900 Series HP 3000 Computers

HP Link Editor/XL Reference Manual

HP Part No. 32650-90030
Printed in U.S.A. 19901201

Fourth
E1290
DRAFT 11/7/97 02:46

Print History

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

First Edition	November 1987	
Second Edition	October 1988	
Third Edition	October 1989	
Fourth Edition	December 1990	HP30315A.04.00

Preface

This manual describes the HP Link Editor/XL subsystem and how you use it with 900 Series HP 3000 computer systems. The manual assumes that you are an experienced programmer, but not necessarily familiar with "linkers" and "loaders".

This manual contains the following chapters:

Chapter 1	Gives an overview of HP Link Editor/XL - what it is, when to use it and how it works. This chapter also summarizes the differences between Link Editor/XL and its MPE counterpart, the MPE V Segmenter.
Chapter 2	Contains a simple tutorial to help you become familiar with the primary functions of HP Link Editor/XL. Since HP Link Editor/XL differs substantially from the MPE V Segmenter, this chapter helps those familiar with MPE V to quickly understand the difference.
Chapter 3	Describes the files used by HP Link Editor/XL and gives the rules for entering Link Editor/XL commands.
Chapter 4	Discusses the HP Link Editor/XL commands that create and display executable program files.
Chapter 5	Discusses the HP Link Editor/XL commands that create and maintain relocatable libraries.
Chapter 6	Discusses the HP Link Editor/XL commands that create and maintain executable libraries.
Chapter 7	Discusses advanced ways to use HP Link Editor/XL.
Appendix A	Lists warning and error messages, along with their remedial actions.
Appendix B	Explains how HP COBOL II/XL programs interface with HP Link Editor/XL.
Appendix C	Explains how HP FORTRAN 77/XL programs interface with HP Link Editor/XL.
Appendix D	Explains how HP Pascal/XL programs interface with HP Link Editor/XL.
Appendix E	Explains how HP C/XL programs interface with HP Link Editor/XL.
Appendix F	Compares HP Link Editor/XL to the MPE V Segmenter.
Appendix G	Contains the HP Link Editor/XL command summary.

Additional Documentation

This manual does not discuss the MPE XL operating system in detail. Only those aspects relevant to HP Link Editor/XL are mentioned. Similarly, details about compiling a program using HP COBOL II, HP FORTRAN 77, HP Pascal, and HP C are only discussed to the extent that they affect how you use HP Link Editor/XL. See the appropriate operating system or language manual for complete information about those subjects. The following is a partial list of the operating system and language manuals:

Manual Title	Manual Part Number	Number to Use to Order Manual
MPE XL Commands Reference Manual	32650-90003	32650-60002
MPE XL Intrinsics Reference Manual	32650-90028	32650-60013
HP COBOL II/XL Reference Manual	31500-90001	31500-60001
HP COBOL II/XL Reference Manual Supplement	31500-90005	31500-60001
HP COBOL II/XL Programmer's Guide	31500-90002	31500-60002
HP FORTRAN 77/XL Reference Manual	31501-90010	31501-60002
HP FORTRAN 77/XL Programmer's Guide	31501-90011	31501-60004
HP Pascal Reference Manual	31502-90001	31502-60005
HP Pascal Programmer's Guide	31502-90002	31502-60006
HP C/XL Reference Manual	92434-90001	31506-60001
HP C/XL Library Reference Manual	30026-90001	31506-60001
HP C Programmer's Guide	92434-90002	31506-60002
HP Symbolic Debugger/XL User's Guide	31508-90003	31508-60003
HP Symbolic Debugger/XL Quick Reference Guide	31508-90005	31508-60004

Conventions

CASE

In a syntax statement, commands and keywords must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example:

```
SHOWJOB
```

can be entered as any of the following:

```
showjob        ShowJob        SHOWJOB
```

It cannot, however, be entered as any of the following:

```
shojwob        Shojob         SHOW_JOB
```

italics

In a syntax statement, a word in italics represents a parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file:

```
RELEASE filename
```

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

```
( filename):( filename)
```

underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, "yes" is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`:

```
SETMSG { ON  }
       { OFF }
```

Commands listed in braces are called *command lists* throughout this manual.

[ ]

In a syntax statement, brackets enclose optional elements. In the following example, `,TEMP` can be omitted:

PURGE *filename*[,TEMP]

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select *devicename* or *deviceclass* or neither. The elements cannot be repeated.

SHOWDEV [*devicename* / *deviceclass*]

`[ ... ]`

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *itemname* zero or more times. Each instance of *itemname* must be preceded by a comma:

`[,` *itemname*`] [...]`

In the example below, you only use the comma as a delimiter if *itemname* is repeated; no comma is used before the first occurrence of *itemname*:

`[` *itemname*`] [,...]`

`| ... |`

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select `A`, `AB`, `BA` or `B`. The elements cannot be repeated.

$$\left\{ \begin{matrix} \texttt{A} \\ \texttt{B} \end{matrix} \right\} \texttt{| ... |}$$

`...` ⋮

In an example, horizontal or vertical ellipses indicate where portions of the example have been omitted.

␣

In a syntax statement, the space symbol ␣ shows a required blank. In the following example, *modifier* and *variable* must be separated with a blank:

`SET [(` *modifier*`)]␣(` *variable*`);`

The symbol indicates a key on the keyboard. For example, RETURN represents the carriage return key.

CNTL *char*

CNTL *char* indicates a control character. For example, CNTLY means you press the control key and the Y key simultaneously.

base prefixes

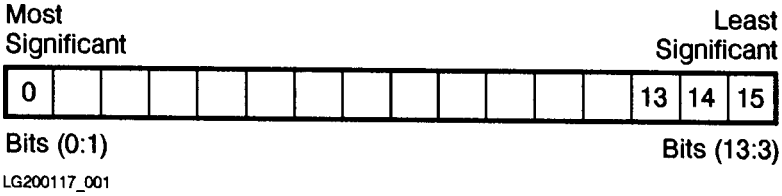
The prefixes %, #, and $ specify the numerical base of the value that follows:

%num specifies an octal number
num specifies a decimal number
$ *num* specifies a hexadecimal number

If no base is specified, decimal is assumed.

`Bits` (*bit:length*)

When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format `Bits` (*bit:length*) *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, `Bits (13:3)` indicates bits 13, 14, and 15:



Contents

Figures

Tables

1

Introduction to HP Link Editor/XL

HP Link Editor/XL is a software tool that prepares compiled programs for execution on HP 3000 Series 900 computers. HP Link Editor/XL also lets you create and maintain libraries containing subprograms that you frequently use.

This chapter explains when to use HP Link Editor/XL and gives an overview of how it works. It also compares it to its MPE V counterpart, the MPE V Segmenter.

When To Use HP Link Editor/XL

Most HP compilers (for example, HP COBOL II) let you compile, link, and execute a program in one step. Or you can compile and link in one step. In these cases, you do not execute HP Link Editor/XL directly to perform the linking function. It is executed automatically.

There are occasions, however, when you may want to compile and link a program yourself in separate steps. You may need to directly execute HP Link Editor/XL when:

- You need to create an *executable program file* (the executable form of a program) which includes several different *modules* that have been compiled separately.
- You need to change one or more of the default parameters associated with the program. For example, you may need to change the execution stack size.
- You want to use one or more library routines in your program. HP Link Editor/XL creates and maintains two kinds of libraries - *relocatable libraries* and *executable libraries*. Routines in relocatable libraries are in their compiled format. Routines in executable libraries are in executable form. Libraries minimize duplication of programming effort while promoting consistency and standardization within a programming organization. They also help to produce easily-maintained programs.

If You're Familiar with MPE V Segmenter . . .

If you've used the MPE V Segmenter, the following list should quickly familiarize you with how it differs from HP Link Editor/XL (for a complete discussion of the differences between MPE V Segmenter and HP Link Editor/XL, see appendix F):

- HP Link Editor/XL processes relocatable object files and relocatable libraries. Relocatable libraries are functionally indistinguishable from user subprogram library (USL) files; they contain one or more relocatable object modules for each program. To simulate the creation of USL files under MPE V (to create a relocatable library), you must compile a program using the `RLFILE` compiler directive.
- HP Link Editor/XL allows more flexibility in the use of relocatable libraries. It searches as many libraries as you need when you create a program.
- Normally, each module in a relocatable library comes from a separate source file. This makes most efficient use of library space and allows each module to be manipulated and linked individually. However, with certain languages, you can create a separate module in the library for each procedure or subprogram in the source file. You do this at compile time by using the `RLFILE` option. (See your compiler manuals for details on `RLFILE`.) This feature simulates the use of relocatable binary modules (RBMs) and may be the most convenient way to place existing source files in a relocatable library.
- HP Link Editor/XL can handle modules of any size. You do not have to segment large modules or a large library.

How HP Link Editor/XL Works

HP Link Editor/XL (the "link editor") processes object code produced by high-level language compilers, such as HP COBOL II. Object code is saved in *relocatable object files*. The link editor *links* relocatable object files for execution by assigning memory locations to them and any external routines that they use.

In addition to creating executable program files, you can use HP Link Editor/XL to create and maintain relocatable and executable libraries.

The next three sections discuss the tasks of creating executable program files, and using relocatable and executable libraries in more detail.

Creating Executable Program Files

Source files can contain one or more programs, procedures or subprograms. If you compile the program using the `RLFILE` option, each program, procedure or subprogram in the source file is a separate *compilation unit* and produces a separate *relocatable object module* in a relocatable library. If you compile without using the `RLFILE` option, the source file (no matter how many programs, procedures or subprograms it contains) is one compilation unit and produces one relocatable object module. After compilation, HP Link Editor/XL transforms the relocatable object file or modules into an executable program file. Figure 1-1 depicts this process.

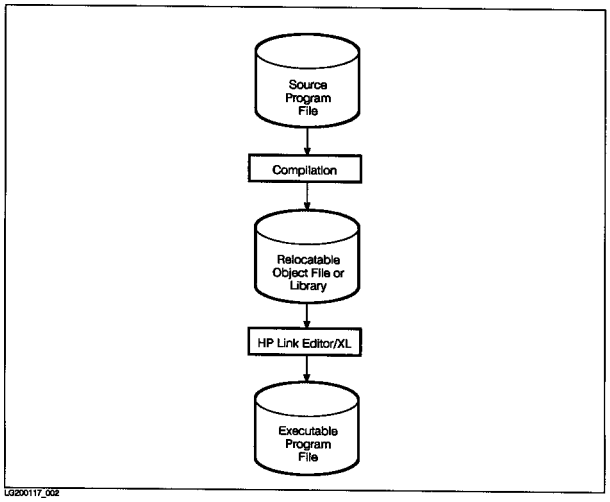


Figure 1-1. Creating an Executable Program File

When the link editor links separately compiled relocatable object files or relocatable modules from a relocatable library, it must be able to find procedure and variable name references (*symbols*) used in the modules. Since compilers process only one compilation unit at a time, they cannot resolve references outside the compilation unit. The unresolved references are called *external references*. Compilers generate a *symbol table* in each relocatable object module which allows the link editor to resolve external references. The symbol table lists all subroutine and variable names that are defined (or *exported*) by that object module. It also lists all subroutine and variable names that are referenced (*imported*) but not defined by that module. The compiler then assigns relative (relocatable) addresses to the exported symbols in the module.

HP Link Editor/XL begins by merging relocatable object modules so that the executable program contains all the code and data in the input files. Figure 1-2 shows how an executable program file is created from three relocatable object files.

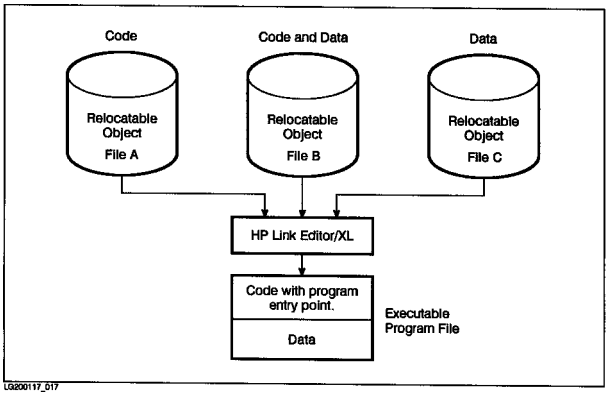


Figure 1-2.
Creating an Executable Program File from Three Relocatable Object Files

If a relocatable library is searched during linking, only those modules that export unresolved symbols are included in the executable program file. When relocatable object modules are merged, many external references are resolved. References that are still unresolved are *external calls* and are resolved when the program is loaded for execution. (The loader subsystem of the MPE XL operating system resolves external calls. It searches executable libraries and the MPE XL System Library.) Finally, the link editor assigns *virtual addresses* to each symbol, assuring that there are no address overlaps. Although the addresses are "final" addresses, they can still be relocated when the program is loaded for execution.

Chapters 3 and 4 give more information about executable program files.

Using Relocatable Libraries

A relocatable library (RL) is a file containing one or more relocatable object modules which can be incorporated into executable program files during linking. Use relocatable libraries for routines that you want to become integral parts of executable program files. When a relocatable object module is linked into a program, the program contains its own unique copy of the module. Changes to the relocatable object module in the library are not reflected in the linked program unless the program is relinked. Routines linked from relocatable libraries can share global data.

Each relocatable library contains a Library Symbol Table (LST) at the beginning of the file. The LST lists each exported symbol in each module of the library.

You can create as many relocatable libraries as you need. You can add modules to a relocatable library from relocatable object files. You can also copy modules from one relocatable library to another or to a relocatable object file, and you can purge and list modules in a relocatable library. For more information on relocatable libraries, see chapters 3 and 5.

Using Executable Libraries

An executable library file contains one or more executable modules that you can load into memory and use at run time. Although the loader searches the executable libraries at run time, you can use the link editor to identify the ones to search in an executable program file. Put routines in executable libraries when the routines are used by programs that are run concurrently. The programs can then use the same physical copy of code.

Note

Routines in executable libraries cannot share global data with the calling program and cannot have outer blocks.

Each executable library contains a Library Symbol Table (LST) at the beginning of the file. During linking, the link editor places unresolved references in an import list in each executable program file. At run time, the loader resolves the symbols in the import list by searching Library Symbol Tables in one or more executable libraries. It builds an External Reference Table (XRT) that tracks externally-called procedures and allows them to be shared.

You can search as many executable libraries as you need. You can add relocatable object modules to an existing library, merging them when necessary. You can copy executable modules from one library to another, purge executable modules from a library, and list the contents of an executable library. For more information about executable libraries, see chapters 3 and 6.

Where To Go from Here

Now that you have read this chapter, you should have a general idea of how HP Link Editor works and the files that it uses.

Continue reading chapters 2 and 3. Chapter 2 gives short examples of common ways to use HP Link Editor/XL and chapter 3 gives details about the files that HP Link Editor/XL uses and how to enter commands.

Use chapters 4, 5 and 6 as reference chapters.

See chapter 7 for information about some of the more advanced ways to use HP Link Editor/XL.

For specific information about how the HP COBOL II, HP FORTRAN 77, HP Pascal and HP C compilers interface with HP Link Editor/XL, see appendices B, C, D, and E respectively.

2

Getting Started with HP Link Editor/XL

This chapter presents simple examples of the basic ways to use HP Link Editor/XL.

The link editor commands are not discussed in detail. Rather, the intent is to give you a quick overview of how to use them to accomplish some common tasks. Chapters 3 through 6 give additional information about the commands that appear in this chapter.

The examples in this chapter show how to do the following:

- Link one relocatable object file.
- Link several relocatable object files.
- Use relocatable libraries.
- Use executable libraries.
- Use both relocatable and executable libraries within a program.

Note 

All of the source files used in the examples in this chapter are listed in the last section of the chapter titled "Sample Programs".

Linking One Relocatable Object File

Link a relocatable object file yourself, rather than have it linked automatically, when you want to use file names or execution-time defaults that vary from the defaults supplied by the compiler. (Execution-time defaults include type checking levels, capability-class attributes, stack size, and heap size).

For example, you can compile, link and execute the HP COBOL II program, `EX1SRC`, using one command:

```
:COB85XLG EX1SRC
```

This command is identical to using the following three commands:

```
:COB85XL EX1SRC
:LINK
:RUN $OLDPASS
```

Both of the above methods use `$OLDPASS` for the relocatable object file and for the executable program file.

Figure 2-1 shows how to compile and link an HP COBOL II source file while explicitly naming the relocatable object file and the executable program file to be used. The `COB85XL` command compiles the source file, `EX1SRC`, producing the relocatable object file, `EX1OBJ`. The MPE XL `LINK` command on the second line in figure 2-1 creates the executable program file, `EX1PROG`.

```
:COB85XL EX1SRC,EX1OBJ
:LINK FROM=EX1OBJ;TO=EX1PROG
```

Figure 2-1. Linking One HP COBOL II Relocatable Object File

Linking Several Relocatable Object Files

When compiling large programs that consist of several separately compiled modules, you must execute HP Link Editor/XL directly. It is a good idea to split a large program into modules because each module can be modified and recompiled independently. You can use the link editor at any time to relink the modules, creating a new executable program file.

Figure 2-2 shows the commands that compile and link two HP FORTRAN 77 source files, `EX2ASRC` and `EX2BSRC`. The HP FORTRAN 77 compiler produces the relocatable object files, `EX2AOBJ` and `EX2BOBJ`. The `:LINK` command creates a new executable program file, `EX2PROG`, consisting of both relocatable object files.

```
:FTNXL EX2ASRC,EX2AOBJ
:FTNXL EX2BSRC,EX2BOBJ
:LINK FROM=EX2AOBJ,EX2BOBJ;TO=EX2PROG
```

Figure 2-2. Linking Two HP FORTRAN 77 Relocatable Object Files

If, after creating the executable program file, you need to update one or more modules, you must modify and recompile those modules, then relink all of them. For example, figure 2-3 shows how to recompile the HP FORTRAN 77 source file, `EX2BSRC` (which was linked in figure 2-2), and to recreate the executable program file, `EX2PROG`. The compile command (`:FTNXL`) overwrites the previous contents of `EX2BOBJ`. `EX2AOBJ` remains unchanged from the previous compilation. (Normally it is a good idea during program development to save relocatable object files. This avoids having to recompile source files that have not changed.)

```
:FTNXL EX2BSRC,EX2BOBJ
:LINK FROM=EX2AOBJ,EX2BOBJ;TO=EX2PROG
```

Figure 2-3. Relinking Two HP FORTRAN 77 Relocatable Object Files

Using a Relocatable Library

Relocatable libraries enable you to share procedures (subprograms) with other subprograms while letting you modify and compile them independently.

The next three sections explain the primary ways to use relocatable libraries. They show how to place the subroutines and functions contained in a source file into a relocatable library and how to use them once they are placed there. To see the relationships of the source files used in the following figures, see the listings for them in the last section of this chapter titled, "Sample Programs".

Creating a Relocatable Library

Figure 2-4 shows how to create a relocatable library and how to add modules to it at the same time. This example also shows how to efficiently organize source files to be added to a relocatable library.

In order to independently compile the subroutines and functions in the HP FORTRAN 77 source file, `EX2BSRC`, it is split into five separate source files: `LIB1SRC`, `LIB2SRC`, `LIB3SRC`, `LIB4SRC`, and `LIB5SRC`. These modules are compiled producing five separate relocatable object files: `LIB1OBJ`, `LIB2OBJ`, `LIB3OBJ`, `LIB4OBJ`, and `LIB5OBJ`. (The modules are placed in different source files because each source file will become one relocatable object module.) After entering HP Link Editor\XL by entering `LINKEDIT` command at the MPE XL prompt, figure 2-4 shows using the link editor `BUILDRL` command to create the relocatable library, `LIBRL`. Then the `ADDRL` command is used to add the relocatable object files to the relocatable library. Finally, use the `EXIT` command to terminate HP Link Editor\XL.

```
:FTNXL LIB1SRC,LIB1OBJ
:FTNXL LIB2SRC,LIB2OBJ
:FTNXL LIB3SRC,LIB3OBJ
:FTNXL LIB4SRC,LIB4OBJ
:FTNXL LIB5SRC,LIB5OBJ
:LINKEDIT
LinkEd> BUILDRL RL=LIBRL
LinkEd> ADDRL FROM=LIB1OBJ,LIB2OBJ,LIB3OBJ,LIB4OBJ,LIB5OBJ
LinkEd> EXIT
```

Figure 2-4. Creating a Relocatable Library and Adding Modules to It

Alternatively, you can have the compiler (rather than the link editor) create the relocatable library and add modules to it. This may be the fastest and easiest choice if you're compiling MPE V source files that contain several subroutines and functions. To do this, use the `$RLFILE` compiler directive (see the *HP FORTRAN 77/XL Reference Manual*).

Searching a Relocatable Library

Figure 2-5 shows how to link the relocatable object file, EX2A0BJ, using the relocatable library, LIBRL, to resolve external references. (This library was created in figure 2-4.) The :LINK command produces the executable program file, EX2PROG.

```
:LINK FROM=EX2A0BJ;RL=LIBRL;TO=EX2PROG
```

Figure 2-5. Searching a Relocatable Library

Updating a Relocatable Library

Figure 2-6 shows how to replace a relocatable object module in a relocatable library. The relocatable module, LIB40BJ, is replaced by a newly-compiled version. The first command in figure 2-6 compiles the HP FORTRAN 77 source file, LIB4SRC. Then, after entering the link editor, the RL command in the third line sets the default relocatable library to LIBRL. To update the existing relocatable module, the old version is purged and a new one added. The PURGERL command purges the existing relocatable module, LIB4SRC, from the library. (The name of the module in the relocatable library is its source file name, LIB4SRC, unless the $RLFILE compiler option is used.) The ADDRL command then adds the updated relocatable object file, LIB40BJ, to the library.

```
:FTNXL LIB4SRC,LIB40BJ
:LINKEDIT
LinkEd> RL RL=LIBRL
LinkEd> PURGERL MODULE=LIB4SRC
LinkEd> ADDRL FROM=LIB40BJ
LinkEd> EXIT
```

Figure 2-6.
Updating a Relocatable Object Module in a Relocatable Library

Alternatively, you can have the compiler (rather than the link editor) update the relocatable library. This may be the fastest and easiest choice if you're compiling MPE V source files that contain several subroutines and functions. To do this, use:

```
:FTNXL LIB4SRC, RL
```

(See the *HP FORTRAN 77/XL Reference Manual* for more information.)

Using an Executable Library

Routines in executable libraries can be shared by many programs; each program uses the same physical copy of code. Though you name executable libraries using the link editor, these libraries are not searched until the executable program file is loaded for execution. (The loader searches the executable libraries, resolving external references, much like the link editor searches relocatable libraries.)

The next three sections show how to create and maintain executable libraries and how to name an executable library to be searched at run time. See the last section in this chapter, "Sample Programs", for listings of the source files used in the examples.

Creating an Executable Library

Figure 2-7 shows how to build an executable library and how to add a module to it at the same time. The link editor `BUILDXL` command creates the executable library, `LIBXL`. The `ADDXL` command adds the relocatable object file, `EX2BOBJ`, to `LIBXL`.

```
:FTNXL EX2BSRC,EX2BOBJ
:LINKEDIT
LinkEd> BUILDXL XL=LIBXL
LinkEd> ADDXL FROM=EX2BOBJ
LinkEd> EXIT
```

Figure 2-7.
Creating an Executable Library and Adding a Module To It

Alternatively, when you add relocatable object modules to an executable library, you can merge several of them into one module. Figure 2-8 shows how to merge five relocatable object files (LIB1OBJ, LIB2OBJ, LIB3OBJ, LIB4OBJ, and LIB5OBJ) into one executable module in the library. (The name of the new executable module is the source file name of the first relocatable object file added, LIB1SRC.)

```
:FTNXL LIB1SRC,LIB1OBJ
:FTNXL LIB2SRC,LIB2OBJ
:FTNXL LIB3SRC,LIB3OBJ
:FTNXL LIB4SRC,LIB4OBJ
:FTNXL LIB5SRC,LIB5OBJ
:LINKEDIT
LinkEd> BUILDXL XL=LIBXL
LinkEd> ADDXL FROM=LIB1OBJ,LIB2OBJ,LIB3OBJ,LIB4OBJ,LIB5OBJ;MERGE
LinkEd> EXIT
```

Figure 2-8.
Creating an Executable Module From Several Relocatable Object Files

Searching an Executable Library

Figure 2-9 shows how to name an executable library to be searched at run time. The :LINK command links the relocatable object file, EX2AOBJ, producing the executable program file, EX2PROG. The XL= option names the executable library, LIBXL, to search at run time.

```
:LINK FROM=EX2AOBJ;TO=EX2PROG;XL=LIBXL
```

Figure 2-9. Naming an Executable Library To Search

Updating an Executable Library

Figure 2-10 shows how to replace an executable module in an executable library. The executable module, `EX2B0BJ` (created in figure 2-7), is replaced by a newly-compiled version. Once the HP FORTRAN 77 source file, `EX2BSRC`, is recompiled (line 1), the link editor `XL` command on the third line sets the default executable library to `LIBXL`. To update the existing executable module, the old version is purged and a new one added. The `PURGEXL` command purges the existing module, `EX2BSRC`, from the library. (The name of the module in an executable library is the same as its source file name, `EX2BSRC`, unless the `$RLFILE` compiler option is used.) The `ADDXL` command then adds the updated relocatable object file, `EX2B0BJ`, to the library.

```
:FTNXL EX2BSRC,EX2B0BJ
:LINKEDIT
LinkEd> XL XL=LIBXL
LinkEd> PURGEXL MODULE=EX2BSRC
LinkEd> ADDXL FROM=EX2B0BJ
LinkEd> EXIT
```

Figure 2-10.
Updating an Executable Module in an Executable Library

Using Both Relocatable and Executable Libraries within a Program

A complex program may use a combination of several types of libraries, and additionally, might also require certain calls to system intrinsic routines to complete its tasks.

The next three sections show how to link a program that incorporates both a relocatable library and an executable library, as well as calls to specific system intrinsics.

Creating the Relocatable Library

Figure 2-11 shows how to create a relocatable library using the HP Pascal source file, `EX3CSRC`. The relocatable object file created is named `EX3COBJ`. The commands entered are identical to those shown in figure 2-4 with the exception that `:PASXL` is used as the compile command.

```
:PASXL EX3CSRC,EX3COBJ
:LINKEDIT
LinkEd> BUILDRL RL=EX3LIBRL
LinkEd> ADDRL FROM=EX3COBJ
LinkEd> EXIT
```

Figure 2-11. Creating a Relocatable Library and Adding a Module

Creating the Executable Library

Figure 2-12 shows how to create an executable library using the HP Pascal source file, `EX3DSRC`. First, a relocatable object file is created named `EX3DOBJ`. Next, the `BUILDXL` command creates the executable library `EX3LIBXL`. Finally, `ADDXL` adds the relocatable object file `EX3DOBJ` to the executable library. The commands entered are identical to those shown in figure 2-7 with the exception that `:PASXL` is used as the compile command.

```
:PASXL EX3DSRC,EX3DOBJ
:LINKEDIT
LinkEd> BUILDXL XL=EX3LIBXL
LinkEd> ADDXL FROM=EX3DOBJ
LinkEd> EXIT
```

Figure 2-12. Creating an Executable Library and Adding a Module

Linking with Libraries and Relocatable Object Files

Figure 2-13 shows how to link object files with a relocatable library using the link editor LINK command while specifying a needed executable library with a parameter to the :RUN command. First, the :PASXL compile command is used to create a relocatable object named EX3AOBJ from the source file EX3ASRC. EX3ASRC represents the main program; see figure 2-23 in the "Sample Programs". Next, a subprogram EX3BSRC is compiled, producing the relocatable object file EX3BOBJ; see figure 2-24 for the source. Its code includes a procedure call to the system intrinsic who; who is used to determine the device number of the current user.

In order to link the above two relocatable object files with the relocatable library module created in figure 2-11, the link editor LINK command is used. After exiting the link editor, the :RUN command executes the program. Note that EX3AOBJ and EX3BOBJ as well as any modules needed from the relocatable library, EX3LIBRL, will be included in the resulting program file. The program file created will still have unsatisfied symbols because the modules on the command line use but do not define those symbols. These symbols will be resolved with the EX3LIBXL or the system libraries; the XL= parameter of the :RUN command is used to resolve these symbols at run time.

Specifically, the procedures and system intrinsics referenced in figures 2-23 through 2-26 are defined as follows:

p1 is defined in EX3BOBJ.

p2 is defined in EX3LIBRL.

p3 is defined in EX3LIBXL.

who and dateline are defined in NL.PUB.SYS or XL.PUB.SYS.

Both p1 and p2 are resolved in the program file whereas p3, who, and dateline are left as unsatisfied symbols in the program file which are resolved at run time.

```
:PASXL EX3ASRC,EX3AOBJ
:PASXL EX3BSRC,EX3BOBJ

:LINKEDIT
LinkEd> LINK FROM=EX3AOBJ,EX3BOBJ;RL=EX3LIBRL
LinkEd> EXIT
:RUN $OLDPASS;XL="EX3LIBXL"
```

Figure 2-13. Linking Libraries and Relocatable Object Files

As an alternative to specifying the executable libraries needed on the :RUN command, it might be useful to specify them at the time of linking. This allows you to simply invoke the :RUN command with no knowledge of the executable libraries needed at run time.

```
:LINKEDIT
LinkEd> LINK FROM=EX3AOBJ,EX3BOBJ;RL=EX3LIBRL;XL="EX3LIBXL"
LinkEd> EXIT
:RUN $OLDPASS
```

Figure 2-14. Alternative Specification for Executable Libraries

Sample Programs

This section lists the HP COBOL II, HP FORTRAN 77, and HP Pascal source files used in the examples in the previous sections of this chapter.

These source files:	are listed in:
`EX1SRC`	Figure 2-15
`EX2ASRC`	Figure 2-16
`EX2BSRC`	Figure 2-17
`LIB1SRC`	Figure 2-18
`LIB2SRC`	Figure 2-19
`LIB3SRC`	Figure 2-20
`LIB4SRC`	Figure 2-21
`LIB5SRC`	Figure 2-22
`EX3ASRC`	Figure 2-23
`EX3BSRC`	Figure 2-24
`EX3CSRC`	Figure 2-25
`EX3DSRC`	Figure 2-26

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EX1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IFILE              ASSIGN "IFILE".
    SELECT PFILE              ASSIGN "PFILE".
DATA DIVISION.
FILE SECTION.
FD  IFILE.
01  IREC.
    05  NAME                 PIC X(30).
    05  SOC-SEC              PIC X(9).
    05  HIRE-DATE.
        10 MO                PIC XX.
        10 DA                PIC XX.
        10 YR                PIC XX.
    05  SALARY               PIC S9(6).
    05                       PIC X(29).
FD  PFILE.
01  PREC.
    05  SOC-SEC              PIC X(9).
    05                       PIC XX.
    05  NAME                 PIC X(30).
    05                       PIC XX.
    05  HIRE-DATE.
        10  MO               PIC XX.
        10                   PIC X.
        10  DA               PIC XX.
        10                   PIC X.
        10  YR               PIC XX.
    05                       PIC X(81).
01  HREC.
    05  HSOC-SEC             PIC X(11).
    05  HNAME                PIC X(32).
    05  HHIRE-DATE           PIC X(89).
```

Figure 2-15. The HP COBOL II Source File, EX1SRC

```
WORKING-STORAGE SECTION.
01  LNCNT                    PIC S9(4) BINARY VALUE 60.
01  W-DATE.
    05  WYR                  PIC XX.
    05                       PIC X(4).
PROCEDURE DIVISION.
P1.
    ACCEPT W-DATE FROM DATE.
    OPEN INPUT IFILE OUTPUT PFILE.
    PERFORM WITH TEST AFTER UNTIL SOC-SEC OF IREC = ALL "9"
      READ IFILE
          AT END MOVE ALL "9" TO SOC-SEC OF IREC
          NOT AT END
              IF WYR = YR OF IREC THEN
                  ADD 1 TO LNCNT
                  IF LNCNT > 50 PERFORM HEADINGS END-IF
                  MOVE SPACES TO PREC
                  MOVE CORR IREC TO PREC
                  WRITE PREC AFTER ADVANCING 1 LINE
              END-IF
          END-READ
      END-PERFORM
    CLOSE IFILE PFILE
    STOP RUN.
HEADINGS.
    MOVE "SOC SEC NO" TO HSOC-SEC.
    MOVE "NAME" TO HNAME.
    MOVE "HIRE DATE" TO HHIRE-DATE.
    WRITE PREC AFTER ADVANCING PAGE.
    MOVE 0 TO LNCNT.
```

Figure 2-15. The HP COBOL II Source File, EX1SRC (Continued)

```
C   This program prints an amortization table for a loan
C   with regular payments on the first of each month.
C   It calculates prepaid interest from the current
C   date until the end of the current month, and begins
C   the amortization at the beginning of the next month.
C   Input to the program is the current date (in month,
C   day, year form), the principal amount, annual interest
C   rate, and the term of the loan in years.

      PROGRAM EX2
      INTEGER TODAY, NXTMON, TERM
      DOUBLE PRECISION PRIN, RATE, PREPD, PAYMNT, PCT
      INTEGER JULIAN
      DOUBLE PRECISION AMORT
      COMMON MONTH, DAY, YEAR
      INTEGER MONTH, DAY, YEAR

      READ (5,*) MONTH, DAY, YEAR
      READ (5,*) PRIN, RATE, TERM

C   Determine the number of days remaining in the current
C   month.  The Julian dates for today and the first of the
C   next month are used for this calculation.

      TODAY = JULIAN(MONTH, DAY, YEAR)
      DAY = 1
      CALL ADDDAT(MONTH, DAY, YEAR, 1, 0, 0)
      NXTMON = JULIAN(MONTH, DAY, YEAR)

C   Calculate the prepaid interest and the monthly payments.
C   The prepaid interest is calculated as simple interest.

      PREPD = PRIN * (NXTMON-TODAY) * (RATE/365.0D0)
      PAYMNT = AMORT(PRIN, RATE/12.0D0, TERM*12)
      PCT = RATE * 100.0D0
      WRITE (6, 100) PREPD, PRIN, PCT, TERM, PAYMNT
  100 FORMAT ('1', 'Prepaid Interest:  ', F10.2/
     *        '0', 'Principal:         ', F10.2/
     *        ' ', 'Interest Rate:     ', F10.2, '%'/
     *        ' ', 'Number of Years:   ', I7/
     *        ' ', 'Monthly Payment:   ', F10.2)

      CALL PRTTAB(PRIN, RATE/12.0D0, TERM*12, PAYMNT)
      STOP
      END
```

Figure 2-16. The HP FORTRAN 77 Source File, EX2ASRC

```
C   Print the amortization table

      SUBROUTINE PRTTAB(PRIN, RATE, TERM, PAYMNT)
      DOUBLE PRECISION PRIN, RATE, PAYMNT
      INTEGER TERM
      DOUBLE PRECISION ACCINT, PPRIN, PINT, RPRIN
      CHARACTER*3 DW, WKDAY
      COMMON MONTH, DAY, YEAR
      INTEGER MONTH, DAY, YEAR
      ACCINT = 0.0
      WRITE (6, 101)
  101 FORMAT ('0', '                 Beginning  Payment to  ',
     *            'Payment to   Accumulated     Remaining'/
     *         ' ', '   Due Date      Principal   Principal  ',
     *            '  Interest      Interest     Principal')
      DO 1 I = 1, TERM
         CALL ADDDAT(MONTH, DAY, YEAR, 1, 0, 0)
         PINT = PRIN * RATE
         PPRIN = PAYMNT - PINT
         ACCINT = ACCINT + PINT
         RPRIN = PRIN - PPRIN
         DW = WKDAY(MONTH, DAY, YEAR)
         WRITE (6, 102) DW, MONTH, DAY, YEAR, PRIN, PPRIN, PINT,
     *                  ACCINT, RPRIN
  102    FORMAT (' ', A3, ' ', I2, '/', I2, '/', I4, 2X, F10.2,
     *          4X, F8.2, 4X, F8.2, 4X, F10.2, 4X, F10.2)
         PRIN = RPRIN
    1 CONTINUE
      RETURN
      END
```

Figure 2-16.
The HP FORTRAN 77 Source File, EX2ASRC (Continued)

```
C   JULIAN returns the Julian date for the given month, day,
C   and year.  The Julian date calculated here is valid from
C   Mar 1, 1900 to Feb 28, 2100.  It is the astronomical date
C   for noon on that day.

      INTEGER FUNCTION JULIAN(MONTH, DAY, YEAR)
      INTEGER MONTH, DAY, YEAR
      PARAMETER (J1900 = 2415020)
      INTEGER JAN1, MON1
      INTEGER MTABLE(12)
      DATA MTABLE /0,31,59,90,120,151,181,212,243,273,304,334/

C   Find Julian date for Jan 1 of given year.

      JAN1 = J1900 + INT(365.25D0 * (YEAR-1900) + 0.75)

C   Find number of days to 1st of given month.

      MON1 = MTABLE(MONTH)
      IF (MOD(YEAR,4) .EQ. 0 .AND. MONTH .GE. 3) MON1 = MON1 + 1
      JULIAN = JAN1 + MON1 + DAY - 1
      RETURN
      END


C   MDY converts a Julian date to month, day, year format.

      SUBROUTINE MDY(JDATE, MONTH, DAY, YEAR)
      INTEGER JDATE, MONTH, DAY, YEAR, YDATE
      PARAMETER (J1900 = 2415020)
      INTEGER MTABLE(12)
      DATA MTABLE /31,28,31,30,31,30,31,31,30,31,30,31/
      YEAR = 1900 + INT((JDATE-J1900) / 365.25D0)
      DAY = JDATE - JULIAN(1, 1, YEAR) + 1
      MTABLE(2) = 28
      IF (MOD(YEAR,4) .EQ. 0) MTABLE(2) = 29
      MONTH = 1
    1 IF (DAY .LE. MTABLE(MONTH) .OR. MONTH .GE. 12) GOTO 2
        DAY = DAY - MTABLE(MONTH)
        MONTH = MONTH + 1
```

Figure 2-17. The HP FORTRAN 77 Source File, EX2BSRC

```
         GOTO 1
      2 RETURN
        END

C     WKDAY returns a 3-letter name of the day of the week
C     given the month, day, and year.

        CHARACTER*3 FUNCTION WKDAY(MONTH, DAY, YEAR)
        INTEGER MONTH, DAY, YEAR, JDATE, DW
        INTEGER JULIAN
        CHARACTER*3 DAYTAB(7)
        DATA DAYTAB /'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'/
        JDATE = JULIAN(MONTH, DAY, YEAR)
        DW = MOD(JDATE+1, 7)
        WKDAY = DAYTAB(DW+1)
        RETURN
        END


C     ADDDAT adds the given number of months, days, and years
C     to the date supplied in the first three arguments.

        SUBROUTINE ADDDAT(MONTH, DAY, YEAR, NMONS, NDAYS, NYRS)
        INTEGER MONTH, DAY, YEAR, NMONS, NDAYS, NYRS
        INTEGER JDATE, JULIAN
        YEAR = YEAR + NYRS
        MONTH = MONTH + NMONS

        IF (MONTH .GT. 12) THEN
           YEAR = YEAR + (MONTH-1)/12
           MONTH = MOD(MONTH-1,12) + 1
        END IF

        IF (NDAYS .GT. 0) THEN
           JDATE = JULIAN(MONTH, DAY, YEAR) + NDAYS
           CALL MDY(JDATE, MONTH, DAY, YEAR)
        END IF

        RETURN
        END
```

Figure 2-17.
The HP FORTRAN 77 Source File, EX2BSRC (Continued)

```
C     AMORT returns the periodic payment for an amortized loan
C     given principal, periodic interest rate, and term.

      DOUBLE PRECISION FUNCTION AMORT(PRIN, RATE, TERM)
      DOUBLE PRECISION PRIN, RATE
      INTEGER TERM

      AMORT = PRIN * RATE / (1.0 - (1.0+RATE) ** (-TERM))

      RETURN
      END
```

Figure 2-17.
The HP FORTRAN 77 Source File, EX2BSRC (Continued)

```
C   JULIAN returns the Julian date for the given month, day,
C   and year.  The Julian date calculated here is valid from
C   Mar 1, 1900 to Feb 28, 2100.  It is the astronomical date
C   for noon on that day.

      INTEGER FUNCTION JULIAN(MONTH, DAY, YEAR)
      INTEGER MONTH, DAY, YEAR
      PARAMETER (J1900 = 2415020)
      INTEGER JAN1, MON1
      INTEGER MTABLE(12)
      DATA MTABLE /0,31,59,90,120,151,181,212,243,273,304,334/

C   Find Julian date for Jan 1 of given year.

      JAN1 = J1900 + INT(365.25D0 * (YEAR-1900) + 0.75)

C   Find number of days to 1st of given month.

      MON1 = MTABLE(MONTH)
      IF (MOD(YEAR,4) .EQ. 0 .AND. MONTH .GE. 3) MON1 = MON1 + 1
      JULIAN = JAN1 + MON1 + DAY - 1
      RETURN
      END
```

Figure 2-18. The HP FORTRAN 77 Source File, LIB1SRC

```
C   MDY converts a Julian date to month, day, year format.

      SUBROUTINE MDY(JDATE, MONTH, DAY, YEAR)
      INTEGER JDATE, MONTH, DAY, YEAR, YDATE
      PARAMETER (J1900 = 2415020)
      INTEGER MTABLE(12)
      DATA MTABLE /31,28,31,30,31,30,31,31,30,31,30,31/
      YEAR = 1900 + INT((JDATE-J1900) / 365.25D0)
      DAY = JDATE - JULIAN(1, 1, YEAR) + 1
      MTABLE(2) = 28
      IF (MOD(YEAR,4) .EQ. 0) MTABLE(2) = 29
      MONTH = 1
    1 IF (DAY .LE. MTABLE(MONTH) .OR. MONTH .GE. 12) GOTO 2
         DAY = DAY - MTABLE(MONTH)
         MONTH = MONTH + 1
         GOTO 1
    2 RETURN
      END
```

Figure 2-19. The HP FORTRAN 77 Source File, LIB2SRC

```
C   WKDAY returns a 3-letter name of the day of the week
C   given the month, day, and year.

      CHARACTER*3 FUNCTION WKDAY(MONTH, DAY, YEAR)
      INTEGER MONTH, DAY, YEAR, JDATE, DW
      INTEGER JULIAN
      CHARACTER*3 DAYTAB(7)
      DATA DAYTAB /'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'/
      JDATE = JULIAN(MONTH, DAY, YEAR)
      DW = MOD(JDATE+1, 7)
      WKDAY = DAYTAB(DW+1)
      RETURN
      END
```

Figure 2-20. The HP FORTRAN 77 Source File, LIB3SRC

```
C   ADDDAT adds the given number of months, days, and years
C   to the date supplied in the first three arguments.

    SUBROUTINE ADDDAT(MONTH, DAY, YEAR, NMONS, NDAYS, NYRS)
    INTEGER MONTH, DAY, YEAR, NMONS, NDAYS, NYRS
    INTEGER JDATE, JULIAN
    YEAR = YEAR + NYRS
    MONTH = MONTH + NMONS

    IF (MONTH .GT. 12) THEN
       YEAR = YEAR + (MONTH-1)/12
       MONTH = MOD(MONTH-1,12) + 1
    END IF

    IF (NDAYS .GT. 0) THEN
       JDATE = JULIAN(MONTH, DAY, YEAR) + NDAYS
       CALL MDY(JDATE, MONTH, DAY, YEAR)
    END IF

    RETURN
    END
```

Figure 2-21. The HP FORTRAN 77 Source File, LIB4SRC

```
C     AMORT returns the periodic payment for an amortized loan
C     given principal, periodic interest rate, and term.

      DOUBLE PRECISION FUNCTION AMORT(PRIN, RATE, TERM)
      DOUBLE PRECISION PRIN, RATE
      INTEGER TERM

      AMORT = PRIN * RATE / (1.0 - (1.0+RATE) ** (-TERM))

      RETURN
      END
```

Figure 2-22. The HP FORTRAN 77 Source File, LIB5SRC

```
{This program queries the system using defined system
 intrinsics in order to print the device number, the user,
 group, and account name, and the current date and time.}

program myprog (input, output);

type
   pac1 = packed array [1..10] of char;
   pac2 = packed array [1..30] of char;

var
   user, group, acct: pac1;
   date             : pac2;
   dev              : shortint;

{"external" signifies these routines will be
 found in other modules.}
procedure p1(var dev: shortint); external;

procedure p2(var user, group, acct: pac1); external;

procedure p3(var date: pac2); external;

begin
    p1(dev);
    p2(user, group, account);
    p3(date);

   {output the required information}
    write('Device number', dev, 'is logged on as ');
    write(user, '.', group, '.', acct);
    writeln('on', date);
end.
```

Figure 2-23. The HP Pascal Source File, EX3ASRC

```
$subprogram$
program sub1;

{Here, who is specified as an intrinsic.}
procedure who; intrinsic;

{This procedure calls the system intrinsic who
 to return the device the current user is logged
 on to. who command defaults are used for the
 1st 7 parameters, as documented in the MPE XL
 Intrinsics Reference Manual.}
procedure p1(var dev: shortint);
begin
   who(,,,,,,,dev);
end;

{The main program is defined elsewhere.}

begin
end.
```

Figure 2-24. The HP Pascal Source File, EX3BSRC

```
$subprogram$

program sub2;

type
   pac1 = packed array [1..10] of char;

procedure who; intrinsic;

{This procedure calls the system intrinsic who to
 return the name of the current user, group, and account.
 who command defaults are used for the 1st 3 parameters,
 as documented in the MPE XL Intrinsics Reference Manual.}
procedure p2(var user, group, acct: pac1);
begin
   who(,,,user, group, acct);
end;

{The main program is defined elsewhere.}

begin
end.
```

Figure 2-25. The HP Pascal Source File, EX3CSRC

```
$subprogram$

program sub3;

type
   pac2 = packed array [1..30] of char;

{dateline is specified as an intrinsic}
procedure dateline; intrinsic;

{This procedure calls the system intrinsic dateline
 to return the current date and time. dateline is
 documented in the MPE XL System Intrinsics Manual.}
procedure p3(var date: pac2);
begin
   dateline(date);
end;

{The main program is defined elsewhere.}

begin
end.
```

Figure 2-26. The HP Pascal Source File, EX3DSRC

3

Using HP Link Editor/XL Files and Commands

This chapter discusses the files that HP Link Editor/XL uses when it links a program, with or without libraries, and when it builds and maintains relocatable and executable libraries.

The chapter also explains how to start and end HP Link Editor/XL and the rules for entering commands.

The Files Used By HP Link Editor/XL

Figure 3-1 shows the files that the link editor uses. The next seven sections in this chapter discuss them in detail.

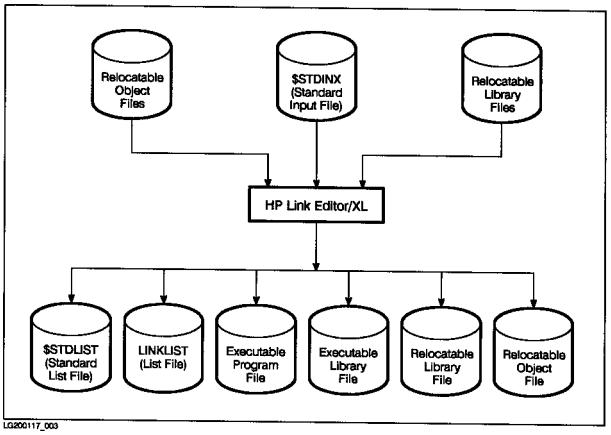


Figure 3-1. The Files Used by HP Link Editor/XL

The Relocatable Object File

A relocatable object file is output from compilation. It can also be created by the link editor EXTRACTRL command. When it is produced by a compiler, it consists of one relocatable object module regardless of the number of procedures or subprograms the source file contains. When it is produced by the EXTRACTRL command, it can consist of many relocatable object modules; however, these modules cannot be accessed individually with the LINK command. Relocatable object files are binary files having the filecode NMOBJ (1461).

Relocatable object files are input to the LINK, LISTOBJ, ADDRL, and ADDXL commands and are created by the EXTRACTRL command. These commands are discussed in chapters 4, 5 and 6.

To use a relocatable object file as input, you must have read access to it. To create a relocatable object file, you must have save access to the group where the file is located.

The $STDINX File

HP Link Editor/XL reads its commands from the standard input file, $STDINX. For an interactive session, this is the terminal keyboard. For a batch job, it is the job stream file.

If you wish, you can change the standard assignment for $STDINX. Enter a :RUN command with the STDIN option, naming an unnumbered ASCII file. The file must contain valid HP Link Editor/XL commands. For example, to use the file SCRIPT as the standard input file, enter the command:

```
:RUN LINKEDIT.PUB.SYS;STDIN=SCRIPT
```

If you start the link editor using the :LINK command or if you execute the link editor by passing a command in the INFO string of the :RUN command, $STDINX is not used. Instead, the single command is executed and the link editor terminates. (See the section in this chapter titled, "Starting HP Link Editor/XL", for information on using :LINK and :RUN to execute the link editor.)

The Relocatable Library File

A relocatable library is a collection of relocatable object modules and a Library Symbol Table. The Library Symbol Table maps exported symbols in each relocatable module. Relocatable library files are binary files, and have the filecode NMRL (1033).

You create a relocatable library using the BUILDRL command and you add modules to it with the ADDRL command (or you can have the compiler create and add modules to a library by using the RLFILE or RLINIT compiler directives). To copy relocatable modules from one relocatable library to another, use the COPYRL command. The PURGERL command deletes modules and the EXTRACTRL command copies selected modules to a relocatable object file. The CLEANRL, HIDERL and REVEALRL commands modify relocatable libraries and the LISTRL command lists the contents of them. All of these relocatable library commands are discussed in chapter 5.

To create a relocatable library file, you must have save access to the group where the file will be located. To modify an existing relocatable library, you must have write access to the file. To list the contents of a library or to copy modules out of it, you must have read access to it.

The $STDLIST File

HP Link Editor/XL writes all prompts, errors, and informational messages to the standard list file, $STDLIST. When running in an interactive session, your terminal is the device used for $STDLIST. When running a batch job, the output spoolfile is used.

If you wish, you can change the standard assignment for $STDLIST. Enter a :RUN command with the STDLIST option to name the file or device to use. (Note that when you do this and run interactively, command prompts do not appear on the screen.) For example, the following command sends link editor output to the printer:

```
:FILE LINKOUT;DEV=LP
:RUN LINKEDIT.PUB.SYS;STDLIST=*LINKOUT
```

The LINKLIST File

HP Link Editor/XL listings and maps are sent to the file, LINKLIST, instead of to the standard list file, $STDLIST. The following listings and maps are sent to LINKLIST:

- The symbol map produced by the MAP option of the LINK command (see chapter 4).
- The listing produced by the LISTPROG command (see chapter 4).
- The listing produced by the LISTOBJ command (see chapter 4).
- The listing produced by the LISTRL command (see chapter 5).
- The listing produced by the MAP option of the ADDXL command (see chapter 6).
- The listing produced by the LISTXL command (see chapter 6).

LINKLIST output is normally sent to the $STDLIST device. You can redirect LINKLIST to another file or device by using the MPE XL :FILE command. For example, the following commands send the listing of the relocatable library, LIBRL, to the line printer:

```
:FILE LINKLIST;DEV=LP
:LINKEDIT
LinkEd> LISTRL RL=LIBRL
LinkEd> EXIT
```

The Executable Program File

Executable program files are created by the LINK command. They are in binary format - ready to be loaded into memory and executed by the MPE XL :RUN command. Executable program files have the filecode NMPRG (1030).

You can use the LISTPROG command (see chapter 4) to list the symbol table of an executable program file.

To use LINK to create an executable program file, you must have save access to the group where the file is located. To list an executable program file, you must have read access to the file.

The Executable Library File

An executable library is a collection of executable modules and a Library Symbol Table. The Library Symbol Table maps exported and imported symbols in each executable module. Executable library files are binary files having the filecode NMXL (1032).

You use the BUILDXL command to create an executable library. Executable modules are added to a library with the ADDXL command. Modules are copied from library to library with the COPYXL command and deleted using the PURGEXL command. The CLEANXL command compacts executable libraries, and the LISTXL command lists the contents of them. All of these executable library commands are discussed in chapter 6.

To create an executable library, you must have save access to the group where the file is located. To modify an existing library, you must have write access to the file. To list the contents of a library or to copy modules from it, you must have read access to the file.

Starting HP Link Editor/XL

There are three ways to start HP Link Editor/XL:

- Enter the `:LINKEDIT` command at the MPE XL prompt:

  ```
  :LINKEDIT
  ```

  HP Link Editor/XL displays its command line prompt, `LinkEd>`, and waits for you to enter a command. Each time you enter a command, it is executed and you are prompted to enter another. This continues until you end the link editor with the `EXIT` command (see the next section).

- Enter a `LINK` command at the MPE XL prompt:

  ```
  :LINK FROM=EX1OBJ;TO=EX1PROG;RL=LIBRL
  ```

  The link editor performs the link operation, then ends. The `LINK` command is discussed in chapter 4 and has the same syntax when used at the MPE XL command level as when entered at the link editor prompt.

- Enter a `:RUN` or a `:LINKEDIT` command, with an `INFO` string, at the MPE XL prompt. The `INFO` string may contain one link editor command:

  ```
  :RUN LINKEDIT.PUB.SYS;INFO="LISTRL RL=LIBRL"
  ```

  Or you can use the short form:

  ```
  :LINKEDIT "LISTRL RL=LIBRL"
  ```

  The command in the `INFO` string is executed and the link editor ends. You can execute any link editor command in this manner.

  For complete information on the `:RUN` command, see the *MPE XL Commands Reference Manual*.

Ending HP Link Editor/XL

Three events terminate HP Link Editor/XL:

- When you explicitly end HP Link Editor/XL, by entering the `EXIT` command:

  ```
  LinkEd> EXIT
  ```

  You can abbreviate the `EXIT` command as `E`, `EX`, or `EXI`. (The commands `QUIT`, `Q` and `BYE` also terminate HP Link Editor/XL.)

- When end-of-file in `$STDINX` is encountered.

  End-of-file can occur when `$STDINX` is redirected to a disc file or when an `:EOD` command is encountered in it.

- When an error occurs in a batch job.

  An error message is printed, the system Job Control Word (JCW) is set to indicate a fatal error and the link editor ends.

Entering HP Link Editor/XL Commands

The following sections discuss the rules for entering HP Link Editor/XL commands. (The link editor reads commands from the standard input file, `$STDINX`. For more information on `$STDINX` see "The $STDINX File" section in this chapter.)

Using Upper and Lower Case Characters

When entering HP Link Editor/XL commands, you can enter them in either upper case or lower case, or a mixture of the two.

There is one instance when case is significant. You must enter entry point (procedure) names exactly as they are found in the relocatable object modules. You specify entry point names using the `ENTRY` parameter of the following commands: `LINK`, `COPYRL`, `EXTRACTRL`, `HIDERL`, `LISTRL`, `PURGERL`, `REVEALRL`, `ADDXL`, `COPYXL`, `LISTXL` and `PURGEXL`. In general, case insensitive languages (including HP FORTRAN 77 and HP Pascal) convert procedure names to lower case. For those languages, specify entry point names in lower case.

Using Keyword or Positional Parameters

HP Link Editor/XL commands described in chapters 4, 5, and 6 are shown in "keyword" form. That is, command parameters are preceded by a keyword followed by an equal sign (`=`). For example, the relocatable object file used by the `LINK` command is preceded by the `FROM=` keyword. You can enter commands using keywords or without them (see the next paragraph). When you use command keywords, you can enter command parameters in any order. Separate keyword parameters by semicolons.

Under certain conditions, you can omit parameter keywords (and the equal sign), and enter them in "positional" form. When you do this, you must enter the parameters in the same order as shown in the command description in this manual. Separate positional parameters by commas or spaces.

You cannot use positional parameters:

- To specify a list of file names, unless the list is in an indirect file. (See the section which follows in this chapter titled "Using Indirect Files" for more information on indirect files.)
- After keyword parameters (keyword parameters must follow all positional parameters).

The following three `LINK` commands use keywords and positional parameters and are all equivalent:

```
LinkEd> LINK FROM=^OBJLIST; TO=PROGFILE; RL=LIBRL; MAP
LinkEd> LINK FROM=^OBJLIST; RL=LIBRL; MAP; TO=PROGFILE
LinkEd> LINK ^OBJLIST,PROGFILE,LIBRL; MAP
```

You can use a positional parameter without preceding it by previous parameters for the command. You must supply the appropriate number of commas to show that the parameters are omitted. The

following command uses default values for the first two parameters of the LINK command, but specifies a value (LIBRL) for the third:

```
:LINK ,,LIBRL
```

Continuing Commands from One Line to Another

If you need more than one line to enter a command, end all lines except the last with an ampersand (&).

This example shows how to enter a LINK command on three lines:

```
LinkEd> LINK FROM=SAMPOBJ1, SAMPOBJ2, SAMPOBJ3, SAMPOBJ4, SAMPOBJ5, &
SAMPOBJ6, SAMPOBJ7; TO=SAMPPROG; RL=LIBRL1, LIBRL2; &
XL=MYXL; MAP
```

Do not use ampersands in indirect files. (See the next section for information about indirect files.)

Using Indirect Files

An indirect file is an ASCII file containing a list of names. You use indirect file names in link editor commands instead of entering each name contained in the file individually. (You can also mix indirect and regular file names in commands.) Indirect files are a convenient way to enter a long list of names for commands that you use frequently. You can use indirect files only with the commands and their parameters shown below (the parameters are shown in parentheses):

LINK	(FROM=, RL=, and XL=)
ADDRL	(FROM= and RL=)
COPYRL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
EXTRACTRL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
LISTRL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
PURGERL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
ADDXL	(FROM=, RL=, ENTRY=, MODULE=, BLOCKDATA= and LSET=)
COPYXL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
LISTXL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
PURGEXL	(ENTRY=, MODULE=, BLOCKDATA= and LSET=)
ALTPROG	(PROG=)

When you create an indirect file, enter one or more names on each line (you can use as many lines as necessary). Separate the names on each line by spaces or commas. Make sure that the link editor has read access to the file.

You can use comments in an indirect file. Use a pound sign (#) at the beginning of a line or after a space to denote that the following characters are part of a comment. Comments are ignored by the link editor. If you want to use a pound sign as a character rather than to denote the beginning of a comment, use double pound signs (##).

To use an indirect file in a command, precede its name by a caret (^). For example, assume that the ASCII file, OBJLIST, contains these lines:

```
LIB1OBJ
LIB2OBJ
LIB3OBJ
LIB4OBJ
LIB5OBJ
```

The following commands use the indirect file, OBJLIST, to add the five relocatable object files shown above to the relocatable library, LIBRL:

```
:LINKEDIT
LinkEd> BUILDRL RL=LIBRL
LinkEd> ADDRL FROM=^OBJLIST
LinkEd> EXIT
```

Re-executing HP Link Editor/XL Commands

Similar to the MPE XL Command Interpreter, HP Link Editor/XL keeps a list of the last 20 commands you entered. You can view this list by entering the `LISTREDO` command. Then, you can enter the `DO` or `REDO` command to execute a command from the list. You use `DO` and `REDO` the same way you do as an operating system command except you cannot use the `EDIT=` parameter.

The following example shows how you can use the `REDO` command to correct a simple typing mistake.

```
LinkEd> LISTRL XL=Linedraw (RETURN)
                    ^
This is not a valid keyword for this command.
 (parserr 07)
```

This is your initial entry.

HP Link Editor/XL reports an error.

`LinkEd> REDO`	You decide to edit the last command.
`LISTRL XL=Linedraw`	HP Link Editor/XL displays last command.
`RR (RETURN)`	You correct the letter `X`.
`LISTRL RL=Linedraw`	HP Link Editor/XL displays the corrected command.
`(RETURN)`	You execute the command.
`LinkEd> LISTRL RL=Linedraw`	

The *MPE XL Commands Reference Manual* contains a complete description of the `DO`, `REDO`, and `LISTREDO` commands.

Checking the Execution Status of Commands

When you're running a batch job and an error occurs, HP Link Editor/XL sets the system Job Control Word (JCW) to `FATAL` (octal 100000, hexadecimal 8000, decimal 32768). This causes MPE XL to flush the remainder of the job. By entering an MPE XL `:CONTINUE` command in the job file, you can continue the job and then test the JCW.

The link editor sets two other Job Control Words when it finishes. Since these Job Control Words are set in session and batch mode, you can test them in command files and UDCs. The job control words are:

- LKEDCMD

  LKEDCMD shows the status of the last command executed. If there is an error, it contains the actual error number. If there is no error, LKEDCMD is set to zero.

- LKEDSTAT

  LKEDSTAT is set to `FATAL` (octal 100000, hexadecimal 8000, decimal 32768) when there is an error. If there is a warning, LKEDSTAT contains `WARN` (octal 40000, hexadecimal 4000, decimal 16384). If there are no errors or warnings, LKEDSTAT is set to zero.

Executing MPE XL Commands

While you're using HP Link Editor/XL, you can enter a programmatically executable MPE XL operating system command. To do this, precede the MPE XL command with a colon (:). For example, to execute the `LISTF` command, type:

```
LinkEd> :LISTF
```

See the *MPE XL Commands Reference Manual* for a complete description of all the MPE XL commands.

Getting Help

HP Link Editor/XL provides an online Help facility similar to that used by the MPE XL Command Interpreter and other MPE XL subsystems. Request Help to clarify the syntax of an HP Link Editor/XL command or to list an example of how to use it.

Request Help by entering the HELP command in this format:

```
LinkEd> HELP [keyword] [ , ALL      ]
                       [ , PARMS    ]
                       [ , EXAMPLES ]
```

The *keyword* parameter can be any of following commands:

ADDRL	COPYRL	LISTOBJ	QUIT
ADDXL	COPYXL	LISTPROG	REDO
ALTPROG	DO	LISTREDO	REVEALRL
BUILDRL	EXTRACTRL	LISTRL	RL
BUILDXL	HELP	LISTXL	SHOWRL
CLEANRL	HIDERL	PURGERL	SHOWXL
CLEANXL	LINK	PURGEXL	XL

The *keyword* parameter can also be one of the following words, LINKWARN or LINKERR, followed by a four-digit number. If you enter one of these words, do not use an option (ALL, PARMS, EXAMPLES) with it:

LINKWARN *nnnn* — Displays a description of the link editor warning whose number (*nnnn*) you enter.

LINKERR *nnnn* — Displays a description of the link editor error whose number (*nnnn*) you enter.

When you ask for Help, you can enter an option that determines the type of information to display. If you do not specify an option, the link editor displays the syntax diagram for the command. The Help options are:

ALL — Provides a full description of the command, including syntax, parameters and an example of how it is used.

PARMS — Describes the parameter(s) for the command.

EXAMPLES — Gives examples of typical ways to use the command.

Valid examples of Help requests are:

```
LinkEd> HELP
LinkEd> HELP BUILDRL, parms
LinkEd> HELP addrl, examples
LinkEd> help LINKERR1001
```

4

Creating Executable Program Files

This chapter discusses executable program files and how HP Link Editor/XL creates and displays them. It also explains how to display symbols in a relocatable object file.

The link editor creates executable program files from relocatable object files and relocatable libraries as follows. First, it merges the specified relocatable object files and libraries into one module and resolves inter-module references. Then, it searches the specified relocatable libraries resolving external references to symbols undefined after the merge operation. When a relocatable object module in the library resolves an external reference, the module is merged into the executable program file that is being built. In the last step, the link editor assigns virtual addresses to all symbols, binds references to the known symbols within each relocatable object module, and puts the resulting executable program in a form that the loader can process.

The Executable Program File Commands

Figure 4-1 shows the link editor commands that are discussed in this chapter along with the files that they use.

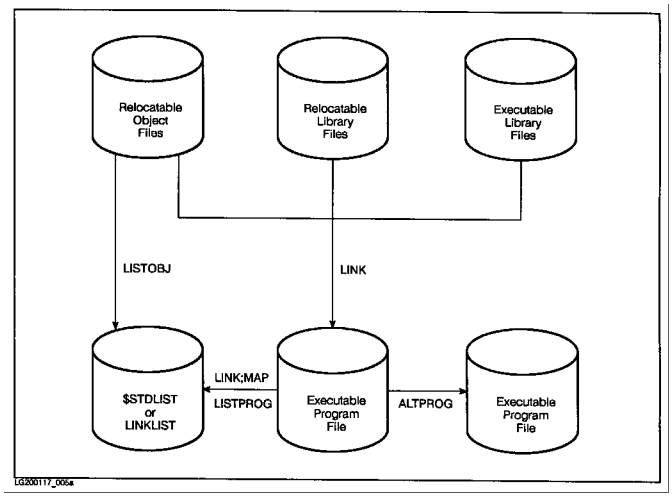


Figure 4-1. Executable Program File Commands

The Executable Program Commands Reference

The link editor commands that create and display executable program files and that display symbols in a relocatable object file are listed below. Each command is discussed in detail in the sections which follow in this chapter.

`ALTPROG`	Alters the user-definable fields of a program file affecting the behavior of the program at run time.
`LINK`	Creates an executable program file.
`LISTOBJ`	Displays symbols in a relocatable object file.
`LISTPROG`	Displays symbols in an executable program file.

ALTPROG

This command allows the user to manipulate those fields of a program file which dictate the behavior of the program at run time. It is especially useful in that programs may be adjusted without having to link them a second time. Most of the options and keywords available can be overridden by the `:RUN` command.

If a keyword is specified but no argument is given, then the corresponding field in the file specified will be reset to its default value.

Syntax

```
ALTPROG [PROG= file] [, file]...
        [;XL= xl_file [, xl_file]...]
        [;CAP= cap_list]
        [;NMSTACK= max_stack_size]
        [;NMHEAP= max_heap_size]
        [;UNSAT= unsat_name]
        [;ENTRY= entry_name]
        [;PRI= priority_level]
        [;MAXPRI= max_priority_level]
```

Parameters

file — Specifies the name of a program file which is to be altered. If no file is given, the file `$OLDPASS` is assumed.

xl_file — Specifies a default executable library to be searched at run time.

If *xl_file* was not previously specified, or has more characters than the previous *xl_file* specified, the link editor will attempt to allocate enough space for the new string. Since this *xl_file* name can be of arbitrary length, it is possible to get an error message from the link editor when not enough space is available. In this case, you may specify the XL list on the `:RUN` command, or else link the program again using the longer *xl_file* name.

To specify the default *xl_file*, use `XL=""`.

cap_list — The capability attribute that the link editor assigns to the executable program file. Enter one or more of the following attributes separated by commas:

`PH` - Process Handling
`DS` - Extra Data Segments
`MR` - Multiple Resources
`PS` - Programmatic Creation of Session
`PM` - Privileged Mode
`IA` - Interactive Access

`BA` - Local Batch Access

Default: If no capabilities are specified, the executable file's capability set will default to `BA` and `IA`.

max_stack_size — Sets the maximum stack size, in bytes, for the resulting executable program. The program uses the stack to store a procedure's local variables and for control purposes. You can override this value using the `NMSTACK` parameter of the `:RUN` command. Default: the system-configured value.

max_heap_size — Sets the maximum heap size, in bytes, for the resulting executable program. The program uses the heap for dynamic storage allocation. You can override this value using the `NMHEAP` parameter of the `:RUN` command. Default: the system-configured value.

unsat_name — Names the procedure which the loader uses to satisfy unresolved externals. Since *unsat_name* is a procedure, it is case sensitive. You can override the parameter by using the `UNSAT` parameter of the `:RUN` command. When the loader cannot resolve external references, it reports an error.

If *unsat_name* was not previously specified, or has more characters than the previous *unsat_name* specified, the link editor will attempt to allocate enough space for the new string. Since this *unsat_name* can be of arbitrary length, it is possible to get an error message from the link editor when not enough space is available. In this case you may specify the *unsat_name* on the `:RUN` command, or else link the program again using the longer *unsat_name*.

To specify the default *unsat_name*, use `UNSAT=""`.

entry_name — Names the point within a program where execution begins. `ENTRY=` lets you override the primary program entry point. If the symbol that matches *entry_name* is not found, an error occurs. *Entry_name* is case sensitive. You can override this parameter using the `ENTRYPOINT` parameter of the `:RUN` command. Default: starts execution from the primary program entry point (corresponding to a program's main procedure or outer

block). *Entry_names* must be primary or secondary entry types.

If *entry_name* was not previously specified, or has more characters than the previous *entry_name* specified, the link editor will attempt to allocate enough space for the new string. Since this *entry_name* can be of arbitrary length, it is possible to get an error message from the link editor when not enough space is available. In this case you may specify the *entry_name* on the `:RUN` command, or else link the program again using the longer *entry_name*.

priority_level — Specifies the execution priority that the program will have at run time. The *priority_level* has to be either `BS`, `CS`, `DS`, `ES`, or a number between 100 and 255 inclusive. This value can be overridden by the `PRI=` keyword on the `:RUN` command.

max_priority_level — Specifies the maximum execution priority that the program can have at run time. The *max_priority_level* has to be either `BS`, `CS`, `DS`, `ES`, or a number between 100 and 255 inclusive. See the `PRI=` keyword of the `:RUN` command for more information.

LINK

This command creates an executable program file. It does this by merging the relocatable object modules from all the files in the FROM= parameter. (These files can be relocatable object files, relocatable library files, or a combination of both.) The link editor also searches the relocatable libraries specified by the RL= parameter and includes the modules in those libraries containing definitions that resolve external references.

Syntax

```
LINK [FROM= source_file [, source_file...]
     [;TO= dest_file]
     [;RL= rl_file [, rl_file]...]
     [;XL= xl_file [, xl_file]...]
     [;CAP= cap_list]
     [;NMSTACK= max_stack_size]
     [;NMHEAP= max_heap_size]
     [;UNSAT= unsat_name]
     [;PARMCHECK= check_level]
     [;PRIVLEV= priv_level]

     [;PRI= priority_level]
     [;MAXPRI= max_priority_level]

     [;ENTRY= entry_name]
     [;NODEBUG]
     [;MAP]
     [;SHOW]
```

Parameters

source_file

Names a relocatable object file or a relocatable library file. The file must be a binary file of type NMOBJ or NMRL. The link editor merges all the relocatable object modules in the FROM= files to form the executable program file named by the TO= parameter. You can use an indirect file name for this parameter. Default: merge the relocatable object modules in the system file $OLDPASS.

dest_file

Names the file where the resulting executable module is placed. If you include the TO= parameter and the link editor finds no file with that name, it creates a new executable program file for you. If the file already exists, it replaces the current contents of the file with the executable module. When *dest_file* is an existing file, it must have filecode NMPRG. Default: place the executable module in the system file $NEWPASS.

rl_file

Names a relocatable library file that resolves external reference(s) contained in the *source_file* or in another *rl_file*. The file must have a filecode of NMRL. The link editor searches the relocatable libraries in the `RL` list according to the order in which you list them. Therefore, if a module from one library calls a routine in another library and that routine refers to a module in the first library, you must name the first library a second time to resolve this circular reference. You can enter an indirect file for this parameter.

xl_file

Names an executable library that resolves external references remaining in the executable program file. The file must have filecode NMXL. You can override the `XL=` list by using the `XL=` parameter of the `:RUN` command. You can enter an indirect file for this parameter.

Since *xl_file* is passed to the system, if *xl_file* is not fully qualified, it will be qualified with a name that is consistent with the program file being loaded. For further information, refer to the `:RUN` command in the *MPE XL Commands Reference Manual*.

cap_list

The capability attribute that the link editor assigns to the executable program file. Enter one or more of the following attributes separated by commas:

`PH` - Process Handling
`DS` - Extra Data Segments
`MR` - Multiple Resources

`PS` - Programmatic Creation of Session

`PM` - Privileged Mode
`IA` - Interactive Access
`BA` - Local Batch Access

Default: If no capabilities are specified, the executable file's capability set will default to `BA` and `IA`.

max_stack_size

Sets the maximum stack size, in bytes, for the resulting executable program. The program uses the stack to store a procedure's local variables and for control purposes. You can override this value using the `NMSTACK`

parameter of the :RUN command. Default: the system-configured value.

max_heap_size — Sets the maximum heap size, in bytes, for the resulting executable program. The program uses the heap for dynamic storage allocation. You can override this value using the NMHEAP parameter of the :RUN command. Default: the system-configured value.

unsat_name — Names the procedure which the loader uses to satisfy unresolved externals. This procedure must reside in an executable library that is specified at run time. Refer to the *MPE XL Commands Reference Manual* for further information.

Since *unsat_name* is a procedure, it is case sensitive. You can override the parameter by using the UNSAT parameter of the :RUN command. Default: when the loader cannot resolve external references, it reports an error.

check_level

Determines the type checking error level that the link editor uses while binding external references to procedures and global variables. All relocatable object modules indicate a checking level for each reference and each definition of a procedure or a global variable. When binding an external reference to a definition, the link editor compares the type information at the lower of the two checking levels specified by the reference and the definition. If a type mismatch is found, it is either a warning or an error. This option determines which type mismatches are warnings and which are errors. The *check_level* entries are:

`0` - All type mismatches are warnings.

`1` - Mismatches of the procedure, function or variable type are errors. All other mismatches are warnings.

`2` - Mismatches of the procedure, function or variable type and mismatches of the number of arguments for procedures or functions are errors. All other mismatches (parameter types, for example) are warnings.

`3` - All type mismatches are errors. Default: 3.

priv_level

Determines the privilege level used by the executable program file. This parameter changes the privilege level of all procedures in the symbol and export tables (of the relocatable object file) that were set during compilation.

The *priv_level* entries are:

`0` - System level access

`1` - Unused

`2` - Privileged level access

`3` - User level access

Default: the privilege levels set during compilation by compiler directives.

priority_level	Specifies the execution priority that the program will have at run time. The priority_level has to be either `BS`, `CS`, `DS`, `ES`, or a number between 100 and 255 inclusive. This value can be overridden by the `PRI=` keyword on the `:RUN` command.
max_priority_level	Specifies the maximum execution priority that the program can have at run time. The priority_level has to be either `BS`, `CS`, `DS`, `ES`, or a number between 100 and 255 inclusive. See the `PRI=` keyword of the `:RUN` command for more information.

entry_name	Names the point within a program where execution begins. ENTRY= lets you override the primary program entry point. If the symbol that matches *entry_name* is not found, an error occurs. *Entry_name* is case sensitive. You can override this parameter using the ENTRYPOINT parameter of the :RUN command. Default: starts execution from the primary program entry point (corresponding to a program's main procedure or outer block). *Entry_names* must be primary or secondary entry types.
NODEBUG	Strips all symbolic debugging information from the resulting executable program file. Debugging information is generated when you use the compiler debug option. Default: debugging information is not stripped from the executable program file.
MAP	Prints a symbol map to the list file, LINKLIST. The symbol map is identical to that produced by the LISTPROG command. Default: do not print a symbol map.
SHOW	Displays on $STDLIST the name of each relocatable object module as it is being merged into the executable program file. You can use this parameter to verify the order in which the link editor processes each module. Default: do not display the names of relocatable object modules.

Examples

```
LinkEd> LINK FROM=OBJCODE;TO=EXECPROG;NMSTACK=30000;MAP;SHOW
```

This command merges the relocatable object module(s) from the file OBJCODE and places them into the executable program file EXECPROG. It assigns a program stack size of 30000 bytes and generates a map of the resulting executable program file. The name of each relocatable object module is also displayed as the executable program file is being built.

```
LinkEd> LINK FROM=^OBJCDE;TO=EXECPROG;RL=LINEDRAW,ARCDRAW;CAP=BA
```

This command merges the relocatable object modules named in the indirect file, OBJCDE, into the executable program file EXECPROG. It searches the relocatable libraries LINEDRAW and ARCDRAW to resolve external references. The resulting executable program file can be executed only in batch mode.

LISTOBJ

This command displays (on `LINKLIST`) the symbols in a relocatable object file.

If you do not specify which symbols to display using the parameters listed below, the following types of symbols are displayed:

- Procedure and program entry points.
- Imported code symbols.
- HP COBOL II chunk symbols.
- Exported data symbols, except compiler-generated symbols beginning with `$`, `S$`, or `C$`.
- Certain compiler-generated static data symbols, beginning with `M$`, which appear in HP COBOL II listings.
- Storage requests (for example, `HP FORTRAN 77 COMMON`).
- Module symbols.

Syntax

```
LISTOBJ OBJFILE= relocatable_object_file
        [;ALL]
        [;CODE]
        [;DATA]
        [;ENTRYSYM]
        [;MILLICODE]
```

Parameters

relocatable_object_file	Names the relocatable object file to display.
`ALL`	Displays the symbols in the relocatable object file, including compiler-generated local symbols.
`CODE`	Displays all imported and exported (not local) code symbols.
`DATA`	Displays all exported data symbols and storage requests.
`ENTRYSYM`	Displays all procedure and program entry points.
`MILLICODE`	Displays all millicode symbols.

Example

```
LinkEd> LISTOBJ EX1OBJ
```

This command displays symbols in the relocatable object file, `EX1OBJ`. (The source program for `EX1OBJ` is `EX1SRC` and is shown in figure 2-11.)

The first part of the listing is a header that gives general information about the relocatable object module. `MODULE NAME` shows the name of the relocatable object module and `VERSION` gives its format version. `LENGTH` shows the number of bytes (in hexadecimal) in the relocatable object module. Symbols in the relocatable object module are listed after the header. See the next section "Understanding the Symbol Listing" for an explanation of the symbols and columns in the symbol portion of the listing. If there are other relocatable object modules in the relocatable object file, they follow and are listed in the same format as the first.

```
MODULE NAME     : EX1SRC
VERSION         : 85082112
LENGTH          : 00000CD4

Sym                        C H X P Sym    Sym  Lset
Name                               Type  Scope Name
----                       - - - - ----  ----- ----

_start                     0   3 3 sec_p univ
ex1                        0   3 3 pri_p univ
M$1                        0       data  local
COB_ACCEPT                 0       code  unsat
COB_CLOSE                  0       code  unsat
COB_CLOSE_FILES            0       code  unsat
COB_OPEN                   0       code  unsat
COB_READSEQ                0       code  unsat
COB_WRITE                  0       code  unsat
TERMINATE                  0       code  unsat
U_EXIT                     0       code  unsat
coboltrap                  0       code  unsat
```

Understanding the Symbol Listing

This section describes the fields that appear in the symbol listing produced by this command.

Column	Description
`Sym Name`	Contains the name of the symbol. If the name exceeds 25 characters, it is truncated and an asterisk appears in the first truncated position.
`C`	Contains the type checking level of the symbol. See the `PARMCHECK=` *check_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`H`	Specifies whether the symbol is hidden or not. If an `H` appears in this column, the symbol was hidden by the `HIDERL` command. If the column is blank, the symbol is not hidden.
`X`	Specifies the *xleast* level of the symbol. See the `XLEAST=` *xleast_level* parameter of the `ADDXL` command for a definition of the values that appear in this column.
`P`	Specifies the privilege or execution level at which this symbol runs. See the `PRIVLEV=` *priv_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`Sym Type`	Contains the symbol type. The symbol types are shown below (see Table 4-1 for the relationship of `Sym Type` values to `Sym Scope` values). `abs` - Absolute `code` - Code `data` - Data `entry` - Entry `milli` - Millicode `mod` - HP Pascal module name `null` - Null `plab` - Procedure label `pri_p` - Primary program entry point `s_req` - Storage request `sec_p` - Secondary program entry point
`Sym Scope`	Specifies the symbol's scope. The symbol scopes are shown below (see Table 4-1 for the relationship of `Sym Scope` values to `Sym Type` values). `local` - Local `univ` - Universal `unsat` - Unsatisfied

`Lset Name`	Specifies the name of the locality set to which this symbol belongs. Only user-defined locality sets are listed.

Table 4-1. Symbol Types and Scopes (LISTOBJ)

Sym Type	Sym Scope	Description
`abs`	`univ`	A symbol that defines a non-relocatable symbol or value and is visible to other object modules.
`abs`	`local`	A symbol that defines a non-relocatable symbol or value and is invisible to other object modules.
`abs`	`unsat`	A symbol that references a non-relocatable symbol.
`code`	`local`	A local label generated by the compiler, a user label or a local label within a millicode routine.
`code`	`univ`	The actual starting point of the code of a level one procedure or function. An `entry univ` symbol must exist for this symbol in order for other object modules to reference the procedure or function. (This symbol appears most frequently in `LISTPROG` and `LISTXL` listings.)
`code`	`unsat`	A symbol which is referenced by an object module, but not defined by it.
`data`	`local`	A data symbol which is visible inside an object module, but invisible to other object modules.
`data`	`univ`	A data symbol defined in an object module that is visible to other object modules.
`data`	`unsat`	A data symbol that is referenced by an object module but not defined in it.
`entry`	`univ`	The export stub for a level one procedure or function. It is visible to other object modules.
`entry`	`local`	The entry point to a nested procedure or program, referenceable only within the module.
`milli`	`univ`	A millicode routine linked into an object module.
`milli`	`unsat`	A reference to a millicode routine that will be linked into a relocatable object module.

Table 4-1.
Symbol Types and Scopes (LISTOBJ) (continued)

Sym Type	Sym Scope	Description
mod	local	An HP Pascal module name.
null	univ	Internal symbol.
null	local	Internal symbol.
null	unsat	Internal symbol.
plab	local	An export stub created for a procedure or function (declared in a relocatable object module) whose address has been taken.
pri_p	univ	The main entry point into an outer block of a program file.
s_req	unsat	A symbol created when an uninitialized HP FORTRAN 77 common block is declared. This symbol is also created for Pascal global data and C globals.
sec_p	univ	The secondary entry point into an outer block of a program file.

LISTPROG

This command displays (on LINKLIST) the symbols in an executable program file. This command produces the same output as the MAP option of the LINK command.

If you do not specify which symbols to display using the parameters listed below, the following types of symbols are displayed:

- Procedure and program entry points.
- Unresolved external symbols.
- Imported code symbols.
- HP COBOL II chunk symbols.
- Exported data symbols, except compiler-generated symbols beginning with $, S$, or C$.
- Certain compiler-generated static data symbols, beginning with M$, which appear in HP COBOL II listings.
- Storage requests (for example, HP FORTRAN 77 COMMON).
- Module symbols.

Syntax

```
LISTPROG PROG= executable_prog_file
         [;ALL]
         [;CODE]
         [;DATA]
         [;ENTRYSYM]
         [;MILLICODE]
         [;STUB]
         [;VALUE]
```

Parameters

executable_prog_file	Names the executable program file to display.
ALL	Displays the symbols in the executable program file, including compiler-generated local symbols.
CODE	Displays all imported and exported (not local) code symbols.
DATA	Displays all exported data symbols and storage requests.
ENTRYSYM	Displays all procedure and program entry points.
MILLICODE	Displays all millicode symbols.
STUB	Displays all stub and plabel symbols.
VALUE	Displays the symbols (within symbol type) by their value rather than alphabetically by their name.

Example

```
LinkEd> LISTPROG EX1PROG
```

This command displays symbols in the executable program file, EX1PROG. (The source program for EX1PROG is EX1SRC and is shown in figure 2-11.)

The first part is the header which gives general information about the executable program file. PROGRAM names the executable program file. XL LIST shows the names of executable libraries specified in the XL parameter of the LINK command. CAPABILITIES shows the capabilities assigned to the program via the CAP parameter of the LINK command. NMHEAP SIZE gives the value specified by the NMHEAP parameter of the LINK command. NMSTACK SIZE shows the value specified for the NMSTACK parameter of the LINK command. And finally, VERSION gives the format version of the executable program file. The symbols in the executable program file are listed next. See the next section "Understanding the Symbol Listing" for explanations of the symbols and columns appearing in the symbol portion of the listing.

```
PROGRAM         : EX1PROG
XL LIST         :
CAPABILITIES    : BA, IA
NMHEAP SIZE     :
NMSTACK SIZE    :
VERSION         : 85082112

Sym                      C H X P Sym    Sym        Sym        Lset
Name                             Type   Scope      Value      Name
----                     - - - - ----   -----      -----      ----
$START$                  0   3 3 sec_p  univ       000059B4
_start                   0   3 3 sec_p  univ       00005A04
ex1                      0   3 3 pri_p  univ       000059E8
M$1                      0       data   local   dp+00000000
```

Understanding the Symbol Listing

This section describes the fields that appear in the symbol listing produced by this command.

Column	Description
`Sym Name`	Contains the name of the symbol. If the name exceeds 25 characters, it is truncated and an asterisk appears in the first truncated position.
`C`	Contains the type checking level of the symbol. See the `PARMCHECK=` *check_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`H`	Specifies whether the symbol is hidden or not. If an `H` appears in this column, the symbol was hidden by the `HIDERL` command. If the column is blank, the symbol is not hidden.
`X`	Specifies the *xleast* level of the symbol. See the `XLEAST=` *xleast_level* parameter of the `ADDXL` command for a definition of the values that appear in this column.
`P`	Specifies the privilege or execution level at which this symbol runs. See the `PRIVLEV=` *priv_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`Sym Type`	Contains the symbol type. The symbol types are shown below (see Table 4-2 for the relationship of `Sym Type` values to `Sym Scope` values). `abs` - Absolute `code` - Code `data` - Data `entry` - Entry `milli` - Millicode `mod` - HP Pascal module name `plab` - Procedure label `pri_p` - Primary program entry point `sec_p` - Secondary program entry point `stub` - Stub
`Sym Scope`	Specifies the symbol's scope. The symbol scopes are shown below (see Table 4-2 for the relationship of `Sym Scope` values to `Sym Type` values). `ext` - External `local` - Local `univ` - Universal
`Sym Value`	Specifies the value of the symbol. For `pri_p`, `sec_p` and `entry univ` symbols, this column contains the address of an export stub. For `stub ext` and `plab`

`local` symbols (values displayed in the `lp+` format), this column shows the address of the XRT entry for this import stub. For `stub local` symbols, this column contains the address of the stub (a promotion stub or an import stub). For all `data univ` symbols, this column contains the address of a literal (if not represented in `dp+` format) or the offset from the `dp` (data pointer) register. For all other symbols, it shows the address of the symbol.

`Lset Name` Specifies the name of the locality set to which this symbol belongs. Only user-defined locality sets are listed.

Table 4-2. Symbol Types and Scopes (LISTPROG)

Sym Type	Sym Scope	Description
`abs`	`univ`	A symbol that defines a non-relocatable symbol or value and is visible to other object modules.
`abs`	`local`	A symbol that defines a non-relocatable symbol or value and is invisible to other object modules.
`code`	`local`	A local label generated by the compiler, a user label or a local label within a millicode routine.
`code`	`univ`	The actual starting point of the code of a level one procedure or function. An **`entry univ`** symbol must exist for this symbol in order for other object modules to reference the procedure or function.
`data`	`local`	A data symbol which is visible inside an object module, but invisible to other object modules.
`data`	`univ`	A data symbol defined in an object module that is visible to other object modules.
`entry`	`univ`	The export stub for a level one procedure or function. It is visible to other object modules.
`entry`	`local`	The entry point to a nested procedure or program, referenceable only within the module.
`milli`	`univ`	A millicode routine linked into an object module.
`mod`	`local`	An HP Pascal module name.
`plab`	`local`	An export stub created for a procedure or function (declared in a relocatable object module) whose address has been taken.
`pri_p`	`univ`	The main entry point into an outer block of a program file.
`sec_p`	`univ`	The secondary entry point into an outer block of a program file.
`stub`	`ext`	A procedure or function which is referenced by an object module but not defined by it. The loader resolves this reference at run time.
`stub`	`local`	A promotion or an import stub.

5

Maintaining Relocatable Libraries

This chapter describes how HP Link Editor/XL creates and maintains relocatable libraries. It begins by describing relocatable libraries and how they are used. The rest of the chapter discusses the link editor commands for manipulating relocatable libraries.

Figure 5-1 shows the files that the link editor uses when creating and maintaining relocatable libraries.

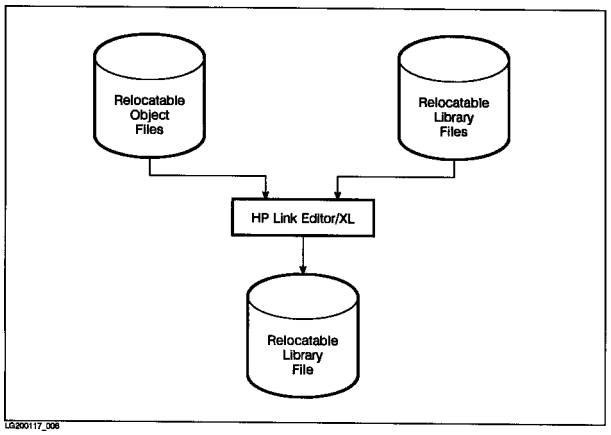


Figure 5-1.
Files Used for Creating and Maintaining a Relocatable Library File

Relocatable Libraries

A relocatable library contains relocatable object modules and a Library Symbol Table. Figure 5-2 illustrates the structure of a relocatable library.

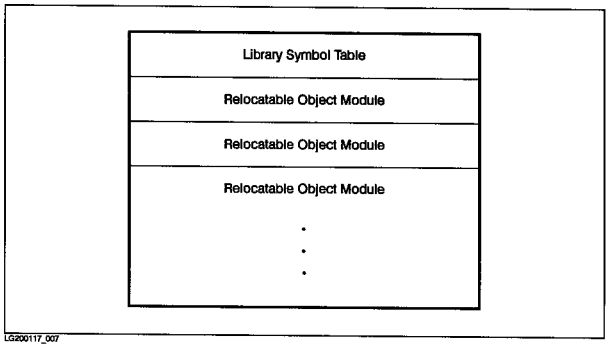


Figure 5-2. The Structure of a Relocatable Library

Relocatable libraries are useful for storing subprograms since subprograms contain common routines that are used frequently. As an example, if several programs use the same routine, you can place it in a relocatable library. Then, to incorporate the routine in each program, name the library when you link the program using `LINK`. The link editor merges the relocatable object module containing the routine into each program file.

Storing routines in relocatable libraries helps eliminate duplication of programming effort and encourages consistency and adherence to local programming standards. Furthermore, since the link editor can search a series of relocatable libraries, you can create different libraries for distinct purposes and then reference only those relocatable libraries that a particular program needs.

The library routine becomes part of the program file when the link editor merges the relocatable object module containing the referenced code into the program file. Once the executable program file is created, the program is insulated from changes made to the library routines. To incorporate library changes into a program, you must relink the program using the modified library.

When the link editor finds a routine in a relocatable library that resolves an external reference, it merges the entire relocatable object module containing that routine into the calling program. If the module contains code that pertains to a single procedure, the link editor adds that procedure to the program file. If the called procedure is one of several procedures in a module, the entire module is added to the program file.

The Relocatable Library Commands

HP Link Editor/XL provides a full set of commands to manipulate relocatable object modules within relocatable libraries. All relocatable libraries start as compiled relocatable object code. Use the `ADDRL` command to place the relocatable object modules produced by a compiler into a relocatable library. You can extract selected modules from a relocatable library and put them into a new relocatable object file with the `EXTRACTRL` command. You can also copy relocatable modules between relocatable libraries or purge selected modules from a specific library. Figure 5-3 shows how these commands relate to relocatable object files and relocatable libraries, as well as the other relocatable library commands that the link editor provides.

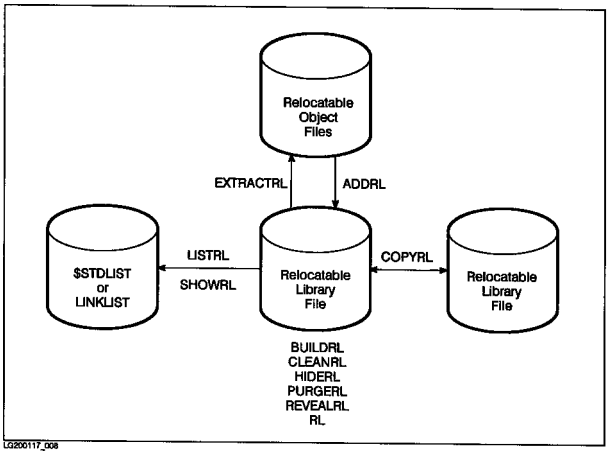


Figure 5-3. Relocatable Library Commands

The Relocatable Library Commands Reference

The following HP Link Editor/XL commands manage relocatable libraries. Each command is discussed in detail in the sections which follow in this chapter.

`ADDRL`	Adds relocatable object modules from a relocatable object file to a relocatable library.
`BUILDRL`	Builds and initializes a file as a new relocatable library. This library becomes the current relocatable library for subsequent interactive commands.
`CLEANRL`	Rebuilds a relocatable library by removing any fragmentation and leaving room for expansion.
`COPYRL`	Copies selected relocatable object modules from one relocatable library to another.
`EXTRACTRL`	Copies selected relocatable object modules from a relocatable library, placing them into a new relocatable object file.
`HIDERL`	Hides a symbol so the loader can no longer use this symbol to resolve external references between executable modules.
`LISTRL`	Lists the symbols that are imported and exported by each relocatable object module within a relocatable library. This directory also shows the module name and entry point of each relocatable object module.
`PURGERL`	Deletes selected relocatable object modules from a relocatable library.
`REVEALRL`	Reveals a symbol that was previously hidden by the `HIDERL` command. This command allows the loader to resolve external references between executable modules.
`RL`	Selects an existing relocatable library to be the current relocatable library.
`SHOWRL`	Displays the name of the current relocatable library.

ADDRL

This command takes relocatable object modules, which are compiled from one or more source files, and puts them into a relocatable library. To add a relocatable object module from another relocatable library, use the COPYRL command.

Syntax

```
ADDRL FROM= source_file [, source_file]...
      [;TO= dest_file]
      [;MERGE [;RL= rl_file [, rl_file]...]]
      [;SHOW]
      [;REPLACE]
```

Parameters

source_file — Names the relocatable object file containing the module(s) to add to the relocatable library. The file must be a binary file with the filecode NMOBJ. When you want to include several relocatable object files, you can name each file individually, or you can provide an indirect file name containing a list of object files by preceding that file name with a caret symbol (^).

dest_file — Names the relocatable library where the relocatable object modules are placed. When *dest_file* is an existing file, it must have the filecode NMRL. Default: the current relocatable library established by the last BUILDRL or RL command.

MERGE — Directs the link editor to merge the relocatable object modules into a single object module and then add that module to the relocatable library. The link editor takes the name of the first object module it encounters in the list of relocatable object files and assigns that name to the relocatable object module being built. The examples, which follow, give more details on how MERGE works. Default: create a separate relocatable object module in the library for each module in the relocatable object file.

rl_file — Names the relocatable library to use during MERGE operations to resolve external references. The file must have a filecode of NMRL. When you want to include several relocatable library files, you can name each library individually, or you can provide an indirect file name by preceding that file name with the caret symbol (^). Default: do not

use a relocatable library to resolve external references.

`SHOW` — Displays (on `$STDLIST`) the name of each relocatable object module added to the relocatable library. All files specified in the `FROM=` and `RL=` parameters are displayed. Default: do not display the names of relocatable object modules.

`REPLACE` — Specifies that when symbols in the module being added are duplicates of symbols in any module in the destination library, then the modules with duplicate symbols residing in the library are removed. The new module is added before any of the modules in the library are removed.

Examples

```
LinkEd> ADDRL FROM=ARC,LINE,TANGENT
```

This command adds each of the relocatable object modules within the relocatable object files `ARC`, `LINE`, and `TANGENT` as distinct relocatable object modules to the current relocatable library.

When using `ADDRL`, you normally omit the `MERGE` parameter. By omitting `MERGE`, you create a separate relocatable object module in the relocatable library corresponding to each relocatable object module in the object file. Figure 5-4 illustrates this.

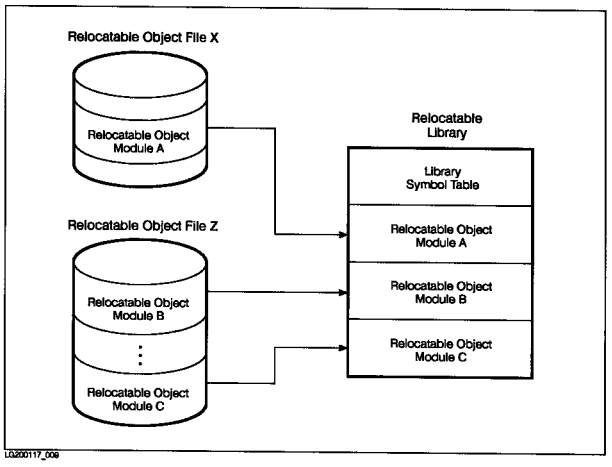


Figure 5-4. The ADDRL Command without the MERGE Option

```
LinkEd> ADDRL FROM=LINEDRAW;TO=BOXDRAW;MERGE;RL=ARC,LINE;SHOW
```

This command merges the relocatable object modules in the relocatable object file LINEDRAW, with the modules from the relocatable libraries ARC and LINE that resolve external references, then adds a single relocatable object module containing this code to the relocatable library BOXDRAW. The link editor also displays the name of each relocatable object module it processes during the MERGE operation.

The MERGE parameter directs the link editor to combine all the relocatable object modules into a single module as shown in Figure 5-5.

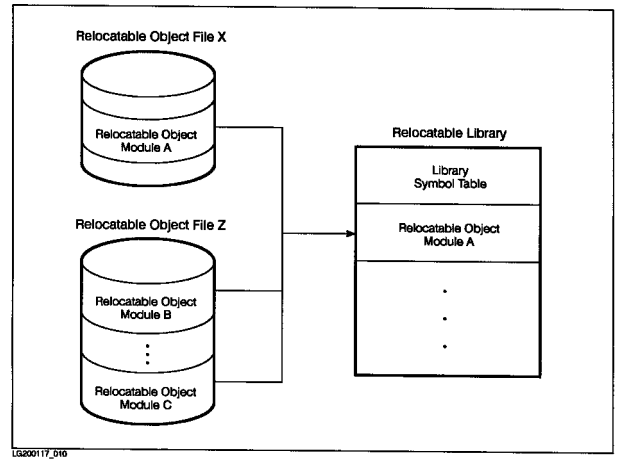


Figure 5-5. The ADDRL Command with the MERGE Option

During a MERGE operation, you can also provide a list of relocatable libraries for the link editor to search to resolve external references. For example, if a relocatable object file contains three relocatable object modules and these modules refer to two relocatable object modules within a relocatable library, the link editor combines all five relocatable object modules into one.

BUILDRL

This command creates a new relocatable library file. The directory of the relocatable library file is initialized and symbol space is reserved. This command also makes the new library the current relocatable library for subsequent interactive commands.

Syntax

```
BUILDRL RL= rl_file
        [;LIMIT= max_modules]
```

Parameters

rl_file — Names the new relocatable library (it is created with the filecode NMRL). The name must conform to the conventions established for all MPE XL file names. The file must not already exist in the account. If it does, an error message is printed.

max_modules — Specifies the maximum number of relocatable object modules that the relocatable library can contain by setting the size of the directory. The maximum number you can enter for this parameter is 400000. Default: 2000.

Examples

```
LinkEd> BUILDRL RL=LINEDRAW;LIMIT=50
```

This command builds a new relocatable file called `LINEDRAW`. The command also sets the maximum number of relocatable object modules that this library can contain at 50.

```
LinkEd> BUILDRL BOXDRAW
```

This command builds a new relocatable library called `BOXDRAW` that can contain a maximum of 2000 relocatable object modules.

CLEANRL

This command eliminates fragmentation that may exist in a relocatable library file.

Although relocatable libraries can expand in size, expansion beyond a certain point fragments the Library Symbol Table (so access to that library is slower). The `CLEANRL` command rebuilds the library, while allocating sufficient space in the library's internal tables for expansion. Thus, you can use this command to allocate more space for a full relocatable library or to conserve disc space by reducing the size of a partially-filled library.

Syntax

```
CLEANRL [RL= rl_file]
        [;COMPACT]
        [;LIMIT= max_modules]
```

Parameters

rl_file	Names an existing relocatable library. The file must have an NMRL filecode. If the relocatable library does not exist, an error results. Default: the current relocatable library established by the last `BUILDRL` or `RL` command.
`COMPACT`	Removes fragmentation and reduces the internal tables of the relocatable library to the minimum size that will accommodate the current contents of the library. Using this parameter does not restrict future use of the library. Default: fragmentation of the internal tables is removed. A pre-determined amount of space is allocated for future expansion.
max_modules	Specifies a new limit for the maximum number of relocatable object modules within the library.

Example

```
LinkEd> CLEANRL BOXDRAW
```

This command rebuilds the relocatable library `BOXDRAW` and restructures its library symbol table so the table can hold more symbols than it currently stores.

COPYRL

This command copies relocatable object modules from one relocatable library to another. You can copy specific modules by their entry point, module, and block data name (HP FORTRAN 77) or by their locality set name.

Syntax

```
COPYRL [ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [, blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;FROM= source_file]
       [;TO= dest_file]
       [;REPLACE]
```

Parameters

The first four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to copy. You can use any one by itself, or you can use them in combination. If you omit these parameters, the entire relocatable library is copied.

entry_name
: Copies the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name
: Copies only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name
: Copies only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name
: Copies those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

source_file	Names the relocatable library containing the modules to copy. The file must have an NMRL filecode. Default: the current relocatable library established by the last `BUILDRL` or `RL` command. (If you use the default, you must enter the `TO=` *dest_file* parameter.)
dest_file	Names the relocatable library where the modules are placed. When *dest_file* is an existing file, it must have the filecode NMRL. Default: the current relocatable library established by the last `BUILDRL` or `RL` command. (If you use the default, you must enter the `FROM=` *source_file* parameter.)

REPLACE	Specifies that when symbols in the module(s) being copied are duplicates of symbols in any module in the destination library, then the modules with duplicate symbols residing in the destination library are removed. The new module is added before any of the modules in the library are removed.

Examples

```
LinkEd> COPYRL LSET=CLIP;TO=WINDOWS
```

This command copies all relocatable object modules which are associated with the CLIP locality set from the current relocatable library and places them into the WINDOWS relocatable library.

```
LinkEd> COPYRL FROM=LINEDRAW
```

This command copies all the relocatable object modules from the relocatable library LINEDRAW and places them into the current relocatable library.

EXTRACTRL

This command extracts selected relocatable object modules from a relocatable library and places them into a new relocatable object file. You can extract specific modules by their entry point, module and block data name (HP FORTRAN 77) or by their locality set name.

This command does not delete the extracted modules from the relocatable library.

Syntax

```
EXTRACTRL [ENTRY= entry_name [ ,entry_name]...]
          [;MODULE= module_name [ ,module_name]...]
          [;BLOCKDATA= blockdata_name [, blockdata_name]...]
          [;LSET= lset_name [ ,lset_name]...]
          [;FROM= source_file]
          [;TO= object_file]
```

Parameters

The first four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to extract. You can use any one by itself, or you can use them in combination. If you omit these parameters, the entire relocatable library is extracted.

entry_name — Extracts the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name — Extracts only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name — Extracts only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name — Extracts those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to

see if locality sets are available). You can enter an indirect file for this parameter.

source_file — Names the relocatable library containing the modules to extract. The file must have the filecode NMRL. Default: the current relocatable library established by the last `BUILDRL` or `RL` command.

object_file Names the relocatable object file to be created (it is created with the filecode NMOBJ). The name must conform to the conventions established for MPE XL file names. The file must not already exist in the specified group. If it does, an error message is printed. Default: the system file, `$NEWPASS`.

Examples

```
LinkEd> EXTRACTRL LSET=CLIP;TO=WINDOWS
```

This command extracts all relocatable object modules which are associated with the `CLIP` locality set from the current relocatable library and places them into the new relocatable object file, `WINDOWS`.

```
LinkEd> EXTRACTRL
```

This command extracts all the relocatable object modules from the current relocatable library and places them into the relocatable object file `$NEWPASS`.

HIDERL

This command hides one or more procedure entry points contained in the relocatable object modules of a relocatable library.

`HIDERL` takes effect when the relocatable object module containing the hidden entry points is added (using `ADDXL`) to an executable library. The entry points are hidden from the loader at run time. Thus, `HIDERL` lets you keep procedure entry points private within a module and avoid name conflicts among procedures.

Syntax

```
HIDERL { ENTRY= entry_name }
       { ;ALL              }
       [;RL= rl_file]
```

Parameters

The first parameter (`ENTRY` or `ALL`), which identifies the entry points to hide, is required.

entry_name	Names a symbol to conceal in the relocatable library. If the *entry_name* does not exist, an error results. *Entry_name* is case sensitive.
`ALL`	Hides all entry points in the relocatable library.
rl_file	Names the relocatable library containing the symbol. The file must have an NMRL filecode. Default: the current relocatable library established by the last `BUILDRL` or `RL` command.

Example

```
LinkEd> HIDERL ENTRY=LineTo;RL=LINEDRAW
```

This command hides the `LineTo` symbol within the `LINEDRAW` relocatable library. If a module from this library is added to an executable library, the loader cannot use this symbol when resolving external references.

LISTRL

This command displays (on LINKLIST) the symbols contained in relocatable object modules of a relocatable library. (You may need this information for the COPYRL, EXTRACTRL and PURGERL commands.)

If you do not specify which symbols to display using the parameters listed below, the following types of symbols are displayed:

- Procedure and program entry points.
- Imported code symbols.
- HP COBOL II chunk symbols.
- Exported data symbols, except compiler-generated symbols beginning with $, S$, or C$.
- Certain compiler-generated static data symbols, beginning with M$, which appear in HP COBOL II listings.
- Storage requests (for example, HP FORTRAN 77 COMMON).
- Module symbols.

Syntax

```
LISTRL [RL= rl_file]
       [;ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [, blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;ALL]
       [;CODE]
       [;DATA]
       [;ENTRYSYM]
       [;MILLICODE]
```

Parameters

rl_file — Names the relocatable library to list. The file must have an NMRL filecode. Default: the current relocatable library established by the last `BUILDRL` or `RL` command.

The next four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to list. You can use any one by itself, or you can use them in combination. If you omit these parameters, the entire relocatable library is listed.

entry_name — Lists the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name — Lists only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name — Lists only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name — Lists those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

`ALL` — Displays the symbols in the relocatable library, including compiler-generated local symbols.

`CODE` — Displays all imported and exported (not local) code symbols.

`DATA` — Displays all exported data symbols and storage requests.

`ENTRYSYM` — Displays all procedure and program entry points.

`MILLICODE` — Displays all millicode symbols.

Example

```
LinkEd> LISTRL RL=LIBRL;CODE;ENTRYSYM
```

This command displays symbols in the LIBRL relocatable library. It also displays all procedure and program entry points. (The library is the one that is created in figure 2-4.)

The first part of the listing is the relocatable library header. LIBRARY NAME gives the file name of the relocatable library and VERSION is its format version. MODULE COUNT shows the number of relocatable modules in the library and MODULE LIMIT gives the maximum number of modules that it holds.

After the relocatable library header is the first relocatable module header. MODULE NAME gives the name of the relocatable object module and VERSION is its format version. LENGTH gives the number of bytes (in hexadecimal) in the relocatable object module. Symbols in the relocatable object module are listed after the header. See the next section "Understanding the Symbol Listing" for an explanation of the symbols and columns in the symbol portion of the listing. If there are additional relocatable object modules in the relocatable library to list, they appear next and are listed in the same format as the first module.

```
LIBRARY NAME   : LIBRL
VERSION        : 85082112
MODULE COUNT   : 5
MODULE LIMIT   : 2000

MODULE NAME    : LIB1SRC
VERSION        : 85082112
LENGTH         : 00000508

Sym                      C H X P Sym    Sym  Lset
Name                             Type  Scope Name
----                     - - - - ----  ----- ----
julian                   3   3 3 entry univ

MODULE NAME    : LIB2SRC
VERSION        : 85082112
LENGTH         : 00000620

Sym                      C H X P Sym    Sym  Lset
Name                             Type  Scope Name
----                     - - - - ----  ----- ----
mdy                      3   3 3 entry univ
julian                   3       code  unsat
```

```
MODULE NAME    : LIB3SRC
VERSION        : 85082112
LENGTH         : 000004B8

Sym                     C H X P Sym    Sym  Lset
Name                            Type  Scope Name
----                    - - - - ----  ----- ----
wkday                   3   3 3 entry univ
julian                  3       code  unsat

MODULE NAME    : LIB4SRC
VERSION        : 85082112
LENGTH         : 00000530

Sym                     C H X P Sym    Sym  Lset
Name                            Type  Scope Name
----                    - - - - ----  ----- ----
adddat                  3   3 3 entry univ
julian                  3       code  unsat
mdy                     3       code  unsat

MODULE NAME    : LIB5SRC
VERSION        : 85082112
LENGTH         : 00000484

Sym                     C H X P Sym    Sym  Lset
Name                            Type  Scope Name
----                    - - - - ----  ----- ----
amort                   3   3 3 entry univ
FTN_DTOI                0       code  unsat
```

Understanding the Symbol Listing

This section describes the fields that appear in the symbol listing produced by this command.

Column	Description
`Sym Name`	Contains the name of the symbol. If the name exceeds 25 characters, it is truncated and an asterisk appears in the first truncated position.
`C`	Contains the type checking level of the symbol. See the `PARMCHECK=` *check_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`H`	Specifies whether the symbol is hidden or not. If an `H` appears in this column, the symbol was hidden by the `HIDERL` command. If the column is blank, the symbol is not hidden.
`X`	Specifies the *xleast* level of the symbol. See the `XLEAST=` *xleast_level* parameter of the `ADDXL` command for a definition of the values that appear in this column.
`P`	Specifies the privilege or execution level at which this symbol runs. See the `PRIVLEV=` *priv_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`Sym Type`	Contains the symbol type. The symbol types are shown below (see Table 5-1 for the relationship of `Sym Type` values to `Sym Scope` values). `abs` - Absolute `code` - Code `data` - Data `entry` - Entry `milli` - Millicode `mod` - HP Pascal module name `null` - Null `plab` - Procedure label `pri_p` - Primary program entry point `s_req` - Storage request `sec_p` - Secondary program entry point
`Sym Scope`	Specifies the symbol's scope. The symbol scopes are shown below (see Table 5-1 for the relationship of `Sym Scope` values to `Sym Type` values). `local` - Local `univ` - Universal `unsat` - Unsatisfied

`Lset Name`	Specifies the name of the locality set to which this symbol belongs. Only user-defined locality sets are listed.

Table 5-1. Symbol Types and Scopes (LISTRL)

Sym Type	Sym Scope	Description
`abs`	`univ`	A symbol that defines a non-relocatable symbol or value and is visible to other object modules.
`abs`	`local`	A symbol that defines a non-relocatable symbol or value and is invisible to other object modules.
`abs`	`unsat`	A symbol that references a non-relocatable symbol.
`code`	`local`	A local label generated by the compiler, a user label or a local label within a millicode routine.
`code`	`univ`	The actual starting point of the code of a level one procedure or function. An `entry univ` symbol must exist for this symbol in order for other object modules to reference the procedure or function. (This symbol appears most frequently in `LISTPROG` and `LISTXL` listings.)
`code`	`unsat`	A symbol which is referenced by an object module, but not defined by it.
`data`	`local`	A data symbol which is visible inside an object module, but invisible to other object modules.
`data`	`univ`	A data symbol defined in an object module that is visible to other object modules.
`data`	`unsat`	A data symbol that is referenced by an object module but not defined in it.
`entry`	`univ`	The export stub for a level one procedure or function. It is visible to other object modules.
`entry`	`local`	The entry point to a nested procedure or program, referenceable only within the module.
`milli`	`univ`	A millicode routine linked into an object module.
`milli`	`unsat`	A reference to a millicode routine that will be linked into a relocatable object module.

Table 5-1.
Symbol Types and Scopes (LISTRL) (continued)

Sym Type	Sym Scope	Description
`mod`	`local`	An HP Pascal module name.
`null`	`univ`	Internal symbol.
`null`	`local`	Internal symbol.
`null`	`unsat`	Internal symbol.
`plab`	`local`	An export stub created for a procedure or function (declared in a relocatable object module) whose address has been taken.
`pri_p`	`univ`	The main entry point into an outer block of a program file.
`s_req`	`unsat`	A symbol created when an uninitialized HP FORTRAN 77 common block is declared. This symbol is also created for Pascal global data and C globals.
`sec_p`	`univ`	The secondary entry point into an outer block of a program file.

PURGERL

This command deletes selected modules from a relocatable library. You can purge specific modules by their entry point, module, and block data name (HP FORTRAN 77) or by their locality set name.

Syntax

```
PURGERL
{ ENTRY= entry_name [,entry_name] ...                          }
{ ;MODULE= module_name [,module_name] ...                      }
{ ;BLOCKDATA= blockdata_name [ , blockdata_name] ...           }
{ ;LSET= lset_name [,lset_name] ...                            }
        [;RL= rl_file]
```

Parameters

The first four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to purge. You can use any one by itself, or you can use them in combination. Modules matching any of the criteria that you enter are purged.

entry_name — Purges the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name — Purges only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name — Purges only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name — Purges those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

rl_file — Names the relocatable library containing the modules to purge. The file must have an NMRL filecode. Default: the current

relocatable library established by the last `BUILDRL` or `RL` command.

Examples

```
LinkEd> PURGERL LSET=CLIP;RL=WINDOWS
```

This command deletes all relocatable object modules that belong to the `CLIP` locality set from the `WINDOWS` relocatable library.

```
LinkEd> PURGERL MODULE=GRAPH
```

This command deletes the relocatable object module named `GRAPH` from the current relocatable library.

REVEALRL

This command reveals hidden symbols in the relocatable object modules of a relocatable library. (`REVEALRL` reverses the effect of the `HIDERL` command.)

This command takes effect when the module containing the symbol is added to an executable library. The symbol can be used by the loader to resolve external references between executable modules. Thus, the effect of `REVEALRL` is to reveal the symbol to the loader at run time.

Syntax

```
REVEALRL { ENTRY= entry_name }
         { ;ALL              }
         [;RL= rl_file]
```

Parameters

entry_name	Names a symbol to reveal in the relocatable Library Symbol Table. If the symbol does not exist, an error results. *Entry_name* is case sensitive.
`ALL`	Reveals all symbols in the relocatable library.
rl_file	Names the relocatable library in which the symbol currently resides. The file must have an NMRL filecode. Default: the current relocatable library established by the last `BUILDRL` or `RL` command.

Example

```
LinkEd> REVEALRL ENTRY=LineTo;RL=LINEDRAW
```

This command reveals the symbol `LineTo` within the `LINEDRAW` relocatable library. When the relocatable object modules containing references to `LineTo` are added to an executable library, the loader can use the symbol to resolve external references between executable modules.

RL

This command makes an existing relocatable library the current (working) relocatable library. This relocatable library becomes the default library in subsequent command operations.

You must have read and write access to the relocatable library.

Syntax

```
RL RL= rl_file
```

Parameters

rl_file — Names an existing relocatable library. The file must have an NMRL filecode.

Example

```
LinkEd> RL RL=BOXDRAW
```

This command makes `BOXDRAW` the current relocatable library.

SHOWRL

This command displays (on `$STDLIST`) the name of the current (working) relocatable library.

To change the current relocatable library, use the `RL` command. To create a relocatable library and make that library the current relocatable library, use the `BUILDRL` command.

Syntax

```
SHOWRL
```

Example

```
LinkEd> SHOWRL
```

This command displays the name of the current relocatable library.

6

Maintaining Executable Libraries

This chapter explains how to build and maintain executable libraries. The chapter begins by describing executable libraries and comparing them to relocatable libraries. The remainder of the chapter provides a detailed description of each of the executable library commands. Since the task of building and maintaining executable libraries resembles the task of building and maintaining relocatable libraries, much of this chapter parallels the information in chapter 5.

Figure 6-1 shows the input and output files that HP Link Editor uses to create and maintain executable libraries.

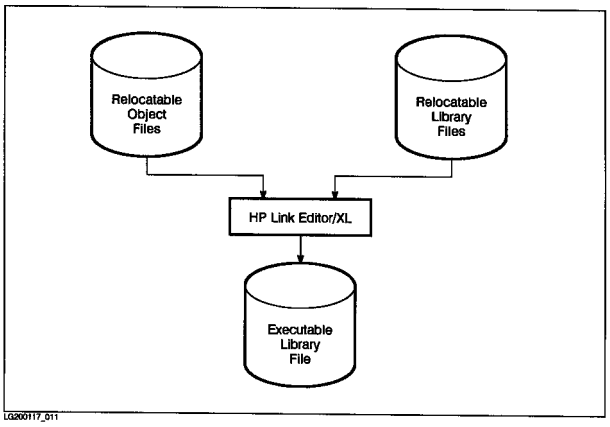


Figure 6-1. Creating an Executable Library File

Executable libraries are composed of one or more executable modules that come from relocatable object modules created by compilers. The executable modules can also come from relocatable object modules in a relocatable library.

Executable Libraries

An executable library contains executable modules and a Library Symbol Table. Figure 6-2 illustrates the structure of an executable library.

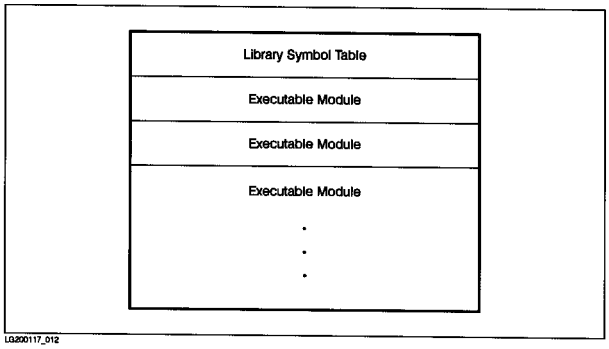


Figure 6-2. The Structure of an Executable Library

Executable libraries contain executable modules having the following characteristics:

- Executable modules are in a form that can be executed directly.
- Executable modules are shared - only one copy of the code need exist on the system. Programs that use an executable module share the same physical copy of code.
- Executable modules have their own global data, separate from the program's global data.
- External references between executable modules and calling programs are resolved at run time.
- Executable modules cannot have outer blocks.

You can store executable libraries in any group and account. At run time, the loader searches the executable libraries that you name in the `XL` list of the `LINK` command or that you specify by the `RUN` command. Besides searching your executable libraries, the loader automatically searches the executable libraries maintained by the system. These libraries are `NL.PUB.SYS`, which contains system routines such as MPE XL intrinsics, and `XL.PUB.SYS`, which contains subsystem support routines.

The Executable Library Commands

Several of the executable library commands resemble relocatable library commands. For example, the `XL`, `LISTXL`, and `SHOWXL` commands (corresponding to the `RL`, `LISTRL`, and `SHOWRL` commands) let you specify the current executable library or display information about an executable library. As you use the `BUILDRL`, `CLEANRL`, `ADDRL`, `COPYRL`, and `PURGERL` commands to manipulate relocatable libraries, you can perform similar operations on executable libraries with the `BUILDXL`, `CLEANXL`, `ADDXL`, `COPYXL`, and `PURGEXL` commands.

Figure 6-3 shows the executable library commands that are discussed in this chapter along with the files that they use.

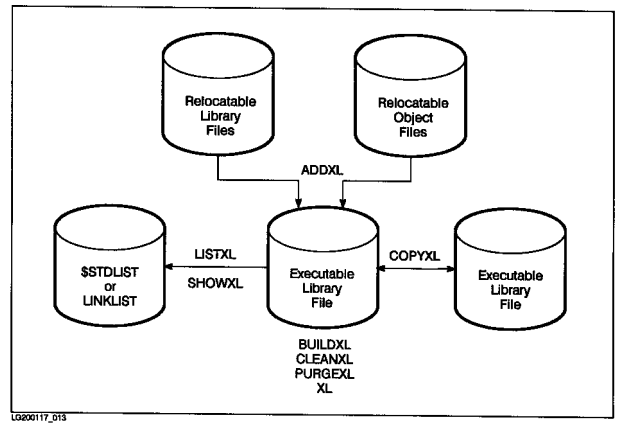


Figure 6-3. Executable Library Commands

The Executable Library Commands Reference

The remainder of this chapter discusses in detail each of the link editor commands that manage executable libraries. The executable library commands are:

`ADDXL`	Adds relocatable object modules from a relocatable object file or from a relocatable library to an executable library.
`BUILDXL`	Builds and initializes a new executable library. This library becomes the current executable library for subsequent interactive commands.
`CLEANXL`	Rebuilds an executable library by removing any fragmentation and leaving room for expansion within that library's internal tables.
`COPYXL`	Copies specified executable modules from one executable library to another.
`LISTXL`	Lists the symbols that are imported and exported by each executable module in an executable library. This listing also includes the module name and procedure entry points for each executable module.
`PURGEXL`	Purges specified executable modules from an executable library.
`SHOWXL`	Displays the name of the current executable library. If you want to select another executable library as the current library, enter an `XL` or `BUILDXL` command using that library's name.
`XL`	Makes an existing executable library the current executable library.

ADDXL

This command adds relocatable object modules to an executable library from either a relocatable object file or a relocatable library.

Syntax

```
ADDXL FROM= source_file [ ,source_file]...
      [;TO= dest_file]
      [;MERGE [;RL= rl_file [, rl_file]...]]
      [;SHOW]
      [;PARMCHECK= check_level]
      [;PRIVLEV= priv_level]
      [;XLEAST= xleast_level]
      [;MAP]
      [;REPLACE]
      [;ENTRY= entry_name [ ,entry_name]...]
      [;MODULE= module_name [ ,module_name]...]
      [;BLOCKDATA= blockdata_name [, blockdata_name]...]
      [;LSET= lset_name [ ,lset_name]...]
      [;NODEBUG]
```

Parameters

source_file

Names either a relocatable object file (from a compiled source file) or a relocatable library file that contains the relocatable object modules you want to add to the executable library. The file must have a filecode of NMOBJ or NMRL. When you want to include several files, you can name each file individually, or you can use an indirect file name containing a list of the file names you want to include. Precede the indirect file name with a caret symbol (^). Note that you must supply at least one file name since the FROM= parameter is required.

dest_file

Names the executable library where the link editor places the executable modules. When *dest_file* is an existing file, it must have the filecode NMXL. Default: the modules are placed in the executable library used in the last XL or BUILDXL command.

MERGE

Directs the link editor to merge all the relocatable object modules together producing a single executable module in the executable library. The link editor uses the first relocatable object module name that it merges as the name for the new module in the library. The examples, which follow,

explain `MERGE` in more detail. Default: do not merge relocatable object modules.

rl_file

Names a relocatable library that the link editor searches during a `MERGE` operation to resolve external references. The file must have an NMRL filecode. When you want to include several relocatable library files, you can name each library individually, or you can provide an indirect file name containing a list of file names. Precede the indirect file name with a caret symbol (^). Default: no relocatable library is used.

`SHOW` — Displays (on `$STDLIST`) the name of each relocatable object module as it is merged into the executable library. Use this parameter to verify the order in which the link editor processes each module. Default: do not display relocatable object modules.

check_level — Determines the type checking error level that the link editor uses while binding external references to procedures and global variables. All relocatable object modules indicate a checking level for each reference and each definition of a procedure or a global variable. When binding an external reference to a definition, the link editor compares the type information at the lower of the two checking levels specified by the reference and the definition. If a type mismatch is found, it is either a warning or an error. This option determines which type mismatches are warnings and which are errors. The *check_level* entries are:

- `0` - All type mismatches are warnings.
- `1` - Mismatches of the procedure, function or variable type are errors. All other mismatches are warnings.
- `2` - Mismatches of the procedure, function or variable type and mismatches of the number of arguments for procedures or functions are errors. All other mismatches (parameter types, for example) are warnings.
- `3` - All type mismatches are errors.

Default: 3.

priv_level — Determines the privilege level of all entry points in the executable module. This parameter changes the privilege level of all procedures in the symbol and export tables (of the relocatable object file) that were set during compilation.

The *priv_level* entries are:

- `0` - System level access

	`1` - Unused `2` - Privileged level access `3` - User level access Default: the privilege levels set during compilation by compiler directives.
xleast_level	Determines the privilege level at which calling procedures must be executing to use the executable module. Enter a value from zero to three (see the values for the *priv_level* parameter, above). Default: use the existing privilege levels of the executable module.
`MAP`	Prints a symbol map to the list file, `LINKLIST`, using the same format as the `LISTXL` command. Default: Do not print a symbol map.

`REPLACE` Specifies that when symbols in the module being added are duplicates of symbols in any module in the destination library, then the modules with duplicate symbols residing in the library are removed. The new module is added before any of the modules in the library are removed.

The next four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify the modules to add from the relocatable library. (Do not use these parameters when the *source_file* is a relocatable object file.) You can use any of the parameters alone, or you can use them in combination. If you omit these parameters, the entire relocatable library is added.

entry_name Adds the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name Adds only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name Adds only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name Adds those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

`NODEBUG` Specifies that all debugging information should be stripped from the output object module before being added to the executable library.

Examples

```
LinkEd> ADDXL FROM=FILEOPEN;TO=FILEREAD
```

This command takes the relocatable object modules from the relocatable object file `FILEOPEN` and adds them to the `FILEREAD` executable library.

When you omit the MERGE parameter, the link editor links each relocatable object module independently, then adds that module to the executable library. It doesn't attempt to resolve references between modules or library routines. Thus, each relocatable object module in the object file has its counterpart in the executable library. Figure 6-4 illustrates this process.

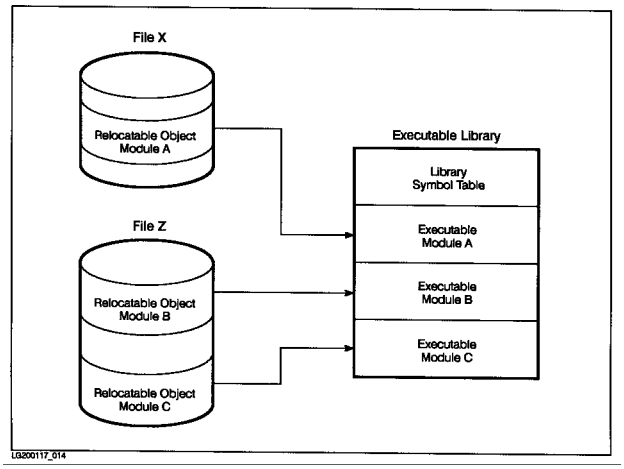


Figure 6-4. The ADDXL Command without the MERGE Option

In this process, the ADDXL command duplicates the operation of the LINK command as the link editor binds the relocatable object module to make it executable. That is, the link editor assigns virtual addresses to all symbols, binds references to the known symbols within each relocatable object module, and puts the resulting executable module in a form that the loader can process.

```
LinkEd> ADDXL FROM=FILEIO,FILEREAD,FILEWRIT;MERGE;RL=FILEUTIL
```

This command merges the relocatable object modules from the relocatable object files `FILEIO`, `FILEREAD` and `FILEWRIT`, as well as using those modules from the `FILEUTIL` relocatable library that resolve external references, and then places a single executable module (called `FILEIO`) into the current executable library.

By specifying the `MERGE` parameter, you can direct the link editor to merge the relocatable modules into one executable module, resolving references between them. (See Figure 6-5.) In the same command, you can also list the relocatable libraries to be searched to resolve external references to library routines.

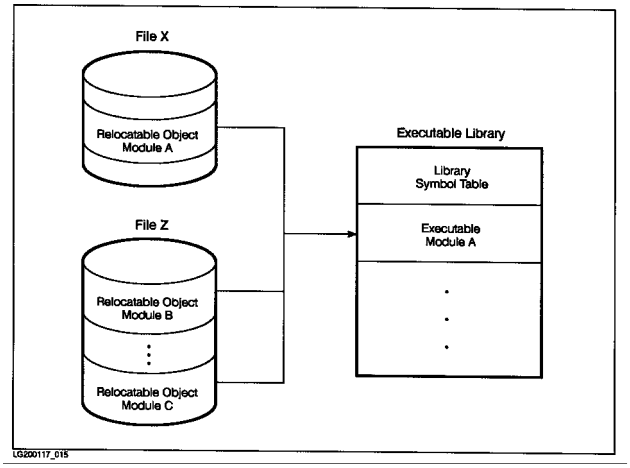


Figure 6-5. The ADDXL Command with the MERGE Option

BUILDXL

This command builds and initializes a new executable library. The new library becomes the current (working) executable library for subsequent interactive commands.

Syntax

```
BUILDXL XL= xl_file
        [;LIMIT= max_modules]
```

Parameters

xl_file — Names the executable library to be created (it is created with filecode NMXL). The name must conform to the MPE XL file naming conventions. If the file already exists, the link editor ignores the command and prints an error message.

max_modules — Specifies the maximum number of relocatable object modules that the executable library can contain. The maximum number that you can enter is 400000. Default: 500.

Examples

```
LinkEd> BUILDXL XL=FILEREAD;LIMIT=200
```

This command creates an executable library called `FILEREAD` that can contain a maximum of 200 executable modules.

```
LinkEd> BUILDXL XL=FILEIO
```

This command creates an executable library called `FILEIO` that can contain a maximum of 500 executable modules.

CLEANXL

This command eliminates fragmentation which may exist in an executable library.

Although executable libraries can expand in size, expansion beyond a certain point fragments the Library Symbol Table so access to that library becomes slower. The `CLEANXL` command takes a fragmented library and rebuilds that library, while allocating sufficient space in the library's internal tables to allow for expansion. You can use this command to allocate more space for an executable library, or to conserve disc space by reducing the size of a partially-filled library.

Syntax

```
CLEANXL [XL= xl_file]
        [;COMPACT]
        [;LIMIT= max_modules]
```

Parameters

xl_file — Names an existing executable library (the file must have an NMXL filecode). If the executable library does not exist, the link editor reports an error. Default: condense the current executable library (established by the last `BUILDXL` or `XL` command).

`COMPACT` — Removes fragmentation and reduces the internal tables of the relocatable library to the minimum size that will accommodate the current contents of the library. Using this parameter does not restrict future use of the library. Default: fragmentation of the internal tables is removed. A pre-determined amount of space is allocated for future expansion.

max_modules — Specifies a new limit for the maximum number of relocatable object modules that the library can contain.

Example

```
LinkEd> CLEANXL XL=FILEIO
```

This command rebuilds the executable library `FILEIO` and restructures its Library Symbol Table so the table can hold more symbols than it currently stores.

COPYXL

This command copies executable modules from one executable library to another. You can copy individual modules by their module, block data subprogram, procedure entry point, or locality set name.

Syntax

```
COPYXL [ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [, blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;FROM= source_file]
       [;TO= dest_file]
       [;REPLACE]
```

Parameters

The first four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to copy. You can use any one by itself, or you can use them in combination. If you omit these parameters, the entire relocatable library is copied.

entry_name — Copies the module(s) that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name — Copies only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name — Copies only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name — Copies those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

source_file	Names the executable library that the link editor searches to find the specified modules. The file must have an NMXL filecode. Default: the current executable library established by the last `BUILDXL` or `XL` command. (If you use the default, you must enter the `TO=` *dest_file* parameter.)

dest_file — Names the executable library where the modules are placed. When *dest_file* is an existing file, it must have the filecode NMXL. Default: the current executable library established by the last `BUILDXL` or `XL` command. (If you use the default, you must enter the `FROM=` *source_file* parameter.)

`REPLACE` — Specifies that when symbols in the module(s) being copied are duplicates of symbols in any module in the destination library, then the modules with duplicate symbols residing in the destination library are removed. The new module is added before any of the modules in the library are removed.

Examples

```
LinkEd> COPYXL LSET=FILEINTRINS;TO=FILEIO
```

This command copies all executable modules which are associated with the `FILEINTRINS` locality set from the current executable library and places them into the `FILEIO` executable library.

```
LinkEd> COPYXL FROM=FILEREAD
```

This command copies all executable modules from the executable library `FILEREAD` and places them into the current executable library.

LISTXL

This command lists (on LINKLIST) symbols contained in selected executable modules of an executable library. (You may need this information for the COPYXL and PURGEXL commands.)

If you do not specify which symbols to display using the parameters listed below, the following types of symbols are displayed:

- Procedure and program entry points.
- Imported code symbols.
- HP COBOL II chunk symbols.
- Exported data symbols, except compiler-generated symbols beginning with $, S$, or C$.
- Certain compiler-generated static data symbols, beginning with M$, which appear in HP COBOL II listings.
- Storage requests (for example, HP FORTRAN 77 COMMON).
- Module symbols.

Syntax

```
LISTXL [XL= xl_file]
       [;ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [, blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;ALL]
       [;CODE]
       [;DATA]
       [;ENTRYSYM]
       [;MILLICODE]
       [;STUB]
       [;VALUE]
```

Parameters

xl_file — Names the executable library to list. The file must have an NMXL filecode. Default: the current executable library established by the last BUILDXL or XL command.

The next four parameters (ENTRY, MODULE, BLOCKDATA, and LSET) identify specific modules to list. You can use any one by itself, or you can use them in combination. If you omit these parameters, the entire executable library is listed.

entry_name	Lists the modules that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.
module_name	If you do not use the RLFILE compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the RLFILE compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.
blockdata_name	Lists only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.
lset_name	Lists those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.
ALL	Displays all symbols in the executable library, including compiler-generated local symbols.
CODE	Displays all imported and exported (not local) code symbols.
DATA	Displays all exported data symbols and storage requests.
ENTRYSYM	Displays all procedure and program entry points.
MILLICODE	Displays all millicode symbols.
STUB	Displays all stub and plabel symbols.
VALUE	Displays all symbols by symbol value rather than alphabetically by symbol name.

Example

```
LinkEd> LISTXL XL=LIBXL;ALL;VALUE
```

This command displays all symbols in the `LIBXL` executable library, including compiler-generated local symbols. Symbols are sorted and displayed by their value. The executable library is the one created in figure 2-7. The source program (`EX2BSRC`) is listed in figure 2-13.

The first part of the listing is the executable library header. `LIBRARY NAME` is the file name of the executable library. `VERSION` is the format version of the library. `MODULE COUNT` shows the number of executable modules in the library and `MODULE LIMIT` is the maximum number that it holds.

After the executable library header, the first executable module header is listed. `MODULE NAME` shows the name of the executable module and `VERSION` shows its format version. `LENGTH` shows the number of bytes (in hexadecimal) in the executable module. Symbols in the executable module are listed after the header. See the next section "Understanding the Symbol Listing" for an explanation of the symbols and columns in the symbol portion of the listing. If there are other executable modules to list in the executable library, they appear next and are listed in the same format as the first module.

```
LIBRARY NAME   : LIBXL
VERSION        : 85082112
MODULE COUNT   : 1
MODULE LIMIT   : 500

MODULE NAME                                    START     LENGTH
-----------                                    -----     ------
EX2BSRC                                     00129000   00004078

MODULE NAME    : EX2BSRC
VERSION        : 85082112
LENGTH         : 00004078

Sym                     C H X P Sym    Sym        Sym       Lset
Name                            Type   Scope      Value     Name
----                    - - - - ----   -----      -----     ----
julian                  3   3 3 entry  univ       0012B4EC
mdy                     3   3 3 entry  univ       0012B508
wkday                   3   3 3 entry  univ       0012B524
adddat                  3   3 3 entry  univ       0012B543
amort                   3   3 3 entry  univ       0012B55C
$neg3                   0       code   local      0012B058
$neg5                   0       code   local      0012B098
$neg6                   0       code   local      0012B0E0
$pos                    0       code   local      0012B120
$pos_for_17             0       code   local      0012B130
$neg10                  0       code   local      0012B168
$neg                    0       code   local      0012B174
$neg_for_17             0       code   local      0012B184
$neg12                  0       code   local      0012B1C0
$neg15                  0       code   local      0012B1F4
$neg17                  0       code   local      0012B224
$u17                    0       code   local      0012B248
$7                      0       code   local      0012B25C
$pos7                   0       code   local      0012B26C
$1                      0       code   local      0012B294
$2                      0       code   local      0012B2A0
$neg7                   0       code   local      0012B2A8
$8                      0       code   local      0012B2AC
$neg7_shift             0       code   local      0012B2B8
$3                      0       code   local      0012B2E0
$4                      0       code   local      0012B2F0
$neg9                   0       code   local      0012B32C
$neg14                  0       code   local      0012B36C
t1                      0       code   local      0012B39C
finish                  0       code   local      0012B4A4
div_ovfl                0       code   local      0012B4B4
julian                  3       code   univ       0012B598
mdy                     3       code   univ       0012B688
wkday                   3       code   univ       0012B7F0
adddat                  3       code   univ       0012B878
```

```
amort                    3       code  univ       0012B96C
$UNWIND_START            0       code  univ       0012BA38
$UNWIND_END              0       code  univ       0012BAA8
$RECOVER_END             0       code  univ       0012BAB8
$RECOVER_START           0       code  univ       0012BAB8
L$2                      0       data  local      0012B4C0
L$3                      0       data  local      0012B4D0
L$6                      0       data  local      0012B4D8
$global$                 0       data  univ    dp+00000000
julian.M$2               0       data  local   dp+00000000
$dp$                     0       data  univ    dp+00000000
mdy.M$3                  0       data  local   dp+00000030
wkday.M$4                0       data  local   dp+00000060
$PFA_C_END               0       data  univ    dp+00000078
$PFA_C_START             0       data  univ    dp+00000078
FTN_DTOI                 0       stub  ext     lp+00000020
$$divide_by_constant     0       milli univ       0012B000
$$divI_2                 0       milli univ       0012B000
$$divI_4                 0       milli univ       0012B010
$$divI_8                 0       milli univ       0012B020
$$divI_16                0       milli univ       0012B030
$$divI_3                 0       milli univ       0012B040
$$divU_3                 0       milli univ       0012B06C
$$divI_5                 0       milli univ       0012B084
$$divU_5                 0       milli univ       0012B0B0
$$divI_6                 0       milli univ       0012B0C8
$$divU_6                 0       milli univ       0012B0F8
$$divU_10                0       milli univ       0012B110
$$divI_10                0       milli univ       0012B154
$$divI_12                0       milli univ       0012B1AC
$$divU_12                0       milli univ       0012B1D0
$$divI_15                0       milli univ       0012B1E4
$$divU_15                0       milli univ       0012B1FC
$$divI_17                0       milli univ       0012B208
$$divU_17                0       milli univ       0012B23C
$$divI_7                 0       milli univ       0012B258
$$divU_7                 0       milli univ       0012B2F8
$$divI_9                 0       milli univ       0012B310
$$divU_9                 0       milli univ       0012B344
$$divI_14                0       milli univ       0012B360
$$divU_14                0       milli univ       0012B364
$$remoI                  0       milli univ       0012B378
```

Understanding the Symbol Listing

This section describes the fields that appear in the symbol listing produced by this command.

Column	Description
`Sym Name`	Contains the name of the symbol. If the name exceeds 25 characters, it is truncated and an asterisk appears in the first truncated position.
`C`	Contains the type checking level of the symbol. See the `PARMCHECK=` *check_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`H`	Specifies whether the symbol is hidden or not. If an `H` appears in this column, the symbol was hidden by the `HIDERL` command. If the column is blank, the symbol is not hidden.
`X`	Specifies the *xleast* level of the symbol. See the `XLEAST=` *xleast_level* parameter of the `ADDXL` command for a definition of the values that appear in this column.
`P`	Specifies the privilege or execution level at which this symbol runs. See the `PRIVLEV=` *priv_level* parameter of the `ADDXL` and `LINK` commands for a definition of the values that appear in this column.
`Sym Type`	Contains the symbol type. The symbol types are shown below (see Table 6-1 for the relationship of `Sym Type` values to `Sym Scope` values). `abs` - Absolute `code` - Code `data` - Data `entry` - Entry `milli` - Millicode `mod` - HP Pascal module name `plab` - Procedure label `stub` - Stub
`Sym Scope`	Specifies the symbol's scope. The symbol scopes are shown below (see Table 6-1 for the relationship of `Sym Scope` values to `Sym Type` values). `ext` - External `local` - Local `univ` - Universal
`Sym Value`	Specifies the value of the symbol. For `pri_p`, `sec_p` and `entry univ` symbols, this column contains the address of an export stub. For `stub ext` and `plab local` symbols (values displayed in the `lp+` format),

this column shows the address of the XRT entry for this import stub. For `stub local` symbols, this column contains the address of the stub (a promotion stub or an import stub). For all `data univ` symbols, this column contains the address of a literal (if not represented in `dp+` format) or the offset from the `dp` (data pointer) register. For all other symbols, it shows the address of the symbol.

`Lset Name` Specifies the name of the locality set to which this symbol belongs. Only user-defined locality sets are listed.

Table 6-1. Symbol Types and Scopes (LISTXL)

Sym Type	Sym Scope	Description
`abs`	`univ`	A symbol that defines a non-relocatable symbol or value and is visible to other object modules.
`abs`	`local`	A symbol that defines a non-relocatable symbol or value and is invisible to other object modules.
`code`	`local`	A local label generated by the compiler, a user label or a local label within a millicode routine.
`code`	`univ`	The actual starting point of the code of a level one procedure or function. An **`entry univ`** symbol must exist for this symbol in order for other object modules to reference the procedure or function.
`data`	`local`	A data symbol which is visible inside an object module, but invisible to other object modules.
`data`	`univ`	A data symbol defined in an object module that is visible to other object modules.
`entry`	`univ`	The export stub for a level one procedure or function. It is visible to other object modules.
`entry`	`local`	The entry point to a nested procedure or program, referenceable only within the module.
`milli`	`univ`	A millicode routine linked into an object module.
`mod`	`local`	An HP Pascal module name.
`plab`	`local`	An export stub created for a procedure or function (declared in a relocatable object module) whose address has been taken.
`stub`	`ext`	A procedure or function which is referenced by an object module but not defined by it. The loader resolves this reference at run time.
`stub`	`local`	A promotion or an import stub.

PURGEXL

This command purges selected modules from an executable library. You can purge individual modules by their entry point, module, and block data name (HP FORTRAN 77) or by locality set name.

Syntax

```
PURGEXL
{ ENTRY= entry_name [,entry_name] ...                                  }
{ ;MODULE= module_name [,module_name] ...                              }
{ ;BLOCKDATA= blockdata_name [, blockdata_name] ...                    }
{ ;LSET= lset_name [,lset_name] ...                                    }
        [;XL= xl_file]
```

Parameters

The first four parameters (`ENTRY`, `MODULE`, `BLOCKDATA`, and `LSET`) identify specific modules to purge. You can use any one by itself, or you can use them in combination. Modules matching any of the criteria that you enter are purged.

entry_name — Purges the modules that define (export) the symbolic *entry_name*. You can enter an indirect file for this parameter. *Entry_name* is case sensitive.

module_name — Purges only those modules having the name, *module_name*. If you do not use the `RLFILE` compiler directive, this is the name of the source file from which the relocatable object module was compiled. If you use the `RLFILE` compiler directive, see the appropriate language appendix (appendix B, C, D, or E) for the definition of this name. You can enter an indirect file for this parameter.

blockdata_name — Purges only those modules having the name, *blockdata_name*. Use this parameter only for HP FORTRAN 77 block data subprograms. You can use an indirect file name for *blockdata_name*.

lset_name — Purges those modules that contain code belonging to the locality set (*lset_name*) that you enter. A module can contain several locality sets, or there can be several modules within a locality set. Each compiler provides its own directives for placing procedures into locality sets (check your language manual to see if locality sets are available). You can enter an indirect file for this parameter.

xl_file — Names the executable library containing the modules to purge. This file must have an NMXL filecode. Default: the current

executable library established by the last BUILDXL or XL command.

Examples

```
LinkEd> PURGEXL LSET=FILEINTRINS;XL=FILEIO
```

This command deletes every executable module that belongs to the FILEINTRINS locality set in the FILEIO executable library.

```
LinkEd> PURGEXL MODULE=SEEK
```

This command deletes the executable module named SEEK from the current executable library.

SHOWXL

This command displays (on $STDLIST) the name of the current executable library established by the last XL or BUILDXL command.

Syntax

```
SHOWXL
```

Example

```
LinkEd> SHOWXL
```

This command displays the name of the executable library established by the last BUILDXL or XL command.

XL

This command selects an existing file as the current executable library. This library is used as the default library for subsequent command operations.

Syntax

```
XL XL= xl_file
```

Parameters

xl_file	Names an existing executable library. The file must have an NMXL filecode.

Example

```
LinkEd> XL XL=FILEIO
```

This command makes `FILEIO` the current executable library.

7

Advanced Topics

In the MPE XL environment, you can program effectively without an explicit knowledge of HP Link Editor/XL and how it works. However, when working on complex applications, you may need to take advantage of certain advanced features of the link editor. This requires that you explicitly run the link editor to override its default values.

The topics discussed in this chapter include:

- A description of the MPE XL programming environment.
- A brief description of the millicode library, `MILLI.LIB.SYS`.
- How to use locality sets to improve program performance.

The MPE XL Programming Environment

The paragraphs which follow in this section cover the aspects of the MPE XL environment that affect HP Link Editor/XL and how it links programs.

Virtual Memory

A program running under MPE XL has at least two *spaces*, where a space is a fixed-length block of virtual memory. One is a code space and the other is a data space. A virtual address is composed of a space identifier and a space offset, both of which are 32 bits long. Thus, virtual memory consists of over 4 billion spaces, each of which can contain over 4 billion bytes (four gigabytes). (Some implementations of HP Precision Architecture restrict space identifiers to 16 bits, allowing only 65,536 spaces.)

Even though a virtual memory address is 64 bits long, most addressing can be done with 32-bit addresses. There are eight *space* registers that hold space identifiers. When loading or storing a word in memory using a 32-bit address, one of the last four space registers is selected automatically. Its selection is based on the high-order two bits of the address. This method of addressing allows one gigabyte (or quadrant) of each space selected by the space registers to be addressed: the first quadrant of the first space, the second quadrant of the second space, the third quadrant of the third space, and the fourth quadrant of the fourth space. Figure 7-1 shows the code and data spaces divided into quadrants:

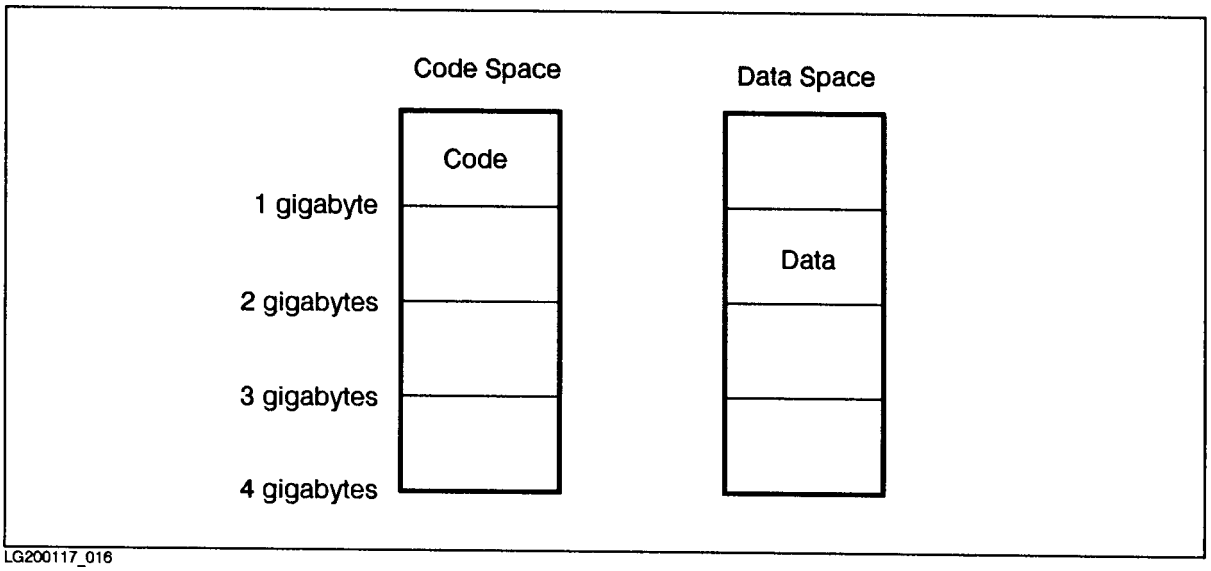


Figure 7-1. Code and Data Quadrants

The link editor places all code and literals in the first quadrant of the code space, and all data in the second quadrant of the data space. Thus, every executable module, whether in a executable program file or in an executable library, consists of one code space and one data space. When the program is loaded, a new code space is created for the code in the executable program file and for each executable library that is loaded to satisfy external references. Only one data space is created. The data spaces in each executable module are loaded one after another into a single data space.

If you run a program that is already loaded (someone else is also running it), the code spaces already loaded can be reused. Only a data space must be created for the second program (process). The code and literals can be shared because they cannot be modified by a process.

External Calls

An external call is a procedure call that transfers control from one executable module to another. These calls are not resolved during the link operation for two reasons. First, the link editor builds (using `LINK` or `ADDXL`) one executable module at a time. Therefore, it does not know where the called procedure is located. Second, since space identifiers are assigned at run time, there is no way to predict what the space identifier for either code space will be.

Because the link editor cannot resolve external calls, it builds an *import stub* during linking for each procedure that is called but not defined in the executable module. It also allocates an entry in the External Reference Table (XRT) for the unresolved procedure. The import stub contains a short sequence of code that is used at run time to transfer to the procedure's entry point. The import stub uses the XRT entry to find both the space identifier and the space offset of the target procedure. The import stub then saves the current value of space register 4 (which corresponds to the first quadrant of the code space), and copies the new space identifier into space register 4. This ensures that the space register always contains the space identifier of the current code space. The loader locates each procedure referenced in the XRT and initializes each XRT entry with the appropriate values.

When a procedure is called externally, it must restore the space identifier of the calling procedure before returning to it. To do this, the link editor builds an *export stub* for every procedure that can be called externally. The export stub gives an alternate entry point to the procedure that is executed by an external call. All internal calls (that is, calls to procedures in the same executable module) use the ordinary procedure entry point. External calls use the export stub as the entry point. On `LISTPROG` and `LISTXL` symbol listings, export stubs are shown as `entry` symbols, while the ordinary entry points are shown as `code` symbols.

Privilege Levels

The HP Precision Architecture provides four levels of privilege. Level 0 is the most privileged and allows complete access to all system resources. Level 3 is the least privileged. MPE XL establishes the following meanings for the privilege levels:

`0`	Restricted to the MPE/XL kernel.
`1`	Reserved for future use.
`2`	Privileged Mode. Programs with PM capability execute at this level. This privilege level provides access to most operating system features and security checks are streamlined.
`3`	User Mode. Most programs run at this level. Programs access operating system features only through documented intrinsics, and access is subject to full security checking.

All procedures have two privilege levels: the privilege level at which it runs, the execution privilege level; and the privilege level at which callers must be running in order for the call to succeed, the call or *xleast* privilege level. When a procedure call occurs, the execution privilege level of the calling procedure must have the same or a higher (numerically lower) privilege level than the call privilege level of the procedure being called. (If not, the calling program is aborted with a privilege violation.) The execution privilege level of the called procedure is promoted to the execution privilege level of the calling procedure, when the calling level is more privileged; otherwise the called procedure's execution privilege level remains unchanged. The execution and call privilege levels for procedure entry points are shown in the symbol listings produced by the link editor commands, `LISTOBJ`, `LISTPROG`, `LISTRL`, and `LISTXL`.

Privilege level promotion and security checking are performed by the operating system during an external procedure call. Internal calls cannot perform promotion. When internal procedure calls require a promotion, the link editor builds a *promotion stub* that turns the internal call into an external call. A promotion stub is a combination of an import stub and an export stub.

Long Branch Stubs for Procedure Calls

Compilers generate a single branch-and-link instruction for all procedure calls. The instruction has a limited addressing range - 256K bytes in either direction from the call. If the call is external, the link editor places the import stub within this range. If the call is internal, the target may be out of range. In this case, the link editor builds a *long branch stub* within reach of the call. A long branch stub consists of two instructions that reach the target anywhere in the code space.

Long branch stubs are also used for millicode calls and interchunk branching for HP COBOL II programs.

Procedure Labels

The address of a procedure is a data item called a procedure label, or *plabel*. A plabel can be passed as a parameter from one procedure to another, so that an indirect call through the plabel might not be an internal call. To accommodate this possibility, all indirect procedure calls are external calls, even if the call happens to be in the same code space as the procedure being called. A plabel is therefore the address of an XRT entry, not the address of the procedure itself. The indirect procedure call obtains the space identifier and offset from the XRT entry just like an import stub.

If you obtain a plabel through the `HPGETPROCPLABEL` intrinsic, the intrinsic creates an XRT entry and returns its address. If you obtain a plabel by taking the address of a procedure in your source program (for example, by passing a subprogram or function name by reference in HP FORTRAN 77) the link editor automatically allocates an XRT entry and creates a *plabel stub* that instructs the loader to initialize the XRT entry appropriately. The code in the plabel stub is identical to an export stub. Plabel stubs are shown in `LISTPROG` and `LISTXL` symbol listings as `plab` symbols.

Note that if a routine resides in an executable library, and its address is taken (as in HP C), it will not necessarily be the same as the address that is taken from the user's program. The addresses *will* be the same if they are being compared in the same program or library.

In other words, if your program compares the addresses of routines within the program, it will work as you expect. Likewise, comparing the addresses of two routines which reside in the same executable library will work as you expect. But, if your program compares the address of a routine taken from within the program code with the address of a routine taken from within the library, those addressses will not be the same. You will be comparing the address of an import stub with the address of an export stub.

HP Link Editor/XL Environment Files

When you use the `LINK` and `ADDXL` commands, the link editor includes two files, `NRTO.LIB.SYS` and `XLO.LIB.SYS`, into the executable modules that are produced. These files define:

- The standard *subspaces* that control how the link editor arranges the code and data spaces.

  Since MPE XL compilers group the various parts of a relocatable object file into separate subspaces, the link editor can combine like subspaces together within each space. Standard subspaces are defined for millicode, literals, code, stack unwind descriptors, Pascal outer block global variables, static initialized data and static uninitialized data. Compilers also define an additional subspace for each locality set.

- The standard symbols that are used by the compiler libraries and system debuggers.

  For example, the symbols `$UNWIND_START` and `$UNWIND_END` declare the beginning and end of the region containing stack unwind descriptors.

Stack Unwinding

Whenever a traceback of procedure calls is made, the process is referred to as *unwinding* the stack. The traceback can occur as a result of a multi-level procedure return in languages that support it (for example, non-local `GOTO` or escape in Pascal), or from a program abort or a debugging request. Regardless of the cause, each stack frame must be examined to determine the procedure that created it and its size. Given the size of the current stack frame, the previous stack frame can be located.

All MPE XL compilers create static tables of *unwind descriptors* that make stack unwinding possible. The tables are placed in pre-defined subspaces so that the link editor can build one combined stack unwind table during a `LINK` or `ADDXL` operation. Each descriptor describes the stack frame for a procedure, which is identified as a range of addresses in the code space. Thus, given any code address as a starting point, the appropriate descriptor can be located. The descriptor, in turn, identifies the size and type of the current stack frame. From this information, the address of the caller of that procedure and the address of the caller's stack frame can be determined, and the unwinding can continue until the bottom of the stack is reached.

Millicode

HP Link Editor/XL automatically searches the standard system relocatable library, `MILLI.LIB.SYS`, when you execute the `LINK` and `ADDXL` commands. This library contains millicode routines that supplement common low-level operations in programs. It is searched after user relocatable libraries.

Improving Performance with Locality Sets

You can improve the performance of large programs by arranging the code so that sets of procedures that call one another frequently are located in one contiguous area of virtual memory. You do this by using compiler directives to assign procedures to a *locality set*. The compilers place locality set information in relocatable object modules, and HP Link Editor/XL uses this information (during linking) to arrange the code.

Dividing a program into locality sets affects performance in two ways:

- It minimizes the number of "long branches" in the program.

  The HP Precision Architecture instruction set has a single-instruction branch with an addressing range of 256K bytes from the point of call. This instruction is used for procedure calls whenever the branch instruction is close enough to its target. If the short branch does not reach the target, the link editor must insert additional instructions (called a "long branch stub"), which degrade program performance slightly. Locality sets, by grouping frequently-called procedures together, help to keep the majority of branches within reach of their targets.

- It reduces paging during program execution.

  The MPE XL operating system divides memory into pages of 4096 bytes each. When a program does not fit into physical memory, the operating system swaps portions (or pages) of it onto disc. When a program references a page that is not in physical memory, the operating system reads the page from disc into physical memory so that the program can continue. Since swapping can slow program execution, you can use locality sets to reduce the number of page swaps. If execution remains in a locality set for a reasonable time, the number of page swaps is reduced and the operating system can better predict the behavior of the program.

To utilize locality sets effectively, study the program's behavior carefully. Assign procedures to locality sets using the following guidelines:

- Keep locality sets small.

  Put procedures that are used seldomly into one locality set, and those that execute for a long period of time into another.

- Keep locality sets tightly coupled.

  Try to design a locality set so there is a high probability that the procedures in it are used together or that they execute for a significant period of time. Put low-level utility routines, used by several other locality sets, in separate locality sets.

- Put only the most frequently-used code in locality sets.

  Place code that is executed infrequently or just once, in the default (none) locality set.

A

Messages

This appendix lists messages that you may encounter while using HP Link Editor/XL. Self-explanatory messages and those which relate to syntax errors, such as missing or extraneous characters in commands, are not listed in this appendix.

To assist you in finding the solution to a problem, several messages may be displayed. Look up each message in this appendix to get complete information about the action to take.

Messages are preceded by unique reference numbers that indicate the error type. Messages, with their message reference numbers, are listed in this order:

`1000-1499`	User Errors
`1500-1999`	Warning messages
`2000-2999`	System errors
`3000-3999`	Language subsystem errors
`4000-4999`	Internal errors

As an example, the following message has a reference number of `1002` and is listed below as it appears in this appendix:

```
1002      MESSAGE      ATTEMPT TO OPEN FILE "!" FAILED
```

The symbol `!`, used in a message, indicates replaceable character positions. For this message, `!` is a place-holder for a file name.

User Errors (1000-1499)

User errors result from entering incorrect commands or from using the commands incorrectly. User errors cause the command that you entered to fail. You must correct the cause of the error and re-enter the command.

1001 MESSAGE `PROGRAM ENTRY POINT "!" NOT FOUND`

CAUSE HP Link Editor/XL could not find an entry point for the procedure.

ACTION Check to make sure an outer block is present in the link or check that the `NRTO.LIB.SYS` file is present on your system and that it has not been overwritten with some other file. Also ensure that you do not have a file equate for `NRTO.LIB.SYS`.

1002 MESSAGE `ATTEMPT TO OPEN FILE "!" FAILED`

CAUSE HP Link Editor/XL cannot open the named file for reading.

ACTION Be sure that you typed the file's name (and group and account) correctly. If the file exists, be sure you have the required capabilities to read this file. If the named file is either `NRTO.LIB.SYS`, `XLO.LIB.SYS`, or `MILLI.LIB.SYS`, one of these required files is missing from your system. In this case, contact your System Manager.

1003 MESSAGE `ATTEMPT TO CREATE FILE "!" FAILED`

CAUSE HP Link Editor/XL cannot create the named file for writing.

ACTION Be sure that you have the required capabilities for creating files in the group and account.

1004

MESSAGE	DUPLICATE SYMBOL "!" IN "!"
CAUSE	HP Link Editor/XL found two relocatable object modules that define the same symbol. (The error message names the file that contains the second definition.)
ACTION	Be sure that you have not linked the same module twice. Also ensure that two source files have not declared the same procedure name, outer block, global variable or BLOCK DATA subprogram. If two files have declared the same symbol, go back to one of the source files, change the name of the symbol, then recompile.

1005	MESSAGE	FOUND ! DUPLICATE SYMBOL(S)
	CAUSE	This message summarizes the number of duplicate symbols originally detected and listed by error message 1004.
	ACTION	Take the appropriate "action" for each duplicate symbol that was identified by error message 1004.

1006	MESSAGE	UNSATISFIED SYMBOLS:
	CAUSE	A relocatable object module that is being linked has referenced an undefined symbol. This may simply be a misspelled name, or it may indicate that an object module was mistakenly omitted from the LINK command. A list of the unsatisfied symbols follows this message. A missing outer block causes the symbol _start to be listed as unresolved. Symbols followed by "(DATA)" indicate a global variable that was not defined; other symbols indicate undefined millicode or absolute symbols. Undefined "code" symbols are not reported as an error, but are passed to the loader for run-time binding to executable libraries.
	ACTION	Either correct the use of the symbol in the source program or include the missing module in the list of modules to be linked together. An undefined millicode symbol probably indicates that the library MILLI.LIB.SYS is either missing or incompatible with your system release or that you have used a file equate for this file.

1010	MESSAGE	FOUND ! TYPE CHECKING ERROR(S)
	CAUSE	One or more type checking mismatches are errors. (Depending on the value

of the PARMCHECK option of the LINK and ADDXL commands, type checking mismatches are either warnings or errors.) Not all of the type mismatches (warning numbers 1502-1507) may actually be errors; this message indicates the number that are errors.

ACTION — Check your source code and recompile, or request a lower type checking level using the PARMCHECK option.

1021	MESSAGE	INDIRECT FILES NESTED TOO DEEPLY (MAXIMUM DEPTH IS 10)
	CAUSE	An indirect file may contain a reference to another indirect file. You can use up to ten levels of indirect files.
	ACTION	Check for an indirect file that references itself, or for a set of indirect files that contains a chain of circular references.
1022	MESSAGE	RL FILE "!" MUST HAVE FILECODE NMRL
	CAUSE	A file that was included in the RL= list as part of a LINK command has an incorrect filecode. This usually indicates that the wrong file was named in the LINK command.
	ACTION	Enter the correct file name.
1023	MESSAGE	OBJECT FILE "!" MUST HAVE FILECODE NMOBJ OR NMRL
	CAUSE	A file that was included in the FROM= list as part of a LINK command has an incorrect filecode. This usually indicates that the wrong relocatable object file was named in the LINK command.
	ACTION	Enter the correct file name.
1024	MESSAGE	INDIRECT FILE "!" MUST BE AN ASCII FILE
	CAUSE	A file that you are using as an indirect file is a binary file rather than an ASCII file.
	ACTION	Check that you have spelled the name of the indirect file correctly.

1025	MESSAGE	EXPECTED FILECODE NMPRG OR NMXL FOR TO= FILE
	CAUSE	You tried to LINK into an existing translated file.
	ACTION	Use a different filename for the TO= keyword of the LINK command, or, purge the existing output file and try the LINK command again.

1042	MESSAGE	INCOMPATIBLE NUMBER OF ARGUMENTS: ! (!, !)
	CAUSE	The named procedure is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible.
	ACTION	Check the source code and correct the incompatibility, or request a lower type checking level through the PARMCHECK parameter of the LINK or ADDXL command (so this error is reported as a warning).

1043	MESSAGE	INCOMPATIBLE PACKING: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or request a lower type checking level through the PARMCHECK parameter of the LINK or ADDXL command (so this error is reported as a warning).

1044	MESSAGE	INCOMPATIBLE ALIGNMENT: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters,

the parameter number is listed on the following line.

`ACTION` Check the source code and correct the incompatibility, or request a lower type checking level through the `PARMCHECK` parameter of the `LINK` or `ADDXL` command (so this error is reported as a warning).

1045	MESSAGE	INCOMPATIBLE MODE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or request a lower type checking level through the PARMCHECK parameter of the LINK or ADDXL command (so this error is reported as a warning).
1046	MESSAGE	INCOMPATIBLE STRUCTURE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or request a lower type checking level through the PARMCHECK parameter of the LINK or ADDXL command (so this error is reported as a warning).
1047	MESSAGE	INCOMPATIBLE TYPE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that

the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.

`ACTION` Check the source code and correct the incompatibility, or request a lower type checking level through the `PARMCHECK` parameter of the `LINK` or `ADDXL` command (so this error is reported as a warning).

1100	MESSAGE	"ENTRY=" NAME IS LONGER THAN 132 CHARACTERS
	CAUSE	HP Link Editor/XL restricts the names for symbols in library symbol tables to 132 characters, but you entered an entry point name that exceeds this limit.
	ACTION	Give the entry point a name that contains 132 characters or less.
1101	MESSAGE	"LSET=" NAME IS LONGER THAN 132 CHARACTERS
	CAUSE	HP Link Editor/XL restricts the names for symbols in library symbol tables to 132 characters, but you entered a locality set name that exceeds this limit.
	ACTION	Give the locality set a name that contains 132 characters or less.
1102	MESSAGE	"MODULE=" NAME IS LONGER THAN 132 CHARACTERS
	CAUSE	HP Link Editor/XL restricts the names for symbols in library symbol tables to 132 characters. You entered a module name that exceeds this limit.
	ACTION	Give the module a name that contains 132 characters or less.
1103	MESSAGE	NO CURRENT ! IS OPEN; YOU MUST SPECIFY A "FROM" FILE
	CAUSE	Certain library maintenance commands let you omit the FROM= file and use a currently active file instead. In this case, you omitted the FROM= file, but no default file is established.

`ACTION`	Either open a default file and enter the same command, or re-enter the command using the `FROM=` parameter.

1104	MESSAGE	NO CURRENT RELOCATABLE LIBRARY IS OPEN; YOU MUST SPECIFY AN RL FILE
	CAUSE	Certain library maintenance commands let you omit the RL= file and use a currently active relocatable library instead. In this case, you omitted the RL= parameter, but no relocatable library file was selected as the currently active relocatable library.
	ACTION	Either use the RL or BUILDRL command to establish a currently active relocatable library, or re-enter the command using the RL= parameter.

1105	MESSAGE	NO CURRENT ! IS OPEN; YOU MUST SPECIFY A "TO" FILE
	CAUSE	Certain library maintenance commands let you omit the TO= file and use a currently active file instead. In this case, you omitted the TO= file, but no default file is established.
	ACTION	Either open a default file and enter the same command, or re-enter the command using the TO= parameter.

1106	MESSAGE	NO CURRENT EXECUTABLE LIBRARY IS OPEN; YOU MUST SPECIFY AN XL FILE
	CAUSE	Certain library maintenance commands let you omit the XL= file and use a currently active executable library instead. In this case, you omitted the XL= parameter, but no executable library file was selected to be the currently active executable library.
	ACTION	Either use the XL or BUILDXL command to establish a currently active executable library, or re-enter the command using the XL= parameter.

1107	MESSAGE	"RL=" FILES MAY NOT BE SPECIFIED WITHOUT ALSO SPECIFYING THE MERGE OPTION
	CAUSE	The ADDRL and ADDXL commands allow you to specify relocatable libraries to resolve external references only when you request the MERGE option.
	ACTION	Re-enter the command using the MERGE option.

1108	MESSAGE	"UNSAT=" NAME IS LONGER THAN 132 CHARACTERS
	CAUSE	HP Link Editor/XL restricts the names for entries in library symbol tables to 132 characters. You have given an UNSAT procedure a name that exceeds this limit.
	ACTION	Give the UNSAT procedure a name that contains 132 characters or less.
1109	MESSAGE	! IS NOT A LEGAL VALUE FOR PARMCHECK. IT MUST BE IN THE RANGE [0..3]
	CAUSE	You have used an illegal value for the PARMCHECK parameter to the LINK or ADDXL command.
	ACTION	Re-enter the command using a value between 0 and 3 for the PARMCHECK parameter.
1110	MESSAGE	ATTEMPT TO ADD MODULE(S) BEYOND LIBRARY LIMIT (!)
	CAUSE	You have exceeded the limit for object modules in a relocatable or executable library. (HP Link Editor/XL sets the limit to 2000 for relocatable libraries and 500 for executable libraries if you do not specify a limit when you build the library.)
	ACTION	Build a new library file using a larger limit, copy the contents of the old library into the new library, then add the object modules that produced this "overflow" condition.
1111		

MESSAGE SYMBOL "!", ENCOUNTERED IN "!", WAS PREVIOUSLY DEFINED

CAUSE When adding modules to a library, HP Link Editor/XL encountered a new definition for a previously defined symbol.

ACTION Check that you have not added the same module twice. Also ensure that two source files have not declared the same procedure name or BLOCK DATA subprogram.

1112	MESSAGE	ATTEMPT TO OPEN/CREATE FILE "!" FAILED
	CAUSE	Certain HP Link Editor/XL commands attempt to open or create a file to serve as an input source or an output destination. In this case, HP Link Editor/XL either tried to open a file that doesn't exist, or tried to create a new file but found a file with that name already exists.
	ACTION	Consult the section of the manual that describes the command you are using and determine whether or not the named file should exist.
1113	MESSAGE	"TO=" FILE "!" IS THE SAME AS "FROM=" FILE "!". THEY MUST BE UNIQUE
	CAUSE	Certain HP Link Editor/XL commands accept both FROM= and TO= files. In this case, you have named the same file for both parameters.
	ACTION	Determine which file you want to be your FROM= file and which file you want to be your TO= file. Then ensure that their names are distinct.
1114	MESSAGE	"! IS NOT A LEGAL VALUE FOR PRIVLEV. IT MUST BE IN THE RANGE [0..3]"
	CAUSE	You have used an illegal value for the PRIVLEV parameter of the LINK or ADDXL commands.
	ACTION	Re-enter the command using a value between 0 and 3 for the PRIVLEV parameter.
1115		

MESSAGE	"! IS NOT A LEGAL VALUE FOR XLEAST. IT MUST BE IN THE RANGE [0..3]"
CAUSE	You have used an illegal value for the XLEAST parameter of the LINK or ADDXL commands.
ACTION	Re-enter the command using a value between 0 and 3 for the XLEAST parameter.

1116	MESSAGE	LINK FAILED
	CAUSE	When HP Link Editor/XL encounters an error during the linking process, it prints this "summary" message as its last message before terminating.
	ACTION	Refer to the "action" description for the previously listed error messages. Once you have resolved those problems, this message goes away.
1117	MESSAGE	LIMIT FOR A LIBRARY MUST BE IN THE RANGE [1..400000]
	CAUSE	When you build a relocatable or executable library, you must specify how many object modules that library can contain. In this case, you have specified a number outside of this range.
	ACTION	Re-enter the BUILDRL or BUILDXL command and use a number between 1 and 400000 for the maximum number of object modules.
1118	MESSAGE	! IS NOT A LEGAL VALUE FOR THE NMSTACK PARAMETER. IT MUST BE A POSITIVE INTEGER
	CAUSE	An option to the LINK command lets you specify a NMSTACK value. This number must be a positive integer.
	ACTION	Re-enter the LINK command using a valid number for the NMSTACK parameter.
1119	MESSAGE	! IS NOT A LEGAL VALUE FOR THE NMHEAP PARAMETER. IT MUST BE A POSITIVE INTEGER

CAUSE	An option to the LINK command lets you specify a NMHEAP value. This number must be a positive integer.
ACTION	Re-enter the LINK command using a valid number for the NMHEAP parameter.

1120	MESSAGE	ONE OF { ENTRY, LSET, MODULE } MUST BE SPECIFIED
	CAUSE	The syntax for the PURGERL and PURGEXL commands requires that you specify which object modules you want to delete by either their entry point name, their module name, or their locality set name. You have omitted this parameter.
	ACTION	Re-enter the PURGERL or PURGEXL command and specify which modules you want to delete.
1121	MESSAGE	FILE "!" IS NOT A VALID "FROM=" FILE FOR ADDXL. LEGAL INPUT IS EITHER A RELOCATABLE OBJECT FILE OR A RELOCATABLE LIBRARY
	CAUSE	The ADDXL command adds relocatable object modules and relocatable libraries to an executable library. You entered a file that does not have the correct filecode (NMOBJ or NMRL) for one of these files.
	ACTION	Re-enter the command and provide the name of a relocatable object file or a relocatable library file for the FROM= file.
1122	MESSAGE	SPECIFIED ENTRY WAS NOT FOUND IN THE RELOCATABLE LIBRARY
	CAUSE	You used a HIDERL or a REVEALRL command and the symbol cannot be found in the relocatable library.
	ACTION	Be sure you have linked together all the relocatable libraries that the program file requires. Also verify that you have spelled the name correctly (entries in symbol tables are case sensitive).

1123	MESSAGE	INDIRECT INPUT FILE "!" MUST BE AN ASCII FILE
	CAUSE	A file that you used as an indirect file is a binary file rather than an ASCII file.
	ACTION	Create your indirect files using a text editor and save them as ASCII files. If you have followed this procedure, check that you have spelled the name of the indirect file correctly.

1124	MESSAGE	EXPECTED FILECODE ! FOR FILE "!"
	CAUSE	The input or output file you used with the link editor command does not have the correct filecode.
	ACTION	Be sure you have spelled the file's name correctly. If you have, check the description for the command you are using to be sure you are using the correct file as your input source or output destination.
1125	MESSAGE	MISSING OPEN QUOTE FOR REDO/DO STRING
	CAUSE	You specified a search string for the REDO/DO command that included a closing quote (either ' or ") but omitted the corresponding opening quote.
	ACTION	Re-enter the REDO/DO command, placing quotes at both ends of the search string.
1126	MESSAGE	MISSING CLOSE QUOTE FOR REDO/DO STRING
	CAUSE	You specified a search string for the REDO/DO command that included an opening quote (either ' or ") but omitted the corresponding closing quote.
	ACTION	Re-enter the REDO/DO command, placing quotes at both ends of the search string.
1127	MESSAGE	SEARCH STRING SPECIFIED FOR REDO/DO IS INVALID
	CAUSE	A search string for a DO or REDO command must be enclosed between a matching set of opening and closing quotes. You may use either a single

quote mark (') or double quotes (\"), but you can't mix the two.

`ACTION` Re-enter the `REDO/DO` command, using the same quotation marks at both ends of the search string.

1128	MESSAGE	REDO/DO NUMBER IS OUT OF RANGE OF REDO STACK
	CAUSE	The DO and REDO commands accept either a search string or an integer parameter that identifies an entry on the redo stack. In this case, the number you supplied does not correspond to any of the values on the redo stack.
	ACTION	Use the LISTREDO command to verify the command number you wish to repeat.
1129	MESSAGE	NO MATCH ENCOUNTERED FOR REDO/DO SEARCH STRING
	CAUSE	The string pattern that you attempted to match on the redo stack does not exist.
	ACTION	Use the LISTREDO command to help you select the command you want to repeat.
1130	MESSAGE	REDO/DO CANNOT BE THE FIRST COMMAND ENTERED
	CAUSE	You entered a DO or REDO command as the first command within a HP Link Editor/XL session. As no commands currently exist on the redo stack, the DO and REDO commands are illegal.
	ACTION	Enter another HP Link Editor/XL command to initiate the current session.
1131	MESSAGE	NO HELP FOUND FOR COMMAND "!"
	CAUSE	You requested help with a command that does not exist.
	ACTION	Verify that you have spelled the command correctly and that the

command is a valid HP Link Editor/XL command (or the word "help").

1132	MESSAGE	ATTEMPT TO REDO FAILED
	CAUSE	You have edited the response from a REDO command, but the resulting command is invalid. The link editor ignores the edited command.
	ACTION	Refer to the description of the REDO command in the *MPE XL Commands Reference Manual*, then use the supported conventions for editing commands.
1133	MESSAGE	ENTRY NAME SPECIFIED BUT NOT FOUND
	CAUSE	You have used the ENTRY= option with the LINK command to request an alternate entry point, but HP Link Editor/XL failed to find the requested symbol.
	ACTION	Check that the specified symbol is spelled correctly (case sensitive) and that the module that defines this symbol is included in the files you are linking together. Current compilers can't generate secondary entry points.
1134	MESSAGE	"!" IS NOT A VALID "XL=" FILENAME
	CAUSE	You have supplied a filename for the XL= option of the LINK command which does not conform to the syntax of a valid filename.
	ACTION	Check that the specified filename is of the right form. The account, group, and file names must all be valid lengths, and they must contain legal characters.
1135		

MESSAGE	ATTEMPT TO WRITE TO FILE LINKLIST FAILED. FILE MAY BE FULL
CAUSE	When attempting to write to a file specified as the LINKLIST file in a file equation, the end of file was encountered.
ACTION	Exit HP Link Editor/XL and increase the size of the LINKLIST file.

1137	MESSAGE	FILE "!" HAS A CORRUPT FILE END
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which the HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that you have spelled the name correctly. If the file name is correct, replace or rebuild the file.
1138	MESSAGE	FILE "!" HAS A CORRUPT STRING AREA
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that you have spelled the name correctly. If the file name is correct, replace or rebuild the file.
1139	MESSAGE	FILE "!" HAS A CORRUPT SYMBOL TABLE
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that you have spelled the name correctly. If the file name is correct, replace or rebuild the file.

1140	MESSAGE	FILE "!" HAS A CORRUPT IMPORT TABLE
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that you have spelled the name correctly. If the file name is correct, replace or rebuild the file.

1141	MESSAGE	FILE "!" HAS A CORRUPT MODULE
	CAUSE	The named file has the correct filecode but contains a corrupted field in its module directory. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that you have spelled the name correctly. If the file name is correct, replace or rebuild the file.
1142	MESSAGE	THE OUTPUT FILE IS FULL
	CAUSE	A listing file, to which HP Link Editor/XL tried to write, is full.
	ACTION	Exit HP Link Editor/XL and check the limit of the listing file. If it is full, recreate the file with a larger limit.
1143	MESSAGE	ILLEGAL CAPABILITY SPECIFICATION
	CAUSE	You do not have the capability you specified on the CAP= keyword of the LINK command. (USER capabilities are not sufficient.)
	ACTION	Either have your System Manager give you the capability you need, or do not specify that capability on the CAP= keyword.
1144	MESSAGE	ATTEMPT TO OPEN LKEDCAT.PUB.SYS FAILED
	CAUSE	HP Link Editor/XL cannot open its error message catalog,

LKEDCAT.PUB.SYS.

ACTION — Check to be sure LKEDCAT.PUB.SYS exists. If not, contact your System Manager.

1145	MESSAGE	LIBRARY "!" IS CORRUPT. USE EITHER A CLEANRL OR A CLEANXL COMMAND TO CLEAN THE LIBRARY
	CAUSE	The library contains errors.
	ACTION	If this is a relocatable library, use the CLEANRL command to clean it. If it is an executable library, use the CLEANXL command.
1146	MESSAGE	LIBRARY "!" IS CORRUPT
	CAUSE	The library contains errors.
	ACTION	Re-create the library.
1147	MESSAGE	OPERATION ON THIS OBJECT MODULE (!) REQUIRES SM CAPABILITY
	CAUSE	The relocatable object module executes at privilege level 0.
	ACTION	You must have SM capability to use ADDXL or to link this module.
1148	MESSAGE	THIS LIMIT HAS TO BE >= THE CURRENT NUMBER OF MODULES (!) OR <= MAX MODULE LIMIT (400000)
	CAUSE	You have used an illegal value for the LIMIT parameter of the CLEANRL or CLEANXL command.
	ACTION	Re-enter the command using a value which is greater than or equal to the number of modules already existing in the library, and less than or equal to the maximum modules allowed (400000)

1149	MESSAGE	THE STRING TABLE IN FILE ! DOES NOT HAVE ENOUGH SPACE FOR THE STRING !
	CAUSE	While doing an ALTPROG command using the XL=, ENTRY=, or UNSAT= keywords, the file you tried to alter does not have enough room in its string table to hold the new string.
	ACTION	Use the :RUN command to override the attributes stored in the program file. Or, link the program file again with the desired keyword specified on the LINK command.
1150	MESSAGE	! IS NOT A LEGAL VALUE FOR THE PRI PARAMETER. IT MUST BE IN THE RANGE 100-255 OR ONE OF [AS, BS, CS, DS, ES]
	CAUSE	While doing an ALTPROG or LINK command you specified an invalid value for the PRI= keyword.
	ACTION	Redo the LINK or ALTPROG command this time specifying a value for the PRI= keyword which is between 100 and 255 inclusive, or the values AS, BS, CS, DS, ES.
1151	MESSAGE	! IS NOT A LEGAL VALUE FOR THE MAXPRI PARAMETER. IT MUST BE IN THE RANGE 100-255 OR ONE OF [AS, BS, CS, DS, ES]
	CAUSE	While doing an ALTPROG or LINK command you specified an invalid value for the MAXPRI= keyword.
	ACTION	Redo the LINK or ALTPROG command this time specifying a value for the MAXPRI= keyword which is between 100 and 255 inclusive, or the values AS, BS, CS, DS, ES.

1154	MESSAGE	THE ! FIELD DOES NOT EXIST IN PROGRAM FILE !. THE FIELD WAS NOT CHANGED.
	CAUSE	While doing an ALTPROG command you specified a keyword which changes a field in the program file which does not exist in the program file you specified. This can only happen if you are trying to alter the PRI or MAXPRI fields in a program file built with an older link editor.
	ACTION	If the fields are necessary for your application, you must link the application with this link editor. Otherwise, if the fields are not relevant to you do not specify them on the ALTPROG command line.

Warning Messages (1500-1999)

Warning messages signal potential error situations. HP Link Editor/XL produces an executable program file, but erroneous results may occur at run time.

1502	MESSAGE	INCOMPATIBLE NUMBER OF ARGUMENTS: ! (!, !)
	CAUSE	The named procedure is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible.
	ACTION	Check the source code and correct the incompatibility, or ignore the warning.
1503	MESSAGE	INCOMPATIBLE PACKING: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or ignore the warning.
1504	MESSAGE	INCOMPATIBLE ALIGNMENT: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.

`ACTION`	Check the source code and correct the incompatibility, or ignore the warning.

1505	MESSAGE	INCOMPATIBLE MODE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or ignore the warning.
1506	MESSAGE	INCOMPATIBLE STRUCTURE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or ignore the warning.
1507	MESSAGE	INCOMPATIBLE TYPE: ! (!, !)
	CAUSE	The named procedure or global variable is referenced in one source file and defined in another source file, but the type checking information indicates that the two declarations are incompatible. If the symbol refers to a procedure and the incompatibility is with its parameters, the parameter number is listed on the following line.
	ACTION	Check the source code and correct the incompatibility, or ignore the warning.

1509	MESSAGE	COMMON BLOCK REQUESTS FOR "!" HAVE DIFFERENT LENGTHS
	CAUSE	The named common block was defined in one subprogram with a longer length than its initialization length.
	ACTION	Since this is a warning, no action is required. HP Link Editor/XL allocates enough space for the common block with the larger size, but only the part declared in the BLOCK DATA subprogram is initialized.

1510	MESSAGE	INVALID SYMBOL TYPE FOR PLABEL (!, !)
	CAUSE	HP Link Editor/XL was asked to create a procedure label for the named symbol in the given source file, but the symbol is not a procedure name.
	ACTION	This message usually indicates a compiler error. Record the steps that produced this error then report the problem to your System Manager.
1511	MESSAGE	INNER-QUADRANT BRANCH IN "!"
	CAUSE	A branch in the code must cross a quandrant boundary. Usually, this is caused by a reference from code to data.
	ACTION	Avoid using branches into data.
1512	MESSAGE	QUADRANT CHANGE IN RELOCATABLE EXPRESSION
	CAUSE	A relocatable expression uses symbols from different quadrants.
	ACTION	Avoid expressions which use both code and data symbols.

System Errors (2000-2999)

System errors are file and system resource errors. They cause the command that you entered to fail. Contact your System Manager to resolve the problem.

2001 MESSAGE: ATTEMPT TO SEEK IN "!" FAILED

CAUSE: This message usually indicates that an input file has been corrupted, or that an output file cannot be written because of system resource limitations. An MPE XL file system error message is usually printed to help clarify the problem.

ACTION: If the problem pertains to a relocatable object file, try recompiling the source file that produced that object module. If you have exceeded the file space limit for your group, either purge unwanted files or ask your System Manager to extend your file space limit. If the problem persists, document the steps that produced this error and report the problem to your System Manager.

2002 MESSAGE: ENCOUNTERED UNEXPECTED END OF FILE IN "!"

CAUSE: This message usually indicates that an input file has been corrupted. An MPE XL file system error message is usually printed to help clarify the problem.

ACTION: If the problem pertains to a relocatable object file, try recompiling the source file that produced that object module. If you have exceeded the file space limit for your group, either purge unwanted files or ask your System Manager to extend your file space limit. If the problem persists, document the steps that produced this error and report the problem to your System Manager.

2003	MESSAGE	ATTEMPT TO READ FROM "!" FAILED
	CAUSE	This message usually indicates that an input file has been corrupted. An MPE XL file system error message is usually printed to help clarify the problem.
	ACTION	If the problem pertains to a relocatable object file, try recompiling the source file that produced that object module. If you have exceeded the file space limit for your group, either purge unwanted files or ask your System Manager to extend your file space limit. If the problem persists, document the steps that produced this error and report the problem to your System Manager.
2004	MESSAGE	ATTEMPT TO WRITE TO "!" FAILED
	CAUSE	This message usually indicates that the file cannot be written because of system resource limitations. An MPE XL file system error message is usually printed to help clarify the problem.
	ACTION	If you have exceeded the file space limit for your group, either purge unwanted files or ask your System Manager to extend your file space limit. If the problem persists, document the steps that produced this error and report the problem to your System Manager.
2005	MESSAGE	OUT OF MEMORY
	CAUSE	HP Link Editor/XL has run out of virtual memory while trying to link a program. This can occur if your system is extremely low on disc space (so insufficient room exists for swapping), or if the program you're linking attempts to declare a very large array.

ACTION

Check with your System Manager to verify that the system has enough free disc space for swapping. If the program declares a very large array, try to revise the program so that the data is allocated dynamically rather than by a static array declaration; otherwise, try to reduce the size of the array.

2100	MESSAGE	ATTEMPT TO CLOSE FILE "!" FAILED
	CAUSE	A file system error has occurred while HP Link Editor/XL was trying to closed the indicated file.
	ACTION	Using the accompanying file system error message, correct the problem and try the command again. If the problem persists, document the steps that produced the error and report the problem to your System Manager.
2101	MESSAGE	ATTEMPT TO OPEN LINKLIST FILE FAILED
	CAUSE	A file system error occurred when the link editor attempted to open LINKLIST.
	ACTION	Using the accompanying file system error message, correct the problem and try the command again. If the problem persists, document the steps that produced the error and report the problem to your System Manager.
2102	MESSAGE	ATTEMPT TO GET FILEINFO FOR LINKLIST FILE FAILED
	CAUSE	A file system error occurred when the link editor attempted this operation.
	ACTION	Using the accompanying file system error message, correct the problem and try the command again. If the problem persists, document the steps that produced the error and report the problem to your System Manager.
2103	MESSAGE	ATTEMPT TO CLOSE LINKLIST FILE FAILED

`CAUSE` A file system error occurred when HP Link Editor/XL tried to close the `LINKLIST` file.

`ACTION` Using the accompanying file system error message, correct the problem and try the command again. If the problem persists, document the steps that produced the error and report the problem to your System Manager.

2105	MESSAGE	ATTEMPT TO OPEN LKEDHELP.PUB.SYS FAILED
	CAUSE	HP Link Editor/XL failed to find the LKEDHELP.PUB.SYS file.
	ACTION	Check that the LKEDHELP.PUB.SYS file is present on your system and that it has not been overwritten by some other file. Also ensure that you do not have a file equate for LKEDHELP.PUB.SYS.
2106	MESSAGE	OUT OF DISK SPACE
	CAUSE	HP Link Editor/XL has run out of disk space. This usually results when insufficient room exists for swapping.
	ACTION	Check with your System Manager to verify that the system has enough free disk space.

Language Subsystem Errors (3000-3999)

Language subsystem errors usually indicate errors in an input relocatable object file. The relocatable object file may have been corrupted after compilation, or there may be a compiler error. Save the corrupt version of the file in case you need it when reporting the error. Recreate the file and retry the command. If the error recurs, document the steps that produced it and contact your System Manager.

3001	MESSAGE	ILLEGAL SYMBOL INDEX IN FIXUP RECORD
	CAUSE	A "fixup request" record in a relocatable object file indicates a reference to a symbol, but the symbol index is outside the range of indices for the symbol table in that relocatable object file. (This problem can arise if a source file compiles with errors and the compiler does not purge the resulting relocatable object file.) This message is followed by another line that indicates the relocatable object file and subspace offset which require the fixup.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3002	MESSAGE	!: SYMBOL ! HAS INVALID STRING INDEX
	CAUSE	An entry in the symbol table of the named relocatable object file contains an invalid pointer to the symbol name.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3003	MESSAGE	NO BSS SUBSPACE DEFINED
	CAUSE	HP Link Editor/XL allocates space for all uninitialized common blocks in a subspace named BSS. This subspace must be defined in one of the input files, and is normally defined in

NRTO.LIB.SYS.

ACTION — Check that the NRTO.LIB.SYS file is present on your system and that it has not been overwritten with some other file. Also ensure that you do not have a file equate for NRTO.LIB.SYS. If the problem persists, document the steps that produced this error and report the problem to your System Manager.

3004	MESSAGE	CAN'T FIND SUBSPACE FOR !
	CAUSE	HP Link Editor/XL automatically defines three symbols for programs: `etext`, `edata` and `end`. These symbols are placed at the end of the code, initialized data, and uninitialized data subspaces, respectively. The `$DATA$` and `$BSS$` subspaces must be defined in one of the input files, and are normally defined in `NRT0.LIB.SYS`.
	ACTION	Check that the `NRT0.LIB.SYS` file is present on your system and that it has not been overwritten with some other file. Also ensure that you do not have a file equate for `NRT0.LIB.SYS`. If the problem persists, document the steps that produced this error and report the problem to your System Manager.
3005	MESSAGE	NO $UNWIND_END$ SUBSPACE DEFINED
	CAUSE	HP Link Editor/XL generates descriptors for all long branch, import and export stubs that it creates. These descriptors are required for unwinding the stack during a stack trace or a non-local escape operation. The `$UNWIND_END$` subspace must be defined in one of the input files, and is normally defined in `NRT0.LIB.SYS`.
	ACTION	Check that the `NRT0.LIB.SYS` file is present on your system and that it has not been overwritten with some other file. Also ensure that you do not have a file equate for `NRT0.LIB.SYS`. If the problem persists, document the steps that produced this error and report the problem to your System Manager.
3006	MESSAGE	FIRST INIT POINTER IS NOT CODE

CAUSE

In order to generate an executable file that the loader can accept, the subspace records for the "code" space must precede those for the "data" space. Normally, the declarations contained in the NRTO.LIB.SYS file ensure that this happens.

ACTION

Check that the NRTO.LIB.SYS file is present on your system and that it has not been overwritten with some other file. Also ensure that you do not have a file equate for NRTO.LIB.SYS. If the problem persists, document the steps that produced this error and report the problem to your System Manager.

3008	MESSAGE	TOO MANY LOADABLE SPACES
	CAUSE	The loader rejects an executable file that contains more than two loadable spaces (a $TEXT$ space for code and a $PRIVATE$ space for data). If a program attempts to declare any other loadable space, HP Link Editor/XL issues this error message.
	ACTION	This is a compiler error. Record the steps that produced this error and report it to your System Manager.
3009	MESSAGE	LOADABLE SPACE MISMATCH
	CAUSE	HP Link Editor/XL has found two relocatable files that declare the same space as both a loadable and an unloadable space.
	ACTION	This is a compiler error. Record the steps that produced this error and report it to your System Manager.
3010	MESSAGE	PRIVATE SPACE MISMATCH
	CAUSE	HP Link Editor/XL found more that one relocatable module that declares the same space as both private and non-private.
	ACTION	This is a compiler error. Record the steps that produced this error and report it to your System Manager.
3012	MESSAGE	ILLEGAL ARGUMENT COMBINATION (!,!)
	CAUSE	A parameter relocation stub cannot be built to transfer the arguments.

ACTION This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3013	MESSAGE	SUBSPACE LENGTH NOT EQUAL INIT LENGTH
	CAUSE	A relocatable object module contains a subspace with a non-zero initialization length but a larger subspace length.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.
3014	MESSAGE	!: UNSUPPORTED SYMBOL TYPE (!) FOR UNSAT SYMBOL
	CAUSE	A relocatable object module contains an unsatisfied symbol with an illegal symbol type. The valid symbol types for unsatisfied symbols are: CODE, DATA, MILLICODE, STORAGE, ABSOLUTE, MILLI_EXT, and NULL.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.
3015	MESSAGE	!: UNSUPPORTED SYMBOL TYPE (!) FOR UNIVERSAL SYMBOL
	CAUSE	A relocatable object module contains a universal symbol with an illegal symbol type. The valid symbol types for universal symbols are: CODE, DATA, PRI_PROG, SEC_PROG, ENTRY, MILLICODE, ABSOLUTE, MILLI_EXT, MODULE, and NULL.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.
3016		

MESSAGE FIXUP HAS INVALID EXPRESSION SELECTOR

CAUSE A relocatable object module contains a corrupt "fixup request" record. The one or two lines that follow this message gives the file and subspace that require the fixup, and the symbol which is referenced by that location.

ACTION This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3017	MESSAGE	FIXUP HAS INVALID FIELD SELECTOR
	CAUSE	A relocatable object module contains a corrupt "fixup request" record. The one or two lines that follow this message gives the file and subspace that require the fixup, and the symbol which is referenced by that location.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3018	MESSAGE	FIXUP HAS INVALID FORMAT SELECTOR
	CAUSE	A relocatable object module contains a corrupt "fixup request" record. The one or two lines that follow this message gives the file and subspace that require the fixup, and the symbol which is referenced by that location.
	ACTION	This is a compiler error. Record the steps that produced this error and report the problem to your System Manager.

3019	MESSAGE	DATA ADDRESS IS OUT OF RANGE FOR SHORT LOAD OR STORE
	CAUSE	A load or store instruction is attempting to reach a memory location that is not reachable through a single instruction. (Normally, compilers always generate a two-instruction sequence for loading or storing data, unless the location is guaranteed to be within the range of a single instruction.) The two lines that follow this message give the file and subspace offset of the load or store instruction, and the symbol which is being referenced.

ACTION This is a compiler error. Record the steps that produced this error and report it to your System Manager.

3020	MESSAGE	TARGET OF CONDITIONAL BRANCH IS OUT OF RANGE
	CAUSE	A conditional branch instruction is attempting to reach a memory location that is not reachable through a single instruction. Normally, HP Link Editor/XL replaces single-instruction branches with a chained branch when necessary. The two lines that follow this message give the file and subspace offset of the branch instruction, and the symbol which is being referenced.
	ACTION	This is a compiler error. Record the steps that produced this error and report it to your System Manager.

3021	MESSAGE	TARGET OF UNCONDITIONAL BRANCH IS OUT OF RANGE
	CAUSE	An unconditional branch instruction is attempting to reach a memory location that is not reachable through a single instruction. Normally, HP Link Editor/XL replaces single-instruction branches with a chained branch when necessary. The two lines that follow this message give the file and subspace offset of the branch instruction, and the symbol which is being referenced.
	ACTION	This is a compiler error. Record the steps that produced this error and report it to your System Manager.

3022	MESSAGE	TARGET OF ABSOLUTE BRANCH IS OUT OF RANGE
	CAUSE	An absolute branch instruction is attempting to reach a memory location that is not reachable through a single instruction. Normally, HP Link Editor/XL replaces single-instruction

branches with a chained branch when necessary. The two lines that follow this message give the file and subspace offset of the branch instruction, and the symbol which is being referenced.

`ACTION` This is a compiler error. Record the steps that produced this error and report it to your System Manager.

3024	MESSAGE	FIXUP REFERS TO INVALID SYMBOL
	CAUSE	A "fixup request" record indicates that a reference is being made to a symbol whose symbol type is not valid for the fixup.
	ACTION	This is typically a compiler error. Document the steps that produced this error and report the problem to your System Manager.

3027	MESSAGE	INVALID FIXUPS EXIST
	CAUSE	When HP Link Editor/XL encounters one or more problems with a "fixup request" record, it prints this summary message at the end of the link operation.
	ACTION	Refer to the "action" description for the previously listed error messages. Once you have resolved those problems, this message goes away.

3028	MESSAGE	!: NOT A VALID LIBRARY (INVALID MAGIC NUMBER)
	CAUSE	The named file has the correct filecode but contains an incorrect "magic number" in its library header record. This usually indicates that the relocatable library file has been corrupted, or that a non-library file has been created with an NMRL filecode.
	ACTION	Verify that you have specified a relocatable library file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, replace or rebuild the file.

3029

MESSAGE `!: CORRUPT LIBRARY FILE (INCORRECT HEADER CHECKSUM)`

CAUSE The named file has the correct filecode but contains an incorrect "checksum" in its library header record. This usually indicates that the relocatable library file has been corrupted, or that a non-library file has been created with an NMRL filecode.

ACTION Verify that you have specified a relocatable library file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, replace or rebuild the file.

3030	MESSAGE	!: MISSING LIBRARY SYMBOL TABLE
	CAUSE	The named relocatable library file has no symbol table. This usually indicates that the relocatable library file has been corrupted, or that a non-library file has been created with an NMRL filecode.
	ACTION	Verify that you have specified a relocatable library file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, replace or rebuild the file.
3031	MESSAGE	!: NOT A VALID OBJECT FILE (INVALID MAGIC NUMBER)
	CAUSE	The named file has the correct filecode but contains an incorrect "magic number" in its header record. This usually indicates that the relocatable object file has been corrupted, or that a non-relocatable object file has been created with an NMOBJ filecode.
	ACTION	Verify that you have specified a relocatable object file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, recreate the file.
3032	MESSAGE	!: NOT A VALID OBJECT FILE (INVALID VERSION ID)
	CAUSE	The named file has the correct filecode but contains an incorrect "version id" in its header record. This usually indicates that the relocatable object file has been corrupted, or that a non-relocatable object file has been created with an NMOBJ filecode.
	ACTION	Verify that you have specified a relocatable object file, that it is the

correct one and that its name is spelled correctly. It the file and filecode are correct, recreate the file.

3033	MESSAGE	!: CORRUPT OBJECT FILE (INCORRECT HEADER CHECKSUM)
	CAUSE	The named file has the correct filecode but contains an incorrect "checksum" in its header record. This usually indicates that the relocatable object file has been corrupted, or that a non-relocatable object file has been created with an NMOBJ filecode.
	ACTION	Verify that you have specified a relocatable object file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, recreate the file.

3034	MESSAGE	!: NOT A VALID OBJECT FILE (INVALID SYSTEM ID)
	CAUSE	The named file has the correct filecode but contains an incorrect "system id" in its header record. This usually indicates that the relocatable object file has been corrupted, or that a non-relocatable object file has been created with an NMOBJ filecode.
	ACTION	Verify that you have specified a relocatable object file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, recreate the file.

3035	MESSAGE	MISSING SYMBOL EXTENSION RECORD: ! (!)
	CAUSE	A relocatable object file has requested HP Link Editor/XL to perform type checking, but it does not contain the required symbol extension records that provide the type checking information.

	ACTION	This message indicates a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3036	MESSAGE	FIXUP HAS A SELECTOR MISMATCH
	CAUSE	An invalid fixup request was found in a relocatable object file during the fixup translation in the linker.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3037	MESSAGE	FIXUP IS NOT A PROCEDURE CALL
	CAUSE	A fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3038	MESSAGE	BAD UNWIND DESCRIPTOR
	CAUSE	A corrupt unwind descriptor was found in a relocatable object file.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3039	MESSAGE	BAD RECOVER DESCRIPTOR
	CAUSE	A corrupt recover descriptor was found in a relocatable object file.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3040	MESSAGE	UNUSED FIXUP
	CAUSE	A fixup request was encountered that was left unused. This typically means that an invalid fixup request exists in the relocatable object file.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3041		

MESSAGE	ILLEGAL COMBINATION OF RELOCATABLE SYMBOLS
CAUSE	A fixup request record in a relocatable object file is invalid.
ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3042	MESSAGE	EXPRESSION STACK UNDERFLOW
	CAUSE	A fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3043	MESSAGE	INVALID FIXUP: "!"
	CAUSE	A fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3044	MESSAGE	INVALID FRAME SIZE: "!"
	CAUSE	A fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3045	MESSAGE	FIXUP APPLIED TO INSTRUCTION: "!"
	CAUSE	A fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3046	MESSAGE	INVALID LOADER FIXUP NEEDED
	CAUSE	The linker has determined that a loader fixup needs to be generated, but the

target of the loader fixup is invalid. It is illegal to have initialized code pointers point into data. Make sure that the code does not contain code pointers which are initialized to point into data (it is possible to make this error when writing in assembly).

ACTION

If the code does not contain initialized code pointers which point into data, or the program consists of only high-level source language (eg. Pascal, COBOL, etc.), then this is a compiler error. Record the steps that produced the error and report the problem to your System Manager.

3047	MESSAGE	INVALID ALIGNMENT OF DATA
	CAUSE	The alignment specified in a fixup request record in a relocatable object file is invalid.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3048	MESSAGE	DIVISION BY 0 IN AN EXPRESSION
	CAUSE	A fixup request in a relocatable file contains a DIV expression that has an invalid (0) divisor.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3049	MESSAGE	TRY-RECOVER STACK UNDERFLOW
	CAUSE	A pop was attempted on an empty try-recover stack.
	ACTION	This is a compiler error. Record the steps that produced the error and report the problem to your System Manager.
3100	MESSAGE	!: NOT A VALID LIBRARY/PROGRAM FILE (CORRUPT FILE ID)
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that its name is spelled correctly.

		It the file and filecode are correct, replace or rebuild the file.
3101	MESSAGE	!: NOT A VALID OBJECT FILE (CORRUPT FILE ID)
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have specified a relocatable object file, that it is the correct one and that its name is spelled correctly. It the file and filecode are correct, replace or rebuild the file.

3102	MESSAGE	!: NOT A VALID PROGRAM FILE (CORRUPT FILE ID)
	CAUSE	The named file has the correct filecode but contains a corrupted field in its header record. This usually indicates that the file has been corrupted, or that an invalid file has been created with a filecode which HP Link Editor/XL recognizes.
	ACTION	Verify that you have entered the correct file and that its name is spelled correctly. It the file and filecode are correct, replace or rebuild the file.
3103	MESSAGE	!: MORE OBJECT MODULES EXPECTED IN THIS OBJECT FILE
	CAUSE	This relocatable object file contains a series of relocatable object modules and one of this series of relocatable object modules is missing. This usually indicates that the file has been corrupted.
	ACTION	Verify that you have included the correct file and that you have spelled the name correctly. If the spelling is correct, replace or rebuild the file.
3104	MESSAGE	HEAP ALLOCATION ERROR
	CAUSE	This message usually indicates that HP Link Editor/XL attempted to allocate heap space and system resource limitations caused the allocation to fail.
	ACTION	If this error persists, document the steps that produced this problem and report the problem to your System Manager.

Internal Errors (4000-4999)

Internal errors are errors caused by the link editor subsystem. Internal errors cause link editor commands to abort. If you encounter any of these messages, document the steps that produced the error, then report the problem and error number to your Hewlett-Packard representative.

B

Using HP Link Editor/XL with HP COBOL II/XL

This appendix discusses the HP COBOL II/XL compiler conventions that relate specifically to HP Link Editor/XL. The following items explain how you use the conventions to successfully create executable program files. (For details about compiler conventions, see the *HP COBOL II Reference Manual*, the *HP COBOL II/XL Programmer's Guide* or the *HP COBOL II/XL Reference Manual Supplement*.)

- **Compilation units**

  If you do not use the `RLFILE` option of the `$CONTROL` directive, the entire source file is treated as one compilation unit and the compiler produces a relocatable object file containing a single relocatable object module.

  If you use `RLFILE`, each concatenated (program unit that is not contained in another program) is a separate compilation unit and results in a separate relocatable object module in the relocatable library.

- **Relocatable object module name**

  If you do not use the `RLFILE` option of the `$CONTROL` directive, the relocatable object module name is the unqualified name of the COBOL source file.

  If you use `RLFILE`, the relocatable object module name is the `PROGRAM-ID` name of that module.

- **Program entry point**

  The program entry point is the name specified in the `PROGRAM-ID` statement. Secondary entry points are specified by the `ENTRY` verb.

- **Scope of data**

  External data and files are relative to the name of the data item or file.

  Internal and local data has local scope.

  Data in the main program is relative to `$global$`.

  Data is OWN (relative to `M$` *n*`$` *name* in the map, where `n` is a number and `name` is the module name) if the `PROGRAM-ID` paragraph does not contain an `INITIAL` clause, or if the program contains one of the following compiler directives:

```
$CONTROL ANSISUB
$CONTROL SUBPROGRAM
```

Data is local (SP relative) if the `PROGRAM-ID` paragraph contains an `INITIAL` clause, or if the program contains the `$CONTROL DYNAMIC` directive.

- **Locality sets**

  When an HP COBOL II relocatable object file contains multiple chunks of code, the locality set name is the name entered for the `PROGRAM-ID`. When the relocatable object file contains a single chunk, no locality set is used.

- **Type checking**

  The compiler generates the following type checking information for parameters in non-intrinsic `CALL` statements:

  1. A type checking level of 3.
  2. The parameter-passing method is by reference or value.
  3. The alignment (but not the type) of each identifier passed by reference or by content.
  4. The type of each identifier passed by value.
  5. The type of identifier in the `GIVING` clause of the `CALL` statement.

  The compiler generates the following type checking information for the formal parameters declared in the `USING` clause of the Procedure Division header or `ENTRY` statement.

  1. A type checking level of 3.
  2. The parameter-passing method is by reference.
  3. The alignment of each parameter. The alignment is assumed to be on byte boundaries unless you use `LINKALIGNED16` or `LINKALIGNED` for the `OPTFEATURES=` parameter of the `$CONTROL` compiler directive. If you use `LINKALIGNED16`, alignment is assumed to be on 16-bit boundaries and if you use `LINKALIGNED`, alignment is assumed to be on 32-bit boundaries.

  To override the type checking level used during compilation, use the
  `PARMCHECK=` *check_level* parameter of the `LINK` command.

C

Using HP Link Editor/XL with HP FORTRAN 77/XL

This appendix discusses the HP FORTRAN 77/XL compiler conventions that relate specifically to HP Link Editor/XL. The following items explain how you use these conventions to successfully create executable program files. (For details about the compiler conventions shown, see the *HP FORTRAN 77/XL Reference Manual*.)

- **Compilation units**

  If you do not use the `$RLFILE` compiler directive, the entire source file is treated as one compilation unit and the compiler produces a relocatable object file containing a single relocatable object module.

  If you use `$RLFILE`, each program unit (main program, subroutine, function, or block data subprogram) is treated as a separate compilation unit and results in a separate relocatable object module in the relocatable library.

- **Relocatable object module name**

  If you do not use the `$RLFILE` compiler directive, the relocatable object module name is the unqualified name of the HP FORTRAN 77 source file.

  If you use `$RLFILE`, the relocatable object module name is the name of that program unit or module. If the program is a block data subprogram, you must use the `BLOCKDATA` parameter (not the `MODULE` parameter) to reference it when using the link editor.

- **Program entry point**

  The program entry point is the first executable statement in the main program. This name is `main__` unless you enter a different name in the `HP FORTRAN 77` program statement.

- **Scope of variables, functions and procedures**

  Names of `COMMON` blocks, subroutines and functions have universal scope. When compiled separately, `HP FORTRAN 77` subroutines and functions do not generate a dummy main block.

- **Locality sets**

  The `HP FORTRAN 77 $LOCALITY` compiler directive associates program units with a locality set. The name of the locality set must begin with a letter. It can contain up to 32 alphanumeric characters. Place the `$LOCALITY` directive before the program

unit(s) to be placed in a locality set (the directive remains in effect until the next `$LOCALITY` directive).

If you do not use the `$LOCALITY` directive, `CODE` is the default locality set name.

- **Type checking**

  Two compiler directives specify the level of type checking that the link editor uses when resolving references between HP FORTRAN 77 subroutines and functions. They are $CHECK_FORMAL_PARM and $CHECK_ACTUAL_PARM and they are described in the next two paragraphs.

  $CHECK_FORMAL_PARM associates a check level with the formal declaration of the subroutine or function. Place $CHECK_FORMAL_PARM before the declaration of the subroutine or function (it remains in effect until a new directive is encountered). If you do not use $CHECK_FORMAL_PARM, the compiler uses type checking level 3.

  $CHECK_ACTUAL_PARM associates a check level with each subroutine or function call encountered. You can place $CHECK_ACTUAL_PARM anywhere in the source file. It remains in effect until a new directive is encountered. If you do not use $CHECK_ACTUAL_PARM, the compiler uses type checking level 3.

  To override the type checking level used during compilation, use the PARMCHECK option of the LINK command (see Chapter 4).

D

Using HP Link Editor/XL with HP Pascal/XL

This appendix discusses the HP Pascal/XL compiler conventions that relate specifically to HP Link Editor/XL. The following items explain how you use these conventions to successfully create executable program files. (For details about the compiler conventions shown, see the *HP Pascal Programmer's Guide*.)

- **Compilation units**

  If you do not use the `$RLFILE` compiler option, the entire source file is treated as one compilation unit and the compiler produces a relocatable object file containing a single relocatable object module.

  If you use `$RLFILE`, each PASCAL MODULE or non-nested procedure is a separate compilation unit and results in a separate relocatable object module in the relocatable library.

- **Relocatable object module name**

  If you do not use the `$RLFILE` compiler option, the relocatable object module name is the unqualified name of the Pascal source file.

  If you use `$RLFILE`, the relocatable object module name is the name of the corresponding non-nested procedure or module in the source file.

- **Program entry point**

  The first executable statement in the outer block is the program entry point. The outer block's name is `PROGRAM`.

- **Scope of variables, functions and procedures**

  Outer block variables (declared with the `GLOBAL` compiler option) and all modules and level one procedures and functions, have universal scope.

  The `SUBPROGRAM` option lets you compile a subset of level one procedures or functions. It also lets you select parts of a large program for compilation. Because each compilation creates a new relocatable object module, you can't recompile into an existing file and retain the old code. The `SUBPROGRAM` option is similar to the `EXTERNAL` option in that the outer block is not compiled. Variables can be coupled with an outer block that is compiled with the `GLOBAL` option.

The EXTERNAL option can be used with the GLOBAL option to compile procedures and functions separately. When EXTERNAL appears in a source file, the compiler generates information about global variables. This allows them to be coupled with identical variables in an outer block that are compiled with the GLOBAL option. The compiler generates object code only for procedures and functions; not for the statement part of the outer block.

Routines in an executable library cannot reference program globals. This includes `INPUT` and `OUTPUT`.

The `GLOBAL` compiler option prepares information about all global variables declared in the outer block. It allows the variables to be coupled with identical variables that are compiled with the `EXTERNAL` option. The compiler generates object code for the outer block, as well as for all procedures and functions.

- **Locality sets**

  The `LOCALITY` option lets you assign the relocatable object to a new or existing locality set. If you enter a name of an existing locality set, the relocatable object code is placed into that set. If the locality set does not exist, a new locality set is created using that name.

  The `LOCALITY` option remains in effect until a new one is encountered. The `LOCALITY` option can appear anywhere in the source program. However, the object code for an entire procedure is placed into the last locality set used. You cannot place part of a procedure in a locality set.

  If you do not enter the `LOCALITY` option, `CODE` is the default locality set name.

- **Type checking**

  Two compiler options specify the level of type checking that the link editor uses when resolving references between Pascal procedures and functions. They are `$CHECK_FORMAL_PARM` *integer*`$` and `$CHECK_ACTUAL_PARM` *integer*`$` and they are described in the next two paragraphs.

  `CHECK_FORMAL_PARM` associates a check level with the formal declaration of the procedure or function. Place `CHECK_FORMAL_PARM` before the declaration of the procedure or function (it applies only to the procedure or function immediately following it). If you do not use `CHECK_FORMAL_PARM`, the compiler uses type checking level 3.

  `CHECK_ACTUAL_PARM` associates a check level with each procedure or function call encountered. You can place `CHECK_ACTUAL_PARM` anywhere in the source file. It remains in effect until a new directive is encountered. If you do not use `CHECK_ACTUAL_PARM`, the compiler uses type checking level 3.

  To override the type checking level used during compilation, use the `PARMCHECK` option of the `LINK` command (see Chapter 4).

E

Using HP Link Editor/XL with HP C/XL

This appendix discusses the HP C/XL compiler conventions that relate specifically to HP Link Editor/XL. The following items explain how you use these conventions to successfully create executable program files. (For details about the compiler conventions shown, see the *HP C/XL Reference Manual Supplement*.)

- **Compilation units**

  The entire source file is treated as one compilation unit and the compiler produces a relocatable object file containing a single relocatable object module.

- **Relocatable object module name**

  The relocatable object module name is the unqualified name of the C source file.

- **Program entry point**

  The first executable statement in the function `main` is the program entry point.

- **Scope of variables**

  Programs cannot share globals with routines in an executable library. Since the standard C library is part of the executable library `XL.PUB.SYS`, C programs cannot directly reference global variables in it. To access these global variables, link the relocatable library, `LIBCINIT.LIB.SYS` with the program. For example, if your program contains the declaration,

  ```
  extern int errno;
  ```

  you access the C library global `errno` by linking `LIBCINIT` with the program.

- **Locality sets**

  The `LOCALITY` pragma lets you assign the relocatable object to a new or existing locality set. If you enter a name of an existing locality set the relocatable object code is placed into that set. If the locality set does not exist, a new locality set is created using that name.

  The `LOCALITY` pragma remains in effect until a new one is encountered. It can appear anywhere in the source file. However, the object code for the entire function is placed into the last locality set specified. You cannot place part of a function in a locality set.

If you do not enter the `LOCALITY` pragma, `CODE` is the default locality set name.

- **Type checking**

  The C compiler specifies 0 (no checking) as the type checking level for all C functions. There are no compiler directives that let you change the level of type checking that the link editor uses when resolving references between C functions.

F

Differences Between HP Link Editor/XL and MPE V Segmenter

This appendix summarizes the differences between linking programs on MPE V systems using the MPE V Segmenter, and linking programs on MPE XL systems using the HP Link Editor/XL. If you are an experienced MPE V user, it should help you to understand the most important differences between MPE V Segmenter and HP Link Editor/XL.

Differences in the Programming Environment

Creating an efficient programming environment implies the effective use of a computer's resources. Thus, utilities that make demands on a system's processing time and memory allocation must efficiently use the architecture of the parent computer. The main differences between HP Link Editor/XL and the MPE V Segmenter relate to differences between the underlying architecture of the Series 900 systems and the segmented architecture of MPE V systems.

Programs running under MPE V are partitioned into variable-sized pieces called *segments*. Segments are limited to 16K instructions and they group code by logical relationships; you can use them to place related procedures into one contiguous area of virtual memory. This property of segments - grouping code by logical relationships - is called *code locality*.

HP Precision Architecture (HPPA) systems do not have a segmented architecture but they do allow code locality with the use of locality sets. (See "Improving Performance with Locality Sets" in Chapter 7 for information on locality sets.) Locality sets allow the use of more intelligent memory management algorithms, which results in fewer page faults during a program's execution. Since HPPA systems do not have segments, they have no code size restraints and the address space is (effectively) unlimited.

USL Files and Relocatable Object Files

Compilers on both MPE V and MPE XL systems read source files and generate object code for them, consisting of blocks of machine instructions. MPE V compilers create relocatable binary modules (RBMs), which are placed in USL files. MPE XL compilers create relocatable object modules, which are placed in relocatable object files or relocatable libraries.

The primary differences between compiling into a USL file and a relocatable object file are:

MPE V Segmenter	HP Link Editor/XL
■ Compilers produce one RBM for each procedure in a source file and place each of these RBMs into one USL file.	■ Unless you use the `RLFILE` compiler directive, compilers produce only one relocatable object module per compilation unit (source file). This module is placed into a relocatable object file. If you need to manipulate individual procedures or subroutines contained in a source file, use the `RLFILE` directive or put the procedures in separate source files. When you compile from separate source files,a relocatable object file is produced for each and can be manipulated separately.
■ RBMs are the smallest units that the MPE V Segmenter can process.	■ Relocatable object modules are the smallest unit that the HP Link Editor/XL can process.
■ You can use subset compilation to selectively replace RBMs in a USL file.	■ Compilation replaces all procedures within a relocatable object file.
■ Version management is possible with USL files. You access a de-activated RBM by using its index number.	■ Version management is not available. The entire relocatable object file is replaced during each compilation.

Relocatable Libraries

Relocatable libraries on both MPE V and MPE XL systems let you efficiently organize code units. They are similar in that they contain a collection of relocatable code units that are used during linking to resolve external references.

The commands used to manipulate relocatable libraries under MPE V and MPE XL have the following *similarities*:

- The `ADDRL` command adds code units to a relocatable library.
- The `BUILDRL` command creates a relocatable library.
- The `LISTRL` command lists the contents of a relocatable library.
- The `PURGERL` command deletes code units from a relocatable library.
- The `HIDERL` and the MPE V `HIDE` commands hide procedures.

Although HP Link Editor/XL and MPE V Segmenter manage libraries in a similar fashion, they differ in noticeable ways. The following list summarizes the *differences*:

MPE V Segmenter	HP Link Editor/XL
■ You specify the size of the relocatable library by the file size parameter of the `BUILDRL` command.	■ You specify the size of the relocatable library by the `LIMIT` parameter of the `BUILDRL` command.
■ You can selectively add RBMs (procedures) from a USL file to a relocatable library.	■ You can add only relocatable object modules (compilation units) to a relocatable library. Procedures in the same compilation unit cannot be added individually.
■ You cannot copy RBMs from one relocatable library to another and you cannot copy RBMs from a relocatable library back to a USL file.	■ You can copy relocatable object modules from one relocatable library to another. You can also extract relocatable object modules from a relocatable library creating one or more relocatable object files.
■ You can specify only one relocatable library in the `PREP` command.	■ You can specify several relocatable libraries in one `LINK` command.
■ Procedures in a relocatable library produce a single code segment which has size limitations.	■ Relocatable object modules have no size limitations.
■ You cannot partially-link a relocatable library.	■ You can partially-link a relocatable library using the `ADDRL` command with its `RL` and `MERGE` parameters.

Segmented and Executable Libraries

Executable libraries on MPE XL systems are similar to segmented libraries (SLs) on MPE V systems. Executable libraries contain executable code that is used by the loader at run time to resolve external references. Modules in executable and segmented libraries are shared by programs running concurrently.

The following list summarizes the *similarities* between the MPE V Segmenter commands that manage segmented libraries and the HP Link Editor/XL commands that manage executable libraries:

- The `ADDSL` command adds code units to a segmented library and the `ADDXL` command adds code units to an executable library.
- The `BUILDSL` command creates a segmented library and the `BUILDXL` command creates an executable library.
- The `COPYSL` command copies one segmented library to another and the `COPYXL` command copies one executable library to another.
- The `LISTSL` command lists the contents of a segmented library and the `LISTXL` command lists the contents of an executable library.
- The `PURGESL` command deletes code units from a segmented library and the `PURGEXL` command deletes code units from an executable library.

Although executable and segmented libraries are similar, executable libraries provide more power and flexibility in managing executable code. The following list summarizes the *differences* between these libraries:

MPE V Segmenter	HP Link Editor/XL
■ Segmented libraries must have the name `SL`.	■ Executable libraries can have any valid MPE XL file name.
■ Using the `LIB=` parameter, you can search up to three segmented libraries.	■ Using the `LIB=` and `XL=` parameters of the `LINK` command, you can search any number of executable libraries at run time.
■ You can add only one segment to a segmented library using the `ADDSL` command.	■ Using `ADDXL`, you can add one or more modules (from an executable program file or from one or more relocatable libraries) to an executable library.
■ You cannot use relocatable libraries to resolve external references when adding modules to a segmented library.	■ When adding modules to an executable library using the `ADDXL` command, you can use relocatable libraries to resolve external references.
■ You cannot merge modules when adding them to a segmented library.	■ When adding modules to an executable library using the `ADDXL` command, you can merge one or more modules into one.

G

HP Link Editor/XL Command Summary

This appendix serves as a quick reference to the syntax of the HP Link Editor/XL commands. The commands are listed alphabetically.

```
ADDRL FROM= source_file [, source_file]...
      [;TO= dest_file]
      [;MERGE [;RL= rl_file [, rl_file]...]]
      [;SHOW]
      [;REPLACE]

ADDXL FROM= source_file [ ,source_file]...
      [;TO= dest_file]
      [;MERGE [;RL= rl_file [, rl_file]...]]
      [;SHOW]
      [;PARMCHECK= check_level]
      [;PRIVLEV= priv_level]
      [;XLEAST= xleast_level]
      [;MAP]
      [;REPLACE]
      [;ENTRY= entry_name [ ,entry_name]...]
      [;MODULE= module_name [ ,module_name]...]
      [;BLOCKDATA= blockdata_name [ ,blockdata_name]...]
      [;LSET= lset_name [ ,lset_name]...]
      [;NODEBUG]

ALTPROG [PROG= file] [, file]...
        [;XL= xl_file [, xl_file]...]
        [;CAP= cap_list]
        [;NMSTACK= max_stack_size]
        [;NMHEAP= max_heap_size]
        [;UNSAT= unsat_name]
        [;ENTRY= entry_name]
        [;PRI= priority_level]
        [;MAXPRI= max_priority_level]

BUILDRL RL= rl_file
        [;LIMIT= max_modules]

BUILDXL XL= xl_file
        [;LIMIT= max_modules]
```

```
CLEANRL [RL= rl_file]
        [;COMPACT]
        [;LIMIT= max_modules]

CLEANXL [XL= xl_file]
        [;COMPACT]
        [;LIMIT= max_modules]

COPYRL [ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;FROM= source_file]
       [;TO= dest_file]
       [;REPLACE]

COPYXL [ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [ ,blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;FROM= source_file]
       [;TO= dest_file]
       [;REPLACE]

DO [ command_id]

EXIT

EXTRACTRL [ENTRY= entry_name [ ,entry_name]...]
          [;MODULE= module_name [ ,module_name]...]
          [;BLOCKDATA= blockdata_name [ ,blockdata_name]...]
          [;LSET= lset_name [ ,lset_name]...]
          [;FROM= source_file]
          [;TO= object_file]

  HELP [keyword] [, ALL      ]
                 [, PARMS    ]
                 [, EXAMPLES ]

  HIDERL { ENTRY= entry_name }
         { ;ALL              }

         [;RL= rl_file]
```

```
LINK [FROM= source_file [, source_file]...]
     [;TO= dest_file]
     [;RL= rl_file [, rl_file]...]
     [;XL= xl_file [, xl_file]...]
     [;CAP= cap_list]
     [;NMSTACK= max_stack_size]
     [;NMHEAP= max_heap_size]
     [;UNSAT= unsat_name]
     [;PARMCHECK= check_level]
     [;PRIVLEV= priv_level]
     [;PRI= priority_level]
     [;MAXPRI= max_priority_level]
     [;ENTRY= entry_name]
     [;NODEBUG]
     [;MAP]
     [;SHOW]

LISTOBJ OBJFILE= relocatable_object_file
        [;ALL]
        [;CODE]
        [;DATA]
        [;ENTRYSYM]
        [;MILLICODE]

LISTPROG PROG= executable_prog_file
         [;ALL]
         [;CODE]
         [;DATA]
         [;ENTRYSYM]
         [;MILLICODE]
         [;STUB]
         [;VALUE]

LISTREDO
```

```
LISTRL [RL= rl_file]
       [;ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [ ,blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;ALL]
       [;CODE]
       [;DATA]
       [;ENTRYSYM]
       [;MILLICODE]

LISTXL [XL= xl_file]
       [;ENTRY= entry_name [ ,entry_name]...]
       [;MODULE= module_name [ ,module_name]...]
       [;BLOCKDATA= blockdata_name [ ,blockdata_name]...]
       [;LSET= lset_name [ ,lset_name]...]
       [;ALL]
       [;CODE]
       [;DATA]
       [;ENTRYSYM]
       [;MILLICODE]
       [;STUB]
       [;VALUE]

PURGERL
{ ENTRY= entry_name [,entry_name] ...                          }
{ ;MODULE= module_name [,module_name] ...                      }
{ ;BLOCKDATA= blockdata_name [, blockdata_name] ...            }
{ ;LSET= lset_name [,lset_name] ...                            }
       [;RL= rl_file]

PURGEXL
{ ENTRY= entry_name [,entry_name] ...                          }
{ ;MODULE= module_name [,module_name] ...                      }
{ ;BLOCKDATA= blockdata_name [, blockdata_name] ...            }
{ ;LSET= lset_name [,lset_name] ...                            }
       [;XL= xl_file]

REDO [ command_id]
```

REVEALRL { ENTRY= *entry_name* | ;ALL }

[;RL= *rl_file*]

RL RL= *rl_file*

SHOWRL

SHOWXL

XL XL= *xl_file*