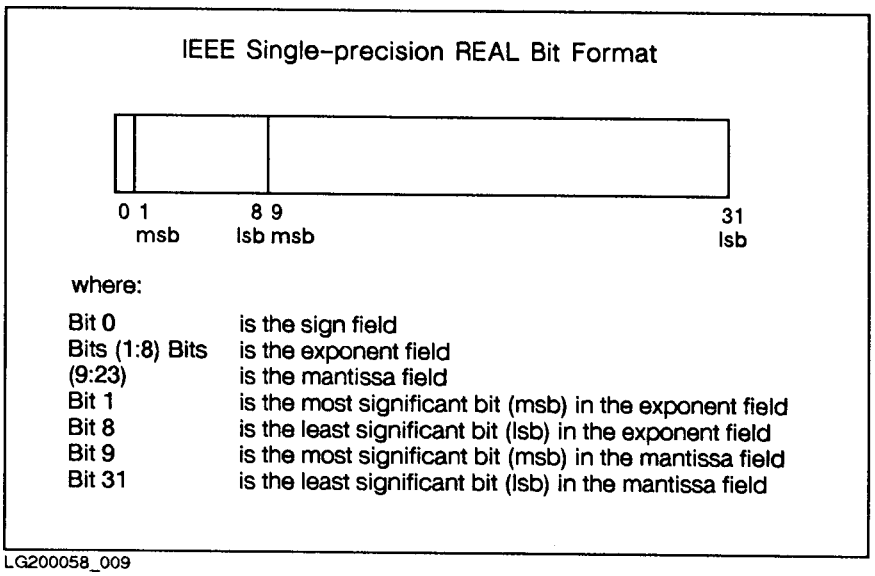

Figure 2-4.
Bit Format: Single-Precision Real in IEEE Floating-Point Notation

Figure 2-5 shows the IEEE real number double-precision bit format.

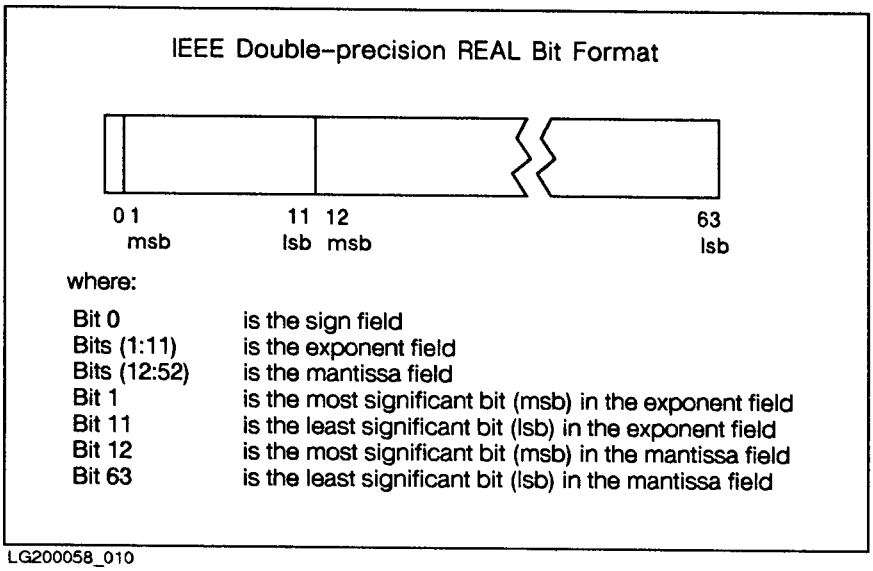

Figure 2-5.
Bit Format: Double-Precision Real in IEEE Floating-Point Notation

HP3000 Real Number Format. Single-precision HP3000 real numbers are 32 bits (2 CM words), and double-precision are 64 bits (4 CM words). When stored in memory, HP3000 reals are aligned on CM word boundaries.

Note

In this manual, bit fields are described as (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (11:3)" refers to bits 11, 12, and 13.

Real numbers in HP3000 floating-point notation contain three fields:

Sign: The sign field is bit (0:1) of the first word. A value of 0 indicates the number is positive and a value of 1 indicates the number is negative. The sign is the only difference between a real number value and its negative.

Exponent: The exponent field is bits (1:9) of the first CM word in the single-precision and double-precision format. The represented exponent range is 0 to 511. Exponents are biased by +256.

Mantissa: The mantissa field is bits (10:6) of the first CM word and bits (0:16) of the other words. MPE stores the mantissa as normalized data of 22 bits for the single-precision format, and 54 bits for the double-precision, with an asssumed 1. leading the field.

A previous section, "Fields of a Real Number", explains biased exponent and normalized mantissa.

Note

In this manual, bit fields are described as (*bit*:*length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (11:3)" refers to bits 11, 12, and 13. Bit 0 is the most significant bit.

Figure 2-6 shows the HP3000 real number single-precision bit format.

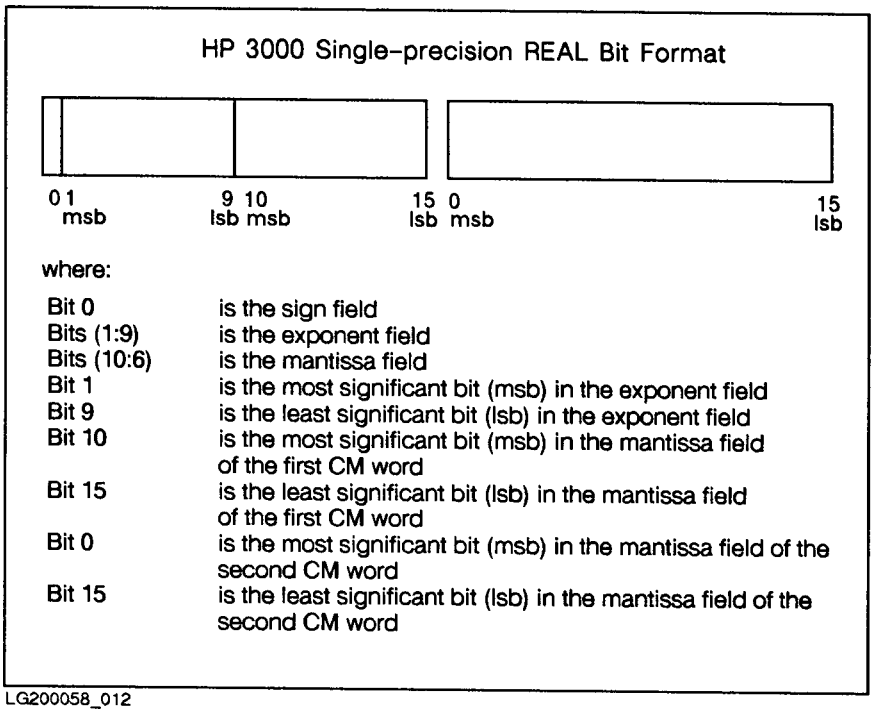

Figure 2-6.
Bit Format: Single-Precision Real in HP3000 Floating-point Notation

Figure 2-7 shows the HP3000 real number double-precision bit format.

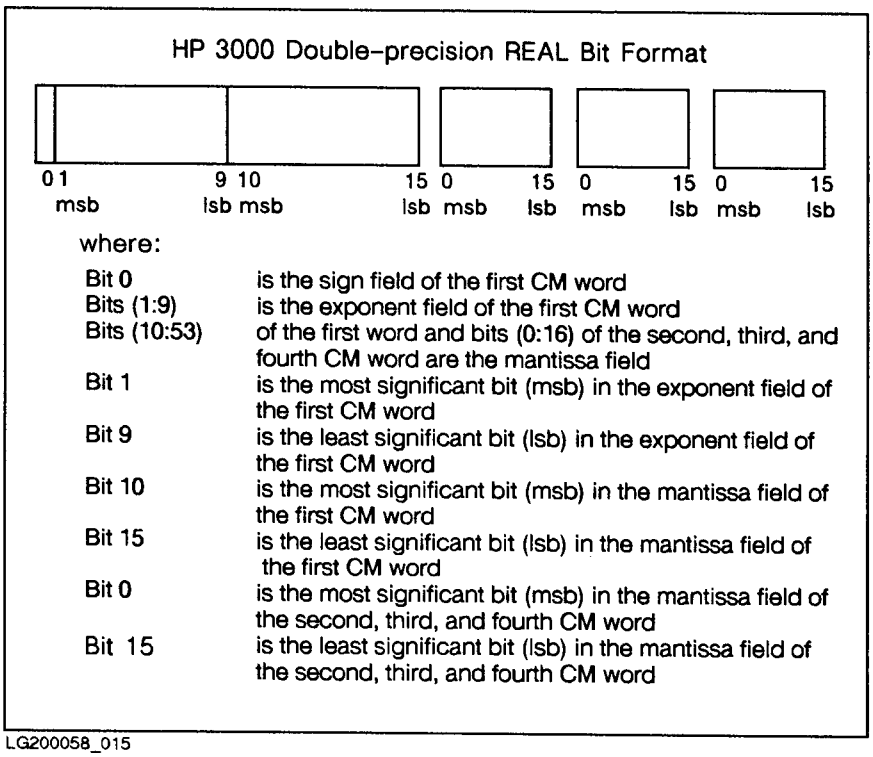

Figure 2-7.
Bit Format: Double-Precision Real in HP3000 Floating-point Notation

Decimals

MPE V has system microcode instructions to handle packed decimals. For compatibility, MPE XL has compiler library procedures that run in NM and emulate the MPE V instruction set.

In MPE XL, three languages use decimal types. COBOL and RPG use packed or unpacked decimals. BASIC has its own type, the floating-point decimal.

In the decimal types, numbers are represented decimal digit by decimal digit. The individual digits of the decimal number are each represented in a BCD (Binary Coded Decimal) nibble. Each nibble is four bits long.

Figure 2-8 shows the bit format for each BCD nibble portion of a decimal.

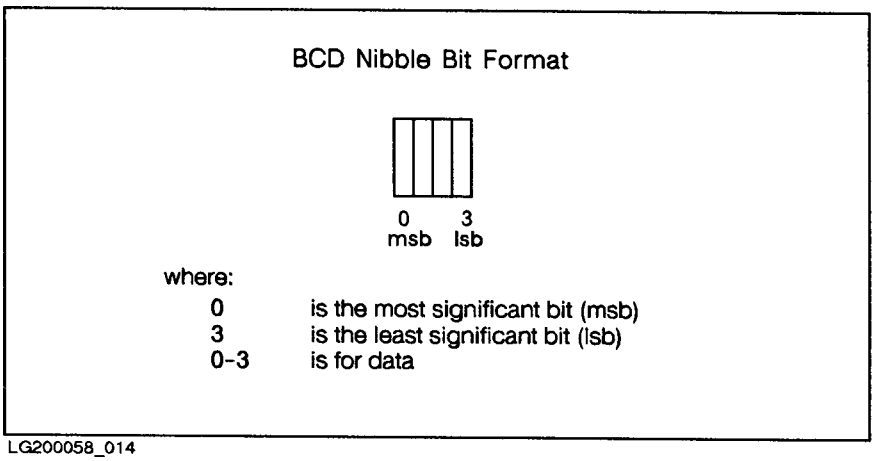

Figure 2-8. Bit Format: BCD Nibble

Packed Decimal Format. Packed decimals represent numbers with BCD (Binary Coded Decimal) nibbles. In packed decimals, each decimal digit of the number is individually represented by a 4-bit BCD.

Decimals are always an even number of nibbles long.

Figure 2-8, above, shows the bit format for each BCD nibble portion of a decimal.

The rightmost (least significant) nibble is for the sign. There are three defined nibble combinations for the sign nibble. The three defined codes are:

hexadecimal C (1100) for positive
hexadecimal D (1101) for negative
hexadecimal F (1111) for unsigned

Since each of the other nibbles represents the decimal digits 0 through 9, the valid nibble combinations are 0000 through 1001 for all but the last nibble.

For example, to represent −52,194 as a packed decimal type, you would use one nibble for each of the five digits and (the last) one for the sign:

```
0101 0010 0001 1001 0100 1101
 5    2    1    9    4   D=negative
```

In COBOL, the PICTURE (PIC) clause specifies the position of the decimal point. For example, the PIC clause 999V99, specifies three digits will be followed by an implied decimal point and two more digits. If you pass the digits 12345 to a variable defined with this PIC clause, its value would be 123.45.

In COBOL and RPG, using packed decimal will probably make your program more efficient than using unpacked. If you do use unpacked decimal, the compiler usually converts to packed for calculations.

Figure 2-9 shows the bit format for the packed decimal.

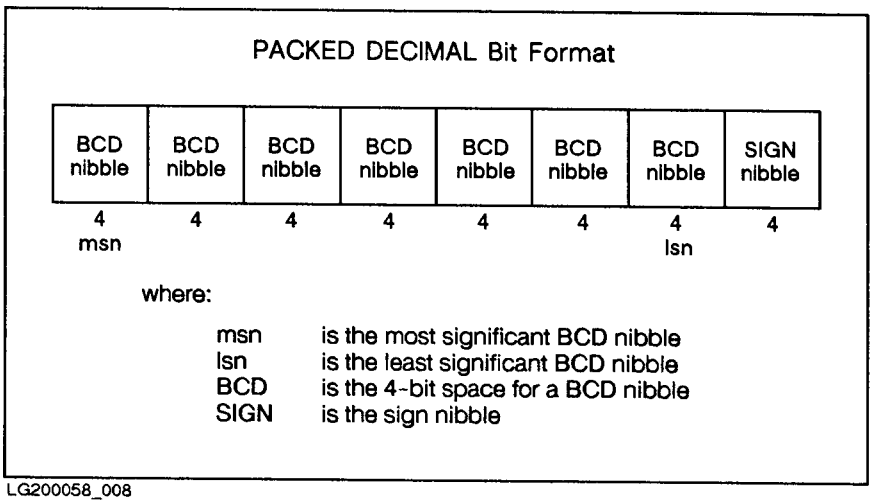

Figure 2-9. Bit Format: Packed Decimal

Unpacked Decimal Format. COBOL and RPG represent numbers with packed and unpacked decimal types. For an unpacked decimal, each decimal digit is one byte long. Unpacked decimals are ASCII characters, interpreted by a correspondence code. The bit format is the ASCII character format in Figure 2-1. For more information, see the notes on COBOL and RPG, "Formatting Data in Programs", later in this chapter.

Floating-Point Decimal Format. HP Business BASIC represents decimal and short decimal types in floating-point decimal notation. The floating-point decimal form is similar to the **E** notation used to represent very small or very large numbers, as when 3.2**E**−27 is used to represent the value 3.2 x 10^{-27}. The BASIC number is normalized (see below).

A decimal in HP Business BASIC/XL is 64 bits long; a short decimal is 32 bits long. Table 2-4, below, shows a summary of the range and accuracy of each.

Table 2-4. Range and Precision for Floating-Point Decimals

	BASIC Decimal	BASIC Short Decimal
Precision:	12 digits	6 digits
Range:	−9.99999999999E511 through −1.00000000000E−511 0 1.00000000000E−511 through 9.99999999999E511	−9.99999E63 through −1.00000−E63 0 1.00000E−63 through 9.9999E63

The representation of the value zero is a special case. To represent the value zero, set all the bits to zero. Since the number is normalized, it is assumed that the mantissa never begins with a zero unless the value of zero is intended.

Fields of BASIC decimals: Floating-point decimals have three fields:

- Exponent.
- Mantissa.
- Sign.

The **exponent field** contains a signed integer, represented in twos complement form. The decimal exponent field is the first 10 bits, bits (0:10), and ranges from −511 to +511. The short decimal exponent field is the first seven bits (bits 0:7) and ranges from −63 to +63.

Note

In this manual, bit fields are described as (*bit*:*length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (11:3)" refers to bits 11, 12, and 13. Bit 0 is the most significant bit.

In the **mantissa field**, each decimal digit of the number is individually represented by a BCD (Binary Coded Decimal) nibble. Each nibble is four bits long. (See Figure 2-8.) Since each nibble in this field represents the decimal digits 0 through 9, the valid mantissa nibble combinations are 0000 through 1001.

The number is normalized. That is,

- The decimal point is implied, or assumed to belong, immediately following the first BCD digit of the mantissa field.
- The first BCD of the mantissa is *never* zero, unless you intend to represent the number zero.

The mantissa field of a 64-bit decimal is bits (12:48). It has the capacity for 12 digits, each represented in a 4-bit nibble. The mantissa field of a 32-bit decimal is bits (8:24). It has the capacity for 6 digits, each represented in a 4-bit nibble.

The **sign field** of a 64-bit decimal is bits (60:4), which are the four least significant bits, or the least significant BCD nibble. The hexadecimal value C (1100) in the sign nibble indicates the number is positive, and D (1101) indicates the number is negative.

The sign field of a 32-bit short decimal is the seventh bit, bit (7:1). A value of 0 in the sign bit indicates the number is positive, and a value of 1 indicates the number is negative.

Figure 2-10 shows the bit format for the floating-point decimal.

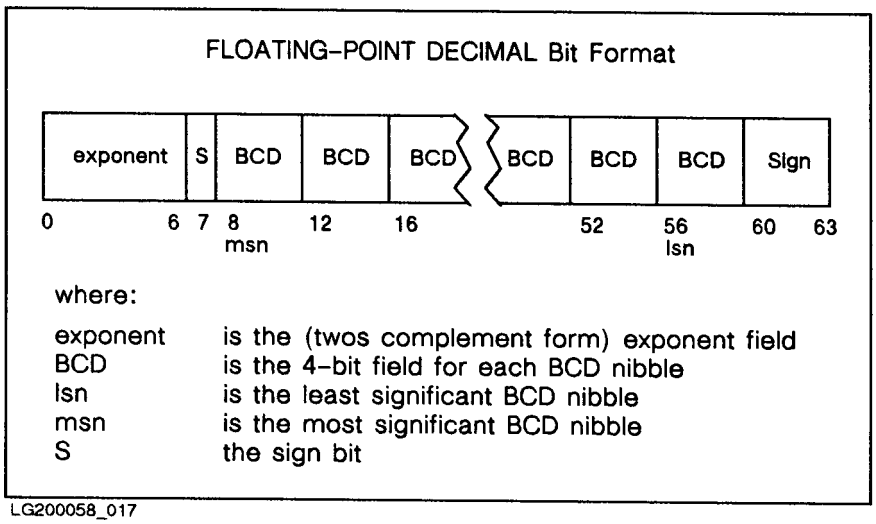

Figure 2-10. Bit Format: Floating-Point Decimal

Figure 2-11 shows the bit format for the short floating-point decimal.

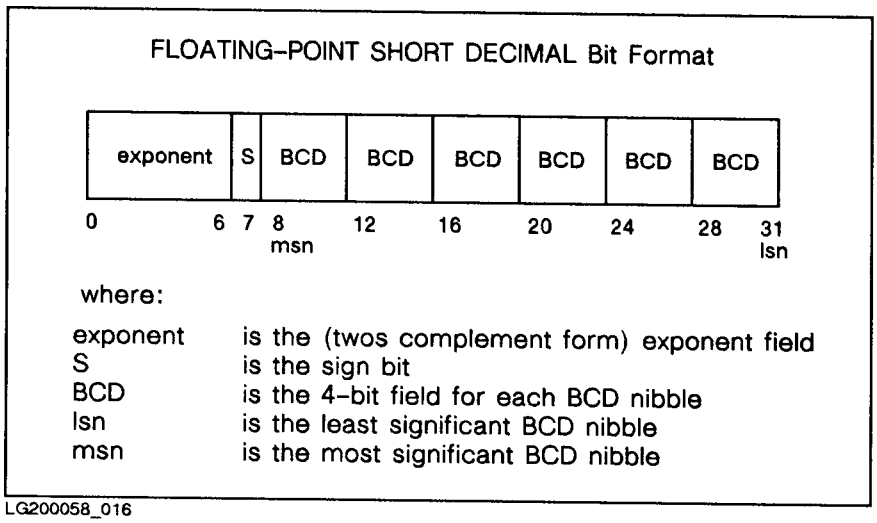

Figure 2-11. Bit Format: Short Floating-Point Decimal

Formatting Data in Programs

The correct format for data in a program depends on the programming environment and the programming language.

NM and CM Programming Environments

MPE XL has two programming environments: NM and CM. NM is based on a 32-bit word. CM emulates MPE V; both are based on a 16-bit word. In NM, data types are aligned on 32-bit boundaries, by default, to improve performance. Many structures that are aligned on 32-bit boundaries in MPE XL NM are aligned on 16-bit boundaries in MPE V/E.

You may have to plan for accurate conversion in mixed-mode applications. Commands and compiler options are provided by supported languages to control alignment.

For example, in COBOL you can choose compiler options for 32-bit NM standard or 16-bit CM standard boundaries. Choose `CALLALIGNED` or `CALLALIGNED[16]` to specify alignment in calling programs, and choose `LINKALLIGNED` and `LINKALLIGNED[16]` to specify program default alignment.

In Pascal and FORTRAN, compiler options HP3000_32 and HP3000_16 specify NM or CM standard alignment and real number format. For example, choosing HP3000_16 will cause the compiler to align data in records on 16-bit CM standard boundaries instead of 32-bit NM standard boundaries, and to format floating point real numbers in HP3000 standard notation instead of IEEE NM notation that is standard in NM.

For an application to use both NM and CM aligned data files, you could specify the program record definitions to force alignment on a structure-by-structure basis. For example, `sync16` or `sync32` in COBOL , or `$alignment$` in Pascal does this.

In this manual, assume that data types are MPE XL Native Mode data types and NM aligned. If CM is meant, CM will be specifically mentioned.

Programming Languages

Table 2-5, following, shows corresponding data types in the different NM languages:

Table 2-5.
Correspondence of Data Types Across Languages: Intrinsics, BASIC, and C

Data Type	Intrinsics	HP Business BASIC/XL	HP C/XL
Character	**C**	$ dimension as 1 character	CHAR or UNSIGNED CHAR
Integer: 16-bit unsigned	**U16**	N/A	UNSIGNED SHORT INT
Integer: 32-bit unsigned	**U32**	N/A	UNSIGNED INT or UNSIGNED LONG INT
Integer: 64-bit unsigned	**U64**	N/A	N/A
Integer: 16-bit signed	**I16**	SHORTINT or subrange [−32768..32767]	SHORT INT
Integer: 32-bit signed	**I32**	INTEGER	INT or ENUM
Integer: 64-bit signed	**I64**	N/A	N/A
Real 32-bit (Single-Precision)	**R32**	SHORTREAL	FLOAT
Real 64-bit (Double-Precision)	**R64**	REAL	DOUBLE
Decimal: Packed	N/A	N/A	N/A
Decimal: Unpacked	N/A	N/A	N/A
Decimal: Floating-Point	N/A	Decimal or Short Decimal	N/A

Table 2-6.
Correspondence of Data Types Across Languages: COBOL, FORTRAN, and Pascal

Data Type	HP COBOL II/XL	HP FORTRAN 77/XL	HP Pascal/XL
Character	DISPLAY or group item	CHARACTER	CHARACTER
Integer: 16-bit unsigned	PIC S9 to PIC S9(4) COMP	LOGICAL OR LOGICAL*2	0..65535 or any 16-bit SUBRANGE
Integer: 32-bit unsigned	PIC S9(5) to PIC S9(9) COMP	LOGICAL OR LOGICAL*4	Any 32-bit subrange
Integer: 64-bit unsigned	PIC S9(10) to PIC S9(18) COMP	N/A	N/A
Integer: 16-bit signed	PIC S9 to PIC S9(4) COMP	INTEGER or INTEGER *2	SHORTINT or any 16-bit subrange
Integer: 32-bit signed	PIC S9(5) to PIC S9(9) COMP	INTEGER or INTEGER *4	INTEGER or any 32-bit subrange
Integer: 64-bit signed	PIC S9(10) to PIC S9(18) COMP	N/A	N/A
Real 32-bit (Single-Precision)	N/A	REAL or REAL*4	REAL
Real 64-bit (Double-Precision)	N/A	DOUBLE PRECISION or REAL*8	LONGREAL
Decimal: Packed	COMP-3	N/A	N/A
Decimal: Unpacked	DISPLAY	N/A	N/A
Decimal: Floating-Point	N/A	N/A	N/A

Language Notes

The following notes relate to the language-specific information in the above table. For further information, consult the manual for the individual languages.

Intrinsics. Intrinsic parameters may require data of type address (`@32` and `@64`). Address is a 32-bit or 64-bit integer that represents a location in memory. The system recognizes by the context of the command that it is to access or operate on the memory cell at that address.

Note

In this manual, bit fields are described as (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, "bits (11:3)" refers to bits 11, 12, and 13. Bit 0 is the most significant bit.

Booleans (`B`) are typically one byte long. Only the rightmost bit, bit (7:1), is interpreted. If this bit is odd (usually 1), the logical value is true; if it is even (usually 0), the boolean value is false. (See the notes for Booleans in HP C/XL).

Records (intrinsic type `REC`) and arrays (`A`) are sets of related data. Each piece of the data is in a field, and the fields are connected. The size and type of the fields are specified when the array or record is created.

The fields in a record can vary in size and can contain various types. The fields in an array, called the elements of the array, are all the same size and all contain the same data types. Arrays are often defined in the supported languages with a notation like [*lb:ub*] or (*lb..ub*), where *lb* is the lower bound of the array, and *ub* is the upper bound.

Arrays and records are complex data types that can themselves be used to build more complex data types, such as an array of records.

For more information about intrinsic data types, refer to *MPE V/E Intrinsics Reference Manual* (32033-90007).

HP Business BASIC/XL. Compiler library routines are designed to work with the packed decimal, not the BASIC floating-point decimal. The BASIC decimal bit format is explained in the previous section on decimals in this chapter.

For more information about HP Business BASIC/XL data types, refer to *HP Business BASIC/XL Reference Manual* (32715-90001).

HP C/XL. HP C/XL notates 32-bit addresses with **name* and 64-bit addresses with *^name*, where *name* is the type of the data being addressed.

In HP C/XL a Boolean is stored as any character type; value of zero represents false, and any non-zero is interpreted as true.

The C language has certain data conversion conventions for parameter passing. It requires that `floats` be converted to `doubles`, that `chars` be converted to `ints`, and that `arrays` of type T be converted to `pointers` to type T. Formal parameters are actually declared as the "converted" type. For example, if you declare a formal parameter as an `array` of type T, it is actually declared as a `pointer` to type T.

For more information about HP C/XL data types, refer to *HP C Reference Manual* (92434-90001) and *HP C/XL Reference Manual Supplement* (31506-90001).

HP COBOL II/XL. HP COBOL II/XL integers may not be an exact match when exchanged with other languages. They match in size and you can store data across languages, but you may have difficulties in computational operations. The COBOL unsigned integer is different from the Pascal integers. COBOL integers may also have a smaller range. (See the section in the next chapter about converting decimals to integers.)

In HP COBOL II/XL, you use clauses to specify three things about data: SIGN, USAGE, and PIC (PICTURE).

The optional SIGN clause indicates whether a number will have an operational sign in non-standard position in its representation.

You indicate the usage of data in the USAGE clause. Usage is DISPLAY (ASCII alphanumeric type) by default. Several COMPUTATIONAL uses can be specified instead.

PIC clauses define type, size and symbols to be inserted into elementary expressions. The PIC clause specifies the number of digits; PIC 9(n) indicates a numeric expression n digits long, and PIC S9 indicates a signed number one digit long. A letter V in the PIC clause indicates the position to insert the the decimal point. (Default is decimal rightmost, or integer.) Other symbols are used for placing such insertions as comma separators.

For example, to set up a field with three digits to the left of the decimal and two to the right, you would specify PIC 999V99, or PIC 9(3)V9(2).

Unpacked Decimals: Unpacked decimals, display type, use ASCII alphanumeric representations for numbers. Represent unsigned digits with the ASCII characters 0 to 9. For signed numbers, use the following ASCII character representations:

{ represents +0

Letters A through I represent digits +1 through +9

} represents −0

Letters J through R represent digits −1 through −9

The COBOL number line for usage display unpacked signed decimal digits looks like this, then:

```
  R  Q  P  O  N  M  L  K  J } { A  B  C  D  E  F  G  H  I
 -+--+--+--+--+--+--+--+--|--+--+--+--+--+--+--+--+--+-
 -9 -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8 +9
```

For further information, refer to *HP COBOL II/XL Programmer's Guide* (31500-90002) or *COBOL II Reference Manual* (31500-90001) and *COBOL II/XL Reference Manual Supplement* (31500-90005).

HP FORTRAN 77/XL. In HP FORTRAN 77/XL, LOGICAL and INTEGER default to 32 bits. Using the `$SHORT` compiler option will cause the default to be 16 bits. Using *2 or *4, however, will override any compiler options.

For further information, refer to *HP FORTRAN 77/XL Programmer's Guide* (31501-90002) or *HP FORTRAN 77/XL Reference Manual* (31501-90010).

HP Pascal/XL. HP Pascal/XL recognizes 32-bit addresses as LOCALANYPTR or any normal pointer type, and 64-bit addresses as GLOBALANYPTR or any pointer type declared with the `$extnaddr$` compiler directive.

For further information, refer to *HP Pascal Reference Manual* (31502-90001), and *HP Pascal Programmer's Guide* (31502-90002).

Language information is in the following manuals:

- *HP Business BASIC/XL Reference Manual* (32715-90001)
- *HP C Reference Manual* (92434-90001) and *HP C/XL Reference Manual Supplement* (31506-90001)
- *HP COBOL II/XL Programmer's Guide* (31500-90002)
- *COBOL II Reference Manual* (31500-90001) and *COBOL II/XL Reference Manual Supplement* (31500-90005)
- *HP FORTRAN 77/XL Reference Manual* (31501-90010)
- *HP FORTRAN 77/XL Programmer's Guide* (31501-90002)
- *HP Pascal Reference Manual* (31502-90001)
- *HP Pascal Programmer's Guide* (31502-90002)

3

Converting Data Types

This chapter discusses converting each of the system data types described in Chapter 2 to each of the others.

Data is information structured in forms that the computer is designed to process. Data conversion is translating that information into another acceptable structure without losing meaning.

If you pass data between routines that do not use the same types or formats, you lose meaning and integrity. The receiving routine may not be able to read the data at all. It may divide the bits it reads into the wrong size chunks. It may interpret the arrangement of bits by its own formatting conventions. The resulting misinterpretation could convey information you did not intend and give you unpredictable results in computations.

Passing Data

To pass data between routines in different languages, you may need to convert it to make it readable and to maintain its integrity. Data created in one routine is formatted according to the language type conventions of that routine; if the receiving routine divides the input bit stream in different places, it will not read the same values. If the receiving and sending routines do not have the same definitions of data types, any data that one routine passes to the other will be meaningless, and any operation on that data will be unpredictable.

Often the programming language has commands or compiler directives the convert or coerce data to the required type. You can sometimes convert input and output data with a command line in a program. Often an assignment statement, like *xtype* := *yvalue* or *xtype*=*yvalue* is sufficient.

Sometimes a more indirect conversion is necessary. Since all data types in all MPE languages are either primitive data types or are built from them, you can translate data from one language into a primitive data type. If necessary, you can then translate the resulting primitive type into the type you need.

High-level languages can access and use most system intrinsics and compiler routines.

MPE V had system-level support of applications written in SPL/V language that perform packed-decimal operations; CM emulates these operations on MPE XL. In NM, compiler library routines can be used

to manipulate decimals. Because the sizes of the operands are passed as parameters, these routines are useful in applications where the field sizes are not known at compile-time, such as general-purpose database applications and report writers.

Packed-decimal procedures must be declared as intrinsics to be called from within high-level NM languages. If speed is a primary concern, consider doing packed-decimal operations within HP COBOL II/XL or HP RPG/XL.

For more information see *MPE XL Intrinsics Reference Manual* (32650-90028) and *Compiler Library/XL Reference Manual* (32650-90029).

Converting from Character:

This section offers suggestions about converting from one character set to another, and from ASCII (character) numbers to numeric data types.

To Other Character

Since all characters are 8 bits long and all are byte-aligned, there is no incompatibility between between languages or environments.

You may want to convert between different character sets. ASCII and EBCDIC are the supported English character sets; several native language character sets like the Japenese EBCDIK and JISCII are also supported.

Between ASCII and EBCDIC

The `CTRANSLATE` intrinsic accepts a string of characters in either ASCII or EBCDIC, and returns a string translated into the other. The translation string is returned the same buffer unless you specify another.

ASCII and EBCDIC are described in Chapter 2 of this book, and their code equivalents are in Appendix A.

Between Native Languages

The Native Language Subsystem (NLS) supports six character sets containing the following native languages:

- USASCII supports NATIVE-3000, an artificial language
- ROMAN8 supports:

  NATIVE-3000
  American English
  Canadian French
  Danish
  Dutch
  English
  Finnish

French
German
Italian
Norwegian
Portuguese
Spanish
Swedish
Icelandic

- KANA8 supports:

  NATIVE-3000
  KATAKANA

- ARABIC8 supports:

  Arabic
  Western Arabic

- Greek8 supports Greek
- Turkish8 supports Turkish

Use the intrinsic `NLINFO` or generate a report from the NLUTIL utility to review information about a character's native language format. For more information, refer to *MPE XL Intrinsics Reference Manual* (32650-90028).

Use the intrinsic `CTRANSLATE` to translate between EBCDIK and JIS(KANA8). EBCDIK is a Hewlett-Packard's Japanese version of EBCDIC. JIS is a Japanese International Standard code (JISCII is a Japanese version of US ASCII); KANA8 is an 8-bit JIS code.

Use the intrinsic `NLREPCHAR` to replace nondisplayable characters. It accepts a character array containing nondisplayable characters and returns another character array with replacement characters. (`NLINFO` can tell you if the character is nondisplayable because it is a control code or an undefined graphic character).

Between Numeric Formats

If you are sending data files to, or receiving data from, a foreign country, you may discover that numbers are represented differently in different native languages. The following intrinsics are used for converting ASCII numbers between native language formats:

- `NLCONVNUM` takes data in a foreign language (one that is supported in NATIVE-3000) and translates it to the system's native language. It accepts a foreign language number with decimal and thousands separators and returns an ASCII number with NATIVE-3000 decimal and thousands separators. As an option, the decimal and thousands separators can be stripped.
- `NLFMTNUM` takes native language data and translates it to a foreign language that is supported by NATIVE-3000. It accepts an ASCII number string, which may include NATIVE-3000 decimal separator, thousands separator, and currency symbol or name. It

returns a string with the same number, but formatted with the decimal separator, thousands separator, and currency symbol or name of the native language you specify.

- `NLNUMSPEC` accepts a string as a logical byte array (minimum 60 bytes) and a language identification number. It returns the string encoded with information (number specifications) about the characteristics of the language, such as

  Direction (left-to-right or right-to-left),
  Digit range.
  Symbols for digits, currency, mathematical operations, and thousands and decimal separators.

For further information about the Native Language Subsystem, refer to *Native Language Programmer's Guide* (32650-90022).

To Integer

Individual languages may have simple assignment commands to accept characters and interpret the symbol as its value. For example, `STRREAD` in Pascal or `READ` in FORTRAN will read an ASCII number and format a variable as the value the character string represents.

The intrinsic `DBINARY` accepts an octal-based, a hexadecimal-based, or a signed decimal-based number in ASCII characters, and returns a 32-bit binary integer in twos complement (signed) form.

A character string beginning with a percent sign (%) is treated as octal (base 8) value. A string beginning with a dollar sign ($) is treated as a hexadecimal (base 16) number. A string begining with a plus sign (+), a minus sign (−), or a number (1 through 9, no leading blanks allowed) is treated as a decimal (base 10) number.

The intrinsic `BINARY` performs a similar operation; it converts a numeric ASCII string into a 16-bit binary value. A parameter indicates the number of input bytes.

The compiler utility procedures, `EXTIN'` (CM) and `HPEXTIN` (NM) can also be used to convert from ASCII to integer. Any fractions in the input number string will be truncated in integer results.

To Real

As with converting from character to integer (above), the simplest way to convert character to real type may be an assignment statement within the language.

The compiler utility procedures `EXTIN'` (CM) and `HPEXTIN` (NM) accept a byte array (a character string of ASCII digits passed by reference) and convert it into an internal representation. Leading blanks in the input string are ignored. A parameter can be set to treat trailing blanks as zeros. Dollar signs and other currency symbols and commas for thousands separators are counted in the input length, but are ignored in the output.

In the *datatype* parameter, you specify which internal representation you want: an integer of 16 bits or 32 bits, or a real of 32 bits or 64

bits. The *result* parameter is a 32-bit pointer to the first word of result storage, according to the type specified. Other parameters let you specify field width, decimal places, exponents, and fractions.

To Packed Decimal

In MPE V, system instructions use packed decimals; for compatibility, MPE XL has compiler library procedures that run in NM and emulate the MPE V instruction set. In MPE XL, the packed decimal type is used only in COBOL or RPG, however. Within COBOL and RPG, use the `MOVE` command to convert easily between types by assignment.

The NM compiler procedure `HPPACCVAD` accepts ASCII digits and returns the packed decimal digits used by MPE V and in COBOL and RPG. The rightmost source digit indicates the sign; all other digits must be unsigned or be leading blanks. Blanks between digits are illegal. Leading blanks are converted to packed-decimal zeros. An all-blank field converts to an unsigned zero. If the source has more digits than the target, the result is left-truncated; if the target is the larger, the result is padded with zeros on the left.

Because this procedure is external and general, it may not be as efficient as code optimized by the NM COBOL compiler. Packed-decimal procedures must be declared as intrinsics to be called from within high-level NM languages. If speed is a primary concern, consider doing packed-decimal operations within HP COBOL II/XL or HP RPG/XL.

Converting from Integer

This section offers suggestions for converting integer data types to character, other integers, real and decimal data types.

To Character

Individual languages may have simple assignment functions to accept integers and store them as characters, such as the `WRITE` and `STRWRITE` command.

Signed integers in MPE XL NM are in twos complement form. MPE XL uses a 32-bit standard word in NM, and a 16-bit standard word in CM, as explained in Chapter 1, Introduction.

The intrinsic `DASCII` accepts a 32-bit signed integer by value. It returns the value as an ASCII string to your character array and gives you the number of characters in the result string. You specify, in the parameters, whether the returned string is to be an octal (base-8), a decimal (base-10), or a hexadecimal (base-16) number. Different bases are returned with different justifications and lengths.

The intrinsic `ASCII` performs a similar operation with a 16-bit integer.

The compiler utility procedures INEXT' and HPINEXT also accept an integer and return a character string of ASCII digits.

To Other Integer

All languages supported on 900 Series HP 3000 Computer Systems with the MPE XL operating system have a way, within the language, to assign value from one integer type to the other integer types.

To Real

Most high-level languages have assignments to convert integers to reals within the language.

The compiler functions DFLOAT and DFLOAT' convert a 32-bit integer into a 64-bit real number.

To Packed Decimal

The compiler procedure HPACCVBD converts a signed binary integer to a packed decimal. The input number is considered to be in twos complement form, from 2 to 12 bytes long.

Packed-decimal procedures must be declared as intrinsics to be called from within high-level NM languages. If speed is a primary concern, consider doing packed-decimal operations within HP COBOL II/XL or HP RPG/XL.

Converting From Real

This section offers some suggestions for converting real numbers in floating-point notation to character, integer, other real, and decimal data types.

To Character

Within languages, there is usually a command or function like WRITE or STRWRITE that will take a real value and print, display, or store it as a string of ASCII characters.

The compiler utility procedures INEXT' and HPINEXT convert a real number to a byte array for an output string of ASCII digits. The resulting ASCII string can be represented in several formats. You can choose options for representing the sign character, decimal points, and the exponent.

To Integer

As mentioned in the section above, most languages contain an internal assignment function to format a real value as an integer. Rounding and truncation rules differ.

The compiler functions INT, INT', IFIX, and IFIX' all accept a 32-bit real number and return it, truncated, as an 16-bit integer.

Similarly, DFIX or DFIX' accept a 64-bit real number, truncate it, and return it the result as a 32-bit integer.

If you round a number equidistant from two adjacent integers, like 1.5 or 2.5, you may find that IEEE and HP3000 return different results. In IEEE, a midpoint number rounds to the integer that has

a least significant bit of zero; in other words, the even integer. For example, −1.5 rounds to −2, and 2.5 rounds to 2. HP3000 rounds to the integer of greatest magnitude. For example, −1.5 rounds to −2 and 2.5 rounds to 3. Rounding directives within a language behave in language-specific ways; consult the language manual, or test a mid-point number if you are doubtful.

To Other Real

As discussed in Chapter 2, there are two formats for real floating-point numbers in MPE XL: IEEE and HP3000. Conversions between the two can be done by choosing a particular compiler, or by calling the intrinsic `HPFPCONVERT`. In addition, there are system procedures that will truncate a fractional real number.

Real floating point numbers in this manual are assumed to be in IEEE format, which is the default representation in NM. In CM data files or programs, floating point real numbers default to HP3000 format, an MPE XL emulation of the MPE V/E format.

If do not want the default IEEE real number format for a particular application, you can force the HP3000 format by specifying the HP3000_16 compiler directive in HP FORTRAN 77/XL and HP Pascal. HP3000_16 selects MPE V/E alignment and HP3000 real number format. As mentioned in Chapter 2, this also changes the alignment.

Although you can use different formats for separate external procedures, you can only use one real number format within an executable module. IEEE and HP3000 single-precision and double-precision real numbers have different accuracies and ranges. You can convert between binary floating-point formats with the intrinsic `HPFPCONVERT`.

You can specify any binary floating-point real number for input to `HPFPCONVERT`, and ask for your output in any legal format. Acceptable legal formats for source and destination are:

- HP3000:
  - 32-bit
  - 64-bit
- IEEE:
  - 32-bit
  - 64-bit

The conversion is performed by regarding the source number as infinitely precise and with unbounded range, and then rounding it to fit the designated destination format. You have some choice in the rounding mode.

The method of rounding and the way exceptions are signalled depends entirely on the destination format, not the source.

Conversion is performed as if all arithmetic traps are disabled. No trapping to user-supplied or system-supplied arithmetic trap routines is done.

You may encounter two types of errors:

- Underflow or Overflow
- Inexact

Overflow and Underflow

Conversion between formats can present a range problem, when the target range is smaller. Thus, overflow and underflow can occur in performing either of the following conversions:

- From a HP3000 single-precision real number to an IEEE single-precision number.
- From an IEEE double-precision real number to an HP3000 double-precision.

You may have to develop new error handling code to prevent this.

Accuracy

The mantissa of an HP3000 double-precision real number provides enough bits for 16 digits of accuracy. The mantissa of an IEEE double-precision real number provides for 15.9 digit of accuracy. Conversion from HP3000 format to IEEE double-precision may cause the least significant digit of a 16-digit real number to be lost.

The loss of numeric precision is extremely small. However, if the requirements of an application depend on the ASCII representation of floating-point results, the effect could be important. For example, if a program assumed 16-digit accuracy and requested 16 digits for formatting output, with trailing zero suppression, the number 64.4 would appear as 64.4 when the system was running in CM, but would appear as 64.40000000000001 when the system was running in NM.

Truncating

The compiler function `AINT` or `AINT'` accepts a 32-bit real number and truncates it to return an integer-like number in 32-bit real representation.

The compiler function `DDINT` or `DDINT'` truncates a 64-bit longreal number to return an integer-like number in 64-bit longreal representation.

To Packed Decimal

In languages other than COBOL and RPG, follow these steps to convert from an input real to a packed decimal:

1. Multiply or divide the real number by an appropriate power of 10.
2. Convert the resulting value to an base-ten integer.
3. Convert that integer to a decimal.

(See the previous sections about making these conversions.)

If your conversion is taking place *within* COBOL or RPG, you cannot operate on a real number, as required in step 1 above. Instead, follow these steps:

1. Convert the real number into a character.
2. Convert the resulting character to a decimal.

(See the previous sections about making these conversions.)

Converting from Packed Decimal

This section offers some suggestions converting packed decimal data types to character, integer, and real data types.

Compiler library routines can be used to manipulate decimals in NM. Because the sizes of the operands are passed as parameters, these routines are useful in applications where the field sizes are not known at compile-time, such as general-purpose database applications and report writers.

The compiler library packed-decimal procedures must be declared as intrinsics if you use them in high-level NM languages. If speed is a primary concern, consider doing packed-decimal operations within HP COBOL II/XL or HP RPG/XL.

If you are working within COBOL or RPG, you would use `PIC` clauses and the `MOVE` command to convert types. The following suggestions are for situations where you have other languages involved.

To Character

`HPPACCVDA` accepts a packed-decimal number and returns an ASCII representation of the number. An unsigned source produces an unsigned result; if the source a signed decimal, you specify whether the target will be signed. You specify the number of digits in the result.

To Integer

`HPPACCVDB` accepts a packed decimal and returns an integer. The integer is a signed binary number in twos complement form, and its size depends on the number of digits in the source.

To Real

If you are working outside COBOL or RPG, you would convert indirectly as follows:

1. Convert the decimal value to an integer.
2. Convert the resulting integer to a real number.
3. Multiply or divide by the appropriate power of ten.

(See the previous sections about making these conversions.)

If you are working in COBOL, you can convert the decimal value to an ASCII integer. Pass this to the routine, and convert it to a real value there.

To Other Decimals

The `MOVE` command is used to change one decimal to another within COBOL or RPG.

Outside of COBOL or RPG, use the compiler library functions `HPPACSRD` and `HPPACSLD` to perform right and left shifts on packed decimals. You specify the amount of offset (the number of digits to be shifted).

To convert a packed decimal to a BASIC decimal, you should convert first to a twos complement integer or type ASCII, and then convert to decimal within BASIC with an assignment. For example, assign an integer value to a decimal with *decval* = *intval* * *n*0, where *n*00 is the appropriate power of 10. To convert between ASCII and decimal, use the `VAL` or `VAL$` internal functions.

A

ASCII and EBCDIC Code Values

The following table shows ASCII and EBCDIC character code values along with their decimal, octal, and hexadecimal equivalents. Control/graphic abbreviations, like NUL and SOH, are spelled out at the end of the table.

ASCII/EBCDIC Character Sets

ASCII Control/ Graphic	EBCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
NUL	NUL	0	000	00
SOH	SOH	1	001	01
STX	STX	2	002	02
ETX	ETX	3	003	03
EOT	PF	4	004	04
ENQ	HT	5	005	05
ACK	LC	6	006	06
BEL	DEL	7	007	07
BS		8	010	08
HT		9	011	09
LF	SMM	10	012	0A
VT	VT	11	013	0B
FF	FF	12	014	0C
CR	CR	13	015	0D
SO	SO	14	016	0E
SI	SI	15	017	0F
DLE	DLE	16	020	10
DC1	DC1	17	021	11
DC2	DC2	18	022	12
DC3	TM	19	023	13
DC4	RES	20	024	14
NAK	NL	21	025	15
SYN	BS	22	026	16
ETB	IL	23	027	17

Table A-1. ASCII/EBCDIC Character Sets

ASCII Control/ Graphic	EDCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
CAN	CAN	24	030	18
EM	EM	25	031	19
SUB	CC	26	032	1A
ESC	CU1	27	033	1B
FS	IFS	28	034	1C
GS	IGS	29	035	1D
RS	IRS	30	036	1E
US	IUS	31	037	1F
SP	DS	32	040	20
!	SOS	33	041	21
"	FS	34	042	22
#		35	043	23
$	BYP	36	044	24
%	LF	37	045	25
&	ETB	38	046	26
'	ESC	39	047	27
(		40	050	28
)		41	051	29
*	SM	42	052	2A
+	CU2	43	053	2B
,		44	054	2C
-	ENQ	45	055	2D
.	ACK	46	056	2E
/	BEL	47	057	2F
0		48	060	30
1		49	061	31
2	SYN	50	062	32
3		51	063	33
4	PN	52	064	34
5	RS	53	065	35
6	UC	54	066	36
7	EOT	55	067	37

ASCII Control/ Graphic	EDCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
8		56	070	38
9		57	071	39
:		58	072	3A
;	CU3	59	073	3B
<	DC4	60	074	3C
=	NAK	61	075	3D
>		62	076	3E
?	SUB	63	077	3F
@	SP	64	100	40
A		65	101	41
B		66	102	42
C		67	103	43
D		68	104	44
E		69	105	45
F		70	106	46
G		71	107	47
H		72	110	48
I		73	111	49
J		74	112	4A
K	.	75	113	4B
L	<	76	114	4C
M	(	77	115	4D
N	+	78	116	4E
O	|	79	117	4F
P	&	80	120	50
Q		81	121	51
R		82	122	52
S		83	123	53
T		84	124	54
U		85	125	55
V		86	126	56
W		87	127	57

ASCII/EBCDIC Character Sets

ASCII Control/ Graphic	EDCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
X		88	130	58
Y		89	131	59
Z	!	90	132	5A
[	$	91	133	5B
\	*	92	134	5C
]	)	93	135	5D
^	;	94	136	5E
_		95	137	5F
`	-	96	140	60
a	/	97	141	61
b		98	142	62
c		99	143	63
d		100	144	64
e		101	145	65
f		102	146	66
g		103	147	67
h		104	150	68
i		105	151	69
j		106	152	6A
k	,	107	153	6B
l	%	108	154	6C
m	_	109	155	6D
n	>	110	156	6E
o	?	111	157	6F
p		112	160	70
q		113	161	71
r		114	162	72
s		115	163	73
t		116	164	74
u		117	165	75
v		118	166	76
w		119	167	77

ASCII/EBCDIC Character Sets, continued

ASCII ASCII Control/ Graphic	EDCDIC EBCDIC Control/ Graphic	Character Code Values Character Code Values Decimal	Octal	Hexadecimal
x		120	170	78
y		121	171	79
z	:	122	172	7A
{	#	123	173	7B
|	@	124	174	7C
}	'	125	175	7D
~	=	126	176	7E
DEL	"	127	177	7F
		128	200	80
	a	129	201	81
	b	130	202	82
	c	131	203	83
	d	132	204	84
	e	133	205	85
	f	134	206	86
	g	135	207	87
	h	136	210	88
	i	137	211	89
		138	212	8A
		139	213	8B
		140	214	8C
		141	215	8D
		142	216	8E
		143	217	8F
	j	144	220	90
	k	145	221	91
	l	146	222	92
		147	223	93
	m	148	224	94
	n	149	225	95
	o	150	226	96
	p	151	227	97

ASCII/EBCDIC Character Sets

ASCII Control/ Graphic	EDCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
	q	152	230	98
	r	153	231	99
		154	232	9A
		155	233	9B
		156	234	9C
		157	235	9D
		158	236	9E
		159	237	9F
		160	240	A0
	~	161	241	A1
	s	162	242	A2
	t	163	243	A3
	u	164	244	A4
	v	165	245	A5
	w	166	246	A6
	x	167	247	A7
	y	168	250	A8
	z	169	251	A9
		170	252	AA
		171	253	AB
		172	254	AC
		173	255	AD
		174	256	AE
		175	257	AF
		176	260	B0
		177	261	B1
		178	262	B2
		179	263	B3
		180	264	B4
		181	265	B5
		182	266	B6
		183	267	B7

ASCII Control/ Graphic	EBCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
		184	270	B8
		185	271	B9
		186	272	BA
		187	273	BB
		188	274	BC
		189	275	BD
		190	276	BE
		191	277	BF
		192	300	C0
	A	193	301	C1
	B	194	302	C2
	C	195	303	C3
	D	196	304	C4
	E	197	305	C5
	F	198	306	C6
	G	199	307	C7
	H	200	310	C8
	I	201	311	C9
		202	312	CA
		203	313	CB
		204	314	CC
		205	315	CD
		206	316	CE
		207	317	CF
		208	320	D0
	J	209	321	D1
	K	210	322	D2
	L	211	323	D3
	M	212	324	D4
	N	213	325	D5
	O	214	326	D6
	P	215	327	D7

ASCII/EBCDIC Character Sets

ASCII Control/ Graphic	EBCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
	Q	216	330	D8
	R	217	331	D9
		218	332	DA
		219	333	DB
		220	334	DC
		221	335	DD
		222	336	DE
		223	337	DF
	\	224	340	E0
		225	341	E1
	S	226	342	E2
	T	227	343	E3
	U	228	344	E4
	V	229	345	E5
	W	230	346	E6
	X	231	347	E7
	Y	232	350	E8
	Z	233	351	E9
		234	352	EA
		235	353	EB
		236	354	EC
		237	355	ED
		238	356	EE
		239	357	EF
	0	240	360	F0
	1	241	361	F1
	2	242	362	F2
	3	243	363	F3
	4	244	364	F4
	5	245	365	F5
	6	246	366	F6
	7	247	367	F7

ASCII Control/ Graphic	EBCDIC Control/ Graphic	Character Code Values		
		Decimal	Octal	Hexadecimal
	8	248	370	F8
	9	249	371	F9
		250	372	FA
		251	373	FB
		252	374	FC
		253	375	FD
		254	376	FE
		255	377	FF

NUL	Null
SOH	Start of Heading
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquiry
ACK	acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal Tabulation
LF	Line Feed
VT	Vertical Tabulation
FF	Form Feed
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1 (X-ON)
DC2	Device Control 2
DC3	Device Control 3 (X-OFF)
DC4	Device Control 4
NAK	Negative Acknowledge
SYN	Synchronous Idle
ETB	End of Transmission Block
CAN	Cancel
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator
SP	Space (Blank)
DEL	Delete

Index

C

D