SPL to HP C/XL Migration Guide

HP 3000 MPE/iX Computer Systems

Edition 2

HEWLETT®
PACKARD

Manufacturing Part Number: 30231-90001
E1089

U.S.A. October 1989

Notice

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 or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing 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.

Restricted Rights Legend

Acknowledgments

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

SPL to HP C/XL Migration Guide

SPL to HP C/XL Migration Guide

Printed in U.S.A.

900 Series HP 3000 Computer Systems
HP Part No. 30231-90001
Edition Second Edition
Printed Oct 1989
E1089

Printing History

The following table lists the various printings of this manual, together with the respective release date for each edition or update. The software code (Product VUF) printed alongside the release date indicates the version, update, and fix level of the software product at the time the manual edition or update was issued. Many software updates and fixes do not require changes to the manual. Therefore, do not expect a one-to-one correspondence between product updates and manual editions or updates.

Edition Number	Release Date	Product VUF	
First Edition	February 1989	SPL HP C/XL	32100A.08.07 31506A.00.02
Second Edition	October 1989	SPL HP C/XL	32100A.08.09 31506A.02.03

Additional Documentation

The following publications provide information that can help you migrate SPL programs to HP C/XL.

Number to Use to Order Manual	Manual Part Number	Manual Title
30000-90024	30000-90024	*Systems Programming Language Reference Manual*
30000-90025	30000-90025	*Systems Programming Language Textbook*
31506-60001	92434-90001	*HP C Reference Manual*
31506-60001	31506-90001	*HP C/XL Reference Manual Supplement*
31506-60001	30026-90001	*HP C/XL Library Reference Manual*
31506-60002	92434-90002	*HP C Programmer's Guide*
30367-60003	30367-90007	*Migration Process Guide*
30367-60004	30367-90005	*Introduction to MPE XL for MPE V Programmers*
32650-60002	32650-90003	*MPE XL Commands Reference Manual*
32650-60013	32650-90028	*MPE XL Intrinsics Reference Manual*
32650-60030	32650-90014	*Switch Programming User's Guide*
31502-60006	31502-90002	*HP Pascal Programmer's Guide*

Preface

The *SPL to HP C/XL Migration Guide* describes how to convert SPL programs to HP C/XL. It is intended for experienced SPL programmers who are also acquainted with the C language.

The guide is organized to parallel the *Systems Programming Language Reference Manual*. Chapter 2 of this guide corresponds to chapter 1 of the reference manual, and so forth. The topics are presented in the same order.

Section	Description
Chapter 1	Provides a general overview of the migration process.
Chapter 2	Highlights the differences between the SPL and HP C/XL source formats.
Chapter 3	Describes the differences in data storage formats, constants, identifiers, arrays, and pointers.
Chapter 4	Describes the differences in global declarations.
Chapter 5	Describes conversions for SPL arithmetic and logical expressions, and assignment, MOVE, and SCAN statements.
Chapter 6	Describes conversions for SPL program control statements.
Chapter 7	Suggests some HP C/XL alternatives for SPL ASSEMBLE statements.
Chapter 8	Describes the conversions required for SPL procedures, local declarations, and subroutines.
Chapter 9	Discusses the conversion of SPL input/output intrinsics to HP C/XL standard functions that perform analogous operations.
Chapter 10	Describes the differences between the SPL compiler commands and the HP C/XL compiler directives.
Chapter 11	Discusses a method for converting SPL programs into HP C/XL.
Appendix A	Lists SPL procedures that are used as a first step toward converting to the HP C/XL macros and functions listed in Appendix B.
Appendix B	Lists HP C/XL functions that emulate special features of the SPL language.

Conventions

This section discusses the notation conventions followed in this manual. "Syntax" deals with the notation used in syntax diagrams. "General" discusses other aspects of textual notation and practices.

Syntax

Notation	**Description**
computer	Letters, digits, and special characters displayed in "computer" type are required and should be entered exactly as shown. SPL permits keywords to be upper- or lowercase. HP C/XL differentiates uppercase from lowercase. In the following example, both the command and the trailing semicolon are required: EXIT ;
italics	Characters in "*italics* ", typically words or compound words, denote elements that you must replace with appropriate values. In the following example, you must replace *filename* with the name of the file you want to close: CLOSE *filename*
[element]	Brackets enclose optional elements.+ When one or more elements are stacked inside brackets, you may select any one or none of the elements. For example: [A] [B] [C] You can select "A" or "B" or neither, and optionally add "C". When brackets are nested, parameters in inner brackets can be specified only if parameters in outer brackets are specified. For example: [X1 [, [X2] [, X3]]] can be entered as any of: *blank* X1 X1, X1,X2 X1,X2,X3 X1,,X3
{ element }	Braces enclose required elements. When one or more

elements are stacked within braces, you must select one of those elements. For example:

```
{A}
{B}
{C}
```

You must select "A" or "B" or "C".

Notation	Description
[...]	A horizontal ellipsis enclosed in brackets indicates that the previous element, usually a selection enclosed in brackets or braces, may be repeated one or more times, separated, if necessary, by spaces. For example: [, *itemname*] [...]
[,...]	If the ellipsis is preceded by a punctuation mark, such as comma or semicolon, you must use that character to separate repetitions of the element. [*item1*] [*item2*] [,...]
" [" "] "	Where special characters that have syntactic meaning, such as the square brackets above, are required to be entered as text, they are shown in "computer" type, enclosed in "right-hand" quotation marks. The syntax: *arrayname " [" subscript* [,...] *"] "* represents the following examples: ABC[25,77] Aardvark [noselength]

General

Notation	Description
...:	Within examples, vertical and horizontal ellipses show where portions of the example have been omitted.
bit *n*	The bits in bytes, bit-fields, words, etc. are numbered from left to right from zero. In a 16-bit SPL "word", bit zero is the high-order left-hand bit and bit 15 is the low-order right-hand bit. In a 32-bit HP C/XL "word", bit zero is the high-order left-hand bit and bit 31 is the low-order right-hand bit.

Chapter 1 SPL Migration

System Programming Language (SPL) is a language that was developed for the older HP 3000 computer systems, which currently run under the MPE V operating system.

HP C/XL is the Hewlett-Packard implementation of the C programming language on the HP Precision Architecture 900 Series HP 3000 computer systems, which run under the MPE XL operating system.

This guide will use the terms MPE V and MPE XL to refer to the two distinct architectures and operating environments.

SPL was designed for systems programmers, in order to give them close control over the hardware stack, registers, and segmentation of the MPE V and earlier operating environments. Many SPL features are hardware-dependent--designed for specific machine instructions and registers. Most SPL special features are inappropriate for the MPE XL environment. Many of them are used chiefly to deal with the lack of space in the MPE V data area, a problem that largely disappears in the MPE XL environment.

For a general discussion of MPE V to MPE XL migration issues, please read the *Migration Process Guide* and the *Switch Programming User's Guide*.

Migration Choices

Many programs and systems developed and written in SPL are difficult to replace. To solve this problem, MPE XL offers a range of migration options for SPL programs:

* Emulate the MPE V environment with Compatibility Mode.

* Convert the program code with the Object Code Translator.

* Convert source programs to a Native Mode language implemented on MPE XL machines.

Compatibility Mode

SPL programs may be compiled and run on MPE XL machines immediately, without code changes. They automatically run in Compatibility Mode, which emulates the MPE V environment. However, emulation lowers efficiency, sometimes dramatically.

CAUTION Applications running in Compatibility Mode **must not** execute privileged instructions; they must call only documented, callable MPE V/E or subsystem intrinsics. However, they may enter Privileged Mode and may call MPE V/E privileged intrinsics from Compatibility Mode.

Object Code Translation

The object code translation program, OCT, which is available on MPE XL machines, translates many of the MPE V instructions in a compiled object file into MPE XL instructions. While such a translated program must still run in Compatibility Mode, it may run faster than an untranslated program. In general, OCT provides higher performance at the expense of a larger program size and greater difficulty in debugging. OCT may be executed with the MPE XL :OCTCOMP command. See the *MPE XL Commands Reference Manual* for details.

Conversion to Another Language

Compatibility Mode and object code translation may be sufficient for many
applications. However, any program that requires maximum efficiency or is enhanced and upgraded regularly should be converted to a language that
generates Native Mode instructions on MPE XL machines. HP C/XL is the recommended migration language. COBOL II/XL, HP FORTRAN 77/XL, and HP Pascal/XL, are suitable alternatives.

This migration guide addresses the option of converting SPL source code to HP C/XL.

Converting SPL to HP C/XL

SPL is a procedure-oriented language. The basic structure of SPL and most of the language constructs are machine-independent. However, machine-dependent constructs are embedded within SPL to allow systems programmers to optimize programs and access system-specific hardware features.

The C language is a portable, machine-independent programming language. Like SPL, C is a procedure-oriented language that uses many similar constructs. This similarity, while making C a good candidate for converting SPL programs, initially may cause some difficulties for experienced SPL programmers. For example: C uses "=" as the assignment operator; SPL uses ":="; C uses "==" as the equality operator; SPL uses "=".

HP C/XL is the Hewlett-Packard implementation of C on MPE XL machines.

HP C/XL is a highly portable version of the C language.

SPL programs that rarely use machine-dependent constructs are easy to translate to HP C/XL. Consequently, the first step in any SPL to HP C/XL conversion is to isolate, and, if possible, eliminate the use of machine-dependent SPL features. Machine-dependent SPL features include direct reference to hardware registers, assembly instructions, and explicit stack manipulation. Many of these operations are used to optimize the MPE V environment and can be easily rewritten in higher level SPL constructs that can be converted directly to HP C/XL.

Machine-dependent SPL features allow access to extra data segments to overcome the limited address space on MPE V machines. This restriction is not present in MPE XL, so these routines may be simplified or eliminated. Such changes can be made (but not tested) in the MPE V environment.

SPL programs sometimes rely upon the hardware stack environment of MPE V machines. MPE XL machines do not have hardware stacks. Although you could emulate a stack in software, using HP C/XL constructs and data structures, usually the better choice is to redesign the algorithm and rewrite the affected program.

Some high-level SPL constructs can be rewritten using alternative SPL operations that are easier to translate into HP C/XL. For example, SPL allows subroutines to be local to procedures. Although HP C/XL does allow nested blocks (compound statements with local data), HP C/XL does not allow any nesting of functions. Rewriting an SPL program to eliminate subroutines, either by placing the code inline, or by converting the SPL subroutine into an SPL procedure, will allow direct translation of the program structure into HP C/XL.

Conversion Strategy

This guide describes a four-step procedure for converting an SPL program to HP C/XL:

1. Remove as many hardware-dependent SPL constructs as possible from the SPL program. Recompile and test.

2. Rewrite other SPL constructs into forms that convert easily to HP C/XL. Recompile and test.

3. Convert the SPL source code to HP C/XL source code, rewriting as little as possible. Compile and test.

4. Make improvements in the HP C/XL source code.

This procedure is described in detail in Chapter 11.

For large programs, you may consider a phased migration. You could convert the main program first and use the switch subsystem to access the remaining SPL code (e.g., in subprograms). See the *Switch Programming User's Guide* for details.

The following chapters parallel the *Systems Programming Language Reference Manual*, section for section, discussing the conversion issues involved.

Major Considerations

MPE V and MPE XL have two areas of incompatibility that may make it difficult for you to convert SPL programs to HP C/XL:

* The representation of floating-point numbers

* Data storage alignment

Floating-Point Numbers

MPE XL floating-point numbers are represented in the industry-standard IEEE format. This format is different from the MPE V format in bit layout, range, and precision. (Range is governed by the size of the exponent; precision is governed by the size of the fraction.)

MPE V 32-bit floating-point numbers:

Bit layout: 1-bit sign, 9-bit exponent, 22-bit fraction
Nonzero range: 8.63617x10-78 to 1.157921x1077

MPE XL 32-bit floating-point numbers:

Bit layout: 1-bit sign, 8-bit exponent, 23-bit fraction
Nonzero range: 1.754944x10-38 to 3.4028235x1038

MPE V 64-bit floating-point numbers:

Bit layout: 1-bit sign, 9-bit exponent, 54-bit fraction
Nonzero range: 8.63618555094445x10-78 to 1.157920892373162x1077

MPE XL 64-bit floating-point numbers:

Bit layout: 1-bit sign, 11-bit exponent, 52-bit fraction
Nonzero range: 2.2250738585072014x10-308 to 1.7976931348623157x10308

MPE XL 32-bit floating point has greater precision but a smaller range than MPE V. Thus, it is possible to have a valid MPE V floating-point number that is not representable in MPE XL floating point.

On the other hand, MPE XL 64-bit floating-point numbers can handle a much higher range than MPE V 32-bit or 64-bit floating point, but they have less precision than MPE V 64-bit floating point.

The data storage formats are quite different, corresponding to the bit representations noted above. Floating-point data stored on disk must be converted or replaced if the programs are converted to HP C/XL.

The MPE XL intrinsic HPFPCONVERT may be used to convert floating point data to and from the various representations. See the *MPE XL Intrinsics Reference Manual* for details.

Data Storage Alignment

On MPE V, a data item whose size is two bytes or greater is aligned on a two-byte boundary.

On MPE XL, a data item is aligned on a boundary not less than the size of the data item itself, that is, a multiple of 1, 2, 4, or 8 bytes.

Thus, a character followed by a 64-bit floating-point number would require 10 bytes in MPE V and 16 bytes in MPE XL.

In MPE V, the character would start at byte 0, there would be one unused byte, and the floating-point number would start at byte 3. In MPE XL, the character would start at byte 0, there would be seven unused bytes, and the floating point number would start at byte 8.

This incompatibility of data storage affects program access to data both in memory and on disk.

Chapter 2 Program Structure

This chapter discusses conversion issues that correspond to sections in Chapter 1 of the *Systems Programming Language Reference Manual*.

Introduction

SPL is particularly designed to access machine-dependent features of the MPE V operating system. The conversion to HP C/XL requires that these machine-dependent features be removed.

Conventions

Table 2-1. Bit Numbering

SPL	HP C/XL Equivalent
Bits are numbered left to right, 0 to 15 in a word, 0 to 31 in a double word, etc. Bit zero is the "high-order" bit.	Not specified. For convenience, this manual will follow the SPL conventions.

An MPE V word is 16 bits long; an MPE XL word is 32 bits. In general, the word size is not a serious problem in the conversion process, since corresponding data types are available. Specific considerations are noted where they apply.

Source Program Format

Table 2-2. Source Program Format

SPL	HP C/XL Equivalent
Records are 80 columns long.	Record lengths are not restricted.
Free field format in columns 1 through 72.	Free field format in all columns.
Columns 73 through 80 may be sequence numbers.	Last eight characters of records are interpreted as sequence numbers if ALL

	eight characters are ASCII numeric.
Statement labels are identifiers followed by colon ("*label*:").	Same as SPL.
A compilation unit is bracketed by the reserved words BEGIN and END and terminated with a period (".")	A compilation unit has no special delimiters. It consists of declarations and one or more function definitions.
Compiler commands are denoted by a "$" in column 1.	Compiler directives are denoted by a "#" in column 1.
A compiler command line is continued to the next line by having "&" as its last nonblank character.	A directive line is continued by having "\" as the last nonblank.
Tokens may not be broken across records.	Same as SPL.
Source input is *not* sensitive to case. (Variable Var1 is the same as var1.)	Source input *is* case sensitive. (Variable Var1 is different from var1.) All HP C/XL keywords must appear in lowercase.

Delimiters

Table 2-3. Delimiters

SPL	**HP C/XL Equivalent**
Blanks and special characters (other than apostophes) act as delimiters to reserved words and identifiers. Apostrophes, "'", may be used in identifiers.	Similar, except that underscore, "_", assumes the role of apostrophe in identifiers and as a nonseparator. That is, change "'" to "_".
Blanks cannot be embedded in reserved words, identifiers, and multicharacter tokens, such as ":=", "<<", and ">>".	Same as SPL.

Comments

Table 2-4. Comments

SPL	**HP C/XL Equivalent**
comment: COMMENT *comment-text* ;	*comment*: /* *comment-text* */

<< *comment-text* >> ! *comment-text to end of record*	Similar to SPL's << *comment-text* >>.

Program and Subprogram Structure

Table 2-5. Program and Subprogram Structure

SPL	HP C/XL Equivalent
An SPL program consists of a single BEGIN-END block that contains global declarations, procedures (which may include subroutines), and a main body of statements. Procedures may have local data declarations; subroutines cannot.	An HP C/XL program consists of declarations and function definitions. The "main body" of a program is a function named main. Functions may have local data declarations. Functions cannot contain subroutines.
A subprogram has the same structure as a main program, except that the block is preceded by the compiler command $CONTROL SUBPROGRAM and it has no main body. Outer blocks of subprograms are not compiled.	A "subprogram" compilation unit has the same structure as a main program, except that it has no main function.

In general, SPL procedures convert directly to HP C/XL functions.

Hardware Concepts

With the exception of the hardware stack structure, which does not exist in MPE XL, the concepts of processes and code/data separation are essentially the same on both MPE V and MPE XL.

Code and Data Segments

Table 2-6. Code and Data Segments

SPL	HP C/XL Equivalent
SPL provides code segmentation and access to the registers and counters (PB, P, and PL) that manage program code.	HP C/XL provides neither segmentation nor register access. $CONTROL SEGMENT compiler commands must be removed. Register references must be recoded.
SPL provides data segmentation and access to the registers (DB, DL, Q, S, and Z) that manage program data.	HP C/XL provides neither segmentation nor register access. Register references must be recoded.

Procedures

Table 2-7. Procedures

SPL	HP C/XL Equivalent
An SPL procedure can be passed parameters, either by reference or by value.	An HP C/XL function can be passed parameters, but always by value. Pass-by-reference is emulated by explicitly passing an address pointer and dereferencing that pointer within the function. (Array identifiers *appear* to be passed by reference; they are passed as pointers.)
Can declare local variables and reference global variables.	Same as SPL.
Can return a value.	Same as SPL.
Can call themselves.	Same as SPL.
Can be called from other procedures and from the main block.	Can be called from other functions and from the main function.
Can contain local subroutines.	Cannot contain nested functions. The closest HP C/XL equivalent is the #define macro directive (see "Subroutines" below).

Subroutines

Table 2-8. Subroutines

SPL	HP C/XL Equivalent
Can appear within procedures and globally.	No direct equivalent.

If possible, you should recode SPL subroutines as HP C/XL #define macro directives, which permit parameters, and result in inline substitution. Where appropriate (i.e., in functions), limit the scope of a #define directive with a subsequent #undef directive.

Otherwise, you must recode the SPL subroutine as an HP C/XL independent function. This can be awkward because variables that were formerly local to the procedure and known to the subroutine have to be made available to the new function. You can make variables available to new functions either by declaring them as global (to all functions) or by passing them as parameters.

See "SUBROUTINE Declaration".

Intrinsics

Table 2-9.

SPL	HP C/XL Equivalent
System and user-defined intrinsics are accessed with the INTRINSIC declaration.	System and user-defined intrinsics are accessed with the #pragma intrinsic and #pragma intrinsic_file directives.

A major advantage of HP C/XL is the large number of functions available in the standard function library. These serve most of the purposes that an SPL program requires intrinsics for (such as I/O). The library also includes numerous routines for byte manipulation, input/output, memory control, and data formatting.

Compound Statements

Table 2-10. Compound Statements

SPL	HP C/XL Equivalent
compound-statement: BEGIN [*statement*] [;...] [;] END	*compound-statement*: " {" [*statement*] [...] " }"
Semicolons are *not* part of statements; they are used to *separate* statements.	Semicolons *are* part of statements; they are required terminators. "Extra" semicolons form null statements, similar to SPL.

Example
 	
BEGIN
A := 10 ;
B := 17 OR B := 17 ;
END
	Example
	
	{
	A = 10 ;
	B = 17 ;
	}

In HP C/XL, a compound statement may be a block. That is, it may contain declarations for data that is local to itself.

Entry Points

Table 2-11. Entry Points

SPL	HP C/XL Equivalent
Main programs and procedures may have multiple entry points.	No equivalent.

In main programs, you may recode existing SPL entry points by using the argc, argv, parm, and info parameters of the HP C/XL main function, and adding a switch statement to jump to the appropriate labels. In HP C/XL, arguments are passed to these parameters with the "INFO=" and "PARM=" parameters of the MPE XL :RUN command.

In functions, you may add a parameter and use a switch statement to jump to the "entry" labels.

Chapter 3 Basic Elements

This chapter discusses conversion issues that correspond to sections in Chapter 2 of the *Systems Programming Language Reference Manual*.

Data Storage Formats

SPL processes six types of data.

Table 3-1. Data Types

SPL	HP C/XL Equivalent
INTEGER	short int
DOUBLE	long int OR int (equivalent in HP C/XL)
REAL	float
LONG	double
BYTE	unsigned char OR unsigned short int (depends on usage)
LOGICAL	unsigned short int

The HP C/XL types float, double, unsigned char, and unsigned short int are not precise equivalents for the SPL types REAL, LONG, BYTE, and LOGICAL. The differences are described below.

INTEGER Format

Table 3-2. INTEGER Format

SPL	HP C/XL Equivalent
Type: INTEGER	Type: short int
16-bit signed integer in two's-complement form.	Same as SPL.
Range is -32768 to 32767.	Same as SPL.

DOUBLE Integer Format

Table 3-3. DOUBLE Integer Format

SPL	HP C/XL Equivalent
Type: DOUBLE	Type: long int OR int (equivalent)
32-bit signed integer in two's-complement form.	Same as SPL.
Range is -2,147,483,648 to 2,147,483,647.	Same as SPL.

REAL Format

Table 3-4. REAL Format

SPL	HP C/XL Equivalent
Type: REAL	Type: float
32 bits (two words) in MPE V floating-point format: 1-bit sign, 9-bit exponent, 22-bit fraction.	32 bits (one word) in IEEE floating-point format: 1-bit sign, 8-bit exponent, 23-bit fraction.
Approximate nonzero range: 8.63617x10-78 to 1.157921x1077	Approximate nonzero range: 1.754944x10-38 to 3.4028235x1038

CAUTION The numeric ranges AND the data storage formats for SPL and HP C/XL 32-bit floating-point data are significantly different. If your application uses REAL floating-point data that depend on extreme values, bit manipulation, or file storage, you may have a problem in migrating to HP C/XL.

However, floating-point values may be translated from MPE V format to MPE XL format and back with the MPE XL HPFPCONVERT intrinsic. See the *MPE XL Intrinsics Reference Manual* for details.

LONG Format

Table 3-5. LONG Format

SPL	**HP C/XL Equivalent**
Type: LONG	Type: double
64 bits (four words) in MPE V floating-point format: 1-bit sign, 9-bit exponent, 54-bit fraction.	64 bits (two words) in IEEE floating-point format: 1-bit sign, 11-bit exponent, 52-bit fraction.
Approximate nonzero range: 8.63618555094445x10-78 to 1.157920892373162x1077	Approximate nonzero range: 2.2250738585072014x10-308 to 1.79769313486231x10308

CAUTION The numeric ranges AND the data storage formats for SPL and HP C/XL 64-bit floating-point data are significantly different. If your application uses LONG floating-point data that depend on bit manipulation or file storage, you may have a problem in migrating to HP C/XL.

However, floating-point values may be translated from MPE V format to MPE XL format and back with the MPE XL HPFPCONVERT intrinsic. See the *MPE XL Intrinsics Reference Manual* for details.

BYTE Format

Table 3-6. BYTE Format

SPL	HP C/XL Equivalent
Type: BYTE	Type: unsigned char OR unsigned short int (depends on usage)
8-bit character stored in high-order byte of 16-bit word.	8-bit character OR 16-bit unsigned integer

In SPL BYTE format, characters are stored as 8-bit bytes,two to a 16-bit word.A single or odd character occupies the high-order byte of the word.

Normally, the HP C/XL unsigned char data type is the correct choice for conversion of both simple BYTE variables and BYTE arrays.

However, a simple BYTE variable may also be used as a 16-bit quantity in many places where an INTEGER or LOGICAL data type is accepted. In that usage, the value is more like an HP C/XL unsigned short int with the character value in the high-order byte.

In the conversion, such uses need to be clearly identified. If the variable is used for both 8-bit and 16-bit operations, it would be wise to divide the uses into separate variables.

LOGICAL Format

Table 3-7. LOGICAL Format

SPL	HP C/XL Equivalent
Type: LOGICAL	Type: unsigned short int
16-bit unsigned integer, ranging from 0 to 65535.	Same as SPL.
In a conditional test, a LOGICAL, BYTE, or INTEGER value is true if it is odd, that is, if bit 15 is on. It is false if it is even, that is, if bit 15 is off.	In a conditional test, a numeric value of any type is true if it is nonzero. It is false if it is zero.
The logical constant TRUE equals 65535 (all 16 bits on); FALSE equals 0 (all 16 bits off).	HP C/XL has no identifiers for true and false. The result of a relational expression is 1 if true, 0 if false.

Constant Types

Table 3-8. Constant Types

SPL	HP C/XL Equivalent
Numeric	Numeric
String	String literal
One-byte string	Character

SPL has two types of constants: numeric and string. You may have to specify the type of the constant with a modifier to avoid errors when mixing types.

HP C/XL has four types of constants: integer, floating point, character, and enumeration. Type mixing is generally allowed in HP C/XL, so you do not need to specify types except when you want to control word size.

NOTE HP C/XL does not permit a leading unary "+" sign, only a unary "-" sign.

Integer Constants

Table 3-9. Integer Constants

SPL	HP C/XL Equivalent
Type: INTEGER	Type: short int
integer-constant: [*sign*] *integer*	*integer-constant*: [-] *integer*

Double Integer Constants

Table 3-10. Double Integer Constants

SPL	HP C/XL Equivalent
Type: DOUBLE	Type: long int or int
double-integer-constant: [*sign*] *integer* D	*long-integer-constant*: [-] *integer* [L]

In HP C/XL, the L (specifying long int) is optional, since int and long int are equivalent and occupy 32 bits. The L may be lowercase.

Based Constants

Table 3-11. Based Constants

SPL	HP C/XL Equivalent
Type: INTEGER DOUBLE LOGICAL BYTE REAL LONG	Type: short int long int or int unsigned short int unsigned char OR unsigned short int float double
based-constant: [*sign*] % [(*base*)] *value* [*type*] *type*: is D, E, or L (for DOUBLE, REAL, or LONG); default is single word, usable as INTEGER, LOGICAL, or BYTE.	*integer-constant*: [-] 0*octal-digits* [L] [-] 0X*hex-digits* [L] Only octal and hexadecimal bases may be specified. Numbers are signed decimal by default. The leading character is a zero. A trailing L forces a long int constant. The L and X may be lowercase. Floating point cannot be specified directly.
Example: %170033 *octal* %(16)F01B D *hexadecimal* %(2)11011011 *binary*	**Example:** 0170033 *octal* 0xF01B L *hexadecimal* *(No equivalent))*

Since HP C/XL can represent only octal, decimal, and hexadecimal values, based constants must be converted into one of those forms.

CAUTION Since MPE XL floating-point format is different from MPE V floating point, REAL and LONG based constants must be carefully translated if they are intended for arithmetic use.

Composite Constants

Table 3-12. Composite Constants

SPL	**HP C/XL Equivalent**
Type: INTEGER DOUBLE LOGICAL BYTE REAL LONG	Type: short int long int or int unsigned short int unsigned char OR unsigned short int float double
composite-constant: [*sign*] " [*"length"* /*"value* [,...]*"*]*"* [*type*] *type* is D, E, or L (for DOUBLE, REAL, or LONG); default is single word usable as INTEGER, LOGICAL, or BYTE.	No direct equivalent. See "Based Constants" above.
Example: +[3/2,12/%5252] *(=* %25252*)* -[3/2,12/%5252] *(=* %152526*)*	**Example:** 025252 *octal* -025252 *octal*

CAUTION Since MPE XL floating-point format is different from MPE V floating point, REAL and LONG composite constants must be carefully translated if they are intended for arithmetic use.

Equated Integers

Table 3-13. Equated Integer Constants

SPL	HP C/XL Equivalent
equated-integer: [*sign*] *identifier* [D]	*defined-constant*: [-] *identifier*
identifier is assigned a numeric value in an EQUATE declaration. It represents a 16-bit INTEGER value. If D is specified, the value is extended on the left with zeros to a 32-bit DOUBLE value.	*identifier* is assigned a literal value in a #define directive. The literal is inserted at the reference point. Note that, while SPL evaluates an equated integer when it is declared, HP C/XL evaluates the literal when the *reference* is compiled.

See "EQUATE Declaration and Reference".

Real Constants

Table 3-14. Real Constants

SPL	HP C/XL Equivalent
Type: REAL	Type: float
real-constant: 1. [*sign*] *fixed-point-number* [E *power*] 2. [*sign*] *decimal-integer* E *power* 3. [*sign*] *based|composite-integer* E	*real-constant*: 1. [-] *fixed-point-number* [E *power*] 2. [-] *decimal-integer* E *power* 3. *(No equivalent; convert to 1 or 2.)* The E may be in lowercase.

CAUTION Since MPE XL floating-point format is different from MPE V floating point, REAL based and composite constants must be carefully translated if they are intended for arithmetic use.

Long Constants

Table 3-15.

SPL	HP C/XL Equivalent
Type: LONG	Type: double
long-constant: 1. [*sign*] *fixed-point-number* L *power* 2. [*sign*] *decimal-integer* L *power* 3. [*sign*] *based|composite-integer* L	*real-constant*: 1. [-] *fixed-point-number* [E *power*]; 2. [-] *decimal-integer* E *power* 3. *(No equivalent; convert to 1 or 2.)* The E may be in lowercase.

HP C/XL uses the same representation for float and double constants.

CAUTION Since MPE XL floating-point format is different from MPE V floating point, LONG based and composite constants must be carefully translated if they are intended for arithmetic use.

Logical Constants

Table 3-16. Logical Constants

SPL	HP C/XL Equivalent
Type: LOGICAL	Type: unsigned short int
TRUE (logical value: 65535; integer value: -1)	No direct equivalent. May be specified with `#define TRUE 1`
FALSE (zero)	No direct equivalent. May be specified with `#define FALSE 0`
INTEGER, LOGICAL, or BYTE constant: * true if bit 15 is on (value is odd) * false if bit 15 is off (value is even)	Any numeric constant (including char): * true if value is nonzero. * false if value is zero.

String Constants

Table 3-17. String Constants

SPL	**HP C/XL Equivalent**
Type: BYTE	Type: string literal OR unsigned char
string-constant: "*characters* "	*string-literal*: "*characters* "
characters is one or more ASCII characters (up to 127). A quotation mark (""") within *characters* is doubled.	*characters* is zero or more ASCII characters. A quotation mark, """, within *characters* is represented by the "escape sequence" "\"", an apostrophe, "'", by "\'", and a backslash, "\", by "\\".
For example, the string *He said, "Hi."* is entered as: "He said, ""Hi."""	For example, the string *He said, "Hi."* is entered as: "He said, \"Hi.\""
Characters are stored two to the 16-bit word, left justified.	Characters are stored as a series of 8-bit bytes. The string literal is terminated by HP C/XL with the ASCII NUL character ('\0', numeric value 0). This fact is used by many HP C/XL string manipulation functions that might be used to emulate SPL string operations.

HP C/XL also has a character constant, in the form:

'*char* '

where *char* is a single character, or a special escape sequence using a leading "\" character, such as those shown above. Escape sequences can be used in character and string constants to represent any of the 256 ASCII character codes. Consult the *HP C Reference Manual* for further details.

Identifiers

Table 3-18. Identifiers

SPL	HP C/XL Equivalent
An identifier consists of one to 15 letters ("A" to "Z" and "a" to "z"), digits ("0" to "9"), and apostrophes ("'"), starting with a letter.	An identifier consists of one to 255 letters ("A" to "Z" and "a" to "z"), digits ("0" to "9", and underscores ("_"), starting with a letter or underscore.
Upper- and lowercase letters are equivalent. The identifier VAR2 is the same as var2.	Upper- and lowercase letters are *not equivalent*. The identifier VAR2 is different from var2.
Identifiers longer than 15 characters are truncated on the right.	Identifiers longer than 255 characters are invalid.

Change apostrophe, "'", to underscore, "_", in identifiers.

Make sure that any SPL identifiers over 15 characters long do not become "unique" due to the extra characters. For example, these two identifiers,

```
A23456789012345
A23456789012345B
```

are the same in SPL but different in HP C/XL.

Arrays

Table 3-19. Arrays

SPL	HP C/XL Equivalent
Type: Array of simple data type.	Type: Array of simple data type.
Arrays are single-dimensional vectors of contiguous storage.	Same as SPL. Arrays may be multi-dimensional, in the sense that arrays of arrays can be declared.
Arrays may be located relative to DB, Q, S, or P registers.	There can be no explicit references to registers, and no read-only constant arrays. (There exists no equivalent to SPL's PB-based arrays.)

Pointers

Table 3-20. Pointers

SPL	HP C/XL Equivalent
Type: Pointer to simple data type	Type: Pointer to simple data type
A pointer is a 16-bit word containing the address of another data item.	A pointer is a 32-bit word containing the address of another data item.
A pointer is declared with the reserved word POINTER.	A pointer is declared by preceding its identifier with the "*" unary operator.
A pointer is dereferenced when its identifier is used alone.	A pointer is dereferenced by preceding its identifier with the "*" unary operator.
The value of a pointer is referenced by preceding its identifier with the "@" unary operator.	The value of a pointer is referenced by using its identifier alone.
The address of a data item is obtained by preceding its identifier with the "@" unary operator.	The address of a data item is obtained by preceding its identifier with the "&" unary operator.

Example:

SPL: HP C/XL:

INTEGER POINTER ptr; short int *ptr; *declares* ptr *as pointer to integer*
@ptr := @ivar; ptr = &ivar; *assigns address of* ivar *to* ptr
ptr := 3; *ptr = 3; *stores* 3 *in ivar (addressed by* ptr*)*
ptr := ptr + 1; *ptr = *ptr + 1;*increments* ivar
va1 := ptr; va1 = *ptr; *stores value from* ivar *into* va1

Labels

Table 3-21. Labels

SPL	HP C/XL Equivalent
A label is an identifier, followed by a colon, that prefixes a statement.	Same as SPL.
Can be declared in a LABEL declaration. Does not need to be declared.	Can not be declared.

Switches

Table 3-22. Switches

SPL	HP C/XL Equivalent
A switch is an ordered list of labels indexed by an identifier and declared with a SWITCH declaration. It uses a GOTO statement to transfer to a label, based on an index value.	Can be emulated with a switch statement. See "SWITCH Declaration" and "GO TO Statement".

Chapter 4 Global Data Declarations

This chapter discusses conversion issues that correspond to sections in Chapter 3 of the *Systems Programming Language Reference Manual*.

Types of Declarations

Table 4-1. Declaration Types

SPL	**HP C/XL Equivalent**
Global declarations occur in the global declaration section, the first section of a program or subprogram.	Global declarations occur in the outer block, outside function definitions. Besides occurring before the first function definition, as in SPL, global declarations may also occur *between* function definitions.
global-data-declaration: [GLOBAL] *data-declaration*	*global-data-declaration*: [static] *data-declaration*
Globally declared identifiers can be accessed from all procedures (and the main body) in the compilation unit.	As with SPL, global identifiers can be accessed by all functions that follow the declarations in the compilation unit. Unlike SPL, an identifier that should be known only within the compilation unit should be preceded by the static storage class specifier.
If an identifier is preceded by the GLOBAL storage attribute, it may also be referenced from a procedure in a different compilation unit. In that external unit, the same identifier is declared in the local declaration section of a procedure with the EXTERNAL storage attribute.	All globally declared identifiers may be referenced from other compilation units. In an external unit, a reference to the same identifier should be declared with the extern storage class specifier.
SPL also allows linking global identifiers between compilation units by the method of including matching global declarations in both program and subprograms. All declarations must be present in the same order, including those for identifiers that are not used in the subprogram. The data types must match; the identifiers *may be different*.	HP C/XL will match up global identifiers that are declared in separate units. The identifiers *must be the same* in all units The unneeded declarations may be deleted.

SPL data declarations have only three general forms: simple, array, and pointer. However, this simplicity is enhanced by the powerful ability to equivalence data of all types and formats and to develop elaborate overlay structures.

It is necessary, therefore, to understand the physical relationships between data elements. Much of that is beyond the scope of this guide. However, it may be useful to you to construct a diagram of the DB-, Q-, and S-relative data areas to determine the correct choice for converting data declarations.

In many cases, you may be able to use HP C/XL pointers in simple emulation of the SPL declarations. In other cases, the data relationships may require an HP C/XL union declaration to ensure the correct interplay of the variables.

In the following sections, the SPL and HP C/XL *type* syntax elements refer to the following simple variable types:

SPL	**HP C/XL Equivalent**
INTEGER	short int
DOUBLE	long int
LOGICAL	unsigned short int
BYTE	unsigned char OR unsigned short int
REAL	float
LONG	double

The rest of this chapter discusses global declarations. Local and external declarations are discussed in Chapter 8.

Simple Variable Declarations

Table 4-2. Simple Variable Declaration

SPL	HP C/XL Equivalent
simple-variable-declaration: [GLOBAL] *type* *variable-defn* [,...] ;	*simple-variable-declaration*: [static] *type* *variable-defn* [,...] ;
variable-defn: 1a. *variable-id* 1b. *variable-id* := *initial-value* 2a. *variable-id* = *register* 2b. *variable-id* = *register sign offset* 3a. *variable-id* = *ref-id* 3b. *variable-id* = *ref-id* *sign offset*	*variable-defn*: 1a. *variable-id* 1b. *variable-id* = *initial-value*

Simple variables in formats 2 and 3 are usually various types of data equivalences. They may be converted to pointers or union equivalences, depending on the requirements of the program. See "ARRAY Declaration" below for further examples.

ARRAY Declaration

Table 4-3. ARRAY Declaration

SPL	HP C/XL Equivalent
array-declaration: [GLOBAL] [*type*] ARRAY [*global-array-defn* ,] [...] {*global-array-defn* *init-global-array-defn* } ;	*array-declaration*: 1b, 1d with *lower* <> 0. [static] [*type*] *array-id* " [" *cells* "]" [*init*] ; 1a, 1c; 1b, 1d with *lower* <> 0. [static] [*type*] *array-ref* " [" *cells* "]" [*init*] ; [static] [*type*] * *array-id* = & *array-ref* " [" *index* "]" ;
global-array-defn: 1a. *array-id* (*lower* : *upper*) 1b. *array-id* (*lower* : *upper*) = DB 2a. *array-id* (@) = DB 2b. *array-id* (@) = DB + *offset* 3a. *array-id* (*) = DB 3b. *array-id* (*) = DB + *offset* 4a. *array-id* (@) 4b. *array-id* (@) = *register sign offset* 5. *array-id* (*) 6. *array-id* (*) = *register sign offset* 7a. *array-id* (*) = *ref-id* 7b. *array-id* (*) = *ref-id sign offset* 8. *array-id* (*) = *ref-id* (*index*)	*init*: = " {" *value* [,...] " }" *index*: is the cell number in *array-ref* of the cell that corresponds to cell zero in the SPL array. _ The other SPL formats establish an equivalence relative to other declared data (not just arrays). Depending on their actual use, they may be converted to HP C/XL pointer or union types, or #define directives. If their relationships are fairly simple, pointers can be used. Some suggestions follow:
init-global-array-defn: 1c. *array-id* (*lower* : *upper*) := *value-group* [,...] 1d. *array-id* (*lower* : *upper*) = DB := *value-group* [,...] *value-group*: {*initial-value* *repeat-factor* (*initial-value* [,...])}	2. union " {" [*type*] * *array-id* ; [*type*] * *other-id* ; : " }" 4. [*type*] * *array-id* 5. [*type*] * *array-id* 7. [*type*] * *array-id* = & *ref-id* 8. [*type*] * *array-id* = & *ref-id* " [" *index* "]"

SPL	HP C/XL
Default type: LOGICAL	Default type: int (= long int)
Array declarations specify one-dimensional vectors of subscripted variables.	Same as SPL.
Array subscript declarations and references use parentheses, "( )"	Array subscript declarations and references use brackets, "[ ]"
An array identifier with no subscript implies a subscript of zero. It is equivalent to *array-id* (0).	An array identifier with no subscript is a *pointer* to cell zero. It is *not* equivalent to *array-id* [0].
Array declarations which allocate space may specify a lower bound other than zero, subject to certain restrictions. Therefore, the SPL zero subscript may refer to some cell other than the lower bound, or to a location outside the array.	The lower bound of HP C/XL arrays is always zero. Therefore, the HP C/XL zero subscript always refers to the "first" (lower bound) cell of the array.
Indirect array-ids are equivalent to pointers. They may be changed to point to a new cell zero location. Direct array-ids are the location of cell zero. Their locations cannot be changed.	Array-ids are identical to pointers in computations, except that the pointer value of an array-id cannot be changed. That is, it will always point to cell zero of the declared array. However, HP C/XL pointers may be subscripted and used like array-ids, allowing them to act like SPL indirect array identifiers.

In HP C/XL, if A is an array-id and P is a pointer-id and P=&A[0], then the following equivalences exist:

```
 A      ==  &A[0]    ==   P      ==  &P[0]
*A      ==   A[0]    ==  *P      ==   P[0]
*A+1    ==   A[0]+1  ==  *P+1    ==   P[0]+1
*(A+1)  ==   A[1]    ==  *(P+1)  ==   P[1]
```

This situation simplifies some of the conversion necessary for changing SPL procedure calls to HP C/XL function calls. If an unsubscripted HP C/XL array-id is passed to a function, it is passed by value as a *pointer* to the array. This is identical to passing cell zero by reference, the equivalent code in SPL. Therefore, conversion is minimal for full arrays passed by reference. To pass a specific cell by reference, convert the SPL cell reference, "*id* (*cell*)", to the HP C/XL address format, "&*id* [*cell*]".

Summary of SPL Array Formats

1a. Indirect; bounded; variable is pointer to cell zero; pointer in next DB primary location; pointer IS allocated; array begins in next DB secondary location; array IS allocated.

1b. Direct; bounded; variable is cell zero; *lower* in next DB primary location; array IS allocated.

1c. Same as 1a; initialized.

1d. Same as 1b; initialized.

2a. Indirect; unbounded; variable is pointer to cell zero; pointer in next DB primary location; pointer NOT allocated; array NOT allocated.

2b. Indirect; unbounded; variable is pointer to cell zero; pointer in specified DB primary location; pointer NOT allocated; array NOT allocated.

3a. Direct; unbounded; variable is cell zero; cell zero in next DB primary location; array NOT allocated.

3b. Direct; unbounded; variable is cell zero; cell zero in specified DB primary location; array NOT allocated.

4a. Indirect; unbounded; variable is pointer to cell zero; pointer in next DB primary location; pointer IS allocated; array NOT allocated.

4b. Indirect; unbounded; variable is pointer to cell zero; pointer in specified Q- or S-relative location; pointer NOT allocated; array NOT allocated.

5. Indirect; unbounded; variable is pointer to cell zero; pointer in next DB primary location; pointer IS allocated; array NOT allocated.

6. Direct; unbounded; variable is cell zero; cell zero in specified Q- or S-relative location; array NOT allocated.

7a. Direct (if *ref-id* is direct array or simple variable); unbounded; variable is cell zero; cell zero in specified location; array NOT allocated.

Indirect (if *ref-id* is pointer or indirect array); unbounded; variable is pointer to cell zero; cell zero in *ref-id* location; pointer in next DB primary location IF one *id* type is BYTE and other is not; ELSE pointer location shared with *ref-id*; pointer IS allocated; array NOT allocated.

7b. Direct; unbounded; variable is cell zero; cell zero in specified location; array NOT allocated.

8. Direct (if *ref-id* is direct array); unbounded; variable is cell

zero; cell zero in specified location; array NOT allocated.

Indirect (if *ref-id* is pointer or indirect array); unbounded; variable is pointer to cell zero; cell zero in specified location; pointer in next DB primary location IF specified location is not *ref-id* cell zero OR IF one array is BYTE and other is not; ELSE pointer location shared with *ref-id*; pointer IS allocated; array NOT allocated.

Array formats 2, 3, 4, 5, 6, 7, and 8 imply methods of data equivalencing or "overlays".

Array formats 4, 5, 6, 7, and 8 cannot have the GLOBAL attribute.

Only array formats 1c and 1d may be initialized.

Comparison of Specific Array Declarations

Array Formats 1a and 1c: Bounded Indirect Arrays.

SPL	HP C/XL Equivalent
1a. INTEGER ARRAY ABC(0:4) ;	short int ABC_REF[5]; short int *ABC = &ABC_REF[0];
1a. INTEGER ARRAY ABC(-3:4) ;	short int ABC_REF[5]; short int *ABC = &ABC_REF[3];
1c. INTEGER ARRAY ABC(0:4) := 0,1,2,3;	short int ABC_REF[5]={0,1,2,3}; short int *ABC = &ABC_REF[0];
1c. INTEGER ARRAY ABC(-3:4) := 6,2,5;	short int ABC_REF[8]={6,2,5}; short int *ABC = &ABC_REF[3];

These are SPL "indirect" arrays. In SPL, the location labeled ABC is a pointer that contains the address (initially) of the zero cell of the array's data.

By converting the SPL indirect array identifier ABC to HP C/XL pointer ABC, all the operations (such as assigning a new address) that may be performed on the SPL array identifier may be performed on the HP C/XL pointer identifier. The HP C/XL pointer may be subscripted to reference array cells.

If the SPL lower bound is zero and the array identifier is not modified, you may use the direct format, as shown below.

The examples above with the nonzero lower bounds show the solution to the SPL capability to specify non-zero lower bounds. Subscripting ABC from -3 through 4 will access the eight cells of ABC_REF from 0 through 7.

Array Formats 1b and 1d: Bounded Direct Arrays.

SPL	HP C/XL Equivalent
1b. INTEGER ARRAY ABC(0:4)=DB;	short int ABC[5];
1b. INTEGER ARRAY ABC(-3:4)=DB;	short int ABC_REF[8]; short int *ABC = &ABC_REF[3];
1d. INTEGER ARRAY ABC(0:4)=DB:=0,1,2,3;	short int ABC[5]={0,1,2,3};
1d. INTEGER ARRAY ABC(-3:4)=DB:=6,2,5;	short int ABC_REF[8]={6,2,5}; short int *ABC = &ABC_REF[3];

These are SPL "direct" arrays: the location labeled ABC refers directly to cell zero of the array allocation.

Note that the examples above having a nonzero lower bound still require an indirect solution, identical to the one used for indirect arrays.

Array Formats 2a, 4a, and 5: Unbounded Indirect Arrays.

SPL	HP C/XL Equivalent
2a. INTEGER ARRAY A1(@)=DB;	short int *A1;
4a. INTEGER ARRAY A2(@);	short int *A2;
5. INTEGER ARRAY A3(*);	short int *A3;

These declarations are equivalent in SPL, and each defines an identifier.
However none of them allocates space for the array data; only one 16-bit word is allocated to be used as a data label referring to an indirect array, that is, as a pointer to space allocated elsewhere. The address contained in this pointer must be initialized by the program at run time.

Simple pointers in HP C/XL are equivalent to this type of declaration.

Array Formats 7a and 8: Unbounded Equivalenced Arrays.

SPL	HP C/XL Equivalent
DOUBLE ARRAY EFG(0:25);	long int EFG[26];
...	...
7a. REAL ARRAY ABC(*) = EFG;	float *ABC = &EFG[0];
8. DOUBLE ARRAY ABC(*) = EFG(0);	#define ABC EFG
8. REAL ARRAY ABC(*) = EFG(10);	float *ABC = &EFG[10];

SPL assigns the *same* pointer location to ABC and EFG: if EFG is indirect, if the index of EFG is zero, and if the type of both arrays or neither is BYTE. HP C/XL allows you to simulate this with a #define only if both arrays are of the identical type. Otherwise, you must use a union data type.

Array Formats 2b and 3b: Unbounded Equivalenced Arrays.

<table>
<tr><th>SPL</th><th>HP C/XL Equivalent</th></tr>
<tr><td>2b. INTEGER ARRAY DEF(@) = DB + 10;</td><td><pre>short int DB[256];
...
short int *DEF = &DB[10];</pre></td></tr>
<tr><td>3b. REAL ARRAY ABC(*) = DB + 10;</td><td><pre>union
   {
   short int DB[256];
   struct
      {
      short int dummy[10];
      float ABC_REF[1]; /*cell zero*/
      }
   }
float *ABC = &ABC_REF[1];</pre></td></tr>
</table>

In SPL, two types of arrays may be equated to the DB-relative area: indirect arrays, in which one word of the DB area is allocated to be used as a pointer to an array; and direct arrays, in which the name of the array refers to the next element of the DB area, which is assumed to be cell zero of an array actually contained within this (DB-relative) area.

If DB-relative addressing is required for an SPL application and cannot be rewritten in a straightforward manner, a DB area may be simulated in HP C/XL.

In the first example, the DB area is simulated as a short int array. The value of the pointer DEF is set to the value in DB[10]. DEF is an indirect array.

In the second example, the DB area is equivalenced in a union with a structure that places cell zero of reference array ABC_DEF at location DB[10]. The pointer ABC is used to reference the array cells of ABC_REF, thus overcoming the undefined subscript range problem presented by the unbounded direct array in SPL. ABC_REF is a direct array.

POINTER Declaration

Table 4-4. POINTER Declaration

SPL	**HP C/XL Equivalent**
pointer-declaration: [GLOBAL] [*type*] POINTER *ptr-defn* [,...] ;	*pointer-declaration*: [static] [*type*] *ptr-defn* [,...] ;
ptr-defn: 1a. *ptr-id* 1b. *ptr-id* := @*ref-id* 1c. *ptr-id* := @*ref-id* (*index*) 2a. *ptr-id* = *ref-id* 2b. *ptr-id* = *ref-id sign offset* 3a. *ptr-id* = *register* 3b. *ptr-id* = *register sign offset* 4. *ptr-id* = *offset*	*ptr-defn*: 1a. * *ptr-id* 1b. * *ptr-id* = & *ref-id* 1c. * *ptr-id* = & *ref-id "[" index "]"*
Default type: LOGICAL	Default type: int (= long int)
Pointers are 16-bit values containing DB-relative addresses.	Pointers are 32-bit values containing standard MPE XL addresses. Overlays of pointers and other data types must be recoded.
Pointers may be initialized to addresses. INTEGER POINTER P := @IVAR; declares a pointer P, as a pointer to type INTEGER data, and initializes it to the address of the integer variable IVAR.	Same as SPL. short int *P = &IVAR; declares a pointer P, as a pointer to type short int data, and initializes it to the address of the short integer variable IVAR.
Pointers may contain either byte addresses or 16-bit word addresses, depending on the data type. The rule is that BYTE pointers contain byte addresses, and all other types contain word addresses. Consequently, many SPL programs contain runtime code to "convert" between byte and	Pointers always contain byte addresses, regardless of the type of data being pointed to. Unlike SPL, HP C/XL automatically scales the operands used in pointer arithmetic, so adding one--"*(*ptr* +1)"--to a type char pointer increments it by one, but adding

word addresses, generally via LSL and LSR shift operators. This also affects pointer arithmetic, since adding one to a byte pointer increments its address to the next *byte*, but adding one to an integer pointer increments its address to the next *word*. A pointer to type DOUBLE or REAL must be incremented by two to advance it to the next DOUBLE or REAL variable. A pointer to type LONG must be incremented by four.	one to a type short int pointer will increment it by two, thus pointing to the next type short int variable. While this is more convenient than the SPL convention, it will require careful examination of any SPL code being converted to HP C/XL to guarantee accurate pointer arithmetic operations.
A pointer location may be equated to a location relative to another variable or a register.	No direct equivalent. Locations may be equated with a union declaration or with pointer arithmetic.

LABEL Declaration

Table 4-5. LABEL Declaration

SPL	HP C/XL Equivalent
label-declaration: LABEL *label-id* [,...] ;	No equivalent.
Label declarations are not required.	Labels are not declared. Remove the SPL label declarations.

SWITCH Declaration

Table 4-6. SWITCH Declaration

SPL	HP C/XL Equivalent
switch-declaration: SWITCH *switch-id* := *label-id0* [,...]	*define-directive*: #define *switch-id* (X) \\ switch (X) \\ " {" \\ case 0: goto *label-id0*; \\ case 1: goto *label-id1*; \\ [...] " }"
A switch declaration defines and names an ordered list of labels that may be transferred to by an indexed GOTO statement in the form: GOTO *switch-id* (*index*) See "GO TO Statement".	The corresponding transfer of control may be executed by specifying the defined macro with the same *index* as in SPL: *switch-id* (*index*)

ENTRY Declaration

Table 4-7. ENTRY Declaration

SPL	HP C/XL Equivalent
entry-declaration: ENTRY *label-id* [,...] ;	No equivalent.

You may emulate multiple entry points into an SPL program by using the argc, argv, parm, and info parameters of the HP C/XL main function, and coding a switch statement to goto the appropriate labels. (See "SWITCH Declaration" above for the format.) In HP C/XL, you may pass arguments to these parameters with the INFO= and PARM= parameters of the MPE XL :RUN command.

DEFINE Declaration and Reference

Table 4-8. DEFINE Declaration and Reference

SPL	HP C/XL Equivalent
define-declaration: DEFINE {*define-id* = *text* #} [,...] ;	*define-directive*: #define *define-id* *text*
All the characters after "=" and up to the next "#" outside a quoted string are assigned to *define-id*. The declaration may use more than one line. No continuation character is needed.	All the characters after *define-id* and up to the end of the last non-continued line are assigned to *define-id*. The directive may use more than one line. Lines are continued by the presence of "\" as the last nonblank character. The "#" character must be in column one.
The declaration is referenced by using its *define-id* anywhere in the subsequent source file.	Same as SPL.
The *define-id* is evaluated and compiled *where it is referenced*, not in the declaration.	Same as SPL.
Example: DEFINE NEXTC = CPTR:=CPTR+1#; ... NEXTC; *expands to:* CPTR:=CPTR+1;	**Example:** #define NEXTC ((*CPTR)++) ... NEXTC; *expands to:* ((*CPTR)++);

In addition to the simple declaration allowed in SPL, HP C/XL also allows macro directives with formal parameters. (See also "SWITCH Declaration" above.) For example,

```
#define next(x)  (*(x)++)
...
next(c);  expands to:  (*(c)++);
next(y);  expands to:  (*(y)++);
```

Please observe a couple of points:

* The left parenthesis,"(", in the HP C/XL directive and the reference must be attached to the *define-id* (no spaces).

* The parameter substitution is literal. The formal parameter (x above) is replaced by the actual parameters (the characters between the parentheses) in the reference (c and y above).

* It is wise to enclose the formal parameters and the entire macro

directive in parentheses to ensure correct evaluation of the actual parameters.

EQUATE Declaration and Reference

Table 4-9. EQUATE Declaration and Reference

SPL	HP C/XL Equivalent
equate-declaration: EQUATE {*equate-id* = *equate-expr* }[,...]	*define-directive*: #define *equate-id equate-expr*
An equate declaration computes the value of the *equate-expr*, left-truncates it if necessary, and assigns it to *equate-id* as a 16-bit INTEGER. The value of *equate-id* is determined *when it is declared*, not when it is referenced.	A #define directive assigns the characters of *equate-expr* to *equate-id* without evaluation. The *equate-id* is evaluated and compiled *where it is referenced*, not in the declaration.
The declaration is referenced by using its *equate-id* anywhere in the subsequent source file.	Same as SPL.

See also "DEFINE Declaration and Reference" above and "Equated Integers".

DATASEG Declaration and Reference

Table 4-10. DATASEG Declaration and Reference

SPL	HP C/XL Equivalent

dataseg-declaration:
 	
DATASEG *dataseg-name* = *dataseg-num*
 	
BEGIN
 	
type dataseg-variable [= *dataseg-offset*]
 	
...
 	
END ;
	No equivalent.
	
	The concept of extra data segments does not exist in HP C/XL.

Remove the DATASEG declaration and convert the variables in the BEGIN-END block to normal HP C/XL variables.

Chapter 5 Expressions, Assignments, and Scan Statements

This chapter discusses conversion issues related to sections in Chapter 4 of the *Systems Programming Language Reference Manual*.

Expression Types

Table 5-1. Expression Types

<table>
<tr><th>SPL</th><th>HP C/XL Equivalent</th></tr>
<tr><td>Variables on either side of an operator must be of the same type. Type transfer functions are used to convert types.</td><td>Variable types may be mixed in expressions. Automatic type conversion is performed prior to execution of an operator. See "HP C/XL Rules for Automatic Numeric Type Conversion" in this chapter.

A "cast" operator may be used to force an expression to a desired data type, perhaps for an actual parameter to a function.</td></tr>
<tr><td>The type transfer functions are the names of the simple variable types, plus two

additions, in the function form:

BYTE (*double* | *integer* | *logical*)
DOUBLE (*byte* | *integer* | *logical*)
FIXR (*real*) rounds to DOUBLE
FIXT (*real*) truncates to DOUBLE
INTEGER (*byte* | *double* | *logical*)
LOGICAL (*byte* | *double* | *integer*)
LONG (*double* | *real*)
REAL (*byte* | *double* | *integer* | *logical* | *long*)

byte, *double*, *integer*, *logical*, *long*, and *real* are the types permitted in the particular functions.

Sometimes more than one function is required, as in the conversion from REAL to INTEGER, which requires either "INTEGER(FIXR(*real*))" or "INTEGER(FIXT(*real*))".</td><td>The corresponding cast operators are similar to SPL, except that the *type names*

are enclosed in parentheses:

(unsigned char) (*expression*)
(long int) (*expression*)
(No equivalent)
(long int) (*expression*)
(short int) (*expression*)
(unsigned short int) (*expression*)
(double) (*expression*)
(float) (*expression*)

The *expression* may have any appropriate character or numeric value. The parentheses around *expression* may be omitted if it is a single entity. Conversion from float or double to any char or int type is by truncation. There is no rounding function. HP C/XL also allows other simple data and pointer types in cast operations.</td></tr>
</table>

HP C/XL Rules for Automatic Numeric Type Conversion

As an expression is evaluated, HP C/XL performs automatic data type conversions on the operands. First, each operand is evaluated and converted, according to Table 5-2.

Table 5-2. Automatic Unary Type Conversions

Original Type	Converted to
char	int
short int	int
unsigned char	unsigned int
unsigned short int	unsigned int
float	double 1
int	int 2
long int	long int 2
unsigned int	unsigned int 2
unsigned long int	unsigned long int 2

1 This conversion from float to double may be prevented with the HP C/XL compiler option "-Wc,-r". See the *HP C/XL Reference Manual Supplement* for details.

2 These types are not converted. They are included here for completeness.

Second, arithmetic operands in binary operations are converted. If the two operands are the same type, the conversion is complete. Otherwise, the process continues row-by-row through Table 5-3 until a conversion makes the operand types equal.

Table 5-3. Automatic Binary Type Conversions

One Operand	Other Operand	Conversion
double	any type	Other becomes double
float	any type	Other becomes float

unsigned long int	any type	Other becomes unsigned long int
long int	unsigned int	Both become unsigned long int
long int	not unsigned int	Other becomes long int

When a value is stored (as in an assignment), it is converted to the destination type.

Variables

Table 5-4. Variables

SPL	**HP C/XL Equivalent**
Syntax of a variable in an expression: 1. *simple-id* 2. *array/ptr-id* 3. *array/ptr-id* (*index*) 4. @*identifier* 5. @*array/ptr-id* 6. @*array/ptr-id* (*index*) 7. TOS 8. ABSOLUTE (*index*)	Syntax of a variable in an expression: 1. *simple-id* 2. *array/ptr-id "["* 0*"]"* OR **array/ptr-id* 3. *array/ptr-id "[" index "]"* 4. &*identifier* 5. &*array/ptr-id "["* 0 *"]"* OR *array/ptr-id* 6. &*array/ptr-id "[" index "]"* 7. *No equivalent.* 8. *No equivalent.*
Syntax of a variable on the left of an assignment operator (:=): a. *simple-id* b. *array/ptr-id* c. *array/ptr-id* (*index*) d. @*ptr-id* e. TOS f. ABSOLUTE (*index*)	Syntax of a variable on the left of an assignment operator (=): a. *simple-id* b. *array/ptr-id "["* 0*"]"* OR **array/ptr-id* c. *array/ptr-id "[" index "]"* d. *ptr-id* e. *No equivalent.* f. *No equivalent.*
The address operator, "@", specifies the location of a variable, rather than its contents. An unsubscripted pointer-id or array-id is assumed to be subscripted by zero. See "Addresses (@) and Pointers" below.	The address operator is "&". An unsubscripted array-id or pointer-id is an address rather than cell zero; either add the subscript or use the "*" operator.
The assignment operator is ":=".	The assignment operator is "=".

	Note: The HP C/XL assignment operator, "=", is the SPL equality operator.
Indexes are enclosed in parentheses, "( )".	Indexes are enclosed in brackets, "[ ]".

SPL allows assignment to the array-id of an indirect array since it is really a pointer. HP C/XL does not permit assignments to any array-id. You may simulate the process by using a pointer to array cell zero. (See "ARRAY Declaration".)

The reserved word TOS and the ABSOLUTE function cannot be translated (see below). Their operations must be recoded entirely.

TOS

Table 5-5. TOS

SPL	**HP C/XL Equivalent**
TOS	No equivalent.
Refers to the top of the hardware stack.	You could write routines to emulate the hardware stack, but a better solution is to recode SPL programs to eliminate stack references.

Addresses (@) and Pointers

Table 5-6. Addresses and Pointers

SPL	**HP C/XL Equivalent**
The address operator, "@", before a simple variable-id yields the address of the variable instead of its contents. If "@" precedes an array or pointer reference, it yields the address of cell zero or of the indexed location if indexed. If "@" precedes an unsubscripted pointer-id or indirect array-id on the left side of the assignment operator, ":=", the right-side expression is stored as the new address value in the identifier. This leads to a potentially confusing feature of SPL: `@ARRAYNAME := @NEWARRAY;` This assigns the address of NEWARRAY(0) to array variable ARRAYNAME. Consequently,	The address operator, "&", before any variable-id yields the address of the variable. Before a subscripted pointer-id or array-id, it yields the address of the referenced location. The dereference operator, "*", before a pointer expression yields the value of the referenced location. An array-id can be used as a pointer in an expression. An unsubscripted pointer-id or array-id yields the address in the identifier.

ARRAYNAME(0) and NEWARRAY(0) both refer to the same location.	
All SPL addresses are 16-bit quantities that may be stored in integer and logical variables. It is preferable to store addresses in pointer variables, but the lack of pointer "arrays" in SPL has led to some applications that store addresses in logical arrays.	All HP C/XL addresses are 32-bit quantities. In many cases, the SPL logical arrays may be converted to HP C/XL pointer arrays without difficulty.

Table 5-7 compares various uses of the SPL "@" operator and the equivalent HP C/XL assignment.

Table 5-7. Assignments Using Pointers and Simple Variables

SPL	**HP C/XL**	**Operation**
POINTER P1,P2; LOGICAL V3;	unsigned int *P1,*P2; unsigned int V3;	Declarations
P1 := P2	*P1 = *P2	Object of P2 stored in object of P1
P1 := @P2	*P1 = P2	Address in P2 stored in object of P1
@P1 := @P2	P1 = P2	Address in P2 stored in P1
@P1 := P2	P1 = *P2	Object of P2 stored in P1
P1 := V3	*P1 = V3	Value of V3 stored in object of P1
P1 := @V3	*P1 = &V3	Address of V3 stored in object of P1
@P1 := @V3	P1 = &V3	Address of V3 stored in P1
@P1 := V3	P1 = V3	Value of V3 stored in P1
V3 := P2	V3 = *P2	Object of P2 stored in V3
V3 := @P2	V3 = P2	Address in P2 stored in V3

Absolute Addresses

Table 5-8. Absolute Addresses

SPL	HP C/XL Equivalent
absolute-address: ABSOLUTE (*index*)	No equivalent.

The use of absolute addresses in MPE V is entirely system-dependent, and only permitted in privileged mode. They must be recoded in HP C/XL.

Function Designator

Table 5-9. Function Designator

SPL	HP C/XL Equivalent
function-designator: 1. *function-id* 2. *function-id* ( ) 3. *function-id* (*actual-parm* [,...])	*function-designator*: 1. *function-id* ( ) 2. *function-id* ( ) 3. *function-id* (*actual-parm* [,...]
actual-parm: a. *simple-variable-id* b. *array/ptr-id* c. *array/ptr-id* (*index*) d. *procedure-id* e. *label-id* f. *arithmetic-expression* g. *logical-expression* h. *assignment-statement* i. *	*actual-parm*: a. *simple-variable-id* b. *array/ptr-id "[" 0"]"* OR **array/ptr-id* c. *array/ptr-id "[" index "]"* d. *function-id* e. *(No equivalent)* f. *numeric-expression* g. *numeric-expression* h. *assignment-expression* i. *(No equivalent)*
A typed procedure (or subroutine) may be used as a function in an arithmetic or logical expression. Formats 1 and 2 are equivalent.	A function may be used in a numeric expression, except if the function is typed as void. As shown in format 1, HP C/XL requires the parentheses even if there are no actual parameters.

See "Procedure Call Statement" and "Data Type" for more details about parameters passed by reference.

Bit Operations

Table 5-10. Bit Operations

SPL	HP C/XL Equivalent
Bit operations can be used in any expression. They include bit extraction, bit concatenation or deposit, bit shifting, and logical masking.	Standard operators handle much of the bit shifting and logical masking. Bit extraction, concatenation, and some other manipulations will require user-supplied functions or #define directives.

Bit operations are commonly used in the limited-space MPE V system to conserve space. With the increased memory of the MPE XL system, it may be more efficient to rewrite bit operations to use full words, resulting in both improved performance and a much more portable program.

NOTE While a simple BYTE variable is *stored* in bits 0-7 of a 16-bit word, the bits are *referenced* in bit operations as 8-15.

Table 5-11 summarizes all the HP C/XL bitwise operators.

Table 5-11. HP C/XL Bit Operators

Operator	Operation
op1 & *op2*	bitwise AND of *op1* and *op2*
op1 | *op2*	bitwise inclusive OR of *op1* and *op2*
op1 ^ *op2*	bitwise exclusive OR of *op1* and *op2*
op1 << *op2*	shift *op1* left *op2* bits
op1 >> *op2*	shift *op1* right *op2* bits
~ *op2*	bitwise negation of *op2*

Bit Extraction

Table 5-12. Bit Extraction

SPL	HP C/XL Equivalent
bit-extraction-operation: *source* . (*sbit* : *len*) *source*: is a 16-bit value. *sbit*, *len*: are values from 0 to 15.	No direct equivalent.

Step 1: Convert the SPL operation to a function procedure, such as BEXTRACT shown in Figure 5-1.

```
LOGICAL PROCEDURE  BEXTRACT ( SOURCE , SBIT , LEN ) ;
   VALUE  SOURCE , SBIT , LEN ;
   LOGICAL  SOURCE ;
   INTEGER  SBIT , LEN ;
BEGIN
   BEXTRACT := ( SOURCE & LSL( SBIT ) ) & LSR( 16 - LEN ) ;
END ;
```

Figure 5-1. SPL BEXTRACT Procedure: Bit Extraction

In the procedure, the formal parameter names correspond to the variables in the syntax above. SOURCE is the word from which to extract bits, SBIT is the starting bit, and LEN is the number of bits.

To use it, replace an expression like

```
Y.(10:4);
```

with

```
BEXTRACT(Y,10,4);
```

Step 2: Replace the SPL function with the #define macro directive in Figure 5-2 or the HP C/XL function in Figure 5-3.

```
#define BEXTRACT( SOURCE , SBIT , LEN ) \
   ( (unsigned short int)             \
     ( ( (SOURCE) << (SBIT) ) ) >> ( 16 - (LEN) ) )
```

```
|                                                              |
|     unsigned short int BEXTRACT( SOURCE , SBIT , LEN )       |
|     unsigned short int SOURCE ,  SBIT ,  LEN ;               |
|     {                                                        |
|        return ( (unsigned short int)                         |
|                 ( ( SOURCE << SBIT ) >> ( 16 - LEN ) ) ) ;|
|     }                                                        |
```

Figure 5-3. HP C/XL BEXTRACT Function: Bit Extraction

Either the macro or the function may be executed with the same format as the SPL function, e.g., "BEXTRACT(Y,10,4)", so further conversion is unnecessary.

Bit Fields.

It is common practice in SPL to pack fields of bits into a single 16-bit word, and refer to them with DEFINE declarations, such as:

```
|                                                  |
|     LOGICAL WORD, A, B, C;                       |
|                                                  |
|     DEFINE FIELD'A = (0:10)#,                    |
|            FIELD'B = (10:4)#,                    |
|            FIELD'C = (14:2)#;                    |
|     ...                                          |
|     WORD := %(16)F30C;   <<set all fields>>      |
|     A := WORD.FIELD'A;   <<bits 0 through 9>>    |
|     B := WORD.FIELD'B;   <<bits 10 through 13>>|
|     C := WORD.FIELD'C;   <<bits 14 through 15>>|
```

A similar operation may be performed in HP C/XL with union and struct declarations:

```
|                                                    |
|     unsigned short A, B, C;                        |
|                                                    |
|     union {                                        |
|           struct {                                 |
|                  FIELD_A : 10;                     |
|                  FIELD_B : 4;                      |
|                  FIELD_C : 2;                      |
|                  } BITS;                           |
|           unsigned short ALL16;                    |
|           } WORD;                                  |
|     ...                                            |
|     WORD.ALL16 = 0xF30C;    /*set all fields*/*    |
|     A = WORD.BITS.FIELD_A;  /*bits 0 through 9*/   |
|     B = WORD.BITS.FIELD_B;  /*bits 10 through 13*/|
|     C = WORD.BITS.FIELD_C;  /*bits 14 through 15*/|
```

Bit Concatenation (Merging)

Table 5-13. Bit Concatenation

SPL	**HP C/XL Equivalent**
bit-concatenation-operation: *dest* CAT *source* (*dbit* : *sbit* : *len*) *source*: is a 16-bit value from which bits are extracted. *dest*: is a 16-bit value in which bits are deposited. *dbit*, *sbit*, *len*: are values from 0 to 15.	No direct equivalent.

The SPL CAT operation is a means of constructing a new 16-bit quantity from two existing 16-bit words. A bit field is extracted from *source* and deposited into a same-length field in *dest*. Thus:

```
A := %(16)ABCD;
B := %(16)1234;
X := A CAT B (4:8:4);
```

Bits 8 through 11 of word B are extracted and deposited in a copy of word A, replacing bits 4 through 7. The resulting value equals %(16)A3CD. The assignment places the value in X. A and B are unchanged.

Step 1: In the SPL program, convert the SPL operation to a function procedure, such as BCONCAT shown in Figure 5-4 performs the same operation.

```
LOGICAL PROCEDURE BCONCAT( DEST , SOURCE , DBIT , SBIT , LEN ) ;
   VALUE DEST , SOURCE , DBIT , SBIT , LEN ;
   LOGICAL DEST , SOURCE ;
   INTEGER DBIT , SBIT , LEN ;
BEGIN
   LOGICAL M ;
   LEN := 16 - LEN ;
   M := ( %(16)FFFF & LSR( LEN ) ) & LSL( LEN - DBIT ) ;
   BCONCAT := ( DEST LAND NOT( M ) ) LOR
                ( IF DBIT < SBIT
                     THEN SOURCE & LSL( SBIT - DBIT )
                     ELSE SOURCE & LSR( DBIT - SBIT ) LAND M ) ;
END ;
```

Figure 5-4. SPL BCONCAT Procedure: Bit Concatenation

In the procedure, DEST is the word where the bits will be deposited, SOURCE is the word from which the bits will be extracted, DBIT is the start bit in the destination word, SBIT is the start bit in the source word, and LEN is the number of bits to be moved.

To use it, replace a CAT expression like

```
A CAT B (4:8:4)
```

with

```
BCONCAT(A,B,4,8,4)
```

Step 2: In the HP C/XL program, replace the SPL function procedure with the HP C/XL function in Figure 5-5.

```
unsigned short int BCONCAT( DEST , SOURCE , DBIT , SBIT , LEN )
   unsigned short int DEST , SOURCE , DBIT , SBIT , LEN ;
{
   unsigned int TEMP ;

   LEN = 16 - LEN ;
   TEMP = ( 0xFFFF >> LEN ) << ( LEN - DBIT ) ;
   return( (unsigned short int)
           ( ( DEST & ~TEMP ) |
             ( ( DBIT < SBIT ? SOURCE << ( SBIT - DBIT )
                             : SOURCE >> ( DBIT - SBIT ) ) & TEMP ) ) ) ;
}
```

Figure 5-5. HP C/XL BCONCAT Function: Bit Concatenation

The function may be executed with the same format as the SPL procedure, e.g., "BCONCAT(A,B,4,8,4)", so further conversion is unnecessary.

Bit Shifts

Table 5-14. Bit Shift Operators

SPL	**HP C/XL Equivalent**
bit-shift-operation: *operand* & *shift-op* (*count*) *operand*: is an arithmetic or logical *primary*. *shift-op*: is one of 17 shift operators, described below. The shift operator is used to determine the participation of the sign bit, regardless of the type of the operand. *count*: is the number of bits to shift.	*bit-shift-operation*: 1. *operand* << *count* 2. *operand* >> *count* Form 1 shifts the bits of *operand* left *count* positions. The sign bit is lost. Zero bits are inserted on the right. Same as SPL's LSL and DLSL. Form 2 shifts the bits of *operand* right *count* positions. If *operand* is unsigned, zero bits are inserted on the left. If *operand* is signed, the sign bit is extended on the left. Almost (but not quite) the same as SPL's LSR, DLSR, ASR, and DASR.

Example: *operand* is LOGICAL or INTEGER: `X := Y & LSL(4) ;`	Example: *operand* is unsigned short int or short int: `X = Y << 4 ;`

Please notice that the examples above demonstrate the only simple exact equivalents between SPL and HP C/XL.

Unlike SPL, the HP C/XL shift operators take note of the data type being shifted, and behave differently for signed and unsigned data. To provide operations similar to the SPL shift operators, some manipulation and type casting are necessary. There are no circular shifts in HP C/XL, and these must be emulated by iteration.

The best solution is to convert the operations to function calls and #define macro references, in the form:

spl-shift-op (*operand* , *count*)

For example, the SPL expression:

```
Y & LSL(4)
```

would become the HP C/XL expression:

```
LSL(Y,4)
```

Suggested macro directives and functions are described in the following sections.

NOTE If necessary, check the source code and ensure that the value of C (*count*) is not negative in these macros and functions. You may wish to use the HP C/XL abs (absolute value) function.

16-Bit Shift Operators.

The six SPL 16-bit (single-word) shift operators are described in Table 5-15.

Table 5-15. SPL 16-Bit Shift Operators

shift-op	Operation
LSL	logical shift left (sign not retained)
LSR	logical shift right (no sign extension)
ASL	arithmetic shift left (sign retained)
ASR	arithmetic shift right (sign extended)
CSL	circular shift left (rotate 16 bits left)
CSR	circular shift right (rotate 16 bits right)

These operations may be performed in HP C/XL by the following #define macro directives and function declarations. X represents the *operand*; C represents the *count*.

```
#define LSL(X,C) ((unsigned short int)((unsigned short int)(X) << (C)))
```

Figure 5-6. HP C/XL LSL Directive: Bit Shift Operation

```
#define LSR(X,C) ((unsigned short int)((unsigned short int)(X) >> (C)))
```

Figure 5-7. HP C/XL LSR Directive: Bit Shift Operation

```
#define ASL(X,C) ((short int)(((short int)(X) & 0x8000) \
                           | ((short int)(X) << (C)) & 0x7FFF))
```

Figure 5-8. HP C/XL ASL Directive: Bit Shift Operation

```
#define ASR(X,C) ((short int)((short int)(X) >> (C)))
```

Figure 5-9. HP C/XL ASR Directive: Bit Shift Operation

```
unsigned short int CSL(X,C)
   unsigned short int X;
   int C;
{
for (;;--C)  /*infinite loop, decrementing C after each iteration*/
   {
   if (C == 0) return(X);  /*exit, returning X*/
   X = ((X & 0x8000) >> 15) | X << 1;
   }
}
```

Figure 5-10. HP C/XL CSL Function: Bit Shift Operation

```
unsigned short int CSR(X,C)
   unsigned short int X;
   int C;
{
for (;;--C)  /*infinite loop, decrementing C after each iteration*/
   {
   if (C == 0) return(X);  /*exit, returning X*/
   X = ((X & 0x0001) << 15) | X >> 1;
   }
}
```

Figure 5-11. HP C/XL CSR Function: Bit Shift Operation

32-Bit Shift Operators.

The six SPL 32-bit (double-word) shift operators are described in Table 5-16.

Table 5-16. SPL 32-Bit Shift Operators

shift-op	Operation
DLSL	logical shift left (sign not retained)
DLSR	logical shift right (no sign extension)
DASL	arithmetic shift left (sign retained)
DASR	arithmetic shift right (sign extended)
DCSL	circular shift left (rotate 32 bits left)
DCSR	circular shift right (rotate 32 bits right)

These operations may be performed in HP C/XL by the following #define macro directives and functions. X represents the *operand*; C represents the *count*.

```
#define DLSL(X,C) ((unsigned int)((unsigned int)(X) << (C)))
```

Figure 5-12. HP C/XL DLSL Directive: Bit Shift Operation

```
#define DLSR(X,C) ((unsigned int)((unsigned int)(X) >> (C)))
```

Figure 5-13. HP C/XL DLSR Directive: Bit Shift Operation

```
#define DASL(X,C) ((int)(((int)X & 0x80000000) | \
                  ((int)X << (C)) & 0x7FFFFFFF))
```

Figure 5-14. HP C/XL DASL Directive: Bit Shift Operation

```
#define DASR(X,C) ((int)((int)X >> (C)))
```

Figure 5-15. HP C/XL DASR Directive: Bit Shift Operation

```
unsigned int DCSL(X,C)
   unsigned int X;
   int C;
{
for (;;--C)  /*infinite loop, decrementing C after each iteration*/
   {
   if (C == 0) return(X);  /*exit, returning X*/
   X = ((X & 0x80000000) >> 31) | X << 1;
   }
}
```

Figure 5-16. HP C/XL DCSL Function: Bit Shift Operation

```
unsigned int DCSR(X,C)
   unsigned int X;
   int C;
{
for (;;--C)  /*infinite loop, decrementing C after each iteration*/
   {
   if (C == 0) return(X);  /*exit, returning X*/
   X = ((X & 0x00000001) << 31) | X >> 1;
   }
}
```

Figure 5-17. HP C/XL DCSR Function: Bit Shift Operation

48-Bit Shift Operators.

The three SPL 48-bit (triple-word) shift operators are described in Table 5-17.

Table 5-17. SPL 48-Bit Shift Operators

shift-op	Operation
TASL	arithmetic shift left (sign retained)
TASR	arithmetic shift right (no sign extension)
TNSL	normalizing shift left

Because there is no triple-word data type in MPE V (early versions of LONG were three words), the use of these operations is extremely rare, and is generally preceded by stack operations, which must be recoded in HP C/XL. The TNSL operation normalizes a triple-word floating point number, and is even more rare in SPL than the first two. If necessary, these operations could be written in HP C/XL in a manner similar to the examples above.

64-Bit Shift Operators.

Finally, the two SPL 64-bit (four-word) shift operators are described in Table 5-18.

Table 5-18. SPL 64-Bit Shift Operators

shift-op	Operation
QASL	arithmetic shift left (sign retained)
QASR	arithmetic shift right (sign extended)

Because the only four-word data type in SPL is LONG (a floating point number in a format unique to the hardware for which SPL was designed), any use of these operators would almost certainly have to be recoded. They could, however, be emulated by slight modification of the DASL and DASR macro directives above.

Arithmetic Expressions

Table 5-19. Arithmetic Expressions

SPL	HP C/XL Equivalent
arithmetic-expression: [*sign*] *primary* [*operator primary*][,...]	Same as SPL, except as noted below.

sign: + -	Same as SPL, except: *(+ is not permitted as a sign)*
operator: + *(addition)* - *(subtraction)* * *(multiplication)* / *(division)* ^ *(exponentiation allows real and long values to integer power)* MOD *(modulus)* Note: The SPL exponentiation operator, "^", is the HP C/XL exclusive OR operator.	Same as SPL, except: *Convert* ^ *to* pow(*x*,*y*) *function.* % *(modulus)*
primary: *variable* *constant* *bit-operation* (*arithmetic-expression*) \ *arithmetic-expression* \ *function-designator* (*assignment-statement*)	Same as SPL, except: *Convert* \... \ *to* abs(*x*) *function.*

The most significant difference between SPL and HP C/XL arithmetic expressions is that SPL allows no type mixing, whereas HP C/XL performs automatic type conversions during the evaluation of an expression. Normally, this is very convenient and produces the desired result. Occasionally, type "cast" operators may be required to force HP C/XL to adhere to SPL-like operations. Particular caution must be observed with any bit manipulations, as an automatic type conversion may result in an unexpected change in word size.

Sequence of Operations (Arithmetic)

Table 5-20. Order of Evaluation of Arithmetic Operators

SPL	HP C/XL Equivalent
Order of evaluation: 1. bit operations expressions in parentheses expressions in backslashes function designators assignment statements in parentheses 2. exponentiation 3. multiply divide modulus	Same as SPL, except for the following: bit operations Implemented as function calls; same sequence level. absolute value (expressions in backslashes) Implemented as function call; same sequence level. exponentiation Implemented as function call; collapses into first level; care needed in converting operands.

4. addition subtraction	

In general, well-formed expressions, with parentheses used to avoid possible confusion, will always yield the same sequence of operations.

Care may be necessary to maintain the same precision, because of implicit data conversion. (See "Type Mixing (Arithmetic)" in this chapter).

Type Mixing (Arithmetic)

Table 5-21. Arithmetic Type Mixing

SPL	**HP C/XL Equivalent**
The mixing of data types across operands is not allowed in SPL, except that real and long values may be exponentiated to integer powers. Type transfer functions (see "Expression Types" above) are used to convert data types.	Arithmetic data types may be mixed. HP C/XL performs automatic type conversions as needed, generally proceeding toward long int and double values. Many type transfer functions can be eliminated or simplified. Where data types need forcing, HP C/XL provides the "cast" operators--data type names in parentheses preceding the value to be converted. See "Expression Types" above for more detail.

As an example, if you need to force a floating point divide of two integers, the cast operator is (float):

```
X = (float)M/(float)N;
```

Cast operators are essential for converting exponentiation involving integers into the pow function. The SPL statement:

```
I := J^K;
```
all integer variables

becomes the HP C/XL statement:

```
I = pow ( (double) J , (double) K ) ;
```

Logical Expressions

Table 5-22. Logical Expressions

SPL	HP C/XL Equivalent
logical-expression: * *logical-elem* [*log-bit-op logical-elem*] * *lower* <= *test* <= *upper* *lower*, *test*, *upper*: are integer expressions.	Same as SPL, except: ((*lower*) <= (*test*) & (*test*) <= (*upper*)) The parentheses may be necessary for correct evaluation if the elements are expressions or if the entire expression is combined with other expressions. &: Same as SPL LAND. See below
logical-elem: *logical-expression* *logical-primary* [*rel-op logical-primary*] *arith-expression rel-op arith-expression* *logical-primary logical-op logical-primary* *byte-comparison*	Same as SPL.
logical-primary is one of: *logical-variable* *logical/integer-constant* *string-constant* *logical-bit-operation* (*logical-expression*) *logical-function-designator* (*logical-assignment*) NOT *logical-primary* *(bitwise negation)*	Same as SPL, except: ~ *logical-primary (bitwise negation;tilde)*
log-bit-op: LAND *(logical bitwise AND)* LOR *(logical bitwise inclusive OR)* XOR *(logical bitwise exclusive OR)*	*log-bit-op*: & *(bitwise AND)* | *(bitwise inclusive OR)* ^ *(bitwise exclusive OR; circumflex)* Note: The HP C/XL exclusive OR operator, "^", is the SPL exponentiation operator.
The bit-wise operators, NOT, LAND, LOR, and XOR, perform Boolean operations on the corresponding bits of their operands and produce a numeric result of type LOGICAL.	Similar to SPL. Operands may be any numeric types. Results correspond to operand types.
logical-op:	Same as SPL, except:

+ *(unsigned addition)* - *(unsigned subtraction)* * *(unsigned multiplication)* / *(unsigned division)* MOD *(unsigned modulus)* ** *(unsigned multiplication)* // *(unsigned division)* MODD *(unsigned modulus)* (**, //, and MODD give DOUBLE result)	% *(unsigned modulus)* * *(unsigned multiplication)* / *(unsigned division)* % *(unsigned modulus)* Use (long int) cast if needed for the conversions from **, //, and MODD.
The *logical-op* operators perform unsigned integer arithmetic on their operands and produce a numeric result of type LOGICAL.	Similar to SPL. Operands may be any numeric types. Results correspond to operand types.
rel-op: < *(less than)* <= *(less than or equal to)* > *(greater than)* >= *(greater than or equal to)* = *(equal to)* <> *(not equal to)* Note: The SPL equality operator, "=", is the HP C/XL assignment operator.	Same as SPL, except: == *(equal to)* != *(not equal to)*
The *rel-op* operators perform arithmetic comparisons on their operands and produce a Boolean result (true or false) of type LOGICAL. True is returned as LOGICAL 65535 (INTEGER -1). False is returned as LOGICAL 0.	Similar to SPL, except: True is returned as int 1. False is returned as int 0. Operands may be any numeric types. Results correspond to operand types.
The reserved word TRUE has the LOGICAL value 65535 (INTEGER -1). The reserved word FALSE has the LOGICAL value 0 (INTEGER 0).	No direct equivalent. You could use #define directives to define SPLTRUE as 65535 and SPLFALSE as 0: #define SPLTRUE 65535 #define SPLFALSE 0 and then change all TRUE and FALSE references to the special names. This would help you to locate instances where they were used in bit or numeric operations.
In tests for true and false, an odd number is true (bit 15 is on); an even number is false (bit 15 is off).	A nonzero number is true; a zero number is false.
Examples: L L + NOT L1 LAND L2 I <= N <= 100 L L1 L XOR L1 MOD L2	Examples: L L + ~L1 & L2 I <= N & N <= 100 L != L1 L ^ L1 % L2

Conversion Issues

SPL NOT Operator.

SPL uses the same operator, NOT, for both bitwise negation and Boolean negation. HP C/XL uses two operators: "~" (tilde) for bitwise negation and "!" for Boolean negation. They give different results, as shown in Table 5-23.

Table 5-23. Logical and Bitwise Negation

SPL	HP C/XL Equivalent
NOT(0) = -1 *(or LOGICAL 65535)* *16 off bits turned on*	~(0) == -1 *(or unsigned 4294967295)* *32 off bits turned on* !(0) == 1 *since 0 means false*
NOT(-1) = 0 *16 on bits turned off*	~(-1) == 0 *32 on bits turned off* !(-1) == 0 *since nonzero means true*
NOT(%(16)F0F0) = %(16)0F0F *16 bits negated* *original value is false* *result value is true*	~(0xF0F0) == 0xFFFF0F0F *32 bits negated* *both values are true* !(0xF0F0) == 0 *since nonzero means true*

The HP C/XL "~" operator is probably the better first-pass replacement for the SPL NOT.

SPL TRUE and FALSE Constants.

SPL returns a 16-bit LOGICAL 65535 (INTEGER -1) for true and 0 for false. However, when *testing* a value for true or false in a condition clause, SPL examines only bit 15 for 1 or 0, ignoring bits 0-14.

HP C/XL returns a 32-bit integer 1 for true and 0 for false. When testing a value for true or false, HP C/XL tests the whole number for nonzero or 0.

These variations will have no effect on the value of a condition clause except if the expressions in the clause use the returned true or false values numerically, as in bit manipulation.

Many SPL programmers have taken advantage of the way SPL tests bit 15 for true or false, and existing SPL code must be carefully examined for examples of this practice. Too direct a translation of bit operations such as these is discouraged, as the resulting HP C/XL code will lack portability and be more difficult to maintain.

Numeric Conversion. Unless a logical expression used in a condition clause results in true or false values that are not 65535 or 0 respectively, or a relational (true/false) result is used in a bitwise or numeric operation (not a recommended coding practice), there should be no problem with a simple substitution of operator symbols.

In other words, if a test for true or false is not really a test for odd or even, and if the values true and false are not used as numbers, the results should be the same.

Converting a Range Test. The conversion of a range test, such as

```
X <= Y <= Z
```

may be performed in two steps.

(The example is true if X is less than or equal to Y, AND Y is less than or equal to Z.)

Step 1: In SPL, change the expression to two "<=" tests joined with LAND, for example:

```
(X) <= (Y) LAND (Y) <= (Z)
```

The parentheses may be needed to ensure the correct evaluation of the expressions.

Step 2: In HP C/XL, replace LAND with either "&" or "&&":

```
(X) <= (Y) & (Y) <= (Z)
(X) <= (Y) && (Y) <= (Z)
```

The "&" bitwise AND is the "precise" conversion operator, but the "&&" Boolean AND operator (described in "Condition Clauses" in this chapter) is more efficient.

Other Notes. Note that the SPL test for equality "=" is the assignment operator in HP C/XL. Failure to convert an SPL "=" to an HP C/XL "==" will result in a statement which compiles without error, but which performs a very different operation at runtime.

SPL uses relational operators to compare byte strings. See "Comparing Byte Strings" below for an explanation and examples.

Sequence of Operations (Logical)

Table 5-24. Order of Evaluation of Logical Operators

SPL	HP C/XL Equivalent
Order of evaluation: 1. logical bit operations logical expressions in parentheses logical function designators logical assignment statements in parentheses 2. *, ** (logical multiply; 16- and 32-bit) /, // (logical divide; 16- and 32-bit) MOD, MODD (logical modulus; 16- and 32-bit) 3. + (logical addition) - (logical subtraction) 4. <, <=, >, >=, =, <> (algebraic and logical comparisons) 5. LAND (logical bitwise AND) 6. XOR (logical bitwise exclusive OR) 7. LOR (logical bitwise inclusive OR) *lower* <= *test* <= *upper* (range test)	Same as SPL, except for the following: bit operations Implemented as function calls; same sequence level. equality tests == and != evaluate below <, <=, >, >=. Parentheses may be needed. range tests The conversion of SPL's X<=Y<=Z construct to HP C/XL's X<=Y & Y<=Z will probably need parentheses around the X, Ys, and Z.

Type Mixing (Logical)

The mixing of data types across operands is not allowed in SPL. Type transfer functions (see "Expression Types" above) are used to convert data types. See "Type Mixing (Arithmetic)" above for more detail.

Comparing Byte Strings

Table 5-25. Comparing Byte Strings

SPL	HP C/XL Equivalent
byte-comparison: 1. *byte-ref rel-op byte-ref* , (*count*) [, *stack-decr*] 2. *byte-ref rel-op* *PB , (*count*) *[, stack-decr*] 3. *byte-ref rel-op string-const* [, *stack-decr*]	*byte-comparison*: 1. strncmp (*byte-ref* , *byte-ref* , *count*) *rel-op* 0 2. *(No direct equivalent;* *convert to format* 1*)* 3. strcmp (*byte-ref* , *string-const*) *rel-op* 0

4. *byte-ref rel-op* (*value-group* [,...] *[, stack-decr*]	4. *(No direct equivalent; convert to format* 3*)*
5a. *byte-variable* = ALPHA	5a. isalpha (*byte-variable*)
5b. *byte-variable* <> ALPHA	5b. !isalpha (*byte-variable*)
5c. *byte-variable* = NUMERIC	5c. isdigit (*byte-variable*)
5d. *byte-variable* <> NUMERIC	5d. !isdigit (*byte-variable*)
5e. *byte-variable* = SPECIAL	5e. !isalnum (*byte-variable*)
5f. *byte-variable* <> SPECIAL	5f. isalnum (*byte-variable*)
byte-reference: a1. *array/pointer-id* a2. *array/pointer-id* (*index*) b. *	*byte-reference*: a1. *array/pointer-id* a2. & *array/pointer-id* " [" *index* "]" b. *(No equivalent; stack reference requires recoding)* The str... functions expect addresses of the strings; hence, the "&" in the indexed format. Note that *array/pointer-id* alone *is* an address (of cell zero).
count: is the number of characters to compare. If *count* is negative, the comparison is right-to-left.	The equivalent syntaxes work left-to-right only. An alternate user-defined function, BYTECMP, that handles both cases is shown below.
stack-decr: is the number of items to remove from the stack. The default value is 3.	The equivalent syntaxes above work only for a decrement of 3. The functionality of other values is provided in the user-defined function BYTECMP, shown below.
value-group: is a numerically defined byte string.	This element and its surrounding parentheses must be converted to an HP C/XL character string.
byte-variable is a reference to a single byte, either as an array or pointer cell reference or as a simple byte variable.	Same as SPL. The is... functions expect a character value.

Here are five examples of the basic forms of byte comparison:

SPL	HP C/XL Equivalent
A < B(3), (5),3	strncmp(A,&B[3],5) < 0
B(5) >= *PB,(5)	*No equivalent*
A <= "string"	strcmp(A,"string") <= 0
B = ("ab",%07)	strcmp(B,"ab\7") == 0
C <> ALPHA	!isalpha(C)

The second example above, which compares bytes to a previously stacked PB-relative address, is a hardware-dependent construct that has no equivalent in HP C/XL.

The isalnum, isalpha, isdigit, strcmp, and strncmp functions are all members of the standard HP C/XL function library.

Some more examples, used here as condition clauses of IF statements:

SPL	HP C/XL Equivalent
IF A = B,(5) THEN...	if (strncmp(A,B,5) == 0)...
IF A <> B,(5) THEN...	if (strncmp(A,B,5) != 0)...
IF A > B,(5) THEN...	if (strncmp(A,B,5) > 0)...
IF A < B,(5) THEN...	if (strncmp(A,B,5) < 0)...
IF A >= B,(5) THEN...	if (strncmp(A,B,5) >= 0)...
IF A(5) = "abc" THEN...	if (strcmp(&A(5),"abc") == 0)...
IF B <> "abc" THEN...	if (strcmp(B,"abc") != 0)...

These HP C/XL statements are equivalent to the SPL versions if the byte strings (character strings) being compared do not contain a NUL character in the range being tested.

The SPL byte comparisons scan exactly the number of characters indicated by *count* or the number of character values in the *string* or *value-group* s.

By definition, an HP C/XL string is terminated by the ASCII NUL character ('\0', numeric value 0). HP C/XL functions that scan strings usually stop scanning when they find a NUL character or when they reach a specified count.

However, because NUL equals zero and is the lowest character value, these comparison functions should work well, except in the following situation. Consider the case where both strings are equal up to a NUL character and different afterward: In HP C/XL notation,

    A == "ab\0de" (character values 'a', 'b',NUL,'d','e')

  and

    B == "ab\0fg" (character values 'a','b',NUL,'f','g')

The SPL comparison "A = B,(5)" would be false, because d is less than f. But the HP C/XL comparison "strncmp(A,B,5)==0" would be true, because strncmp stops scanning at the NULs.

The HP C/XL functions strcmp and strncmp return a value less than zero if the string pointed to by the first parameter compares less than the string pointed to by the second parameter, greater than zero if the first is greater than the second, and equal to zero if they are equal.

The three HP C/XL library functions isalpha, isdigit, and isalnum are not affected by this NUL "problem". They provide equivalents for all the corresponding SPL byte tests.

If the NUL character can be an embedded character, or if the *count* is negative, requiring a right-to-left scan, or if you wish to make use of the values left on the stack by the SPL byte comparisons, then the user-defined function BYTECMP can help. See Figure 5-18 in this chapter.

BYTECMP accepts the first byte-reference, the comparison code, the second byte reference, the count, and the stack decrement, as given in SPL syntax form 1. It also accepts the addresses where it can return the byte count and the left and right byte addresses where the comparison ended.

Also see "SPL BYTECMP Procedure: Byte Comparison" and "HP C/XL BYTECMP Function: Byte Comparison" for further details.

```
enum CMP { LSS, LEQ, EQU, NEQ, GEQ, GTR };

int BYTECMP(left,cmp,right,count,sdec,caddr,laddr,raddr)
   char *left, *right, **laddr, **raddr;
   enum CMP cmp;
   int count, sdec, *caddr;

{
#define ADJ {if (count > 0) {--count;++left;++right;} \
                       else {++count;--left;--right;}}

switch (cmp)
   {
   case LSS:  /* compare < */
              while ((count != 0) && (*left < *right))  ADJ;
              break;
   case LEQ:  /* compare <= */
              while ((count != 0) && (*left <= *right))  ADJ;
              break;
```

```
        case EQU:  /* compare == */
                   while ((count != 0) && (*left == *right))  ADJ;
                   break;
        case NEQ:  /* compare != */
                   while ((count != 0) && (*left != *right))  ADJ;
                   break;
        case GEQ:  /* compare >= */
                   while ((count != 0) && (*left >= *right))  ADJ;
                   break;
        case GTR:  /* compare >  */
                   while ((count != 0) && (*left > *right))  ADJ;
                   break;
        }

     switch (sdec)
        {
        case 0:  *raddr = right;
        case 1:  *laddr = left;
        case 2:  *caddr = count;
        case 3:  ;  /* nil */
        }
     return (count == 0)

     #undef ADJ
     }
```

Figure 5-18. HP C/XL BYTECMP Function: Byte Comparison

Condition Clauses

Table 5-26. Condition Clauses

SPL	**HP C/XL Equivalent**
condition-clause: *cond-term* [{AND OR} *cond-term*][...]	*condition-clause*: *cond-term* [{&& ||} *cond-term*][...]
cond-term is one of: *cond-primary* (*cond-primary* [OR *cond-primary*][...])	*cond-term* is one of: *cond-primary* (*cond-primary* [|| *cond-primary*][...])
cond-primary is one of: *logical-expression* *branch-word*	Only *logical-expression* is permitted.
branch-word is one of: CARRY NOCARRY OVERFLOW NOOVERFLOW IABZ DABZ IXBZ DXBZ < <= <> = > >=	No equivalent. These refer to MPE V hardware constructs and must be recoded. Some condition code testing is possible with the HP C/XL function ccode. See the *HP C/XL Library Reference Manual* for details.

In tests for true and false, an odd value is true (bit 15 is on); an even value is false (bit 15 is off).	A nonzero value is true; a zero value is false.

Condition clauses in SPL may appear in IF expressions and in IF, DO, and WHILE statements.

The SPL hardware branch words (CARRY, NOCARRY, etc.) test hardware registers built into the MPE V-based architecture. These hardware-dependent constructs will have to be rewritten using the intrinsic library routines.

Logical expressions may be combined using AND and OR. These Boolean operators generate branches to optimize runtime performance by suspending evaluation of an expression as soon as it is determined to be true or false. That is, as soon as any logical expression combined with AND is found to be false, the false branch is taken immediately.

SPL programmers use this feature, aware of the possible differences in side effects as a result of incomplete evaluation of a condition clause.

The SPL AND operator has a higher precedence than OR. This precedence can be overridden by parentheses. However, parentheses cannot be placed around items combined by the AND operator.

In HP C/XL, the "logical AND" operator is "&&". and the "logical OR" operator is "||". These are identical to the SPL AND and OR respectively, including the rules of precedence and partial evaluation. HP C/XL does not restrict parentheses around "&&".

CAUTION In SPL, the Boolean value of a logical expression is determined only by bit 15 of the value. If bit 15 is on, the expression is true. If bit 15 is off, the expression is false.

In HP C/XL, the Boolean value of a logical expression is determined by its numeric value. If it is nonzero, the value is true. If it is zero, the value is false.

Since a logical expression may be the result of numeric and logical as well as Boolean operations, you must be careful in converting it. See "Logical Expressions" above for further details.

IF Expressions

Table 5-27. IF Expressions

SPL	HP C/XL Equivalent
if-expression: IF *condition-clause* THEN *true-expression* ELSE *false-expression*	*conditional-expression*: *condition-clause* ? *true expression* : *false-expression*
Example: X + (IF A < B THEN 5 ELSE 6)	Example: X + (A < B ? 5 : 6)
In both cases above, the expression evaluates to X+5 if the condition clause "A < B" is true; otherwise, it evaluates to X+6.	
An IF expression may be used in any expression where the value of the result is allowed.	Same as SPL.

The HP C/XL syntax may look cryptic to SPL programmers. It can be beneficial to add parentheses to make the sections stand out, such as:

```
X + ( (A < B) ? ( 5 ) : ( 6 ) )
```

The HP C/XL "? :" conditional expression has lower precedence than "||" (logical OR) and higher precedence than "=" (assignment).

Assignment Statement

Table 5-28. Assignment Statement

SPL	HP C/XL Equivalent
assignment-statement: 1. *variable* [:= *variable*][...] := *expression* 2. *variable* (*left-bit* : *len*) [:= *variable*][...] := *expression*	*assignment-statement*: 1. *variable* [= *variable*][...] = *expression* ; 2. *(No direct equivalent;* *see* BDEPOSIT *function below.)* Note: The HP C/XL assignment operator, "=", is the same as the SPL equality operator.
The type of *expression* may be different from the types of the *variable* s and they may be different from each other, except they must all be the same length. Type BYTE is treate as a 16-bit quantity.	The types of the *variable* s and *expression* may be different They do not have to have the same length. HP C/XL performs automatic type conversions as assignment proceeds from right to left.
The leftmost assigned-to variable may	Bit-field assignment is not allowed.

specify a bit field in itself where the value will be deposited.	This operation may be performed separately with the user-defined function BDEPOSIT, described below.
May be used as an expression. Its value is the value stored into the leftmost operand. Its type is the type of the leftmost operand.	Same as SPL.

For compatability with very old systems, SPL accepts the "_" (underscore) character as an alternate to the ":=" assignment symbol. (Early terminals and printers labeled and displayed what now is the underscore as a "left arrow" symbol, "<--".)

SPL Examples:

```
Z := B * F;
```
arithmetic expression assignment
```
F1 := F2 = F3;
```
logical expression assignment
```
Z.(5:6) := P := B;
```
multiple assignment, bit deposit
```
Z := (B := B + 1) * 2;
```
assignment in expression
```
Z _ B;
```
underscore replacing ":="

HP C/XL Examples:

```
i = k * l;              /*arithmetic expression assignment*/
l1 = l2 == l3;          /*logical expression assignment*/
i = (k = k + 1) * 2;    /*assignment in expression*/
i = (++k) * 2;          /*same operation*/
```

The SPL bit deposit operation may be emulated in SPL and converted to HP C/XL in two steps.

Step 1: In SPL, add the BDEPOSIT procedure in Figure 5-19 to the compilation unit.

```
PROCEDURE BDEPOSIT(dw,sb,nb,expr);
VALUE dw, sb, nb, expr;
LOGICAL dw, sb, nb, expr;
BEGIN
  LOGICAL M;
  POINTER P;
  nb := 16-nb;
  sb := nb-sb;
  M := (%(16)FFFF & LSR(nb)) & LSL(sb);
  @p := dw;
  p := (p LAND NOT m) LOR (expr & LSL(sb) LAND m);
END;
```

Figure 5-19. SPL BDEPOSIT Procedure: Bit Assignment

Here dw is the address of the destination word, sb is the starting bit of the deposit field, nb is the number of bits to be deposited, and expr is the value to be deposited into the field.

Then separate the bit deposit from any multiple assignments and convert it to a procedure call. For example,

```
I.(5:6) := J + K ;
```

would become

```
BDEPOSIT(@I,5,6,J+K);
```

Note that the address of the first parameter is formed with the "@" operator, and that the parameter has been declared type LOGICAL (16 bit word), and passed by value. Within BDEPOSIT, this value is assigned to a pointer to allow the actual value to be accessed. This rather unconventional approach (normal SPL practice would be to pass this parameter by reference), is to simplify later conversion to the HP C/XL function described below.

Step 2: In HP C/XL, replace the SPL procedure with the HP C/XL BDEPOSIT function shown in Figure 5-20.

```
void BDEPOSIT(dw,sb,nb,exp)
   unsigned short *dw, sb, nb, exp;
{
   unsigned short m;
   nb = 16-nb;
   sb = nb-sb;
   m = (0xFFFF>>nb)<<sb;
   *dw = (*dw & ~m) | (exp<<sb & m);
}
```

Figure 5-20. HP C/XL BDEPOSIT Function: Bit Assignment

Then replace the converted SPL call to BDEPOSIT:

```
BDEPOSIT(@I,5,6,J+K);
```

with:

```
BDEPOSIT(&I,5,6,J+K);
```

Note that the only difference in the calls is that "@" is changed to "&".

MOVE Statement

Table 5-29. MOVE Statement

SPL	HP C/XL Equivalent

move-statement:
1. MOVE *target* := *source* , (*count*)
       [, *stack-decr*]
2. MOVE *target* := *[PB] , (*count*)
       [, *stack-decr*]
3. MOVE *target* := *string-const*
       [, *stack-decr*]
4. MOVE *target* := (*value-group* [,...])
       [, *stack-decr*]
5. MOVE *target* := *source* WHILE *cond*
       [, *stack-decr*]
6. MOVE *target* := * WHILE *cond*
       [, *stack-decr*]
	No direct equivalents.
	1. *(See the* MOVEB *and* MOVEW
	    functions below.)
	2. *(Convert to format* 1.*)*
	3. *(See the* MOVEB*,* MOVEW*, and* MOVESB
	    functions below.)
	4. *(Convert to format* 3.*)*
	5. *(See the* MOVEBW *function below.)*
	6. *(Convert to format* 5.*)*

target: *array/pointer-ref* * *source*: *array/pointer-ref*	
May be used (without *stack-decr*) as an integer expression. Its value is the number of words or bytes moved.	

MOVE statements in SPL are designed to utilize several sophisticated hardware move instructions. There are byte and word moves which can be performed unconditionally or dependent upon a test condition. The destination of the move must be an array or pointer, and the source may be an array, a pointer, a string constant, or a group of values. Two of the SPL moves are not directly translatable, for example:

```
MOVE arrayname := *PB,(count)

MOVE array name := (10(" "),"string",5(""))
```

The first is non-translatable because there is no register-relative addressing in HP C/XL; the second, because repeat factors and grouping of constants into a list are not available. The second case may be handled by multiple move operations or manual expansion of the repeti- tions into a string constant.

NOTE The str... amd mem... series of HP C/XL standard library functions may also be useful here. The str... functions expect the string to be terminated with NUL ('\0', numeric value 0). The mem... functions do not use NUL. See the *HP C/XL Library Reference Manual* for details.

Unconditional byte moves may be emulated in HP C/XL by the MOVEB function, shown in Figure 5-21.

```
int MOVEB(to,from,count,sdec,source_adr,dest_adr)
   char *to, *from, **source_adr, **dest_adr;
   int count, sdec;
{
   int c;
   c = 0;
   if (count>0)  /* left-to-right move */
      do *to++ = *from++; while (++c < count);
      else if (count<0)  /* right-to-left move */
         {
         count = -count;
         do *to-- = *from--; while (++c < count);
         }
   switch (sdec)
      {
      case 0:  ;  /*  fall through to case 1 */
      case 1:  *source_adr = from;
      case 2:  *dest_adr = to;
      case 3:  ;  /* nil */
      }
   return(c);
}
```

Figure 5-21. HP C/XL MOVEB Function: MOVE Bytes Statement

In MOVEB, to is the *target* address, from is the *source* address, count isthe number of bytes to be moved (a positive value means a left-to-right move, negative means right-to-left), and sdec is is the value which would have been used as an SPL stack decrement. In this context, sdec = 3 will cause the function to ignore the last two parameters, which need not be present. An sdec = 2 will set the value for dest_adr, sdec = 1 or 0 will set both dest_adr and source_adr. The parameter source_adr is the address of the next character beyond the final character moved, dest_adr is the address of the next character beyond the final character moved, and the return value of the function is the number of bytes moved.

The following emulates the MOVE statement in SPL for byte moves with no information removed from the stack:

```
MOVE A1 := A2, (CNT), 0
LEN := TOS;               will always be zero
@S1 := TOS;
@D1 := TOS;
NUM := @D1 - @A1;         number of bytes moved
```

This may be converted to HP C/XL as:

```
NUM = MOVEB(&A1,&A2,CNT,0,&S1,&D1);
```

The other variants of byte moves (removing one, two, or all three of the words normally left on the stack after a MOVE) may all be emulated by this function.

Word moves of 16-bit quantities may be emulated by a minor variation of MOVEB, the HP C/XL function, MOVEW, shown in Figure 5-22.

```
int MOVEW(to,from,count,sdec,source_adr,dest_adr)
   unsigned short *to, *from, **source_adr, **dest_adr;
{
   int c;
   c = 0;
   if (count>0)  /* left-to-right move */
      do *to++ = *from++; while (++c < count);
      else if (count<0)  /* right-to-left move */
         {
         count = -count;
         do *to-- = *from--; while (++c < count);
         }

   switch (sdec)
      {
      case 0:  ;  /*  fall through to case 1 */
      case 1:  *source_adr = from;
      case 2:  *dest_adr = to;
      case 3:  ;  /* nil */
      }

   return(c);
}
```

Figure 5-22. HP C/XL MOVEW Function: MOVE Words Statement

The MOVE statement with a WHILE condition may be emulated by the HP C/XL MOVEBW function, shown in Figure 5-23.

MOVEBW is used similarly to MOVEW, but, instead of a count, a condition is supplied. The condition is chosen from the enum declared as COND that matches the SPL options.

The SPL operation:

```
LEN := MOVE B1 := B2 WHILE AS;
@S1 := TOS;
@D1 := TOS;
```

may be replaced with the HP C/XL function call:

```
LEN = MOVEBW(B1,B2,AS,0,&S1,&D1);
```

In SPL, a MOVE-WHILE operation sets a condition code to indicate the type of the last character of the source that was examined (but not moved). This is easily tested by standard HP C/XL character functions. For example, if an SPL MOVE-WHILE statement is followed by:

```
IF > THEN...<<move stopped on a digit 0-9>>
```

you may use the HP C/XL equivalent:

```
if isdigit(*(s1-1)).../* move stopped on a digit */
```

```
enum COND { A, AN, AS, N, ANS };

int MOVEBW(to,from,cond,sdec,source_adr,dest_adr)
   enum COND cond;
   char *to, *from, **source_adr, **dest_adr;
   int sdec;
{
   char *temp;
   temp = to;
   switch (cond)
      {
      case   A:  while (isalpha(*from)) *to++=*from++;
                 break;
      case  AN:  while (isalnum(*from)) *to++=*from++;
                 break;
      case  AS:  while (isalpha(*from)) *to++=toupper(*from++);
                 break;
      case   N:  while (isdigit(*from)) *to++ = *from++;
                 break;
      case ANS:  while (isalnum(*from)) *to++=toupper(*from++);
                 break;
      }

   switch (sdec)
      {
      case 0:  ;  /*  fall through to case 1 */
      case 1:  *source_adr = from;
      case 2:  *dest_adr = to;
      }

   return(to-temp);
}
```

Figure 5-23. HP C/XL MOVEBW Function: MOVE Bytes WHILE Statement

Moving a string constant into a byte array or through a byte pointer may require the HP C/XL MOVESB function, shown in Figure 5-24.

```
int MOVESB(to,str,sdec,source_adr,dest_adr)
   char *to, *str, **source_adr, **dest_adr;
   int sdec;
{
   char *temp;
   temp = to;
   while (*str != '\0') *to++ = *str++;
   switch (sdec)
      {
      case 0:  ;  /*  fall through to case 1 */
      case 1:  *source_adr = str;
      case 2:  *dest_adr = to;
      case 3:  ;  /* nil */
      }
   return(to - temp);
}
```

Figure 5-24. HP C/XL MOVESB Function: MOVE String Bytes Statement

This function makes use of the fact that HP C/XL terminates a string with the NUL character ('\0', numeric value 0).

Consequently, the SPL code

```
LEN := B1 := "test string",0;
CNT := TOS;  <<always zero>>
@S1 := TOS;
@D1 := TOS;
```

may be replaced with:

```
LEN = MOVESB(S1,"test string",0,&S1,&S1);
```

MOVEX Statement

This SPL statement is available only to privileged users accessing extra data segments. Any use of extra data segments should be recoded, utilizing the larger memory space available in HP C/XL.

SCAN Statement

Table 5-30. SCAN Statement

SPL	HP C/XL Equivalent
scan-statement: SCAN *byte-ref* {WHILE UNTIL} *testword* [, *stack-decr*]	No direct equivalent.
byte-ref is one of: *array/pointer-id* *array/pointer-id* (*index*) *	Same as SPL, except * stack reference must be recoded.
testword is one of: integer constant INTEGER or LOGICAL variable string constant of one or two characters *	
First character of *testword* is *terminal-char*. Second character of *testword* is *test-char*. If *terminal-char* is omitted, it is NUL (numeric 0).	

In SCAN-UNTIL, scan starts at *byte-ref* and continues until either *test-char* or *terminal-char* is found. In SCAN-WHILE, scan starts at *byte-ref* and continues until either *terminal-char* is found or character NOT matching *test-char* is found. Carry bit in status register is set to one if *terminal-char* was found; otherwise, it is set to zero. The address of the terminating byte is placed on the stack.	
May be used (without *stack-decr*) as an arithmetic function. Its value is the number of words or bytes scanned.	

The SCAN statement in SPL searches a string of bytes for either of two characters, a test character and a terminating character. The statement may be used either as a function to return the number of bytes scanned, or with a stack decrement value to leave information on the stack.

The HP C/XL library contains string search functions which perform similar operations. For example, the SPL statements

```
SCAN B1 WHILE " ",0;  <<scan while zero or blank>>
T := TOS;             <<testword, always unchanged>>
@S1 := TOS;           <<address of first blank>>
```

may be duplicated in HP C/XL by

```
s1 = strchr(b1,' ');
```

The strchr function searches for a single character, returning an address where it was found. To look for two characters, as SCAN does, another function may be used:

```
s1 = b1 + strcspn(b1,"% ");
```

The function strcspn returns a count of the number of characters which were *not* any of the characters in the second parameter. This value added to the address being searched yields the address of the first occurrence of a character in the string supplied as the second parameter.

SCAN may be used as a function. For example,

```
NUM := SCAN B1 UNTIL " ";
```

or

```
NUM := SCAN B1 UNTIL "% ";
```

In this case, these statements might become:

```
NUM = strchr(B1,' ') - B1;"
```

or

```
NUM = strcspn(B1,"% ");
```

NOTE The HP C/XL library function memchr can be used to scan strings that are not terminated by NUL ('\0', numeric value 0). For more information on memchr and its related functions and on the str... series of functions, see the *HP C/XL Library Reference Manual*.

The HP C/XL SCANU function, shown in Figure 5-25, duplicates the SCAN-UNTIL operation.

```
int SCANU(ba,test,sdec,scan_adr)
   char *ba, *scan_adr;
   unsigned short test;
   int sdec;
{
   char termc, testc, *temp;
   temp = ba;
   termc = (char)test >> 8;
   testc = (char)test & 0xFF;
   while ((*ba != testc) && (*ba != testc)) ba++;
   switch (sdec)
      {
      case 0:  ;  /*  fall through to case 1 */
      case 1:  *scan_adr = ba;
      case 2:  ;  /* nil */
      }
   return(ba - temp);
}
```

Figure 5-25. HP C/XL SCANU Function: SCAN-UNTIL Statement

Chapter 6 Program Control Statements

This chapter discusses conversion issues that correspond to sections in Chapter 5 of the *Systems Programming Language Reference Manual*.

Program Control

SPL has nine basic methods of altering the normal sequential execution of instructions: CASE, DO, FOR, GOTO, IF, RETURN, and WHILE statements, and procedure and subroutine call statements.

HP C/XL has equivalents for all these control statements, with some variations in syntax. In general, HP C/XL provides more options and control than SPL.

GO TO Statement

Table 6-1. GOTO Statement

SPL	HP C/XL Equivalent
goto-statement: 1. GO [TO] *label* 2. GO [TO] [*] *switch-id* (*index*)	Similar to SPL: 1. goto *label* ; 2. *switch-id* (*index*) ;
1. This syntax may transfer to a label in the current routine (main or procedure) or to a label outside a procedure that was passed to the procedure as a parameter. 2. This syntax transfers to a label declared in a SWITCH declaration for the current routine (main or procedure). The "*" option turns off bounds checking.	1. Same as SPL, except that labels cannot be passed to functions. Passed labels must be recoded, perhaps as a function return value. 2. The SWITCH declaration should be recoded as a #define macro directive, as described in "SWITCH Declaration" and "Local SWITCH Declarations". Then the conversion syntax above will execute an HP C/XL switch transfer to the correct label. The SPL "*" option has no HP C/XL equivalent. Just delete it.

Table 6-2. GO TO Statement Examples

<table>
<tr><th>SPL</th><th>HP C/XL Equivalent</th></tr>
<tr><td><pre>
1. GO LABEL1;
    GOTO LABEL1;
    GO TO LABEL1;

2. SWITCH SWITCHLABEL:=L0,L1,L2;
...
    GOTO SWITCHLABEL(JUMP);
</pre></td><td><pre>
1. goto LABEL1;
    goto LABEL1;
    goto LABEL1;

2. #define SWITCHLABEL(X) \
         switch (X) \
              { \
              case 0: goto L0; \
              case 1: goto L1; \
              case 2: goto L2; \
              }
...
    SWITCHLABEL(JUMP) ;
</pre></td></tr>
</table>

DO Statement

Table 6-3. DO Statement

SPL	HP C/XL Equivalent
do-statement: DO *loop-statement* UNTIL *condition-clause*	Similar to SPL: do *loop-statement* while (! (*condition-clause*))
The *loop-statement* (which may be compound) is executed until the *condition-clause* becomes true. It is always executed at least once.	The *loop-statement* (which may be compound) is executed until the expression after while becomes false. It is always executed at least once. Note that this test is the reverse of the SPL version. As shown in the syntax above, the easiest conversion is to enclose the SPL *condition-clause* in parentheses, precede it with the logical NOT operator "!", and then add the outer parentheses required by HP C/XL. You could also just invert the condition, if it's a simple one. For example, "==" would become "!=" and ">=" would become "<".

Table 6-4. DO Statement Examples

SPL	HP C/XL Equivalent

<pre>DO BEGIN
   X := X + 1;
   A(X) := B(X);
   END
   UNTIL X=100;</pre>
	<pre>do {
	   X = X + 1;
	       /*could also be: X++; */
	   A[X] = B[X];
	   }
	   while (!(X==100));
	       /*test could be: (X!=100)*/</pre>

WHILE Statement

Table 6-5. WHILE Statement

SPL	HP C/XL Equivalent
while-statement: WHILE *condition-clause* DO *loop-statement*	Similar to SPL: while (*condition-clause*) *loop-statement*
The *loop-statement* (which may be compound) is executed only if and while the *condition-clause* remains true. If *condition-clause* is false to begin with, *loop-statement* is not executed at all.	Same as SPL. The *condition-clause* must be enclosed in parentheses. Also, remove the keyword DO.

Table 6-6. WHILE Statement Examples

SPL	HP C/XL Equivalent

<pre>WHILE X <> 100 DO
   BEGIN
   X := X + 1;
   A(X) := B(X);
   END;</pre>
	<pre>while (X != 100)
	   {
	   X = X + 1; /*could be: X++; */
	   A[X] = B[X];
	   }</pre>

FOR Statement

Table 6-7. FOR Statement

SPL	HP C/XL Equivalent
for-statement:	*for-statement*:
1. FOR *test-var* := *init-val* UNTIL *end-val* DO *loop-statement*	1. for (*test-var* = *init-val* ; *test-var* <= *end-val* ; *test-var* ++) *loop-statement*
2. FOR *test-var* := *init-val* STEP *step-val* UNTIL *end-val* DO *loop-statement*	2a. for (*test-var* = *init-val* ; *test-var* <= *end-val* ; *test-var* += *step-val*) *loop-statement* 2b. for (*test-var* = *init-val* ; *test-var* >= *end-val* ; *test-var* += *step-val*) *loop-statement*
3. FOR * *test-var* := *init-val* UNTIL *end-val* DO *loop-statement*	3. for (*flag* = 1 , *test-var* = *init-val*; *flag* || *test-var* <= *end-val*; *flag* = 0 , *test-var* ++) *loop-statement*
4. FOR * *test-var* := *init-val* STEP *step-val* UNTIL *end-val* DO *loop-statement*	4a. for (*flag* = 1 , *test-var* = *init-val*; *flag* || *test-var* <= *end-val*; *flag* = 0 , *test-var* += *step-val*) *loop-statement* 4b. for (*flag* = 1 , *test-var* = *init-val*; *flag* || *test-var* >= *end-val* ; *flag* = 0 , *test-var* += *step-val*) *loop-statement*
	The generic syntax is: for (*init-expr* ; *test-expr* ; *incr-expr*) *statement*
init-val, *step-val*, and *end-val* are evaluated and stored. *test-val* is set to *init-val*. *test-val* is compared to *end-val*. If *step-val* is positive or omitted, and *test-val* is less than or equal to *end-val*, *loop-statement* is executed. If *step-val* is negative, and *test-val* is greater than or equal to *end-val*, *loop-statement* is executed. If the test fails, the for statement terminates. After *loop-statement* is executed, *test-val* is incremented by *step-val* or 1, and it is compared with *end-val* as above.	In general, the HP C/XL expressions are independent, and may even be omitted! The values in the expressions may be changed in the body of the for statement. The first expression (e.g., *init-expr*) is evaluated only once, on initial entry into the for statement. For each iteration, the second expression (e.g., *test-expr*) is evaluated. If it is false, the for statement is terminated. If it is true, the *loop-statement* is executed, the third expression (e.g., *incr-expr*) is evaluated, and the for statement iterates. In the simplest case, *init-expr* initializes a *test-var*, *test-expr* tests it, and *incr-expr* increments it.

	Formats 2a and 4a deal with the case where *step-val* is positive. Formats 2b and 4b handle the case where *step-val* is negative. There is no easy way to combine the formats.

In HP C/XL, the three expressions can actually contain multiple expressions, separated by commas. The last or right-most becomes the value of the expression. This is the method used to solve the SPL "*" alternative, in formats 3 and 4. An arbitrary variable, *flag* is set to 1. Since *flag* is true on the first pass, it forces the execution of *loop-statement*. On subsequent passes, it is 0 or false, so the normal end testing takes over.

Table 6-8. FOR Statement Examples

SPL	HP C/XL Equivalent
`FOR I:=ABC STEP 1 UNTIL 99` `   DO A(I):=B(I)-X;`	`for ( I = ABC ; I <= 99 ; I++ )` `   A[I] = B[I]-X;`

`FOR * I:=ABC STEP -1 UNTIL 0`
`   DO A(I):=B(I)-X;`
	`for ( ONCE = 1 , I = ABC ;`
	`          ONCE || I >= 0 ;`
	`          ONCE=0 , I-- )`
	`             A[I] = B[I]-X;`

The SPL FOR * construct may also be easily emulated by an HP C/XL do-while statement, as illustrated by the following statements:

```
I = ABC ;
do A[I] = B[I]-X ; while (--I >= 0);
```

IF Statement

Table 6-9. IF Statement

SPL	HP C/XL Equivalent
if-statement IF *condition-clause* THEN *true-statement* [ELSE *false-statement*]	*if-statement*: if (*condition-clause*) *true-statement* [else *false-statement*]
If *condition-clause* is true, *true-statement*	Same as SPL.

is executed. If it is false, and the ELSE clause is present, *else-statement* is executed; if the ELSE clause is omitted, execution falls through to the statement after *true-statement*.	
If the ELSE clause is present, *true-statement* *must not* end with a semicolon.	Regardless of whether the else clause is present, a simple *true-statement* *must* end with a semicolon. (The terminating "}" of a compound statement implies the semicolon.) Note that the *condition-clause* is enclosed in parentheses and that THEN is deleted.

Table 6-10. IF Statement Examples

<table>
<tr><th>SPL</th><th>HP C/XL Equivalent</th></tr>
<tr><td><pre>
IF X<0 THEN Y:=0;

IF X>0 THEN
   BEGIN
   Y:=0;
   T:=V+10;
   END;

IF X=0 THEN X:=21
       ELSE
          BEGIN
          Y:=0;
          T:=V+10;
          END;
</pre></td><td><pre>
if (X<0) Y=0;

if (X>0)
   {
   Y=0;
   T=V+10;
   }

if (X==0) X=21;
   else
      {
      Y=0;
      T=V+10;
      }
</pre></td></tr>
</table>

CASE Statement

Table 6-11. CASE Statement

SPL	HP C/XL Equivalent
case-statement: CASE [*] *index* OF BEGIN *statement0* ; *statement1* [;...][;] END	*switch-statement*: switch (*index*) " {" case 0: *statement0* break ; case 1: *statement1* break ; [...] [default: *exception-statement*] " }"
The statements in the BEGIN-END clause are implicitly numbered from 0. They may be compound statements.	The statements in the { } clause are explicitly numbered with case *number* labels. *number* may be any integer constant, including a character constant. There may be multiple simple, structured, or compound statements between labels. The break statement is required to emulate the operation of the SPL CASE statement, except if the SPL statement contains a GOTO statement.

index is evaluated and the corresponding statement in the BEGIN-END clause is executed. Then execution drops through to the statement after the CASE statement.	*index* is evaluated and execution transfers to the case label with the corresponding *number*. Statements are executed in sequential order from that point until
	(1) the end of the { } clause is reached,
	(2) a break statement is executed, or
	(3) a goto statement is executed.
	If (1) or (2) occurs, execution drops through to the statement after the switch statement.
	If (3) occurs, execution continues at the label specified.

If *index* is out-of-range, execution simply drops through to the statement after the CASE statement.	The optional default label can be used to trap out-of-range index values. If it is omitted, execution simply drops through to the statement after the switch statement.
The "*" option turns off bounds checking.	No equivalent. Just delete the "*".

Table 6-12. CASE Statement Examples

SPL	HP C/XL Equivalent

CASE N OF
BEGIN
A:=100; <<case 0, N=0>>
; <<case 1, N=1>>
BEGIN <<case 2, N=2>>
A:=90;
B:=1;
END;
B:=100; <<case 3, N=3>>
END;
	switch (N)
	{
	case 0: A=100; /* N==0 */
	break;
	case 1: break; /* N==1 */
	case 2: A=90; /* N==2 */
	B=1;
	break;
	case 3: B=100; /* N==3 */
	break;
	}

Case 1 is a null statement. It is required to fill out the range of values for N, even if N would never equal 1.	Case 1 could be omitted entirely, since an index not represented by a case label terminates the switch. To emulate the SPL operation, each case ends with a break statement to terminate the switch. Note that case 2 does not require braces around the two statements, although they could be used to clarify the BEGIN-END translation.

Again, please note the following:

In SPL, after each "case" of a CASE statement is executed, there is an automatic transfer to the end of the CASE statement. In HP C/XL, execution by default "falls through" to the next case. The break statement causes control to transfer to the statement following the switch statement, emulating SPL's action.

This and other features of the HP C/XL switch statement may afford opportunities to simplify older SPL algorithms once the code has been implemented in HP C/XL.

In the HP C/XL switch statement, if you include a case labelled default, invalid indexes will transfer to this label. Using a default label is good programming practice.

The HP C/XL case labels are simply entries into a series of statements. They may occur in any order and there may be gaps in the numeric sequence.

Procedure Call Statement

Table 6-13. Procedure Call Statement

SPL	HP C/XL Equivalent
procedure-call-statement: 1. *procedure-id* 2. *procedure-id* ( ) 3. *procedure-id* (*actual-parm* [,...])	*function-call-statement*: 1. *function-id* ( ) ; 2. *function-id* ( ) ; 3. *function-id* (*actual-parm* [,...]);
A procedure call causes a control transfer to a procedure, supplying any required parameters. Formats 1 and 2 are equivalent.	Same as SPL, using a function call. As shown in format 1, HP C/XL requires the parentheses even if there are no actual parameters.
actual-parm: a. *simple-variable-id* b. *array/pointer-id* c. *procedure-id* d. *entry-id* e. *label-id* f. *array/pointer-id* (*index*) g. *arithmetic-expression* h. *logical-expression* i. *assignment-statement* j. *	*actual-parm*: a-r. &*simple-variable-id* a-v. *simple-variable-id* b-r. *array/pointer-id* b-v. *array/pointer-id* " [" 0 "]" c-r. *function-id* d-r. *(No equivalent; must be recoded)* e-r. *(No equivalent; must be recoded)* f-r. &*array/pointer-id* " [" *index* "]" f-v. *array/pointer-id* " [" *index* "]" g-v. *arithmetic-expression* h-v. *logical-expression* i-v. *assignment-expression* j. *(No equivalent; must be recoded)*
Parameter formats a, b, and f may be pass-by-reference or pass-by-value. Their pass-by-value use is also included in formats g and h. Formats c, d, and e are pass-by-reference only. Formats g, h, and i are pass-by-value only. Format j may be either.	Parameter formats marked with "-r" are pass-by-reference. Formats marked with "-v" are pass-by-value. In HP C/XL, while all parameters are pass-by-value, pass-by-reference is achieved by passing a pointer value to a pointer parameter. Function-ids are passed as pointers; unsubscripted array-ids are passed as pointers to their first elements. Array, pointer, and function formal parameters expect pointer actual parameters.
Whether an *actual-parm* is pass-by-value or pass-by-reference depends on the definition of the procedure. SPL performs strict type-checking to ensure that parameters match.	Similar to SPL, except that HP C/XL performs no type checking whatsoever. The programmer must ensure that pointers are passed to pointers, integers are passed to integers, and reals are passed to reals. Note that all char, enum, and int types are expanded to [unsigned] long int types, and float is expanded to double when the actual parameters are evaluated. The passed long int, double, and pointer values are converted to the declared formal type when they are received by the function.

	Fortunately, because SPL is so strict, the conversion boils down to getting the pass-by format correct.

Table 6-14. Procedure Call Statement Examples

SPL	**HP C/XL Equivalent**
PROCEDURE P1 (VALP);	void P1 (VALP)
VALUE VALP; INTEGER VALP;	unsigned short VALP;
BEGIN	{
GVAR := VALP;	GVAR = VALP;
END;	}
PROCEDURE P2 (REFP);	void P2 (REFP)
INTEGER REFP;	unsigned short *REFP;
BEGIN	{
GVAR := REFP;	GVAR = *REFP;
END;	}
...	...
<<main program>>	/*main function*/
...	...
P1(V); <<pass-by-value>>	P1(V); /*pass-by-value*/
P2(V); <<pass-by-reference>>	P2(&V); /*pass-by-reference*/
...	...

In the examples above, notice that when P2 was called (in HP C/XL), the *address* of the variable was explicitly specified with the "&" operator. If the actual parameter had been an unsubscripted array-id or a string literal, the "&" would have been omitted. In HP C/XL, if an identifier A is declared to be an array, the following function call expressions are equivalent:

P3(A); *and* P3(&A[0]);

The data type void specifies that the function does not return a value. See "PROCEDURE Declaration" for details.

String Literals

HP C/XL allows string literals to be passed as actual parameters, which is not possible in SPL. Thus, the following SPL code

```
MOVE BARRAY:="test string";
PROCB(BARRAY);  <<called with byte array BARRAY>>
```

used to pass a string to an SPL procedure via a byte array, may be rewritten in HP C/XL as:

```
PROCB("test string");
```

HP C/XL will create storage for the string, and pass its address to PROCB as a "pointer to char" (a byte address). This is a more straightforward means of accomplishing the same operation as the SPL example.

Stacking Parameters

By directly manipulating the hardware stack in MPE V, SPL programmers can set up parameters to a procedure directly and then call the procedure using "*" actual parameters.

There is no equivalent in HP C/XL. However, replacing the "*"s in the parameter list with the stacked values is functionally equivalent. (Usually, the procedure call is preceded by assignments to TOS, which can be thus eliminated.)

This technique is used in SPL mostly to optimize runtime performance, not
to gain otherwise unavailable functionality. A simple rewrite will eliminate any explicit references to the stack.

Missing Parameters in Procedure Calls

If an SPL procedure is declared with OPTION VARIABLE, parameters may be omitted from the actual parameter list when the procedure is called.

HP C/XL provides the varargs macros to enable variable-length actual parameter lists. This feature is described further in "Options", and in the *HP C/XL Library Reference Manual*.

Passing Labels as Parameters

SPL has an elaborate facility for passing labels to procedures as actual parameters. When control is transferred to the label, the procedure automatically performs an exit from itself (and from any other procedures in the calling sequence between this one and the one containing the passed label) prior to transferring control to the label location. This effectively "unwinds" a stack of procedure calls, and is most often used in error recovery.

HP C/XL does not permit labels to be passed as parameters. These situations can (and must) be rewritten, possibly by declaring a global flag variable to indicate error conditions. This flag should be tested by functions to determine if processing is to be terminated prematurely.

Another approach is to use the longjmp and setjmp functions described in the *HP C/XL Library Reference Manual*.

Passing Procedures as Parameters

An SPL procedure (e.g., A) may be passed to another procedure (e.g., B) as a pass-by-reference parameter. When A is called from B, the actual parameters supplied in the parameter list at the time of the call are assumed to be pass-by-reference. Pass-by-value actual parameters *must* be placed on the stack and specified with the "*" symbol in the procedure call. OPTION VARIABLE passed procedures require more work, including the fabrication on the stack of a special mask word.

In HP C/XL, a function-id may be passed as an actual parameter. There are no particular restrictions on the actual parameter list when the passed function is called. For example,

```
main()
{
   void callf(), calledf(); /* declares two functions */

   callf(calledf); /* execute callf, passing calledf */
} /* end of main */

void callf(func) /* function to call a function */
  void (*func)(); /* func is pointer to function returning void */
{
   (void) func(); /* call the passed function */
} /* end of callf */

void calledf() /* function that will be passed */
{
   printf("called calledf!\n");
} /* end of calledf */
```

For further information on OPTION VARIABLE cases, see "Missing Parameters in Procedure Calls" above.

Subroutine Call Statement

SPL subroutines may be called from the procedures in which they are declared.

HP C/XL does not allow nested functions. Subroutines must be converted either to #define macro directives to generate code inline, or to functions that may be callable by other functions. See "SUBROUTINE Declaration" for further details.

The subroutine call itself may not require any modification at all. If you use a #define macro directive, make sure the left parenthesis in both the macro directive and the macro reference follows the identifier with no spaces. e.g., "mysubcall(arg)".

RETURN Statement

Table 6-15. RETURN Statement

SPL	HP C/XL Equivalent
return-statement: 1. RETURN [*count*] 2. *procedure-id* := *procedure-id-value* : RETURN [*count*]	Similar to SPL: 1. return ; 2a. return *procedure-id-value* ; 2b. *return-id* = *procedure-id-value* : return *return-id* ;
Format 1 is a return from a procedure that does not return a value. Format 2 is a return from a function procedure that returns a value assigned to the *procedure-id*. The *count* is the number of words to delete from the stack.	Format 1 is equivalent to SPL. Format 2 is equivalent to SPL. Format 2b is a simple way to convert the SPL code. Simply change the use of the *procedure-id* inside the procedure to another, local same-type identifier, here called *return-id*. Then append this *return-id* to the return statement. In any case, *count* must be recoded or ignored.
RETURN is used to exit from a procedure at a point other than the END of the procedure body.	Same as SPL for void functions. Functions used in expressions require a returned value. The only way to return a value is with the return statement. For functions that return a value, add a return statement before the final brace.

See "Data Type" for examples and additional information.

Chapter 7 Machine Level Constructs

This chapter discusses conversion issues related to sections in Chapter 6 of the *Systems Programming Language Reference Manual*.

ASSEMBLE Statement

Table 7-1. ASSEMBLE Statement

SPL	**HP C/XL Equivalent**
assemble-statement: ASSEMBLE ({[*label-id* :] *instruction* }[;...])	No equivalent.
Allows direct access to MPE V machine instructions.	Many of the instructions have functional equivalents in HP C/XL. See the example below. In general, register manipulation instructions will have to be redesigned and rewritten, whereas memory reference instructions frequently have straightforward replacements.
Example: ASSEMBLE(INCM ivar); *increment memory*	HP C/XL version: ++ivar; *same operation*

DELETE, PUSH, SET, and WITH Statements

The SPL DELETE, PUSH, SET, and WITH statements directly manipulate the MPE V hardware stack and registers.

In the absence of any assumed stack environment, HP C/XL has no direct equivalent constructs. A stack could be emulated in an array, but, in most cases, a simple redesign is preferable.

Chapter 8 Procedures, Intrinsics and Subroutines

This chapter discusses conversion issues that correspond to sections in Chapter 7 of the *Systems Programming Language Reference Manual*.

Subprogram Units

Table 8-1. Subprogram Units

SPL	HP C/XL Equivalent
Procedure	Function
Intrinsic	Intrinsic
Subroutine, global	static function or #define directive.
Subroutine, local (in procedure)	No equivalent. Convert to inline code, #define directive, or separate static function.

Much of the information about declarations has been discussed in detail in Chapter 4. This chapter will focus primarily on the special requirements of procedures. For more on subroutines, see "SUBROUTINE Declaration" below.

In the following sections, the SPL and HP C/XL *type* syntax elements refer to the following simple variable types:

SPL	HP C/XL Equivalent
INTEGER	short int
DOUBLE	long int

LOGICAL	unsigned short int
BYTE	unsigned char OR unsigned short int
REAL	float
LONG	double

PROCEDURE Declaration

Table 8-2. PROCEDURE Declaration

SPL	**HP C/XL Equivalent**
procedure-declaration: 1. [*type*] PROCEDURE *procedure-id* (*formal-parm* [,...]) ; [VALUE *formal-parm* [,...] ;] *formal-parm-decl* [;...] ; [OPTION *option* [,...] ;] [*procedure-body* ;] 2. [*type*] PROCEDURE *procedure-id* [OPTION *option* [,...] ;] [*procedure-body* ;] 3a. [*type*] PROCEDURE *procedure-id* OPTION EXTERNAL [, *option*] [...] ; 3b. [*type*] PROCEDURE *procedure-id* OPTION FORWARD [, *option*] [...] ;	*function-definition*: 1. [static] [*type* void] *function-id* (*formal-parm* [,...]) *formal-parm-decl* [;...] ; *function-body* 2. [static] [*type* void] *function-id* ( ) *function-body* 3. extern [*type* void] *function-id* ( )
formal-parm-decl: a. *type* *formal-parm* [,...] b. [*type*] ARRAY *formal-parm* [,...] c. [*type*] POINTER *formal-parm* [,...] d. [*type*] PROCEDURE *formal-parm* [,...] e. LABEL *formal-parm* [,...]	*formal-parm-decl*: a.[*type*] *formal-parm* [,...] b.[*type*]{*formal-parm* " [" "]" } [,...] c. [*type*] {* *formal-parm* } [,...] d. [*type*] {*formal-parm* ( )} [,...] e. *(labels cannot be passed)*
option: The CHECK, EXTERNAL, FORWARD, INTERNAL, INTERRUPT, PRIVILEGED, SPLIT, UNCALLABLE, and VARIABLE options are discussed in	*storage*: The extern and static storage classes are discussed in "Options" below.

"Options" below.	
procedure-body: a. *statement* b. BEGIN [*local-data-declarations*] [*external/intrinsic-declarations*] [*local-subroutine-declarations*] [*statement* [;...]] END	*function-body*: a. " {" *statement* " }" b. " {" [*local-declarations*] [*statement* [...]] " }"
A procedure declaration: * defines a procedure identifier * specifies whether the procedure will return a value (*type*) * describes the parameters: number, type, pass-by-value or pass-by-reference * specifies any options * declares local variables * includes the statements of the procedure body	A function definition: * defines a function identifier * specifies whether the function will NOT return a value (void) * describes the parameters: number, type (all are pass-by-value) * specifies a storage class * declares local variables * includes the statements of the function body
Procedure declarations cannot be nested, except that a procedure with OPTION EXTERNAL and no body may be declared in a procedure's local declarations.	Same as SPL, using the extern storage class.

Data Type

Table 8-3. Data Type

SPL	**HP C/XL Equivalent**
Default type: None	Default type: int (= long int)
If *type* is specified, the procedure is a function procedure, which may be called in an expression. The value returned is the type specified.	Functions normally return a value and may be called in expressions. The value returned may be any type except array or another function.
If *type* is omitted, the procedure does not return a value and cannot be used in expressions.	If the void type is specified, the function does not return a value and cannot be used in expressions.
A value is returned by assigning it to the *procedure-id* in the body of the procedure:	A value is returned in a return statement:

procedure-id := *expression*
 	
For example:
 	
`INTEGER PROCEDURE FUNC ;`
`   BEGIN`
`...`
`   FUNC := Y + Z ;`
`...`
`   RETURN ;`
`...`
`   FUNC := A - B ;`
`...`
`   END ;`
	return *expression* ;
	
	For easier conversion, declare a local variable, e.g., returnvalue, and replace the *procedure-id* with it in the function body. Then replace all the SPL RETURN statements with "return returnvalue" Add one before the final "}":
	
	`short int FUNC () ;`
	`   {`
	`   short int returnvalue ;`
	`...`
	`   returnvalue := Y + Z ;`
	`...`
	`   return returnvalue ;`
	`...`
	`   returnvalue := A - B ;`
	`...`
	`   return returnvalue ;`
	`   }`

Parameters

Table 8-4. Parameters

SPL	**HP C/XL Equivalent**
Formal parameters are defined by type (in the *formal-parm-decl* section) and by whether they are pass-by-reference (the default) or pass-by-value (named in the VALUE section).	Formal parameters are defined by type (in the *formal-parm-decl* section). All are pass-by-value.
Simple variable and pointer formal parameters may be pass-by-value or pass-by-reference. Array, procedure, and label formal parameters are pass-by-reference only. "Reference" formal parameters expect the address of the actual parameter. "Value" formal parameters expect the value of the actual parameter.	Simple variable formal parameters expect a value. Array, function, and pointer formal parameters expect a pointer value. Consequently, the operation for arrays and functions is functionally equivalent to SPL pass-by-reference. The operation for simple variables and pointers is functionally equivalent to SPL pass-by-value. (Labels cannot be passed.)
At the procedure call, if the formal parameter is pass-by-value, the value of the actual parameter is passed. If the formal parameter is pass-by-reference and the actual parameter is an appropriate identifier, array reference or pointer reference, the address of the actual parameter is passed; if the actual parameter is a constant or expression, then its value is passed as the address. It is possible to pass addresses in SPL as type LOGICAL or INTEGER parameters. Such operations must be examined carefully to determine the function performed in the original SPL code.	At the function call, the actual parameters are evaluated and converted to standard types. (See also "HP C/XL Rules for Automatic Numeric Type Conversion".) * [unsigned] char and short int become [unsigned] int (= [unsigned] long int) * float becomes double * array identifiers become "pointers to array of type T" * function identifiers become "pointers to function returning type T" * pointers are unchanged * structures are unchanged (they are copied into the function space) The conversions are based on the actual parameters, not on the corresponding formal

	parameters. The formal parameters "expect" the converted forms and reconvert them accordingly. HP C/XL does not check parameters.
Addresses are 16-bit pointers.	Addresses are 32-bit pointers.

The HP C/XL equivalent of a *formal* "reference" simple variable or pointer parameter is a pointer to simple variable or pointer to pointer, respectively. This amounts (mostly) to the addition of a leading "*" dereference operator everywhere the formal parameter is used in the function, in the form "**formal-parm* ".

The HP C/XL equivalent of an *actual* "reference" simple variable or pointer parameter is the address of the simple variable or pointer, respectively. The address is obtained with the "&" address operator, in the form "&*actual-parm* ".

Since array-ids (no subscript) are passed as pointers, they are implicitly pass-by-reference. That is, they may be passed as actual parameters and used as formal parameters without the "&" and "*" operators. If an array *cell* is passed by reference to an array or pointer formal parameter, it requires the "&" operator, as in "&*array-id* [*cell*]".

Options

Table 8-5. Options

SPL	**HP C/XL Equivalent**
CHECK *level* Specifies varying degrees of parameter checking for an external procedure.	No equivalent. HP C/XL performs no parameter checking.
EXTERNAL (Table 8-2, format 3a) Defines the name, type, and parameters of a procedure which exists external to the current program.	extern storage class (Table 8-2, format 3) The *formal-parm* list is omitted because HP C/XL performs no parameter checking on functions.
FORWARD (Table 8-2, format 3b) Specifies that the procedure will be declared fully later in the program. Allows a procedure to be called prior to its declaration.	extern storage class (Table 8-2, format 3) The function may be declared elsewhere in the same compilation unit or in a separate unit. If a function is not declared before it is called, its type defaults to "function returning int".
INTERNAL	No direct equivalent.

Prevents the procedure from being called from another segment. Generally used to keep the procedure-id local.	The static storage class provides similar functionality. A static function-id will not be exported to the linker, and therefore will be unknown to other compilation units.
INTERRUPT Specifies an external interrupt procedure. The purpose is highly hardware-dependent.	No equivalent.
PRIVILEGED Allows the procedure to execute in privileged mode.	No equivalent.
SPLIT Aids privileged users running in split-stack mode.	No equivalent.
UNCALLABLE Prevents the procedure from being called by code not executing in privileged mode.	No equivalent.
VARIABLE Lets the procedure be called with a varying number of actual parameters. The mechanism for determining how many actual parameters are passed uses Q-register addressing.	No direct equivalent. The HP C/XL library header file varargs.h contains macros that allow you to write functions with varying actual parameters. Insert the file in your program with the directive: `#include <varargs.h>` See the *HP C/XL Library Reference Manual* for details.

Local Declarations

Table 8-6. Local Declarations

SPL	HP C/XL Equivalent
All variables declared within a procedure are "local" to that procedure; they may not be referenced outside of the scope of the procedure.	Same as SPL.

Table 8-7 lists the three types of local variables in SPL, along with their HP C/XL equivalents.

Table 8-7. Local Variable Storage Classes

SPL	HP C/XL Equivalent
standard	[auto] (the default case)
OWN	static
EXTERNAL	extern

OWN Variables

Table 8-8. OWN Variables

SPL	HP C/XL Equivalent
Standard variables declared local to a procedure are assigned new space each time a procedure is invoked, the space being released when the procedure is exited.	Same as SPL, using auto variables (the default).
If a variable is declared as OWN, space is allocated outside of the dynamic scope of the procedure, in the DB-relative area. The variable is still known only to the procedure, and it retains its value between successive calls to the procedure. If an OWN variable is initialized, it is initialized once, at the start of the program, not every time the procedure is called.	Same as SPL, using static variables.

Local Simple Variable Declarations

Standard Local Variables.

Table 8-9. Standard Local Simple Variables

SPL	HP C/XL Equivalent
standard-local-simple-variable-declaration: *type* *variable-decl* [,...] ;	*simple-variable-declaration*: [*type*] *variable-decl* [,...] ;
variable-decl: 1a. *variable-id* 1b. *variable-id* := *initial-value* 2a. *variable-id* = *register* 2b. *variable-id* = *register sign offset* 3a. *variable-id* = *ref-id*	*variable-decl*: 1a. *variable-id* 1b. *variable-id* = *initial-value*

3b. *variable-id* = *ref-id sign offset*	
type is required.	Default type: int (= long int)
Storage is allocated each time the procedure is called. If an initial value is defined, it will be assigned each time the procedure is called.	Same as SPL.

Simple variables in forms 2 and 3 are usually various types of data equivalences. They may be converted to pointers or union equivalences, depending on the requirements of the program. See "ARRAY Declaration" for further examples.

OWN Simple Variables.

Table 8-10. OWN Local Simple Variables

SPL	**HP C/XL Equivalent**
own-simple-variable-declaration: OWN *type variable-decl* [,...] ;	*static-simple-variable-declaration*: static [*type*] *variable-decl* [,...] ;
variable-decl: 1a. *variable-id* 1b. *variable-id* := *initial-value*	*variable-decl*: 1a. *variable-id* 1b. *variable-id* = *initial-value*
type is required.	Default type: int (= long int)
An OWN local variable is allocated storage global to the procedure, in the DB-relative area. It retains its values between successive calls to the procedure. If an initial value is declared for an OWN variable, the variable is initialized once, at the start of the program, not every time the procedure is called.	Similar to SPL, using a static local variable.

EXTERNAL Simple Variables.

Table 8-11. EXTERNAL Local Simple Variables

SPL	HP C/XL Equivalent
external-simple-variable-declaration: EXTERNAL *type* *variable-id* [,...] ;	*extern-simple-variable-declaration*: extern [*type*] *variable-id* [,...] ;
type is required.	Default type: int (= long int)
An EXTERNAL local variable refers to a global variable that is declared GLOBAL in a separate compilation unit. The storage is allocated by the other unit.	Similar to SPL. An extern local variable refers to a global variable that is *not* declared static in a separate compilation unit. The storage is allocated by the unit that defines it.

See "Types of Declarations" for more detail.

Local Array Declarations

Standard Local Arrays.

Table 8-12. Standard Local Arrays

SPL	HP C/XL Equivalent
standard-local-array-declaration: [*type*] ARRAY [*local-array-decl* ,] [...] {*local-array-decl* *constant-array-decl* } ; *local-array-decl*: 1a. *array-id* (*lower* : *upper*) 1b. *array-id* (*lower* : *upper*) = Q 2. *array-id* (*var-lower* : *var-upper*) 3. *array-id* (@) = Q 4. *array-id* (*) = Q 5a. *array-id* (@) 5b. *array-id* (@) = *register sign offset* 6. *array-id* (*)	*array-declaration*: 1b with *lower* = 0. [*type*] *array-id* " [" *cells* "]" ; 1a; 1b with *lower* <> 0. [*type*] *array-ref* " [" *cells* "]" ; [*type*] * *array-id* = & *array-ref* " [" *index* "]" ; 10 with *lower* = 0. static [*type*] *array-id* " [" *cells* "]" *init:* 10 with *lower* <> 0. static [*type*] *array-ref* " [" *cells* "]"*init*; static [*type*] * *array-id*

```
|      7.  array-id  (*) = register sign offset|                                          |
|                                             |          = & array-ref  " [ "  index  " ] "  ; |
|      8a. array-id  (*) = ref-id              |                                          |
|                                             |                                          |
|      8b. array-id  (*) = ref-id  sign  offset| init:                                    |
|                                             |   = " { "  value  [,...]  " } "          |
|      9.  array-id  (*) = ref-id  ( index  ) |                                          |
|                                             |                                          |
|                                             | index:                                   |
| constant-array-decl:                        |   Cell number in array-ref  of cell that |
|                                             |   corresponds to cell zero in SPL array. |
|      10. array-id  ( lower  : upper  ) = PB |                                          |
|                                             | _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ |
|               := value-group  [,...]        |                                          |
|                                             |                                          |
|                                             | The other SPL forms establish an         |
| value-group:                                | equivalence relative to other declared data |
|                                             | (not just arrays).  Depending on their   |
|      {initial-value                         | actual use, they may be converted to HP  |
|       repeat-factor ( initial-value [,...] )}| C/XL pointer or union types, or #define |
|                                             | directives.  If their relationships are  |
|                                             | fairly simple, pointers can be used.     |
|                                             |                                          |
-------------------------------------------------------------------------------------------
|                                             |                                          |
| Default type:  LOGICAL                      | Default type:  int (= long int)          |
|                                             |                                          |
-------------------------------------------------------------------------------------------
```

The general rules for global array declarations also apply to local array declarations. See "ARRAY Declaration" for details and other conversion suggestions.

Standard arrays declared local to a procedure are allocated each time the procedure is called, and may not be referenced outside of the procedure.

Standard arrays (except for form 10) cannot be initialized.

Array form 10 is a special constant array declaration that is stored in the code segment and cannot be modified while the program is running. The suggested conversion to a static array (equivalent to an OWN array) should be effective. Care must be taken with subsequent code changes, since the converted static array can be modified by the program.

Summary of SPL Local Array Forms.

1a. Indirect; bounded; variable is pointer to cell zero; pointer in next Q-relative location; pointer IS allocated; array begins in next Q+ location; array IS allocated.

1b. Direct; bounded; variable is cell zero; *lower* in next Q+ location; array IS allocated.

2. Indirect; variable bounds; variable is pointer to cell zero; pointer IS allocated when procedure is called; array IS allocated when procedure is called.

3. Indirect; unbounded; variable is pointer to cell zero; pointer in next Q-relative location; pointer NOT allocated; array NOT allocated.

4. Direct; unbounded; variable is cell zero; cell zero in next Q-relative location; array NOT allocated.

5a. Indirect; unbounded; variable is pointer to cell zero; pointer in next Q-relative location; pointer IS allocated; array NOT allocated.

5b. Indirect; unbounded; variable is pointer to cell zero; pointer in specified DB-, Q-, or S-relative location; pointer NOT allocated; array NOT allocated.

6. Indirect; unbounded; variable is pointer to cell zero; pointer in next Q-relative location; pointer IS allocated; array NOT allocated.

7. Direct; unbounded; variable is cell zero; cell zero in specified DB-, Q-, or S-relative location; array NOT allocated.

8a. Direct (if *ref-id* is direct array or simple variable); unbounded; variable is cell zero; cell zero in specified location; array NOT allocated.

Indirect (if *ref-id* is pointer or indirect array); unbounded; variable is pointer to cell zero; cell zero in *ref-id* location; pointer in next Q-relative location IF one %id% type is BYTE and other is not; ELSE pointer location shared with *ref-id*; pointer IS allocated; array NOT allocated.

8b. Direct; unbounded; variable is cell zero; cell zero in specified location; array NOT allocated.

9. Direct (if *ref-id* is direct array); unbounded; variable is cell zero; cell zero in specified location; array NOT allocated.

Indirect (if *ref-id* is pointer or indirect array); unbounded; variable is pointer to cell zero; cell zero in specified location; pointer in next Q-relative location IF specified location is not *ref-id* cell zero OR IF one array is BYTE and other is not; ELSE pointer location shared with *ref-id*; pointer IS allocated; array NOT allocated.

Array forms 1a, 1b, 3, 4, 5a, 5b, 6, 7, 8a, 8b, and 9 correspond directly to global array forms 1a, 1b, 2a, 3a, 4a, 4b, 5, 6, 7a, 7b, and 8, respectively, except that they are Q-relative rather than DB-relative.

Array forms 3, 4, 5, 6, 7, 8, and 9 imply various methods of data equivalencing or "overlays".

Only array form 10 may be initialized.

Comparison of Specific Local Array Declarations. See also "ARRAY Declaration".

Array Format 2: Bounded Indirect Variable Array.

SPL	HP C/XL Equivalent
INTEGER ARRAY ABC (LOW'VAR : HIGH'VAR) This is an SPL "indirect" array with variable bounds. The bounds are evaluated each time the procedure is called, and storage is allocated accordingly.	No direct equivalent. Dynamic arrays are not allowed, but there are library routines, such as malloc, to allocate memory dynamically and assign an address to an array name. See the *HP C/XL Library Reference Manual* for details.

Array Format 10: Bounded Direct Constant Array.

SPL	HP C/XL Equivalent
REAL ARRAY ABC(0:9) = PB := 1,2,3,4,5,6,7,8,9,10; This is a "constant" array. It is initialized in the code segment and cannot not be modified.	static float ABC [10] = {1,2,3,4,5,6,7,8,9,10}; This is not an exact equivalent. There is no protection against inadvertent modification.

OWN Local Arrays.

Table 8-13. OWN Local Arrays

SPL	HP C/XL Equivalent
own-array-declaration: OWN [*type*] ARRAY [*own-array-decl* ,] [...] *init-own-array-decl* ; *own-array-decl*: 1. *array-id* (*lower* : *upper*) *init-own-array-decl*: 2. *array-id* (*lower* : *upper*) := *value-group* [,...] *value-group*: {*initial-value* *repeat-factor* (*initial-value* [,...])}	*static-array-declaration*: 1 with *lower* = 0. static [*type*] *array-id* " [" *cells* "]" ; 1 with *lower* <> 0. static [*type*] *array-ref* " [" *cells* "]" static [*type*] * *array-id* = & *array-ref* " [" *index* "]" ; 2 with *lower* = 0. static [*type*] *array-id* " [" *cells* "]" *init* ; 2 with *lower* <> 0. static [*type*] *array-ref* " [" *cells*"]"*init*:

	static [*type*] * *array-id* = & *array-ref* " [" *index* "]" ; *init*: = " {" *value* [,...] " }" *index*: Cell number in *array-ref* of cell that corresponds to cell zero in SPL array.
Default type: LOGICAL	Default type: int (= long int)
An OWN local array is allocated storage global to the procedure, in the DB-relative area. It retains its values between successive calls to the procedure. If an initial value is declared for an OWN array, the variable is initialized once, at the start of the program, not every time the procedure is called.	Same as SPL, using a static local array.

EXTERNAL Local Arrays.

Table 8-14. EXTERNAL Local Arrays

SPL	HP C/XL Equivalent
external-array-declaration: EXTERNAL [*type*] ARRAY {*array-id* {(*) (@)}} [,...] ; (*) signifies a direct array. (@) signifies an indirect array.	*extern-array-declaration*: Direct with *lower* = 0. extern [*type*] *array-id* " [" "]" ; Indirect, or direct with *lower* <> 0. extern [*type*] * *array-id*
Default type: LOGICAL	Default type: int (= long int)
An EXTERNAL local array refers to a global array that is declared GLOBAL in a separate compilation unit. The storage is allocated by the other unit.	Similar to SPL. An extern local array refers to a global array that is *not* declared static in a separate compilation unit. The storage is allocated by the other unit.

See "Types of Declarations" for further details.

Local Pointer Declarations

See "POINTER Declaration" for further details.

Standard Local Pointers.

Table 8-15. Standard Local Pointers

SPL	**HP C/XL Equivalent**
standard-local-pointer-declaration: [*type*] POINTER *ptr-decl* [,...] ; *ptr-decl*: 1a. *ptr-id* 1b. *ptr-id* := @*ref-id* 1c. *ptr-id* := @*ref-id* (*index*) 2a. *ptr-id* = *ref-id* 2b. *ptr-id* = *ref-id* *sign* *offset* 3a. *ptr-id* = *register* 3b. *ptr-id* = *register* *sign* *offset* 4. *ptr-id* = *offset*	*pointer-declaration*: [*type*] *ptr-decl* [,...] ; *ptr-decl*: 1a. * *ptr-id* 1ba.* *ptr-id* = *ref-id* 1bv.* *ptr-id* = & *ref-id* 1c.* *ptr-id* = & *ref-id* " [" *index* "]" 1ba: *ref-id* is an array or pointer id. 1bv: *ref-id* is a simple variable.
Default type: LOGICAL	Default type: int (= long int)
Pointers are 16-bit values containing DB-relative addresses.	Pointers are 32-bit values containing standard MPE XL addresses. Overlays of pointers and other data types must be recoded.

OWN Local Pointers.

Table 8-16. OWN Local Pointers

SPL	**HP C/XL Equivalent**
own-local-pointer: OWN [*type*] POINTER *ptr-decl* [,...] ; *ptr-decl*: *ptr-id*	*static-local-pointer*: static [*type*] *ptr-decl* [,...] ; *ptr-decl*: * *ptr-id*

Default type: LOGICAL	Default type: int (= long int)
An OWN local pointer is allocated storage global to the procedure, in the DB-relative area. It retains its values between successive calls to the procedure.	Same as SPL, using a static local array.
OWN pointers cannot be initialized.	static pointers may be initialized, using the syntax given for forms 1b and 1c in "POINTER Declaration". The pointer is initialized once, at the start of the program, not every time the function is called.

EXTERNAL Local Pointers.

Table 8-17. EXTERNAL Local Pointers

SPL	**HP C/XL Equivalent**
external-local-pointer: EXTERNAL *type* *ptr-id* [,...] ;	*extern-local-pointer*: extern *type* *ptr-id* [,...] ;
Default type: LOGICAL	Default type: int (= long int)
An EXTERNAL local pointer refers to a global pointer that is declared GLOBAL in a separate compilation unit. The storage is allocated by the other unit.	Similar to SPL. An extern local pointer refers to a global pointer that is *not* declared static in a separate compilation unit. The storage is allocated by the other unit. See "Types of Declarations".

Local LABEL Declarations

See "LABEL Declaration" for further details.

Table 8-18. Local LABEL Declaration

SPL	**HP C/XL Equivalent**
label-declaration: LABEL *label-id* [,...] ;	No equivalent.
Declaration of labels is optional.	Labels are not declared.

	Remove the SPL label declarations.
The scope of a local label is the procedure.	Same as SPL.

Local SWITCH Declarations

See "SWITCH Declaration" for further details.

Table 8-19. Local SWITCH Declaration

SPL	**HP C/XL Equivalent**
switch-declaration: SWITCH *switch-id* := *label-id0* [,...] ;	*define-directive*: #define *switch-id* (X) \\ switch (X) \\ " {" \\ case 0: goto *label-id0*; \\ case 1: goto *label-id1*; \\ [...] " }" ... #undef *switch-id*
switch-reference: GOTO *switch-id* (*index*)	*define-reference*: *switch-id* (*index*)
The scope of a local SWITCH declaration is the procedure.	The scope of a #define directive is not local. It is known to all following source code. To turn it off, insert the #undef directive at the end of the function.

Local ENTRY Declaration

Table 8-20. Local ENTRY Declaration

SPL	**HP C/XL Equivalent**
entry-declaration: ENTRY *label-id* [,...] ;	No direct equivalent.

You may emulate multiple entry points into an SPL procedure by adding a parameter to the HP C/XL function, and coding a switch statement in the function to goto the appropriate labels based on the value of the parameter. See "Local SWITCH Declarations" above for the format.

Entry point identifiers used in calling routines must be changed to the procedure identifier. Alternatively, global #define directives could be used to equate the entry point identifiers with the procedure identifier.

You might also create #define macro directives with different names, each of which calls the original function with the index parameter supplied as a constant.

Or you might rewrite the procedure as several HP C/XL functions named by the entry point identifiers.

Local DEFINE Declaration and Reference

See "DEFINE Declaration and Reference" for further details.

Table 8-21. DEFINE Declaration and Reference

SPL	HP C/XL Equivalent
define-declaration: DEFINE {*define-id* = *text* #} [,...] ;	*define-directive*: #define *define-id* *text* ... #undef *define-id*
The scope of a local DEFINE declaration is the procedure.	The scope of a #define directive is not local. It is known to all following source code. To turn it off, insert the #undef directive at the end of the function.

Local EQUATE Declaration and Reference

See "EQUATE Declaration and Reference" for further details.

Table 8-22. Local EQUATE Declaration and Reference

SPL	HP C/XL Equivalent
equate-declaration: EQUATE {*equateid* =*equate-expr* }[,...];	*define-directive*: #define *equateid* *equate-expr* ...

	#undef *equate-id*
The scope of a local EQUATE declaration is the procedure.	The scope of a #define directive is not local. It is known to all following source code. To turn it off, insert the #undef directive at the end of the function.

Procedure Body

See syntax for *procedure-body* and *function-body* in Table 8-2.

Table 8-23. Procedure Body

SPL	HP C/XL Equivalent
Contains the local declarations and statements of the procedure.	Same as SPL.
The end of the body generates an exit instruction. Additional exit points may be specified with the RETURN statement.	Same as SPL. Additional exit points may be specified with the return statement.

See also "RETURN Statement" and "Data Type" above.

INTRINSIC Declarations

Table 8-24. INTRINSIC Declarations

SPL	HP C/XL Equivalent
intrinsic-declaration: INTRINSIC [(*file*)] *intrinsic-id* [,...]	*pragma-directive*: 1. #pragma intrinsic {*intrinsic-id* [*user-id*]} [,...] 2. #pragma intrinsic_file "*file* " 3. #pragma intrinsic_file ""
Without *file*, the *intrinsic-id* is sought in the system intrinsic file. If *file* is given, the *intrinsic-id* is sought in the user-defined file *file*.	Similar to SPL. The intrinsic_file pragma establishes the intrinsic file where all subsequent intrinsic pragmas will search. If intrinsic_file is not given, or if form 3 is used, intrinsic definitions are sought in the file SYSINTR.PUB.SYS.

	The *user-id* option allows you to rename the *intrinsic-id* for references in your compilation unit.

Table 8-25. INTRINSIC Declaration Examples

SPL	HP C/XL Equivalent
INTRINSIC FREADDIR; INTRINSIC (MYINTRS) MYFREAD;	#pragma intrinsic FREADDIR #pragma intrinsic_file "MYINTRS" #pragma intrinsic MYFREAD #pragma intrinsic_file ""
The first declaration seeks the intrinsic FREADDIR in the system intrinsic library. The second seeks MYFREAD in the user library named MYINTRS.	The first pragma directive seeks the intrinsic FREADDIR in the system intrinsic library SYSINTR.PUB.SYS. The second redirects subsequent searches to the user intrinsic file named MYINTRS. The third seeks the intrinsic MYFREAD in MYINTRS. The fourth resets the search to the system file SYSINTR.PUB.SYS.

The HP C/XL pragmas are described further in the *HP C/XL Reference Manual Supplement*. The construction of user intrinsic files is discussed in the *HP Pascal Programmer's Guide*.

SUBROUTINE Declaration

Table 8-26. SUBROUTINE Declaration

SPL	HP C/XL Equivalent
subroutine-declaration: 1. [*type*] SUBROUTINE *subroutine-id* [VALUE *formal-parm* [,...]] *formal-parm-decl* [;...] ; *statement* ; 2. [*type*] SUBROUTINE *subroutine-id* *statement* ; *statement* may be any SPL statement, including compound (BEGIN-END).	Global. *function-definition*: 1. static [*type* void] *function-id* (*formal-parm* [,...]) *formal-parm-decl* [;...] ; *"* {*"* *statement* *"* }*"* 2. static [*type* void] *function-id* ( ) *"* {*"* *statement* *"* }*"* Local. *define-directive*: 1. #define *function-id* (*formal-parm* [,...]) *statement-process*

	: #undef *function-id* 2. #define *function-id statement-process* : #undef *function-id*
A subroutine is like a procedure that has no option or local declaration sections.	No direct equivalent.
Declared at the global level, it is available only to the main body of the compilation unit. It may access global identifiers.	A static function is the closest global equivalent. It is available to all functions in the compilation unit. It may access global identifiers.
Declared at the local level, it is available only to the procedure body where it is declared. It may access global and local identifiers.	The "best" local equivalent is a #define macro directive, which will be expanded inline wherever it is called. See "DEFINE Declaration and Reference" for the #define syntax rules. Note that there is no control whatsoever over the data types of the macro's formal and actual parameters. The alternate solution is a static function at the global level. This can be awkward because the local procedure variables that were known to the subroutine are no longer automatically available. You could change the local variables to global, or pass them as parameters to the new function.

Table 8-27. SUBROUTINE Declaration Example

SPL Subroutine	HP C/XL Function	HP C/XL define Directive

INTEGER SUBROUTINE A(B,C);
VALUE B,C;
INTEGER B,C;
A := B+C;
	static short int A(B,C);
	short int B,C;
	{
	return B+C;
	}
		#define A(B,C) ((B)+(C))
		...
		#undef A

This example shows the conversion of an SPL subroutine to an HP C/XL function and a #define macro directive. In the #define directive version, the parentheses around B and C and the summation are necessary to ensure correct evaluation of the parameters when the substitutions for B and C are expressions.		

Careful examination of an SPL procedure may reveal that local variables have been declared in the procedure for the sole purpose of providing them to the subroutine. In that case, the variable declarations may simply be moved to the new function.

It is permitted (but rarely used) to execute a GOTO statement from an SPL subroutine to a label within the body of the enclosing procedure. HP C/XL restricts the goto statement to labels within the same function declaration.

Chapter 9 Input/Output

This chapter discusses conversion issues that correspond to sections in Chapter 8 of the *Systems Programming Language Reference Manual*.

Introduction to Input/Output

SPL has no input/output (I/O) statements; instead, it uses MPE V intrinsics to perform all I/O operations.

Similarly, HP C/XL has no I/O statements; it does have its own library header file, <stdio.h>, that provides a comprehensive set of macros and functions for I/O capabilities, including high level formatting. HP C/XL also has a special library header file, <mpe.h>, that provides an interface to the MPE XL I/O intrinsic library. This arrangement allows HP C/XL programmers to choose either HP C/XL I/O functions and macros, MPE XL I/O intrinsics, or a combination of both.

In general, the MPE XL I/O intrinsics are identical to or extensions of the MPE V versions. The differences are described in the *Introduction to MPE XL for MPE V Programmers* migration guide. Consult the *MPE XL Intrinsics Reference Manual* | for the complete specification of all MPE XL intrinsics.

There are strong arguments in favor of adopting the HP C/XL style of I/O operations. Programmer convenience and program portability are high on the list. Programs that use HP C/XL library functions can usually be transferred to HP C/HP-UX with little or no modification. The source code changes that are required anyway to provide parameters to the MPE XL intrinsics can just as easily be revised to use HP C/XL library functions instead.

It is recommended that SPL programs being translated into HP C/XL adopt as many of the HP C/XL I/O facilities as possible. Where there are necessary operations that cannot be performed by the HP C/XL standard library header file, <stdio.h>, MPE XL intrinsics may be declared with the #pragma intrinsic directive and called directly.

CAUTION You cannot use the HP C/XL I/O system and another I/O system concurrently to write data to the same disk file (except for the stdout and stderr file streams). Please consult the *HP C/XL Library Reference Manual* for details.

Example

Since all I/O operations by MPE V intrinsics use 16-bit data, it is common to equivalence a BYTE array to a previously declared LOGICAL word array. Then data is stored into or extracted from the byte array, while the equivalent word array is passed to the MPE V intrinsics.

As an example of the convenience of the HP C/XL constructs, consider the following SPL program fragment and the identical operation in HP C/XL:

SPL	HP C/XL Equivalent
LOGICAL ARRAY BUFW(0:40); <<equate byte to word array>> BYTE ARRAY BUF(*)=BUFW; INTEGER X; INTRINSIC PRINT, ASCII; <<move 18 bytes>> MOVE BUF:="value of X = "; <<convert to ASCII>> ASCII(X,10,BUF(13)); <<output word array copy>> PRINT(BUFW,9,0);	short int X; printf("value of X = %d\n",X);

Record Format

The "normal" SPL file has fixed-length records, although files with variable length records can be created and used. The "normal" HP C/XL file, called a "stream", has variable-length records; files with fixed-length records can be created and used.

File References

There are *three* distinct variables that specify a file, depending on which HP C/XL function or MPE intrinsic opened it. These variables are used to identify the file access to other functions or intrinsics. The HP C/XL function open returns *filedes*, an int file descriptor; the HP C/XL function fopen returns *stream*, a pointer to type FILE; and the MPE intrinsic FOPEN returns *filenum*, a 16-bit integer file number. The MPE XL intrinsic HPFOPEN also returns *filenum*, but as a 32-bit integer file number, equal to the FOPEN value.

Fortunately, there is a relationship among them. A *stream* file pointer can be obtained from a *fildes* file descriptor with the HP C/XL <stdio.h> library function fdopen:

```
#include <stdio.h>
stream = fdopen(fildes )
```

An MPE *filenum* can be obtained from *fildes* with the HP C/XL <mpe.h> library function _mpe_fileno:

```
#include <mpe.h>
filenum = _mpe_fileno(filedes )
```

See the *HP C/XL Library Reference Manual* for more details.

Conflicting Function and Intrinsic Identifiers

Five of the functions in the HP C/XL standard library have the same names as MPE intrinsics: fopen, fclose, fread, fwrite, and read. If any of the MPE intrinsics of the same name are used, it is recommended that you rename them with the #pragma intrinsic directive to avoid confusion. For instance:

```
#pragma intrinsic FREAD MPE_FREAD
```

Although case sensitivity would render FREAD distinct from fread, the use of MPE_FREAD is much more descriptive. It's probably a good idea to apply the same renaming scheme to all the MPE intrinsics your program uses, just to make them easier to find.

Error Reporting

The MPE intrinsics vary in how errors are reported. Some return an error value for a function value or parameter, but most have a side effect of setting the condition code. The HP C/XL library function ccode returns the most recent setting of the condition code.

The HP C/XL I/O functions report an error by returning an error value, and sometimes by setting an external variable errno. The value of errno will indicate the error which caused the most recent intrinsic or library function error. Its value is not changed or reset until the next instance of an error, so errno should not be interrogated unless a function that sets it reports an error.

Summary of Intrinsics, Macros, and Functions

Table 9-1 lists the MPE XL I/O intrinsics. Note that all but HPFOPEN are equivalent to the MPE V versions. HPFOPEN is only available in MPE XL. It has clearer ways of passing parameters than FOPEN, as well as having more options. See the *MPE XL Intrinsics Reference Manual* for details.

Table 9-1. MPE XL I/O Intrinsics

Intrinsic	**Description**
FCHECK(*filenum*,*fserr*,*translog*,*block*,*nrec*)	Get details on I/O errors
FCLOSE(*filenum*,*disp*,*seccode*)	Close file
FCONTROL(*filenum*,*controlcode*,*param*)	Perform control operation on file or terminal
FOPEN(*formdesig*,*foptions*,*aoptions*,...)	Open file; return *filenum*, 16-bit file number
FREAD(*filenum*,*buffer*,*length*)	Read logical record from sequential file; return count
FREADDIR(*filenum*,*buffer*,*length*,*lrecnum*)	Read logical record from direct access file
FSPACE(*filenum*,*disp*)	Space forward or backward on file

FUPDATE(*filenum*,*buffer*,*length*)	Update logical record in file
FWRITE(*filenum*,*buffer*,*length*,*ctlcode*)	Write logical record to sequential file
FWRITEDIR(*filenum*,*buffer*,*length*,*lrecnum*)	Write logical record to direct access file
HPFOPEN(*filenum*,*status* [,*itemnum*,*item*][...])	Open file; return *filenum*, 32-bit file number
PRINT(*message*,*length*,*ctlcode*)	Write string to $STDLIST
READ(*message*,*expectedlength*)	Read string from $STDIN; return actual length
READX(*message*,*expectedlength*)	Read string from $STDINX; return actual length

Table 9-2 and Table 9-3 describe briefly the HP C/XL standard library I/O macros and functions that you may wish to use in converting your SPL programs. See the *HP C/XL Library Reference Manual* for details.

Table 9-2. HP C/XL I/O Macros

Macro	Description
getc(*stream*)	Read one character from file *stream*
getchar()	Read one character from stdin
putc(*c*,*stream*)	Write one character *c* to file *stream*
putchar(*c*)	Write one character *c* to stdout

Table 9-3. HP C/XL I/O Functions

Function	Description
access(*filename*,*access*)	Test accessibility of file
clearerr(*stream*)	Clear error and eof conditions on file *stream*
close(*fildes*)	Close file *fildes*
dup(*fildes*)	Duplicate file descriptor *fildes*
fclose(*stream*)	Close file *stream*; flush buffer
fdopen(*fildes*)	Get *stream* pointer from *fildes* file descriptor
feof(*stream*)	Test file *stream* for end-of-file
ferror(*stream*)	Test file *stream* for error
fflush(*stream*)	Flush buffer to file *stream*
fgetc(*stream*)	Read one character from file *stream*
fgets(*string*,*n*,*stream*)	Read *n* -1 chars from file *stream* (or up to '\n')
fopen(*filename*,*type*)	Open file *filename*; return *stream* (pointer to FILE)
fprintf(*stream*,*format* [,*item*][...])	Convert from internal *item*; write to file *stream*
fputc(*c*,*stream*)	Write one character *c* to file *stream*
fputs(*string*,*stream*)	Write *string* (up to '\0') to file *stream*
fread(*ptr*,*size*,*nitems*,*stream*)	Read fixed-length binary records from file *stream*
freopen(*filename*,*type*,*stream*)	Change file attached to *stream*
fscanf(*stream*,*format* [,*item*][...])	Read from *stream*; convert to internal *item*
fseek(*stream*,*offset*,*ptrname*)	Set byte position in file *stream*
ftell(*stream*)	Return byte position of file *stream*
fwrite(*ptr*,*size*,*nitems*,*stream*)	Write fixed-length binary records to file *stream*
gets(*string*)	Read *string* from stdin
getw(*stream*)	Read int word from file *stream*
lseek(*fildes*,*offset*,*ptrname*)	Set byte position in file *fildes*
open(*filename*,*oflag*,*mode*,*mpeopts*)	Open file *filename*; return *fildes* (int file descriptor)
printf(*format* [,*item*][...])	Convert from internal *item*; write to stdout
puts(*string*)	Write *string* (up to '\0') to stdout
putw(*word*,*stream*)	Write int *word* to file *stream*

read(*fildes*,*buf*,*nbyte*)	Read fixed-length binary records from file *fildes*
remove(*filename*)	Purge file *filename*
rename(*oldname*,*newname*)	Rename file
rewind(*stream*)	Reset byte position to beginning of file *stream*
scanf(*format* [,*item*][...])	Read from stdin; convert to internal *item*
setbuf(*stream*,*buffer*)	Define buffer for file *stream*
setvbuf(*stream*,*buffer*,*type*,*size*)	Define buffer for file *stream*
sprintf(*string*,*format* [,*item*][...])	Convert from internal *item*; write to *string*
sscanf(*string*,*format* [,*item*][...])	Read from *string*; convert to internal *item*
tmpfile()	Open unnamed tempfile
tmpnam(*string*)	Create temp *filename* in *string*
ungetc(*c*,*stream*)	Push back character *c* to input file *stream*
unlink(*filename*)	Purge file *filename*
write(*fildes*,*buf*,*nbyte*)	Write fixed-length binary records to file *fildes*

Opening a New Disk File

SPL uses the MPE V intrinsic FOPEN to create and open disk files. FOPEN allows SPL to have complete control over the definition of a new file. It returns a file number, *filenum*, which is used to identify the access to this file for subsequent I/O operations by other intrinsics, such as FREAD and FWRITE. If an error occurs, FOPEN returns zero and sets the condition code to CCL.

In HP C/XL, a file may be created and opened with the library functions open and fopen or with the MPE XL intrinsics FOPEN and HPFOPEN.

The most preferred and portable is HP C/XL fopen, which returns a *stream* pointer that is used by all of the HP C/XL standard data formatting and character transfer functions, such as fscanf and fprintf. If fopen fails, it returns a null pointer.

If you need the HP C/XL binary read and write functions, or more file creation control, the open function provides more system-specific capabilities, which make it less portable. open returns *fildes*, a 32-bit int file descriptor, which may be used to obtain both a *stream* pointer and a *filenum* file number.

Reading a File in Sequential Order

SPL uses the FREAD intrinsic to read all or part of a record in a sequential file with fixed- or variable-length records. A logical record pointer points to the next record to be read. When any part of a record is read, the pointer advances to the next record.

HP C/XL has several ways to read records. The fgets function is probably the closest to the SPL action: a requested number of bytes or all the characters up to the end of the record are read into a buffer. If the end of record was reached, HP C/XL marks it by appending a '\n' (linefeed) character to the record data in the buffer. In any case, HP C/XL appends a '\0' (NUL) character to mark the end of the data in the buffer.

Alternatively, the HP C/XL fread function can be used to read fixed-length binary data into a structure, such as an array or struct. Since fread does not recognize file record boundaries, you need to be sure the sizes you supply add up correctly.

The conventional means of performing I/O in HP C/XL is to view data files as a "stream" of text (ASCII characters) in variable-length records.

The functions contained in the HP C/XL library provide a rich set of formatted I/O operations, greatly simplifying the multiple steps which are necessary in SPL. Consideration of fixed-length record operations is required only to read files created in this manner by other programs, or to create files for programs that expect to read fixed-length records.

Writing Records into a File in Sequential Order

SPL uses the FWRITE intrinsic to write all or part of a record in a sequential file with fixed- or variable-length records. A logical record pointer points to the next record to be written. When any part of a record is written, the pointer advances to the next record.

The HP C/XL functions provide many choices for output. You may use fputs, fwrite, or write to emulate the SPL record structure. Or you may use the printf and fprintf to write formatted, variable-length text records. With the latter two, you must identify the end of each record to HP C/XL by writing a '\n' (linefeed) character. The '\n' is not actually stored in the file.

Updating a File

SPL uses the MPE V intrinsic FUPDATE to replace the record last accessed in the file by any intrinsic. This is commonly used to update part of a record that has been located by some identifying data in the same record. The records cannot be variable-length.

HP C/XL has no direct equivalent of the FUPDATE intrinsic. You will have to emulate it by using the HP C/XL function ftell to give you the record start byte *before* you read it. Then you can reposition the file with the fseek function and rewrite the record with fputs, fwrite, or write. You can even write records with fprintf and the rest as long as you remember to write a '\n' character, signaling end-of-record to HP C/XL .

Numeric Data Input/Output

SPL programs use four MPE V intrinsics to convert ASCII data to and from binary format. They are:

ASCII	Converts 16-bit binary number to ASCII.
DASCII	Converts 32-bit binary number to ASCII.
BINARY	Converts ASCII byte string to 16-bit binary.
DBINARY	Converts ASCII byte string to 32-bit binary.

Converting floating-point numbers requires other intrinsics.

Calls to these intrinsics have to be combined with building byte strings "by hand", equating them to word arrays, and then passing these to MPE V I/O intrinsics. (See the example above in "Introduction to Input/Output").

HP C/XL standard library functions perform these operations as an extension to the normal I/O operations.

sscanf	Converts ASCII string data into all the binary formats: signed and unsigned long and short int, char and unsigned char, float, and double.
sprintf	Converts all the binary forms above into their text character representations, combining them with string variables and constants.

Both of these functions allow complete format conversions. They are considerably more powerful than the MPE intrinsic equivalents.

Four HP C/XL library functions perform both the physical I/O operation *and* the format conversions at the same time. They are:

scanf	Reads text from the standard input file, stdin, and converts the ASCII text into binary numeric variables, character variables, and strings under the control of a format specification.
fscanf	Does the same as scanf, but reads text from a specified *stream* file.
printf	Converts binary numeric values, character values, and string values into ASCII text under the control of a format specification, and writes the text to the standard output file, stdout.
fprintf	Does the same as printf, but write the text to a specified *stream* file.

The simplicity and flexibility of these routines render the direct use of the MPE intrinsics a highly questionable option. The use of standard HP C/XL library functions in general will greatly improve portability to another operating system such as HP-UX.

File Equations

Standard attributes of a file used by an HP C/XL program may be modified through the use of MPE XL :FILE commands, just as for SPL.

An additional feature available to HP C/XL is the redirection of the standard (default) input and output files. This is accomplished by supplying alternate MPE XL file names in the INFO= string of the MPE XL :RUN command. For example:

```
RUN HPCPROG; INFO="<myinput >myoutput"
```

The "<" parameter causes all standard input operations--the MPE XL intrinsics such as READ and READX and the HP C/XL functions such as scanf and getc--to access the file MYINPUT instead of $STDIN. Likewise, the ">" parameter causes any standard output to be directed to the file MY-OUTPUT instead of $STDLIST. For more information, see the *HP C/XL Reference Manual Supplement*.

Chapter 10 Compiler and MPE Commands

This chapter discusses conversion issues that correspond to sections in Chapter 9 and 10 of the *Systems Programming Language Reference Manual*.

Compiler Format

The compiler listing format for HP C/XL is different from SPL's. For complete information about the HP C/XL compiler, refer to the *HP C Reference Manual* and the *HP C/XL Reference Manual Supplement*.

Use and Format of Compiler Commands

Table 10-1. Compiler Command Format

SPL	HP C/XL Equivalent
compiler-command: $*command-name* [*parameter*][,...] $$*command-name* [*parameter*][,...]	*compiler-directive*: #*directive-name* [*parameter*][...]
The "$" must be in column 1. The "$$" form has no HP C/XL equivalent. Ignore the second "$".	The "#" must be in column 1.
If a compiler command must be continued on subsequent lines, each continued line ends with "&" and the following line begins with "$" in column 1.	If a directive must be continued on subsequent lines, each continued line ends with "\".
The command may contain comments, enclosed in double angle brackets, "<<*comment* >>".	The directive cannot include comments. They become part of the text.

Some of the SPL compiler commands are paralleled in HP C/XL as compiler options that are specified in the MPE XL :RUN command used to invoke the HP C/XL compiler.

See the *HP C/XL Reference Manual Supplement* for further details.

$CONTROL Command

The SPL $CONTROL command has 22 options. Table 10-2 describes the available equivalent HP C/XL directives or compiler options.

Table 10-2. $CONTROL Commands

SPL	HP C/XL Equivalent
$CONTROL LIST	#pragma LIST ON (default)
$CONTROL NOLIST	#pragma LIST OFF
$CONTROL SOURCE	(no equivalent; listing is on by default)
$CONTROL NOSOURCE	(no equivalent; must direct output to $NULL)
$CONTROL WARN	+w*n* compiler option (default) 1
$CONTROL NOWARN	-w compiler option 1
$CONTROL MAP	+m and +o compiler options 1
$CONTROL NOMAP	(default)
$CONTROL AUTOPAGE	#pragma AUTOPAGE ON|OFF
$CONTROL CODE	(no equivalent)
$CONTROL NOCODE	(no equivalent)
$CONTROL LINES=*nnnn*	#pragma LINES *nnn*
$CONTROL ERRORS=*nnn*	(no equivalent)
$CONTROL USLINIT	(no equivalent)
$CONTROL DEFINE	+H*n* compiler option 1
$CONTROL SEGMENT=*segname*	(no equivalent)
$CONTROL ADR	+m and +o compiler options 1
$CONTROL INNERLIST	(no equivalent)
$CONTROL MAIN=*name*	(no equivalent)
$CONTROL UNCALLABLE	(no equivalent)
$CONTROL PRIVILEGED	(no equivalent)
$CONTROL SUBPROGRAM	(implied by the absence of a main function)

The *HP C/XL Reference Manual Supplement* describes the compiler options and #pragma directives listed above, as well as others not available to SPL.

----

1 Where an SPL $CONTROL compiler command is replaced by an HP C/XL compiler option, please be aware that the compiler option applies to all of the source being compiled at the same time. These compiler options are convenient because the source remains unmodified, but you do lose the line-by-line toggling of the SPL compiler command.

$IF Command (Conditional Compilation)

Table 10-3. $IF Command

SPL	HP C/XL Equivalent

if-command:
 	
`$IF [Xn ={OFF`
`ON}]`
	if-directive:
	
	`#if` *constant-expression*
	`...`
	`[#else`
	`...]`
	`#endif`

SPL predefines ten switches named X0,...,X9, whose initial values are OFF. The switches may be changed with the $SET command (see below) and tested with the $IF command.	The SPL switches may be emulated by defining them equal to zero in #define directives.
	
	`#define OFF 0`
	`#define ON 1`
	`#define X0 OFF`
	`...`

When a $IF command is executed, and the switch test is true (or the test is omitted), then all the following source lines are compiled, down to the next $IF command. If the test is false, the same source lines are skipped. Note that there is no form of "else" except a $IF with the opposite test. A $IF with no parameter serves to end the conditional block.	When an #if directive is executed, and the expression is true (nonzero), then all the following source lines are compiled, down to the next #else or #endif directive. If the test is false (zero), the same source lines are skipped. The #else directive marks the start of a block of lines that are compiled only if the test is false. They are skipped if the test is true. The #else block is terminated by an #endif directive.

For example, X3 is used to control a choice between two DEFINE declarations:
 	
`$IF X3 = ON`
`DEFINE GLOBALVAL = 99#;`
`$IF X3 = OFF`
`DEFINE GLOBALVAL = 101#;`
`$IF`
	Assuming appropriate initialization, as above, the corresponding example in HP C/XL could be coded as:
	
	`#if X3 == ON`
	`#define GLOBALVAL 99`
	`#else`
	`#define GLOBALVAL 101`

	#endif
In a case where two switches are used in series, you might see: $IF X3 = ON DEFINE GLOBALVAL = 99#; $IF X5 = OFF DEFINE GLOBALVAL = 101#; $IF	This is rendered in HP C/XL as: #if X3 == ON #define GLOBALVAL 99 #endif #if X5 == OFF #define GLOBALVAL 101 #endif

The conditional compilation facility in HP C/XL is considerably more powerful than that available in SPL. Instead of ten fixed switches, you can define arbitrary names as defined variables, and can test an expression composed of these variables and constants.

The directives "#ifdef *id* " and "#ifndef *id* " are also available to test whether or not an identifier, *id*, has been defined with a #define directive. You can use #ifdef and #ifndef in place of #if. See the *HP C Reference Manual* for more information.

$SET Command (Software Switches for Conditional Compilation)

Table 10-4. $SET Command

SPL	**HP C/XL Equivalent**
set-command: $SET [X*n* ={OFF ON}][,...]	Emulated with the #define directive: #define X*n* {OFF ON} [...]
The ten switches, X0,...,X9, used for conditional compilation are initially set to OFF. They are turned ON and OFF with the $SET command.	The #define directive emulates the $SET command. The syntax above assumes that you have defined the ten switches and the values ON and OFF at the beginning of the compilation unit, as follows: #define ON 1 #define OFF 0 #define X0 OFF ... #define X9 OFF

$TITLE Command (Page Title in Standard Listing)

Table 10-5. $TITLE Command

SPL	HP C/XL Equivalent
title-command: $TITLE ["*title-string* " [,...]]	*title-pragma*: #pragma TITLE "*title-string* "
The combined strings become the title on subsequent listing pages. $TITLE with no strings turns off the title.	The string becomes the title on subsequent listing pages. To turn off the title, use an empty string.

$PAGE Command (Page Title And Ejection)

Table 10-6. $PAGE Command

SPL	HP C/XL Equivalent
page-command: $PAGE ["*title-string* " [,...]]	*page-pragma*: [#pragma TITLE "*title-string* "] #pragma PAGE
A new listing page is started. The combined strings become the title on the new page. If the strings are omitted, the previous title is retained.	A new listing page is started. You may change the title on the new page by preceding #pragma PAGE with a #pragma TITLE directive.

$EDIT Command (Source Text Merging and Editing)

The SPL process of merging text files, checking sequence fields, and editing text files has no equivalent in HP C/XL.

$SPLIT/$NOSPLIT Commands

The toggled version of the SPL procedure option OPTION SPLIT has no equivalent in HP C/XL.

$COPYRIGHT Command

Table 10-7. $COPYRIGHT Command

SPL	**HP C/XL Equivalent**
copyright-command: $COPYRIGHT "*string* "[,...]	*copyright-pragma*: #pragma COPYRIGHT "*string* "
The combined strings are written to the object module and the compiled program as a copyright notice.	A predefined copyright notice is written to the object module and the compiled program, using *string* as the company name.
SPL allows the data to be split over lines by having separate strings delimited by quotes and separated by commas. "&" is the line continuation character.	HP C/XL lets you split the string internally with the "\" continuation character. The string continues in column one of the next line.

NOTE The SPL command is quite different from the HP C/XL directive. In SPL, the combined strings *are* the copyright notice. In HP C/XL, the string is assumed to be a company name that is inserted into predefined text.

Cross Reference Listing

There is no equivalent of the MPE V CROSSREF program available on MPE XL.

$INCLUDE Command

Table 10-8. $INCLUDE Command

SPL	HP C/XL Equivalent
include-command: $INCLUDE *filename*	*include-directive*: 1. #include "*filename* " 2. #include <*filename* >
The text from *filename* is inserted in the source stream at the point of the $INCLUDE command.	Same as SPL.
A full file-id is *filename*.*group*.*account*. If ".*account* " or ".*group*.*account* " is omitted, it defaults to the logon group and account.	Form 1 is the same as SPL, except that the default group and account is that of the source file, and HP C/XL will continue the search in other groups and accounts. See the *HP C/XL Reference Manual Supplement* for a complete description of the file search algorithm. Form 2 implies that the file was supplied with the system. The default group is H and the default account is SYS.

MPE Commands

Many of the MPE V commands described in Chapter 10 of the *Systems Programming Language Reference Manual* are identical to MPE XL commands.

However, the commands required to compile and run an HP C/XL program are different in name and parameters from those used for SPL. Please consult the *HP C/XL Reference Manual Supplement* for the commands and parameters you will need.

Chapter 11 Step-by-Step SPL HP C/XL Conversion

This chapter describes a suggested method for converting SPL programs into HP C/XL. It is by no means the only method, but it is one that works well in a number of common circumstances. The person assigned to the conversion should have a good working knowledge of SPL and the tools (that is, editors) that are used to maintain SPL programs. That person should also be acquainted with the C programming language. The SPL program being converted should be currently correct, and there should be a method of testing it for continued correct behavior.

It is preferable to convert an SPL program in a series of steps, actually retaining the program in SPL source for as long as possible. The primary steps are:

1. Remove as many of the hardware-dependent SPL constructs as possible from the SPL version, recompile, and test.

2. Rewrite certain SPL constructs to be more like HP C/XL, recompile, and test.

3. Convert the source code to HP C/XL (rewriting as little as possible), then compile, debug, and test.

4. Examine the HP C/XL source for improvements that can take advantage of constructs and capabilities not available in SPL.

Step One: Remove Hardware Dependencies

Many SPL constructs are highly hardware-dependent, such as ASSEMBLE statements and references to hardware registers. In most cases, these constructs were used by SPL programmers for reasons of efficiency, not for lack of higher level alternatives. Rewriting these portions and testing the program again should be a first step. Normally, this is a matter of determining exactly what the old statements are intended to do, and implementing the same function in SPL statements that do not dependon specific hardware instructions or registers.

The direct use of the stack, via PUSH, DEL, and TOS operators is done for one of two reasons: either to avoid declaring temporary variables, or to retrieve information left on the stack after an operation such as SCAN. In the first case, simply declaring extra variables will allow the stack references to be eliminated. In the case of operations such as SCAN, see the relevant areas of this guide for SPL procedures that isolate these operations and may later be replaced by equivalent HP C/XL functions.

Step Two: Rewrite SPL to Look Like HP C/XL

The case sensitivity of HP C/XL is one of the first differences between these two languages that an SPL programmer is liable to notice. Because SPL ignores case, some SPL programs have examples of the same reserved word or variable name appearing in both upper- and lowercase at various points in the source. In HP C/XL, these names will be interpreted as *different* entities. HP C/XL keywords *must* be expressed in lowercase. Many HP C/XL programmers tend to specify #define macro identifiers in uppercase to distinguish them from function names, but there are no universally accepted standards.

As a first step, convert the SPL source to all uppercase (except for strings, of course). When you convert to HP C/XL in the next step, reserved and keywords will shift to lowercase, and the identifiers of variables, etc., will remain in uppercase, thereby avoiding any possible conflict with HP C/XL reserved words and library function names.

There are a number of other changes which may be made to SPL programs, causing them to conform more closely to HP C/XL forms, and rendering them easier to translate.

For example, HP C/XL does not allow nested functions, so SPL subroutines will be awkward to translate. Careful examination now of any subroutines used in your SPL program will give you a head start on determining how best to eliminate the subroutines.

Possibilities are: moving the subroutine code inline (meaning that the code will be repeated wherever the subroutine is called, possibly by means of a DEFINE), or converting the subroutine to an SPL procedure. In the latter case, variables in the procedure that are accessed by the subroutine will have to be supplied as parameters, or declared global to both the new procedure and its caller. In many cases, these variables were declared in the procedure simply for use by the subroutine, which means they may be declared within the new procedure.

Also, be alert for the possibility that identical subroutines were declared local to more than one procedure; they could all be replaced by one global procedure.

Another change that may be easier to debug prior to converting to HP C/XL is the elimination of any pass-by-reference procedure parameters. By changing such parameters to pass-by-value pointers, and then changing the actual parameters to addresses (generated with the "@" operator), the process of passing and accessing parameters in the same manner as HP C/XL may be tested in an SPL environment. Remember that unsubscripted array, pointer, and procedure identifiers passed by reference do not require any modification.

Because the natural data size of HP C/XL is 32 bits, you should convert as many SPL INTEGER variables to SPL DOUBLE as possible. This will result in a more efficient final HP C/XL program.

There are a few HP C/XL reserved words that are also reserved words in SPL, and there always exists the possibility that an SPL variable name has a unique meaning as an HP C/XL reserved word. All keywords in HP C/XL must be in lowercase, but relying on case differences to differentiate between reserved words and variable names is bad practice.

The following is a list of words which are reserved in both languages, but do not always mean the same thing:

CASE — This is a statement in SPL, but, as case, it is used to label switch alternatives in HP C/XL.

DO — This statement is very similar, but SPL performs a DO-WHILE test, while HP C/XL performs a do-until test.

DOUBLE — This is a 32-bit signed integer in SPL, but a 64-bit IEEE floating point number in HP C/XL. Variables declared DOUBLE in an SPL program *must* be converted to type [long] int when converting to HP C/XL.

ELSE — This word is very similar in both languages, therefore simply make certain that else is in lowercase at the time of conversion. Also, make sure the statement before the else ends with either a semicolon, ";", or a right brace, "}".

FOR — This is a similar statement that has different syntax. See "FOR Statement".

GOTO — This is an identical operation, but SPL allows both GO and GOTO. Changing both to lowercase goto is valid SPL and prepares for the move to HP C/XL.

IF — This statement is identical in both languages, but has slightly different syntax. Change the word to lowercase.

LONG — In SPL, this is a 64-bit floating point number in the MPE V format. In HP C/XL, long is a 32-bit integer. Be certain to convert it to double in HP C/XL. Also remember that the 64-bit floating-point internal formats are quite different on the two systems.

RETURN — This causes a return from a procedure or subroutine in SPL, and also causes a return from a function in HP C/XL. Remember to remove the SPL parameter and add the HP C/XL return value. (See "RETURN Statement").

SWITCH — In SPL, this declares a list of labels to be branched to by an indexed GOTO. In HP C/XL, switch is a statement type, analogous to the SPL CASE. (See "GO TO Statement").

WHILE — This function is identical in both languages, having only a slight difference in syntax.

The following are HP C/XL reserved words. Examine the SPL source for any use of these words as variable names.

auto	default	extern	int	sizeof	union
break	do	float	long	static	unsigned
case	double	for	register	struct	void
char	else	goto	return	switch	while
continue	enum	if	short	typedef	

You should also avoid the following proposed ANSI C reserved words:

const	signed	volatile

SPL array declarations should be examined for cases that have a nonzero lower bound. This is not allowed in HP C/XL, and should be recoded to work properly with a lower bound of zero. Remember that indirect (and many direct) arrays can be coded in HP C/XL with a pointer to cell zero.

BYTE arrays used for storing ASCII strings should be examined for how they are used and, if possible, a NUL ('\0', numeric value zero) should be placed in the last byte. This is done to facilitate later use of the HP C/XL convention, which expects a NUL to terminate a string.

Be especially careful with cases where word pointers were converted to byte pointers (and vice versa) by means of shift, multiply, or divide operations. All pointers in HP C/XL will refer to byte addresses, so these operations will rarely translate without careful recoding.

Bit operations in SPL are performed for two reasons. One is to unpack data words read by the program from external files, and the other to conserve data storage for variables used by the program. In the latter case, consider declaring whole words for the individual fields and eliminating the bit operations entirely.

The SPL switch declaration may be left alone at this stage. It will be converted into an HP C/XL #define macro directive in step 3. See "GO TO Statement" and "CASE Statement".

Certain operations, such as passing labels as parameters, are not permitted in HP C/XL, so now could be the time to recode the necessary operation in more translatable constructs. In general, operations that use extra data segments and split stack operations, should be removed and rewritten (if possible), or at least isolated into separate procedures.

As a final consideration to HP C/XL, move all of the SPL program's outer block executable statements into a new procedure named main, and make the new outer block consist of a single statement, calling this procedure. These changes will bring an SPL program as close to HP C/XL conventions as possible, and should be thoroughly tested in this form before making the plunge into HP C/XL itself.

Step Three: Convert the Source to HP C/XL

If the preceding two steps have been performed carefully, conversion of the SPL source to something acceptable to the HP C/XL compiler may take less time and effort than expected. The major structural changes will be to remove the initial BEGIN, the outer block (call to procedure main), and the final END. HP C/XL will generate code to initiate the running of the program by calling function main. As the order of declaration of HP C/XL functions is not as critical as in SPL, it is common practice to declare function main as the first function, immediately after any global data declarations, followed by the rest of the function declarations. Thus, all FORWARD declarations should be removed, but the type of any function used prior to its declaration should be specified in the function where it is called.

There are certain obvious changes to be made at this point. They include deleting the word PROCEDURE, changing BEGIN to "{", END to "}" (make sure the preceding statement ends in ";"), and replacing any "'" (apostrophe) characters within variable names with "_" (underscore).

Conversion of the data types should be undertaken with some caution; refer to the SPL data types in this document for suggestions.

As you make the syntax changes in statements such as IF and DO, remember to downshift the keywords. This will serve as a reminder of what has been converted. It's also necessary so the HP C/XL compiler can recognize them.

After converting $CONTROL lines to their equivalent HP C/XL constructs, the first attempts to compile the program may be made.

With the exception of rewriting any code designed to use features such as extra data segments and split stack operations, the most difficult and time consuming work will be assuring that equated declarations and pointer operations behave as they did in SPL. If the equating is necessary, it may be emulated via the union declaration. Pointer operations, especially pointer arithmetic and storing numeric values into pointers, will require the most care in converting. SPL allowed many extremely dangerous operations to be performed, and pointer adjustments were done assuming very specific hardware-dependent rules. HP C/XL, while allowing a great deal of freedom to manipulate pointers, has much more consistent rules regarding the effects of operations on pointers. The differences, however, must be accommodated. The resulting HP C/XL code should be clearer and easier to maintain than the original SPL.

Step Four: Improve the Translated Source

After following the first three steps in this chapter, you will be tempted to "leave well enough alone". Resist this temptation. Any program written initially in SPL, and translated more or less literally into HP C/XL according to these guidelines, is unlikely to be one which a proficient HP C/XL programmer would create directly. The SPL "heritage" will be apparent in the use of union declarations, old SPL equivalencing operations, awkward I/O functions, and so on.

A frequent reason for equating variable names to arrays in SPL is to overcome the lack of any form of record or structure variables. Wherever possible, the use of union to emulate SPL equivalencing should be examined to determine if the HP C/XL struct declaration is more natural and appropriate.

Because HP C/XL performs implicit type conversions during expression evaluation, many type cast operations (which required type conversion functions in SPL) may have been inserted in the converted program where they are no longer needed.

In SPL, strings are simply ASCII characters in arrays of type BYTE. Various conventions were devised by SPL programmers to determine and store the length of these strings, such as keeping a count in a separate variable, or possibly within the first byte of the array. These same operations may be converted to HP C/XL, but the accepted convention in HP C/XL is to delimit a string by appending a NUL character to the string. A NUL character (numeric value 0) is represented in HP C/XL by "'\0'". Once you adopt this convention, a large library of string manipulation routines becomes available, both simplifying and optimizing string operations.

The wide range of formatted I/O routines available in HP C/XL may be utilized, frequently allowing many SPL operations to be replaced with a simple function call. At first, the HP C/XL I/O functions may look simple, and therefore limited. However, the generality of these functions means that they may be combined in ways that are just as powerful as system intrinsics with seemingly more complex options. Remember that reliance on any specific operating system intrinsics restricts the program to that operating system. While this may be unavoidable in some cases, the use of the HP C/XL high level I/O functions will increase program portability, even across operating systems.

Storage allocation (and deallocation) in SPL is quite straightforward, but restrictive. In HP C/XL, a compound statement may contain local variable declarations that are allocated on entry to the statement and released when it ends. There may be instances in a program translated from SPL where this is a more natural structure for the program. It can simplify the source code by defining the scope of specific local variables better. Also in HP C/XL, there are several functions which allow programmatic allocation and deallocation of storage at runtime. No similar features are easily available to the SPL programmer, leading to the occasional clumsy use of dynamic arrays within procedures declared only to allocate space dynamically, or worse, to manipulate hardware registers to force access to memory regions not otherwise available. Use of HP C/XL functions such as malloc, realloc, and free makes it possible to dynamically allocate, reallocate, and release storage at will. HP C/XL programs need not retain the SPL "flavor" that results from a literal translation. The features and operations that performed very efficiently under MPE V now may be needlessly complex and quite possibly less efficient. By using the high level constructs of HP C/XL and its extensive library functions, you can develop programs that are maintainable, portable, and will result in extremely efficient runtime code with the optimizing features of the HP C/XL compiler.

Appendix A SPL Procedures to Replace Special Features

The SPL procedures in this appendix perform many of the same operations as the HP C/XL macros and functions in Appendix B. Using these procedures in an SPL program will help to isolate special hardware-dependent operations and greatly simplify the transition to HP C/XL.

SPL BCONCAT Procedure: Bit Concatenation

```
 <<  BCONCAT              SPL BIT CONCATENATION                     >>
<<                                                                  >>
<<  This emulates the SPL bit concatenation operation, for example: >>
<<           X := A CAT B (4:8:4);                                  >>
<<  This procedure performs the same operation without use of       >>
<<  the CAT operator.                                               >>
<<                                                                  >>
<<  The parameters used by BCONCAT are:                             >>
<<    a  --  1st 16 bit word to be merged into.                     >>
<<    b  --  2nd 16 bit word with field to be merged.               >>
<<    sa --  Starting bit in word "a".                              >>
<<    sb --  Starting bit in word "b".                              >>
<<    n  --  Number of bits to merge.                               >>
<<                                                                  >>
<<  The 16 bit value returned by the function is the result of      >>
<<  the concatenate operation.                                      >>

LOGICAL PROCEDURE BCONCAT(a,b,as,bs,n);
  VALUE a,b,as,bs,n;
  LOGICAL a,b;
  INTEGER as,bs,n;
BEGIN
  LOGICAL M;
  n := 16-n;
  M := (%(16)FFFF & LSR(n)) & LSL(n-as);
  BCONCAT := (a LAND NOT(M)) LOR
                (IF as<bs THEN
                    b & LSL(bs-as) ELSE
                    b & LSR(as-bs) LAND M);
END;
```

SPL BDEPOSIT Procedure: Bit Deposit

```
 <<  BDEPOSIT           SPL BIT DEPOSIT                             >>
<<                                                                  >>
<<  This emulates the SPL bit deposit operation, for example:       >>
<<           I.(5:6) := J + K;                                      >>
<<  as an SPL procedure:                                            >>
<<           BDEPOSIT(@i,5,6,j+k);                                  >>
<<                                                                  >>
<<  The parameters used by BDEPOSIT are:                            >>
<<    dw  --  The address of the destination word.                  >>
<<    sb  --  The starting bit of the deposit field.                >>
<<    nb  --  The number of bits to deposit.                        >>
<<    exp --  The expression to deposit into the field specified.   >>
<<                                                                  >>
```

```
PROCEDURE BDEPOSIT(dw,sb,nb,exp);
  VALUE dw, sb, nb, exp;
  LOGICAL dw, sb, nb, exp;
BEGIN
  LOGICAL M;
  POINTER P;
  nb := 16-nb;
  sb := nb-sb;
  M := (%(16)FFFF & LSR(nb)) & LSL(sb);
  @p := dw;
  p := (p LAND NOT m) LOR (exp & LSL(sb) LAND m);
END;
```

SPL BEXTRACT Procedure: Bit Extraction

```
<<  BEXTRACT          SPL BIT EXTRACTION                              >>
<<                                                                    >>
<<  This procedure emulates the SPL bit extraction, for example:      >>
<<     x := y.(10:4);                                                 >>
<<  as an SPL procedure:                                              >>
<<     x := BEXTRACT(y,10,4);                                         >>
<<                                                                    >>
<<  The parameters to BEXTRACT are:                                   >>
<<     wd   --  Word (unsigned short) to extract bits from.           >>
<<     sb   --  Starting bit of field (0 through 15, left to right).>>
<<     nb   --  Number of bits in field.                              >>
<<                                                                    >>
<<  The return value will be the extracted field, right justified     >>
<<  in a 16 bit (unsigned short) word.                                >>

LOGICAL PROCEDURE BEXTRACT(wd,sb,n);
  VALUE wd, sb, nb;
  LOGICAL wd;
  INTEGER sb, nb;
BEGIN
  BEXTRACT := (wd & LSL(sb)) & LSR(16-nb);
END;
```

SPL BYTECMP Procedure: Byte Comparison

```
 <<  BYTECMP             SPL COMPARE BYTE STRINGS                      >>
<<                                                                    >>
<<  This emulates the byte string compare expression in SPL,          >>
<<  for example:                                                      >>
<<          IF A < B,(N),0;                                           >>
<<          NN := TOS;  {count}                                       >>
<<          @AA := TOS; {left address after compare}                  >>
<<          @BB := TOS; {right address after compare}                 >>
<<  This may be converted to C with:                                  >>
<<          if (BYTECMP(a,LSS,b,n,0,&nn,&aa,&bb))....>>
<<                                                                    >>
<<  The parameters to BYTECMP are:                                    >>
<<    left       --  The left address to be compared.                 >>
<<    cmp        --  The comparison to be made.  Here the following >>
<<                  syntax exists.                                    >>
<<                      LSS     means <                               >>
<<                      LEQ     means <=                              >>
<<                      EQU     means ==                              >>
<<                      NEQ     means <>                              >>
<<                      GEQ     means >=                              >>
<<                      GTR     means >                               >>
<<    right      --  The right address to be compared.                >>
<<    count      --  The maximum number of bytes to compare.          >>
<<    sdec       --  The SPL stack decrement.  In this context, the >>
<<                  value of this parameter will determine if the     >>
<<                  function accesses the last parameter, as          >>
<<                  follows:                                          >>
```

```
<<                      sdec = 3  -- Ignore the last three          >>
<<                                  parameters (in SPL, this is     >>
<<                                  the default case, deleting      >>
<<                                  three stack words).             >>
<<                      sdec = 2  -- Expect only one parameter after>>
<<                                  this, cnt.                      >>
<<                      sdec = 1  -- Expect two parameters after    >>
<<                                  this, cnt and laddr.            >>
<<                      sdec = 0  -- Expect three parameters after  >>
<<                                  this, cnt, laddr, and raddr.    >>
<<    cnt        --  The value of "count" at the conclusion of the  >>
<<                  comparison.  If the strings compare for count   >>
<<                  bytes, cnt will equal zero.                     >>
<<    laddr      --  The address of the char within the left string >>
<<                  which failed to match.                          >>
<<    raddr      --  The address of the char within the right string>>
<<                  which failed to match.                          >>

DEFINE LSS=0#, LEQ=1#, EQU=2#, NEQ=3#, GEQ=4#, GTR=5#;

INTEGER PROCEDURE BYTECMP(left,cmp,right,count,sdec,cnt,laddr,raddr);
  VALUE left, cmp, right, count, sdec, cnt, laddr, raddr;
  LOGICAL left, right, laddr, raddr;
  INTEGER cmp, count, sdec, cnt;
BEGIN
  DEFINE ADJ =
    DO BEGIN
         IF count > 0
           THEN BEGIN count:=count-1; @lftp:=@lftp+1; @rhtp:=@rhtp+1; END
           ELSE BEGIN count:=count+1; @lftp:=@lftp-1; @rhtp:=@rhtp-1; END;
       END#;
  BYTE POINTER lftp, rhtp, laddrp, raddrp;
  INTEGER POINTER cntp;
  @lftp := left;
  @rhtp := rht;
  @cntp := cnt;
  @laddrp := laddr;
  @raddrp := raddr;
  CASE cmp OF
    BEGIN
      <<LSS:   compare <  >>
            BEGIN WHILE (count <> 0) AND (lftp < rhtp) ADJ END;
      <<LEQ:   compare <= >>
            BEGIN WHILE (count <> 0) AND (lftp <= rhtp) ADJ END;
      <<EQU:   compare == >>
            BEGIN WHILE (count <> 0) AND (lftp == rhtp) ADJ END;
      <<NEQ:   compare <> >>
            BEGIN WHILE (count <> 0) AND (lftp <> rhtp) ADJ END;
      <<GEQ:   compare >= >>
            BEGIN WHILE (count <> 0) AND (lftp >= rhtp) ADJ END;
      <<GTR:   compare >  >>
            BEGIN WHILE (count <> 0) AND (lftp > rhtp) ADJ END;
    END;
  CASE sdec OF
    BEGIN
      << 0 >>  GOTO sdec 0;
      << 1 >>  GOTO sdec 1;
      << 2 >>  GOTO sdec 2;
      << 3 >>  GOTO sdec 3;
    END;
  sdec0:  raddrp := rhtp;
  sdec1:  laddrp := lftp;
  sdec2:  cntp := count;
  sdec3:  ;  << nil >>
  BYTECMP := IF count = 0 THEN 1 ELSE 0;
END;
```

Appendix B   HP C/XL Funtions to Emulate SPL Operations

The HP C/XL macro directives and function definitions in this appendix emulate SPL operations that are performed by special features of the SPL language, usually designed to access specific instructions available under the MPE V operating system. If an SPL program has had these operations replaced by the SPL procedures in Appendix A, simple replacement of those procedure declarations by these HP C/XL macros and functions are all that will be necessary to perform the same operation in HP C/XL. Note that variable names are compatible with respect to case and special characters.

The HP C/XL macro directives and function definitions in this appendix emulate SPL operations that are performed by special features of the SPL language, usually designed to access specific instructions available under the MPE V operating system. If an SPL program has had these operations replaced by the SPL procedures in Appendix A, simple replacement of those procedure declarations by these HP C/XL macros and functions are all that will be necessary to perform the same operation in HP C/XL. Note that variable names are compatible with respect to case and special characters.

HP C/XL BCONCAT Function: Bit Concatenation

```
 /***************************************************************
  BCONCAT          SPL BIT CONCATENATION

  This emulates the SPL bit concatenation operation, for example:
           X := A CAT B (4:8:4);
  Using this function, this may be converted to HP C with:
           x = BCONCAT(a,b,4,8,4);
  The parameters used by BCONCAT are:
    a -- 1st 16 bit word to be merged into.
    b -- 2nd 16 bit word with field to be merged.
    sa-- Starting bit in word "a".
    sb-- Starting bit in word "b".
    n -- Number of bits to merge.

  The 16 bit value returned by the function is the result of
  the concatenate operation.
****************************************************************/

unsigned short int BCONCAT(a,b,sa,sb,n)
unsigned short int a, b, sa, sb, n;
{
  unsigned int m;
  n = 16-n;
  m = (0xFFF>>n)<<(n-sa);
  return((unsigned short int)((a & ~m) |
         ((sa<sb ? b<<(sb-sa) : b>>(sa-sb)) & m)));
}
```

HP C/XL BDEPOSIT Function: Bit Deposit

```
 /************************************************************
  BDEPOSIT          SPL BIT DEPOSIT

  This emulates the SPL bit deposit operation, for example,
           I.(5:6) := J + K;
  Using this function, this may be converted to HP C with:
           BDEPOSIT(&i,5,6,j+k);

  The parameters used by BDEPOSIT are:
    dw  -- The address of the destination word.
    sb  -- The starting bit of the deposit field.
    nb  -- The number of bits to deposit.
    exp -- The expression to deposit into the field specified.
*************************************************************/

void BDEPOSIT(dw,sb,nb,exp)
unsigned short *dw, sb, nb, exp;
{
  unsigned short m;
  nb = 16-nb;
  sb = nb-sb;
  m = (0xFFFF>>nb)<<sb;
  *dw = (*dw & ~m) | (exp<<sb & m);
}
```

HP C/XL BEXTRACT Macro and Function: Bit Extraction

```
 /***********************************************************
  BEXTRACT          SPL Bit Extraction

  This macro and function perform the SPL bit extraction:
     x := y.(10:4);
  which may be replaced in HP C by:

     x = BEXTRACT(y,10,4);

  The parameters to BEXTRACT are:
    wd -- The word (unsigned short int) from which to extract bits.
    sb -- Starting bit of field (0 through 15, left to right).
    nb -- Number of bits in field.
  The return value will be the extracted field, right
  justified in a 16 bit (unsigned short int) word.

************************************************************/

#define BEXTRACT(w,s,n) (((unsigned short int)((w)<<(s)))>>(16-(n)))

/***********************************************************/

unsigned short int BEXTRACT(sw,sb,nb)
unsigned short int sw, sb, nb;
{
  return((unsigned short int)((sw<<sb))>>(16-nb));
}
```

HP C/XL BYTECMP Function: Byte Comparison

```
 /************************************************************
  BYTECMP           SPL COMPARE BYTE STRINGS

  This emulates the byte string compare expression in SPL,
  for example:
          IF A < B,(N),0;
          NN := TOS;  <<count>>
          @AA := TOS;  <<left address after compare>>
          @BB := TOS;  <<right address after compare>>
  This may be converted to C with:
          if (BYTECMP(a,LSS,b,n,0,&nn,&aa,&bb))...

  The parameters to BYTECMP are:
    left        --  The left address to be compared.
    cmp        --  The comparison to be made, where:
                    LSS     means  <
                    LEQ     means  <=
                    EQU     means  ==
                    NEQ     means  !=
                    GEQ     means  >=
                    GTR     means  >
    right        --  The right address to be compared.
    count           --  The maximum number of bytes to compare.
    sdec       --  The SPL stack decrement.  In this context,
                 the value of this parameter will determine if
                 the function accesses the last parameter
                 as follows:
                 sdec = 3  -- Ignore last three parameters
                             (in SPL, this is the default
                             case, deleting 3 stack words).
                 sdec = 2  -- Expect only one parameter
                             after this:  caddr.
                 sdec = 1  -- Expect two parameters after
                             this:  caddr and laddr.
                 sdec = 0  -- Expect three parameters after
                             this:  caddr, laddr, and
                             raddr.
    caddr        --  The value of count at the conclusion of the
                 comparison.  If the strings compare for
                 count bytes, caddr will equal zero.
    laddr      --  The address of the char within the left
                 string which failed to match.
    raddr      --  The address of the char within the right
                 string which failed to match.

**************************************************************/

enum CMP {LSS, LEQ, EQU, NEQ, GEQ, GTR };
short int BYTECMP(left,cmp,right,count,sdec,caddr,laddr,raddr)
  char *left, *right, **laddr, **raddr;
  enum CMP cmp;
  int count, sdec, *caddr;
{
#define ADJ  {if (count>0) {--count;++left;++right;}  \
                      else {++count;--eft;--right;}}
  switch (cmp)  {
    case LSS:  /* compare < */
               while ((count != 0) && (*left < *right))  ADJ;
               break;
    case LEQ:  /* compare <= */
               while ((count != 0) && (*left <= *right))  ADJ;
               break;
    case EQU:  /* compare == */
               while ((count != 0) && (*left == *right))  ADJ;
               break;
```

```
    case NEQ:  /* compare != */
               while ((count != 0) && (*left != *right))  ADJ;
               break;
    case GEQ:  /* compare >= */
               while ((count != 0) && (*left >= *right))  ADJ;
               break;
    case GTR:  /* compare > */
               while ((count != 0) && (*left > *right))  ADJ;
               break;
  }
  switch (sdec) {
    case 0:  *raddr = right;
    case 1:  *laddr = left;
    case 2:  *caddr = count;
    case 3:  ;  /* nil */
  }
  return (count == 0);
#undef ADJ
}
```

HP C/XL MOVEB Function: Move Bytes

```
/**************************************************************
 MOVEB                SPL MOVE BYTES

 This emulates the MOVE statement in SPL for byte moves with
 no information removed from the stack, for example:
         MOVE B1 := B2, (CNT), 0
         LEN := tos;
         @S1 := tos;
         @D1 := tos;
 This may be converted to C with:
         LEN := MOVEB(B1,B2,CNT,0,&S1,&D1);

 The parameters to MOVEB are:
   to         --  The address to be moved to.
   from       --  The address to be moved from.
   count      --  Number of bytes to be moved.  A positive value
                 means left to right move, negative means
                 right to left.
   sdec       --  The SPL stack decrement.  In this context, the
                 value of this parameter will determine if
                 the function accesses the last two
                 parameters, as follows:
                 sdec = 3  -- Ignore the last two parameters
                             (in SPL, this is the default
                             case, deleting 3 stack words).
                 sdec = 2  -- Expect only one parameter
                             after this, dest_addr.
                 sdec = 1  -- Expect two parameters after
                             this, dest_addr and source_addr.
                 sdec = 0  -- Same as 1.  This is never a
                             meaningful operation in SPL,
                             as the TOS, or count value,
                             is always zero after the MOVE
                             instruction.
   source_addr -- The address of the next char of "from" beyond
                 the final character moved.
   dest_addr   -- The address of the next char of "to" beyond the
                 final character moved.

 The return value of the function is the number of bytes moved
```

```
**************************************************************/

short int MOVEB(to,from,count,sdec,source_addr,dest_addr)
  char *to, *from, **source_addr, **dest_addr;
  int count, sdec;
{
  int c;
  c = 0;
  if (count>0)          /* left-to-right move */
    do *to++ = *from++; while (++c < count);
  else if (count<0)     /* right-to-left move */
         {
           count = -count;
           do *to-- = = *from--; while (++c < count);
         }
  switch (sdec) {
    case 0:  ;   /*  fall through to case 1  */
    case 1:  *source_addr = from;
    case 2:  *dest_addr = to;
    case 3:  ;  /* nil */
  }
  return(c);
}
```

HP C/XL MOVEBW Function: Move Bytes While

```
/**************************************************************
 MOVEBW            SPL MOVE BYTES WHILE

 This emulates the MOVE while statement in SPL
         MOVE B1 := B2, WHILE A, 0;
         @S1 := tos;
         @D1 := tos;
 which may be converted to C with:
         LEN := MOVEBW(B1,B2,A,0,&S1,&D1);

 The parameters to moveb are:
   to          --  The address to be moved into.
   from        --  The address to be moved from.
   cond        --  The move while condition, where:
                     A    means alphabetic
                     AN   means alphanumeric
                     AS   means alphabetic, upshift
                     N    means numeric
                     ANS  means alphanumeric, upshift
   sdec        --  The SPL stack decrement.  In this context, the
                  value of this parameter will determine if
                  the function accesses the last two
                  parameters, as follows:
                  sdec = 3  -- Ignore the last two parameters
                              (in SPL, this is the default
                              case, deleting 3 stack words).
                  sdec = 2  -- Expect only one parameter
                              after this, dest_addr.
                  sdec = 1  -- Expect two parameters after
                              this, dest_addr and source_addr.
                  sdec = 0  -- Same as 1.  This is not a
                              meaningful operation in SPL,
                              as the TOS, or count value,
                              is always zero after the MOVE
                              instruction.
   source_addr --  The address of the next char of from beyond
                  the final character moved.
   dest_addr   --  The address of the char of from beyond the
                  final character moved.

 The return value of the function is the number of bytes moved.
```

```
***********************************************************/

enum COND {A, AN, AS, N, ANS};
short int MOVEBW(to,from,cond,sdec,source_addr,dest_addr)
  enum COND cond;
  char *to, *from, **source_addr, **dest_addr;
  int sdec;
{
  char *temp;
  temp = to;
  switch (cond)  {
    case   A:  while (isalpha(*from)) *to++ = *from++;
                    break;
    case  AN:  while (isalnum(*from)) *to++ = *from++;
                    break;
    case  AS:  while (isalpha(*from)) *to++ = toupper(*from++);
                    break;
    case   N:  while (isdigit(*from)) *to++ = *from++;
                    break;
    case ANS:  while (isalnum(*from)) *to++ = toupper(*from++);
                    break;
  }
  switch (sdec) {
    case 0:  ;  /*  fall through to case 1  */
    case 1:  *source_addr = from;
    case 2:  *dest_addr = to;
  }
  return(to-temp);
}
```

HP C/XL MOVESB Function: Move String Bytes

```
/***********************************************************
 MOVESB            SPL MOVE STRING BYTES

 This emulates the MOVE statement in SPL for string moves,
 for example:
       MOVE A1 := "constant string",0;
       LEN := tos;
       S1 := tos;
       D1 := tos;
 which may be converted to C with:
       LEN := MOVESB(A1,"constant string",0,&S1,&D1);

 The parameters to MOVESB are:
  to          --  The address to be moved into,
                 right to left.
  sdec        --  The SPL stack decrement.  This parameter will
                 determine if the function accesses the last
                 two parameters, as follows:
                 sdec = 3  -- Ignore the last two parameters
                             (in SPL, this is the default
                             case, deleting 3 stack words).
                 sdec = 2  -- Expect only one parameter
                             after this, dest_addr.
                 sdec = 1  -- Expect two parameters after
                             this, dest_addr and source_addr.
                 sdec = 0  -- Same as 1.  This is never meaningful
                             in SPL, because the TOS (count)
                             is always zero after a MOVE.
  source_addr --  The address of the next char of str beyond.
                 the final character moved.
  dest_addr   --  The address of the next char of to beyond the
                 final character moved.

 The return value is the number of bytes moved.
```

```
************************************************************/

short int MOVESB(to,str,sdec,source_addr,dest_addr)
  char *to, *str, **source_addr, **dest_addr;
  int sdec;
{
  char *temp;
  temp = to;
  while (*str != '\0') *to++ = *str++;
  switch (sdec) {
    case 0:  ;  /*  fall through to case 1  */
    case 1:  *source_addr = str;
    case 2:  *dest_addr = to;
    case 3:  ;  /* nil */
  }
  return(to-temp);
}
```

HP C/XL MOVEW Function: Move Words

```
/*********************************************************************
  MOVEW          SPL MOVE WORDS

  This emulates the MOVE statement in SPL for word moves with
  no information removed from the stack, for example:
          MOVE W1 := W2, (CNT), 0
          LEN := tos;
          @S1 := tos;
          @D1 := tos;
  This may be converted to C with:
          LEN := MOVEW(W1,W2,CNT,0,&S1,&D1);

  The parameters to MOVEW are:
   to          --  The address to be moved into.
   from        --  The address to be moved from.
   count       --  Number of bytes to be moved; a positive value
                  means left to right move, negative means
                  right to left.
   sdec        --  The SPL stack decrement.  In this context, the
                  value of this parameter will determine if the
                  function accesses the last two parameters,
                  as follows:
                  sdec = 3  -- Ignore the last two parameters
                              (in SPL, this is the default
                              case, deleting 3 stack words).
                  sdec = 2  -- Expect only one parameter after
                              this, dest_addr.
                  sdec = 1  -- Expect two parameters after
                              this, dest_addr and source_addr.
                  sdec = 0  -- Same as 1.  This is not a meaningful
                              operation in SPL because the TOS,
                              or count value, is always zero
                              after the MOVE instruction.
   source_addr --  The address of the next char of "from" beyond
                  the final character moved.
   dest_addr   --  The address of the next char of "to" beyond the
                  final character moved.

   The return value of the function is the number of bytes moved.

*********************************************************************/

short int MOVEW(to,from,count,sdec,source_addr,dest_addr)
  short int *to, *from, **source_addr, **dest_addr;
  int count, sdec;
{
  int c;
```

```
  c = 0;
  if (count>0)              /* left to right move */
    do *to++ = *from++; while (++c < count);
  else if (count<0)         /* right to left move */
         {
           count = -count;
           do *to-- = *from--; while (++c < count);
         }
  switch (sdec) {
    case 0:  ;  /* fall through to case 1 */
    case 1:  *source_addr = from;
    case 2:  *dest_addr = to;
    case 3:  ;  /* nil */
  }
  return(c);
}
```

HP C/XL SCANU Function: Scan Until

```
/*******************************************************************
  SCANU              SPL SCAN UNTIL

  This emulates the SCAN until statement in SPL, for example:
          NUM := (SCAN B1 UNTIL TEST, 0);
          T := TOS;  <<test word -- unchanged>>
          @S1 := TOS;
  This may be converted to C with:
          LEN := SCANU(B1,TEST,0,&S1);

  The parameters to SCANU are:
    ba         --  The address to be scanned.
    test       --  The testword, two bytes.  The first is the
                  terminate character, the second is the
                  test character, either of which will
                  cause the scanning to continue.
    sdec       --  The SPL stack decrement.  In this context, the
                  value of this parameter will determine if the
                  function accesses the last parameter,
                  as follows:
                  sdec = 2  -- Ignore the last parameter.  This
                              parameter need not be present.
                  sdec = 1  -- Expect one parameter after this:
                              scan_addr.
                  sdec = 0  -- Same as 1.  In SPL, an sdec of 1
                              or 2 deletes the test word from
                              the stack, which is always unchanged
                              after the SCAN operation.
    scan_addr  --  The address of the char which stopped the SCAN
                  operation.  This equals either the terminal or
                  the test character.

  The return value of this function is the number of bytes moved.

*******************************************************************/

short int SCANU(ba,test,sdec,scan_addr)
  char *ba, **scan_addr;
  unsigned short test;
  int sdec;
{
  char termc, testc, *temp;
  temp = ba;
  termc = (char)test >> 8;
  testc = (char)test & OxFF;
  while ((*ba != testc) && (*ba != testc)) ba++;
  switch (sdec) {
    case 0:  ;  /* fall through to case 1 */
```

```
    case 1:  *scan_addr = ba;
    case 2:  ;  /* nil */
  }
  return(ba-temp);
}
```

HP C/XL SCANW Function: Scan While

```
/*********************************************************************
  SCANW              SPL SCAN WHILE

  This emulates the SCAN while statement in SPL, for example:
          NUM := (SCAN B1 WHILE TEST, 0);
          T := TOS;  <<test word -- unchanged>>
          @S1 := TOS;
  which may be converted to C with:
          LEN := SCANW(B1,TEST,0,&S1);

  The parameters to SCANW are:
    ba         --  The address to be scanned.
    test       --  The testword.  This is two bytes; the first is
                  the terminate character, the second is the test
                  character.  Either of these will terminate the
                  scan operation.
    sdec       --  The SPL stack decrement.  In this context, the
                  value of this parameter will determine if the
                  function accesses the last parameter, as follows:
                  sdec = 2  -- Ignore the last parameter (which
                              need not be present).
                  sdec = 1  -- Expect one parameter after this,
                              scan_addr.
                  sdec = 0  -- Same as 1.  In SPL, an sdec of 1 or
                              2 deletes the test word from the
                              stack, which is always unchanged
                              after the SCAN operation.
    scan_addr  --  The address of the char which stopped the SCAN
                  operation (i.e. failed to equal either the terminal
                  or the test character).
    The return value of the function is the number of bytes moved.

*********************************************************************/

short int SCANW(ba,test,sdec,scan_addr)
  char *ba, **scan_addr;
  unsigned short test;
  int sdec;
{
  char temc, testc, *temp;
  temp = ba;
  termc = (char)test >> 8;
  testc = (char)test & OxFF;
  while ((*ba == testc) || (*ba == testc)) ba++;
  switch (sdec) {
    case 0:  ;  /* fall through to case 1 */
    case 1:  *scan_addr = ba;
    case 2:  ;  /* nil */
  }
  return(ba-temp);
}
```

HP C/XL Bit Shift Macros and Functions

```
 /*******************************************************/
#define LSL(x,c)  ((unsigned short) ((unsigned short) x << c))
/*******************************************************/
#define LSR(x,c)  ((unsigned short) ((unsigned short) x >> c))
/*******************************************************/
#define ASL(x,c)  ((short) ( ((short)x & 0x8000) | \
                             ((short)x << c) & 0x7FFF) )
/*******************************************************/
#define ASR(x,c)  ((short) ((short)x >> c))
/*******************************************************/
unsigned short CSL(x,c)
unsigned short x;
int c;
{
  for (;;--c)   {
    if (c == 0) return(x);
    x = ((x & 0x8000) >> 15) | x << 1;
  }
}
/*******************************************************/
unsigned short CSR(x,c)
unsigned short x;
int c;
{
  for (;;--c)   {
    if (c == 0) return(x);
    x = ((x & 0x0001) << 15) | x >> 1;
  }
}
/*******************************************************/

/*******************************************************/
#define DLSL(x,c)  ((unsigned int) ((unsigned int) x << c))
/*******************************************************/
#define DLSR(x,c)  ((unsigned int) ((unsigned int) x >> c))
/*******************************************************/
#define DASL(x,c)  ((int) ( ((int)x & 0x80000000) | \
                            ((int)x << c) & 0x7FFFFFFF) )
/*******************************************************/
#define DASR(x,c)  ((int) ((int)x >> c))
/*******************************************************/
unsigned int DCSL(x,c)
unsigned int x;
int c;
{
  for (;;--c)   {
    if (c == 0) return(x);
    x = ((x & 0x80000000) >> 31) | x << 1;
  }
}
/*******************************************************/
unsigned int DCSR(x,c)
unsigned int x;
int c;
{
  for (;;--c)   {
    if (c == 0) return(x);
    x = ((x & 0x00000001) << 31) | x >> 1;
  }
}
/*******************************************************/
```