eld Manual

Abstract

This manual describes how programmers can use `eld`, the object file linker for TNS/E, to create loadfiles for execution on HP Integrity NonStop™ NS-series servers.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs and H06.01 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
527255-009	February 2012

Document History

Part Number	Product Version	Published
527255-004	N.A.	July 2005
527255-005	N.A.	May 2010
527255-007	N.A.	August 2010
527255-008	N.A.	May 2011
527255-009	N.A.	February 2012

Legal Notices

Printed in the US

eld Manual

Glossary | Index | Tables

What's New in This Manual

Manual Information

Abstract

This manual describes how programmers can use `eld`, the object file linker for TNS/E, to create loadfiles for execution on HP Integrity NonStop™ NS-series servers.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs and H06.01 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
527255-009	February 2012

Document History

Part Number	Product Version	Published
527255-004	N.A.	July 2005
527255-005	N.A.	May 2010
527255-007	N.A.	August 2010
527255-008	N.A.	May 2011
527255-009	N.A.	February 2012

New and Changed Information

Changes to the H06.24/J06.13 manual:

- In the section, The Steps in Looking for Archives and DLLs on page 2-17:
  - Updated the fourth and fifth step and also added a new content on page 2-17.
  - Updated information on 2-18.
- In the section, Additional rules about -alf on page 4-3:
  - Updated the list on page 4-4.
- In the table, The -set and -change Options on page 4-8:
  - Added a new entry on page 4-8.

- Updated the information on the attributes for -set and -change option on pages 4-9 and 4-10.

- Added eld Functionality for 64-Bit on page 4-12.
- In the table, Set Attributes on page 5-8:
  - Added a new entry on page 5-8.
- In the title, Output Listings and Error Handling on page 6-1:
  - Updated the message 1557 on page 6-95.
  - Added a new message 1510 on page 6-77.
  - Added a new message 1665 on page 6-124.
  - Added a new message 1666 on page 6-124.
  - Added a new message 1667 on page 6-125.
  - Added a new message 1668 on page 6-125.
  - Added a new message 1669 on page 6-125.
  - Added a new message 1670 on page 6-125.
  - Added a new message 1672 on page 6-125.

Changes to the 527255-008 manual:

- Added a new message 1132 on page 6-20.

Changes to the 527255-007 manual:

- Added the following linker options:
  - -NS_extent_size extent-size on page 5-6
  - -NS_max_extents max_extents on page 5-7
  - -warn_common on page 5-12
- Updated `systype` attribute information under Target Platforms on 2-2.
- Added Using the -cross_dll_cleanup option on page 3-19.
- Added information about -cross_dll_cleanup on page 5-2.
- Added the following new messages:
  - 1233 on page 6-39 to 1236 on page 6-39
  - 1389 on page 6-69
  - 1391 on page 6-69
- Updated error message 1519 on page 6-79.

- Added applicability note for `data2protected` parameter support on page 2-7 and 3-16.
- Updated list of sections under Binding of References on page 3-1.

Changes to the 527255-005 Manual:

- Updated information on how the linker searches for indirect DLLs on page 2-12 and 2-19.
- Added consideration for `-set libname` on page 3-10.
- Added consideration for `-change libname` on page 4-10.
- Added the following error messages:
  - 1229 to 1232 on page 6-38
  - 1657, 1659, and 1660 on page 6-124

About This Manual

This publication describes how programmers can use `eld`, the object file linker for TNS/E, to create loadfiles for execution on H-series software NonStop servers.

Section 1, Introduction to eld consists of the following topics:

- eld Overview - explains the general functionality of the product.
- Native Object Files - introduces the different types of object files.
- The Linker Command Stream - shows the conventions for entering tokens (options, parameters and filenames) on the command line.
- Example of Use - presents an example of using `eld` to link a main program and a DLL .

Section 2, eld Input and Output consists of the following topics:

- Host Platforms - where the linker may be used.
- Target Platforms - where the output from the linker may be used.
- Output Object Files - what forms (libraries, loadfiles and DLLs) that output may take.
- The Creation of Output Object Files - how you control that process.
- Creating Segments of the Output Loadfile - how parts of a loadfile are created.
- Using a DLL Registry - how you can manage DLL addressing.
- Input Object Files - which files you can use as input.
- Using Archives - how you can group multiple input or output files together.

Section 3, Binding of References consists of the following topics:

- Overview - an overview of symbol resolution and code relocation.
- Presetting Loadfiles - the process of resolving references to DLLs at linktime.
- To Preset or Not to Preset, and Creation of the LIC - `eld` rules for presetting.
- Handling Unresolved References - what happens if a symbol is not found in any loadfile in the linker's search list?
- Using User Libraries - introduces the libname options.
- Creating Import Libraries - three types are available.
- Ignoring Optional Libraries - a command stream toggle is available.

- Finding and Reading The Public DLL Registry (ZREG) File - three ways to find it.

Section 4, Other eld Processing consists of the following topics:

- Adjusting Loadfiles: The -alf Option - how to repeat the presetting of a loadfile when DLLs change.
- The -set and -change Options - how to set various options within the loadfile.
- eld Functionality for 64-Bit - how the linker performs consistency checks.
- Renaming Symbols - how the linker treats each input file.
- Updating Or Stripping DWARF Symbol Table Information - from the input and output object files.
- Modifying the Data Sections that Contain Stack Unwinding Information - when concatenating sections to create a new loadfile.
- Creating the MCB - the Master Control Block contains key settings such as product version numbers, valid file types, language dialects, and so on.
- Processing of Floating Point Versions and Data Models - more consistency checks.
- Specification of the Main Entry Point - there are two ways to specify the main entry point.
- Specifying Runtime Search Path Information for DLLs - `eld` tells *rld* where to find the DLLs.
- Merging Source RTDUs - used with SQL/MP.

Section 5, Summary of Linker Options consists of a list and description of every linker option.

Section 6, Output Listings and Error Handling consists of the following topics:

- General Information - when and how messages are created.
- Error Messages - individual cause, effect and recovery information.
- Glossary of Errors - a glossary of terms used in the error messages.

Appendix A, TNS/E Native Object Files consists of the following topics:

- The Object File Format - the types of object files and their content.
- Code and Data Sections - the "ordinary" code and data sections that come from application source code, possibly with additions by the compiler or linker.

- Relocation Tables - when code is relocated, who resolves the address and prepares relocation tables?
- Finding Information About Procedures and Subprocedures in Linkfiles - an introduction to the .procinfo and .procnames sections of linkfiles.
- The DWARF Symbol Table - this table contains information used by debuggers and the Cobol compiler.
- Archives - contains an extension of material covered in a previous section of this manual.
- Tools That Work With Object Files - a quick look at which HP NonStop operating system tools use object files.

Notation Conventions

The specific conventions for `eld` may be found in The Linker Command Stream on page 1-5.

The conventions shown below are generic, applying to most manuals in the NonStop Technical Library.

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

> This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in most NonStop manuals. Some of these conventions may not apply to this particular manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *Italic computer type* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

pathname

<filename>. TThis represents the name of a file. Filenames follow whatever rules apply to the corresponding host platform when the linker needs to do something with the file. For example, filenames are not case sensitive on Guardian or NT, but are case sensitive on OSS. The linker keeps all filenames in the same form that they were specified, unless otherwise stated in this document.

<symbol name> . This represents the name of a symbol as it appears within an object file. It is case sensitive. There are rules that depend upon the source language for mapping between the way the symbol appears in the source code and the way the symbol appears within the object file. Each source language has its own rules for this.

<path> . This is the kind of string used in the *-rld_L* and *-rld_first_L* options.

<dllname>. This is the kind of string used in the -dllname option.

<location> . This is the kind of string used in the -L and -first_L options.

<number> . This indicates a 64-bit numerical value. It is interpreted as a hexadecimal number if it begins with “0x”, “0X”, “%h”, or “%H”, in which case the rules given below for a <hexadecimal number> apply. If it has none of these prefixes then it is interpreted as a decimal number and the rest of the token must be a sequence of decimal digits. The TNS/E linker does not accept octal numbers.

<hexadecimal number> . This indicates a 64-bit hexadecimal number, and may optionally begin with “0x”, “0X”, “%h”, or “%H”. The letters “a” through “f” (representing the values 10 through 15) are not case sensitive. Periods are allowed in hexadecimal numbers to subdivide the number for readability, using one of the following two methods. There may be one period in the number, and then the period is assumed to divide the number into two 32-bit portions. Or, there may be three periods, in which case the periods are assumed to divide the number into four 16-bit portions.

<attribute> and <value>. These notations are explained under the description of the *-set* option.

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on

each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Introduction to eld

This section contains the following information topics:

eld Overview introduces the general functionality of the product.

Native Object Files introduces the different types of object files.

The Linker Command Stream shows the conventions for entering tokens (options, parameters and filenames) on the command line.

Example of Use presents an example of using `eld` to link a main program and one DLL.

eld Overview

The primary use of the linker `eld`, in the TNS/E development environments (Guardian, OSS or PC), is to combine one or more TNS/E position-independent code (PIC) object files into a new single loadfile.

`eld` manipulates both code and data, and then places all of the loadfile's adjustable references in tables outside the code to make them available to `rld`, the run-time loader. This process, called linking, must be applied to linkfiles after they have been compiled and before they can be loaded for execution.

Linkfiles are produced by a TNS/E compiler or assembler. The new loadfile created by `eld` is either a program or a DLL. A loadfile contains certain information used to bind references among loadfiles at load time. The linker may also look at other DLLs in order to resolve references to them at link time (this is called "presetting"). A main program, together with the DLLs that it needs directly or indirectly, is executed on the HP NonStop operating system. A file type of 800 indicates that the code can only execute on the TNS/E HP NonStop operating system platform.

The equivalent product for TNS/R PIC object files is known as `ld` and is documented in the *ld Manual*.

`rld` is the run-time loader that may be used in either TNS/R or TNS/E environments. `rld` is documented in the *rld Manual*.

As Guardian commands are case-insensitive you may use either upper (for example, `ELD`) or lower-case (for example, `eld`) commands to invoke the linker. To comply with UNIX conventions of lower-case usage, you may only use lowercase in the OSS environment. This manual mostly uses lower-case versions of the linker names.

PIC native object files are generated by the native C, native C++, native COBOL, and pTAL compilers. See the following manuals for information about these compilers:

- *C/C++ Programmer's Guide*
- *HP COBOL For NonStop Systems*
- *pTAL Reference Manual.*

Example Command Line

The following is an example of a linker command line:

```
eld myobj1 myobj2 -o myprog -lib mydll
```

This command specifies linkfiles *myobj1* and *myobj2* as inputs to the linker, which is named `eld`. The linker will combine them into the program named *myprog*, and will bind references against a DLL found from *mydll*.

For a longer example of using `eld`, see the Example of Use on page 1-9.

eld Functionality

`eld` can also :

- obtain its input linkfiles from archives.
- combine existing linkfiles into a new linkfile with the `-r` option.
- create an import library, including one that represents all the “implicit” libraries.
- bind references against user libraries and import libraries.
- update an existing loadfile with the `-alf`, `-change`, and `-strip` options.

The linker does not have a programmatic interface other than starting it as a new process, providing it the appropriate inputs, and looking at the outputs that it produces.

There are various ways of invoking the linker, such as directly from the command line, indirectly from other command line tools such as the C compiler, or through graphical user interfaces. You can run `eld` in these ways:

Manually, at a command prompt.

Automatically, when using these compilers:

Environment	Compiler
Guardian	CCOMP
	CPPCOMP
	ECOBOL
OSS	`c89`
	`c99`
	`ecobol`
Windows NT* Windows 2000* Windows XP*	All of the above and eptal

* By means of the Cross-Compiler CDs, ETK

This publication explains only how to run `eld` manually. For information on running `eld` automatically, see:

- *C/C++ Programmer's Guide*
- *HP COBOL For NonStop Systems*

`eld` runs in the following environments:

- Guardian
- OSS
- Windows NT, Windows 2000, and Windows XP, using either the Cross-Compiler CDs or the HP Enterprise Toolkit—NonStop Edition (ETK), which is based on Microsoft Visual Studio .NET

Note. `eld` has the same capabilities and syntax in each environment, but each environment has its own rules, for example, filename syntax.

Linker Version Information

`eld` writes information about its version number into the *.tandem_info* section of its output object file.

`eld` also contains a VPROC procedure that is a standard feature of Tandem tools.

The PC version of `eld` contains version information that you can see by looking at the "properties" of the `eld.exe` file.

Native Object Files

`eld` operates on TNS/E native object files. Native object files are in Executable and Linking Format (ELF), a standard format used for object files, with some HP extensions.

This manual presents only basic information about these files. For details on the structure of native object files for TNS/E, see Appendix A, TNS/E Native Object Files and the *enoft Manual*.

Native object files are either Linkfiles or Loadfiles, but not both.

	Can Be Linked to Produce a Loadfile	Can Be Executed
Linkfiles	Yes	No
Loadfiles	No	Yes

The native compilers create native object files called linkfiles from source code. `eld` produces native object files called loadfiles from such linkfiles.

Native object files of TNS/E in the Guardian environment have a file code of 800. Native object files of TNS/R in the Guardian environment have a file code of 700.

There are four types of TNS/E object file:

Type of Object File	Description
Linkfile	Object files that are produced by a compiler or by the assembler that can be given as input to the linker. It is also possible for the linker to produce a linkfile as output when run with the `-r` option.
Program	This is the main program. There is one program in every process.
DLL	This is a dynamic-link library. It is an object file that is not a program but can also be part of a process. A process can contain any number of DLLs. DLLs are also used by the linker when building other programs or DLLs.
Import Library	This is a file that contains just part of a DLL that is needed at link time to build other DLLs or programs.

Collectively, programs and DLLs are called loadfiles. Loadfiles and import libraries are built by the linker.

The main distinctions occur between linkfiles and loadfiles. There is little difference between a program and a DLL as far as the file format is concerned, and an import library is a subset of what is in a DLL.

A loadfile may refer by name to symbols that exist in other loadfiles in the same process. Such references are resolved when the loadfiles are brought into memory by the runtime loader (`rld`) or by the C/C++ runtime procedure named `dlopen()`. When the loadfile was originally built by the linker it is also possible that the linker tried to resolve such references. A loadfile whose references have been resolved by the linker is said to be preset.

A process can also use one user library. A user library is a DLL. Nothing within a user library distinguishes it from other DLLs, and a DLL that serves as the user library for one program can also be used like any other DLL by other programs. The only difference between the user library and other DLLs is in the way the program identifies the user library that it uses. For a DLL to be used as a user library at runtime its filename must be in the Guardian name space.

An import library can take the place of a DLL at link time. One use of import libraries is to save space. Another use is for security, when it is necessary for the linker to read the header information but it is not desirable for others to be able to see the code. Import libraries are further categorized as complete or incomplete. The difference is that an incomplete import library need not contain the correct addresses for symbols. A complete import library can be used by the linker when presetting a loadfile. The linker can use an incomplete import library to check for unresolved references, but not to preset.

DLLs and import libraries can also be used at compile time by the COBOL compiler to find out information about procedure call interfaces.

Some DLLs are called public libraries because they are provided as part of the TNS/E implementation and are found in a special way by the linker and runtime loader. A public library has the same format as any other DLL, and can have an import library to represent it.

Some of the public libraries are called implicit libraries because they are used at link time and run time without explicit mention on the part of the user. There are several implicit libraries, and there is a bit in a DLL that tells if it is an implicit library. A single implicit library never has an import library to represent it to the linker. Rather, at link time, when building a loadfile that is not an implicit library, a single import library represents the entire set of implicit libraries. That is called the import library that represents the implicit libraries, and it is always a complete import library.

For detailed information about PIC programs and developing DLLs, see the *DLL Programmer's Guide for TNS/E Systems*.

There is also an equivalent manual for TNS/R systems, the *DLL Programmer's Guide for TNS/R Systems*.

The Linker Command Stream

The following definitions of terms such as option, parameters and filenames is specific only to `eld`. In other environments and products, similar concepts might go under different names, for example OSS calls options "flags".

The linker obtains tokens from the command line by using the `argc` and `argv` functions of C. The way this works may be dependent on the C runtime implementation, but the general idea is that tokens are separated by spaces. On the PC it may be useful to include a space within a filename, and that can be done by placing quotation marks around the name. The C runtime gives the name to the linker without the surrounding quotation marks.

Certain options, such as the `-obey` option, cause the linker to obtain tokens from files, and the linker mostly treats them the same as if they were on the command line. The linker's command line, plus the other tokens it obtains in such ways from files, are collectively called the linker's command stream.

Tokens are categorized as options, parameters, or filenames. An option always starts with a hyphen. A filename or parameter never starts with a hyphen. Each option requires a certain number of parameters to immediately follow it. `eld` reports an error if the linker cannot parse the command stream into a sequence of options and/or filenames, where each option is followed by the required number of parameters.

Names of options and parameters must be spelled exactly as given in this manual. Except for the options named `-l` and `-L`, options and keyword parameters are not case sensitive.

Options may be placed into one of three categories:

- Repeatable options are options such that each occurrence of the option is independent, such as providing another element of a list of information or making the linker do a certain activity again.
- Toggle options are options that modify the linker's behavior for the remainder of the command stream, or until toggled again.
- One-time options are everything else.

Descriptions of repeatable options and toggle options within this manual explain the significance of each occurrence of the option. If this manual does not explain why an option might be given more than once in the command stream, it is a one-time option. It is okay to specify a one-time option more than once if it doesn't take any parameters, and the repeated occurrences are ignored by the linker. It is similarly okay if the option has a parameter and you specify the option more than once with the same parameter, or with another parameter that is a "synonym" of it. `eld` reports an error if you specify a one-time option more than once with non-synonymous parameters. With regard to one-time options that take string parameters, "synonymous" means exactly the same, including the same case of all the characters, even in situations (e.g., filenames on Guardian) where the case wouldn't be significant. Except, because the parameter to `-set libname` is converted to upper case by the linker anyway, this particular check is not case sensitive. On the other hand, when there is a numerical parameter, "synonymous" means that the value comes out the same, regardless of how it is written.

With regard to the `-set` option, each combination of the `-set` option with one of its attributes is treated as a single one-time option as mentioned in the previous paragraph. In other words, there are no restrictions on how many times the `-set` option can be given with different attributes. If `-set` is specified more than once with the same attribute, and a value is required, that is okay if a synonymous value is given each time, otherwise `eld` reports an error.

With regard to the `-b` option, this is really treated as two different options. There is a one-time `-b` option, whose possible parameter values are `globalized`, `localized`, `semi_globalized`, and `symbolic`, where `symbolic` is a synonym for `semi_globalized`. And there is also a toggle `-b` option, where the possible parameter values are `dllsonly`, `dynamic`, and `static`.

Throughout this manual, rules are given for which combinations of options are legal. Even if not stated explicitly, `eld` reports an error if you specify multiple options that are mutually exclusive from their descriptions.

Whenever the name of an option is a single letter and the option has a single parameter it is permissible to omit the space between the option name and the parameter, combining them into a single token. This would be ambiguous for an option that takes a filename or symbol name as a parameter when the result is the name of some other option, so the rule is that it is only permitted to leave out the space if this does not cause such an ambiguity. For instance, if the output file is to be named b, it can be specified as either -o b or -ob. However, if the output file is to be named bey then it must be written -o bey because there is another option named `-obey`.

If no tokens are given to the linker in its command stream then the linker writes out messages to the output listing to give a one-line summary of each of the available options and does nothing else.

If some tokens are given, and based on these tokens the linker should create a new object file from one or more linkfiles, but no linkfiles are brought into the link, `eld` reports an error.

Obey Files and the Use of Standard Input

The `-obey` option lets the user put tokens into a file that is read by the linker. This file is a text file, and must be either a file code 101, or a code 180 file. The parameter to the `-obey` option is the name of the file, and anything that could be given on the linker command line is permissible in that file. In simple cases the contents of the file are treated the same as if they were on the command line, in the place of the `-obey` option. White space in the obey file, including ends of lines, serves to separate tokens.

Within obey files, if a token begins with two consecutive hyphens, those hyphens and the rest of the line are treated as a comment (ignored).

There is also a special rule with regard to (double) quotation marks within obey files. The special rule only applies to a quotation mark that comes at the beginning of a token, i.e., it is either the first character in the obey file or it follows white space. In this case, if it is the last quotation mark on that line, `eld` reports an error. Otherwise, the characters between it and the next quotation mark are considered to be a single token, even if they include white space. The linker will then start looking for the next token immediately after the second quotation mark. For example, if a line in an obey file contains the following:

```
-o “abc -def”ghi
```

then the linker will consider this to contain three tokens, namely, “-o”, “abc -def”, and “ghi”. Note that there is a space between the ‘c’ and the ‘-’, and that the name of the output file created by the linker is “abc -def”. On some platforms this will work, and on other platforms it won’t. Also note that no space is required between the second quotation mark and the ‘g’. As another example, if a line in an obey file contains the following:

```
-o abc“d e”f
```

then this contains two tokens after the `-o`, where the first token is abc”d and the second one is e”f. The quotation marks do not fall under the special rule here, because they are not at the beginnings of tokens. Thus, the space between the d and e serves to separate tokens.

Note that the above examples show that it is possible to put a space in the middle of a token, and that it is possible to put a quotation mark in the middle of a token, but it is not possible to have both of these things in the same token.

There can be multiple `-obey` options on the command line, each being processed as explained above. There can be `-obey` options within obey files, with no limit on the

nesting. Recursive nesting is handled by the rule that an `-obey` option is ignored if its parameter is identical to the parameter of an `-obey` option that is currently being processed.

The `-obey` option has a synonym, `-FL`.

There is also an option named `-stdin`. This is the same as an `-obey` option except that it doesn't take a parameter and it signifies instead that the contents of the standard input file are read at this point in the command stream. The linker reads from standard input until it encounters end of file. If the runtime environment allows a process to read standard input up to an end of file, and then read it some more until another end of file, etc., then there can be more than one `-stdin` option in the linker command stream, each one being processed the same way.

It is not an error if a `-stdin` option is found within the standard input file. It is simply ignored, as a special case of the rule that recursive `-obey` options are ignored. The linker doesn't require input from the standard input file and the linker is often used where standard input is interactive, with nothing being entered. Accordingly, it would be a mistake for the linker to unconditionally read from standard input because then the linker might wait forever, with the user not realizing that the linker was waiting for input. That is why a special option such as `-stdin` is necessary, to say when input is coming from standard input.

A special case is made, however, for the Guardian platform, where the linker distinguishes an EDIT file from other types of input files. On the Guardian platform, if the standard input file is an EDIT type file and nothing is specified on the command line, the linker reads that EDIT type file to obtain its command stream.

Note that the `-obey` option is expanded like a macro, and can even be used to provide the parameter to an option. For example, it is possible to say `-rpath -obey x`, and then the initial token within the file named `x` would be used as the parameter for the `-rpath` option. If the file named `x` contained additional contents after the initial token, those additional tokens would be interpreted as additional filenames, options, etc., following the `-rpath` option.

Similar considerations apply when an `-obey` option is a parameter to another `-obey` option, for example, if you write `-obey -obey x` as part of the command stream. In this case, the linker will first assume that `x` is the name of an obey file and expand the contents of the file `x` in place of the `-obey x`. So, if the first token found within the file `x` is, say, `y` (not starting with a hyphen), so that the expanded command stream now looks like `-obey y ...`, the linker will assume that `y` is the name of an obey file and expand the contents of the file `y` in place of the `-obey y`. Thus the command stream will contain the contents of the file `y` , followed by the remaining contents of the file named `x` after its initial token "y".

Example of Use

This section shows an example of using the `eld` linker. This example shows the use of a main program, *mainstrc,* and a library called *mystrngc*. Both will be compiled using `ccomp`, then linked using `eld`. *mystrngc* will be loaded as a DLL.

Display the Source Code

Here is the code for the main program, *mainstrc*

```
#include <stdio.h>  nolist
#include <stdlib.h> nolist
#include <string.h> nolist
int StrRev (char *s, char *r); /* declaration of external procedure */

char s[100];

/*********************************************************
|    main: given a list of strings, print out them reversed
|       argv[1]...argv[argc-1] point to strings
|
|    if no string passed, put out usage message and quit.
|    for each string
|     reverse it
|      display it
|
\*********************************************************/
int main(int argc, char *argv[]) {
   char **ppStr;
   int strLeft;
   int outcome;

   if (argc < 2)  /* no args passed */
     {
     printf("Usage: run rev <str1> [<str2>] ....\n \
             \twhere <str> is a string to reverse\n \
             \texample: run rev abc zyxw\n");
       exit(1);
     }

   for (strLeft= argc-1, ppStr=argv+1;
        strLeft;
        ppStr++, strLeft-- ) {
        strcpy(s, *ppStr);
        outcome = StrRev( s, s );
        (outcome == 0) ? printf( "Reverse(%s) = (%s)\n", *ppStr, s ) :
                         printf( "error in reversing the string\n");
        } /* for */


   printf("Hit enter to finish\n");

   getchar( );

} /* of proc main */
```

Here is the source code for the library, *mystrngc*

```
#include <string.h> nolist
#include <stdlib.h> nolist

int StrRev (char *s, char *r ) {
```

```
  char *pBegin;
  char *pEnd;
  char c;
  strcpy(r, s);
  pBegin = r;
  pEnd = r + strlen(r);
  while (--pEnd > pBegin )
        {
        c = *pBegin;
        *pBegin++ = *pEnd;
        *pEnd = c;
        }
  return (0);

} /* StrRev */
```

Compile the Program and Library

The first step is to compile the programs using `ccomp`, the native mode TNS/E compiler, on the HP NonStop operating system to create the two object files, *mainstro* and *mystro*.

We are using a fully-qualified filename to get to the TNS/E compiler, `ccomp`. On your system, the pathname showing the location of your development tools will be quite different.

```
run $data01.toolsy02.ccomp /in mainstrc /mainstro; suppress
TNS/E C - T0549H01 - 30AUG2004 (Oct 25 2004 14:47:23)
(C)2004 Hewlett Packard Development Company, L.P.


0 remarks were issued during compilation.
0 warnings were issued during compilation.
0 errors were detected during compilation.
Object file: mainstro
Compiler statistics
  phase     CPU seconds   elapsed time   file name
  CCOMP                                  \SPEEDY.$DATA01.TOOLSY02.CCOMP
  CCOMBE            0.2       00:00:07   \SPEEDY.$DATA01.TOOLSY02.CCOMBE
  total             0.2       00:00:09
All processes executed in CPU 05 (NSR-Y)
Swap volume: \SPEEDY.$DATA01
```

Here's the creation of the object file called *mystro*.

```
run $data01.toolsy02.ccomp /in mystrngc /mystro;suppress
TNS/E C - T0549H01 - 30AUG2004 (Oct 25 2004 14:47:23)
(C)2004 Hewlett Packard Development Company, L.P.

0 remarks were issued during compilation.
0 warnings were issued during compilation.
0 errors were detected during compilation.
Object file: mystro
Compiler statistics
  phase     CPU seconds   elapsed time   file name
  CCOMP                                  \SPEEDY.$DATA01.TOOLSY02.CCOMP
  CCOMBE            0.2       00:00:06   \SPEEDY.$DATA01.TOOLSY02.CCOMBE
  total             0.2       00:00:06
All processes executed in CPU 04 (NSR-Y)
Swap volume: \SPEEDY.$DATA01
```

Build the DLL and the Program

First we build the DLL, then the main executable file called *revstr.* It has to be in that order because the main executable could not refer to a DLL that did not yet exist. (If the linker's `-allow_missing_libs` option is specified, the main executable could be linked befoe the DLL is linked.)

Note that the `-lib` option references the DLL called *mystrdll.* Note the `-export_all` option. We could also individually reference the items to be exported, as follows:

```
            -export StrRev
```

Note the `-shared` option sent to `eld`. This creates the DLL. The `-dll` option can be used to do the same task, and is probably more descriptive of what we want to achieve. Either option can be used.

The following command input creates the DLL:

```
run $data01.toolsy02.eld mystro -o mystrdll -shared -export_all
eld - TNS/E Native Mode Linker - T0608H01 - 26OCT04
Copyright 2004 Hewlett-Packard Company

eld command line:
   \speedy.$data01.toolsy02.eld mystro -o mystrdll -shared -export_all

**** INFORMATIONAL MESSAGE **** [1530]:
   Using 'ImpImp' file: \speedy.$data01.toolsy02.zimpimp.

Output file: mystrdll (dll)
Output file timestamp: Nov  8 13:59:43 2004

No errors reported.
No warnings reported.
1 informational message reported.
Elapsed Time:  00:00:01
```

Now Build the Program

The next step is to create the loadfile (the whole program) by use of the linker.

`ccplmain` contains initialization code for the C and C++ run-time libraries. Your version of that file will probably be located in `$system.system`.

`ccplmain` contains external references to errno and environ (which are defined in ZCREDLL) and C_INT_INIT_COMPLETE_ , C_INT_INIT_START_ , and exit (which are defined in ZCRTLDLL).

Note that each DLL must use an individual `-lib` option to be linked with `eld`. The command syntax does not allow for a single `-lib` option followed by a list of DLLs, for example: `-lib zcredll, zcrtdll, mystrdll` is not valid syntax.

```
60> run $data01.toolsy02.eld $data01.toolsy02.ccplmain mainstro -lib
mystrdll&

60> & -lib zcredll -lib zcrtldll -o revstr -L $users.patrick -L
$data01.toolsy02 -verbose

eld - TNS/E Native Mode Linker - T0608H01 - 26OCT04

Copyright 2004 Hewlett-Packard Company


eld command line:
   \speedy.$data01.toolsy02.eld $data01.toolsy02.ccplmain mainstro -lib
   mystrdll -lib zcredll -lib zcrtldll -o revstr -L $users.patrick -L
   $data01.toolsy02 -verbose


**** INFORMATIONAL MESSAGE **** [1019]:
   Using DLL: $users.patrick.mystrdll.
**** INFORMATIONAL MESSAGE **** [1019]:
   Using DLL: $data01.toolsy02.zcredll.
**** INFORMATIONAL MESSAGE **** [1019]:
   Using DLL: $data01.toolsy02.zcrtldll.
**** INFORMATIONAL MESSAGE **** [1530]:
   Using 'ImpImp' file: \speedy.$data01.toolsy02.zimpimp.

Output file: revstr (program file)
Output file timestamp: Nov  9 14:59:42 2004

No errors reported.
No warnings reported.
4 informational messages reported.
Elapsed Time:  00:00:02
```

Run The Program

```
RUN REVSTR XYZ
Reverse(XYZ) = (ZYX)
Hit enter to finish
```

2 eld Input and Output

This section contains the following information:

Host Platforms - where the linker may be used.

Target Platforms - where the output from the linker may be used.

Output Object Files - what forms (libraries, loadfiles and DLLs) that output may take.

The Creation of Output Object Files - how you control the process.

Creating Segments of the Output Loadfile - how parts of a loadfile are created.

Using a DLL Registry - how you can manage DLL addressing.

Input Object Files - which files you can use as input.

Using Archives - how you can group multiple linkfiles together for `eld` access.

Host Platforms

The TNS/E linker (`eld`) runs on several platforms, as follows.

- The TNS/E linker runs on the TNS/E version of the HP NonStop operating system, with both the Guardian and OSS personalities.
- The TNS/E linker runs on TNS/R versions of the HP NonStop operating system. This means you can link PIC object files into TNS/E loadfiles as part of your development cycle on that platform - but you will not be able to execute those loadfiles on TNS/R.
- The TNS/E linker runs on appropriate versions of the Windows operating system on PC's.

The TNS/E linker's features are the same for all host platforms unless otherwise specified in this manual. Differences often relate to the different types of filenames and file characteristics on different platforms.

The Guardian namespace also exists as a subset of the OSS file system, where a Guardian file named $a.b.c corresponds to an OSS file named /G/a/b/c.

The Guardian namespace is a set of rules use for naming files in the Guardian filesystem. A complete description of those rules may be found in the OSS `filename (5)` reference page in the *OSS System Calls Reference Manual*.

Those rules include:

- The Guardian namespace ignores lowercase, the OSS namespace uses lowercase.
- $ and \ are not recognized in the OSS namespace
- If a filename contains a period in it, and a file of that name is to be created in a Guardian subvolume, the period is deleted from the name.

- Text files in Guardian subdirectories of OSS are code 180 files, not edit files. When the linker creates a text file in a Guardian subvolume of OSS it is a code 180 file. When the linker reads a text file in a Guardian subvolume of OSS, code 180 files always work, and edit files do not necessarily work.
- OSS has the concept of a “file mode” used to control UNIX file security and access rules. This file mode does not apply to files in the Guardian namespace.

Target Platforms

The loadfiles created by the TNS/E linker can only be loaded and run on the TNS/E versions of the HP NonStop operating system. A process can be run in either one of two environments “Guardian” and “OSS”, dependent on it’s process ID.

The `-set systype` option specifies the target environment for the loadfile being built by the TNS/E linker. The systype attribute has no meaning for a DLL. A DLL may be usable by a Guardian process, or an OSS process, or both, depending on how the various parts of it were written and compiled. However, you must be aware of how the different parts were written and compiled to use it appropriately. The target environment does not affect anything else the linker does unless otherwise specified in this document. The target environment is indicated within the loadfile by the *EF_TANDEM_SYSTYPE* bit in the *e_flags* field of the ELF header.

The default target environment is `guardian` for the linker hosted on Guardian. The default is `oss` on the PC. The default on OSS is either `guardian` or `oss`, depending on whether the object file is being created in a Guardian subvolume.

Filenames and The File Identifier

In the Guardian environment, “filename” is defined as the complete descriptor (pathname) of \NODE.$VOL.SUBVOL.FILEID. The actual file’s name is contained in the FILEID, the file identifier.

In the OSS environment, filename id defined as any component of the pathname, that is /filename/filename/filename. The actual file’s name or file identifier comes after the rightmost slash.

Sometimes the linker is required to put or pull off the file identifier of a filename, and that means the following:

- On Guardian, it is the part of the filename after the last period. If there is no period in a name, the entire name is used.
- On OSS, it is the part of the filename after the last slash, if any.
- On the PC, it is the part of the filename after the last slash, backslash, or colon, if any.

Sometimes the linker is required to put a filename back together from its two pieces. The linker concatenates the two pieces using a period on Guardian, a slash on OSS, or a backslash on the PC.

A Note About MAP DEFINES

A DEFINE is a collection of attributes to which a common name has been assigned. These attributes can be passed to a process simply by referring to the DEFINE name from within the process. The =_DEFAULTS DEFINE is an example of such a DEFINE; this DEFINE passes the default node name, volume, and subvolume to a process.

The DEFINE mechanism can be used for passing file names to processes; this kind of DEFINE is called a CLASS MAP DEFINE. The following example creates a CLASS MAP DEFINE called =MYFILE and gives it a FILE attribute equal to \SWITCH.$DATA.MESSAGES.ARCHIVE:

```
1> SET DEFINE CLASS MAP, FILE \SWITCH.$DATA.MESSAGES.ARCHIVE
2> ADD DEFINE =MYFILE
```

Whenever your process accesses the DEFINE =MYFILE, it gets the name of the file specified in the DEFINE. For example, when your process opens =MYFILE, the file that actually gets opened is \SWITCH.$DATA.MESSAGES.ARCHIVE.

There are various items on the `eld` command line that are filenames. These include the parameters of various options, such as `-o, -l, -strip`, etc., as well as filenames that are just written directly on the command line. For such command line items, `eld` checks if they begin with equal signs. If so, in the Guardian case, the linker will immediately do the expansion of the DEFINE, so that all uses thereafter are the same as if the expanded name had been given originally (with one special case described below). The expansion of the name should also be done in upper-case, and the linker will put out an informational message. If the specified string cannot be expanded as a MAP DEFINE, that is an error. And, on other platforms, such as the PC or OSS, if a filename parameter begins with an equal sign, that is unconditionally an error.

On the other hand, there are certain items on the command line that are not filenames, although they look similar to filenames. In such cases, if the string starts with an equal sign, that is always an error, even on Guardian. Examples of this include the names of subvolumes specified in options such as `-L` and `-rpath`, and the DLL name specifed by the `-soname` option.

When it comes to parameters that are symbol names, no such rules apply. An equal sign at the start of a symbol name has no special significance to the linker.

There is a special case. In the case of the `-libname` (or `-set libname`, or `-change libname`) option, usually, it is an error if the parameter is not exactly of the form $a.b.c. However, a DEFINE can be used for this, even though a DEFINE always expands to the form \system.$a.b.c. In these contexts, after expanding the DEFINE, the linker also removes the system name.

Output Object Files

The linker can create a new object file or update an existing one in certain ways. When the linker is creating a new object file, by default it creates a loadfile, but the `-r` option instead tells the linker to create a linkfile. The TNS/E linker can also create an import library, as described in Creating Import Libraries on page 3-11.

When the linker creates a new linkfile with the `-r` option, and there was only one input file, the output file may be considered a new version of the input file. There are two reasons why you might do this:

- To strip the file in place, with the `-s` or `-x` option.
- To change the specified floating point type with the `-set floattype` option.

When the linker creates a new linkfile with the `-r` option, and there was only one input file, the linker is required to create the new file so that it has the same fingerprint as the original file.

The two types of ELF loadfiles produced by the linker are programs and DLLs. These are both PIC (position independent code). The default is to create a PIC program. The option named `-call_shared` means this, and so is the default for the TNS/E linker. The option to create a DLL is `-shared`. The option `-dll` is accepted as a synonym for `-shared`.

`eld` reports an error if you specify more than one of the `-call_shared`, `-r`, and `-shared` options.

When a DLL is created, its DLL name can be specified with the `-soname` option. If the `-soname` option is used, the linker accepts whatever string is given for the DLL name, exactly as-is, and without imposing any rules as to which strings are legal DLL names.

If the `-soname` option is not specified then the DLL name is determined to be the file identifier of the output file. The name of the output file is determined as described in The Creation of Output Object Files on page 2-5. And, in this case, when the linker is running on a host platform where the case of filenames is not significant (i.e., Guardian or the PC), the linker converts the DLL name to lower case. Note that, on OSS, the default name for the output file is “a.out”, so the default DLL name is similarly “a.out”, and that is true even if this is a Guardian subvolume.

The linker places the DLL name into the DT_SONAME entry of the .dynamic section of the DLL.

The option named `-dllname` is accepted as a synonym for `-soname`. `eld` reports an error if you specify this option when not building a DLL.

A user library is a DLL that is found in a special way by programs, but otherwise is no different from any other DLL. The `-ul` option is intended to be used when creating a DLL that is used as a user library.

The `-ul` option is synonymous with `-shared` plus `-export_all`, that is, to create a DLL and export all its symbols. See Using User Libraries on page 3-10 for an explanation of how the linker uses user libraries.

Certain DLLs are called implicit libraries. If the linker is creating a DLL, it can also be told to make it an implicit library with the `-make_implicit_lib` option. This option causes the linker to set the *EF_TANDEM_IMPLICIT_LIB* bit in the ELF header and to impose certain other rules, as mentioned in various other places in this manual.

The Creation of Output Object Files

The name of the output object file is specified by the `-o` option. If the `-o` option is not specified, but the `-soname` option is specified, then the name specified for the `-soname` option is also used for the `-o` option.

If neither the `-o` nor `-soname` option is specified, the default output file name is “aout” on the Guardian host and “a.out” elsewhere. Note that, in a Guardian subvolume of OSS, the created file would actually be named “aout”, because the period automatically goes away.

On Guardian, and in Guardian subvolumes on OSS, the linker creates object files with a file code of 800.

Output files on Guardian are odd unstructured files. The same is true for files created in Guardian subvolumes on the OSS host platform.

On the TNS/E OSS host platform, when a loadfile is created with OSS as its target personality, the linker gives it mode 777 because it is executable there. In all other cases when the linker creates an object file on any version of OSS it gives it mode 666. These modes are octal values, and they are AND’ed with the value returned by the `umask` system call.

Whenever the TNS/E linker creates an object file it first creates a work file. This file is in the same directory or subvolume as the output object file and will have a name of the form ZLDAF*nnn*, where *nnn* is a 3-digit integer of the form 000, 001, etc. The linker will choose the first name of this form that is not the name of an existing file, and it is an error if all names of this form are already taken. The linker will attempt to do this in an atomic way so that multiple links creating output object files for the same directory or subvolume won’t choose the same name. The linker will remove the work file if it detects any error before the work file is complete.

The `-temp_o` option specifies the name of an intermediate file. If the specified name is just the file identifier then the name is interpreted to be within the same directory or subvolume as the output object file. If the specified name is not the file identifier then `eld` reports an error if you do not specify the same directory or subvolume as the output object file.

If no error has occurred and the `-temp_o` option has been specified then the linker will rename the completed work file to this intermediate file name. If there already was a file with the name as specified in the `-temp_o` option, or if for any other reason the linker is unable to rename the completed work file to the intermediate file name, then

the name of the work file is unchanged and the linker puts out a warning message. Similarly, if the `-temp_o` option is not specified then the name of the work file is unchanged.

Thus, the `-temp_o` option gives the user a way to specify the name of a file that will only come into existence if the link is successful and will still be in existence if something goes wrong when the linker tries to put the file in the designated output file location.

Next, if there already was a file with the name of the output object file, the linker will try to remove that file. If the linker cannot remove it, the linker will put out a warning message saying that the output file of the link is in the location of the work file (possibly renamed by `-temp_o`). If there was no file with the same name as the output file, or if the linker was successful in removing that file, the work file is renamed to the output file name.

The name of the output object file can be the same as the name of an input file. That input file would therefore be removed if the link was successful.

It is possible that the linker might terminate unexpectedly, after creating a complete work file, but before being able to rename it to the final output file name. You might want to know where that file is, and that is the reason for the `-temp_o` option. In other words, it tells the linker which name to use, so that you don't have to search through all files with names of the form ZLDAF*nnn*.

Alternatively, if you specify the `-must_use_oname` option, that means `eld` reports an error if `eld` cannot delete the existing file and rename the workfile to it. This is a better behavior if `eld` is called from an automated script that knows how to watch out for errors, but not how to check for the file being left in a different place. The `-must_use_oname` option cannot be specified with the `-temp_o` option.

Creating Segments of the Output Loadfile

A segment is a contiguous portion of memory. The linker creates the segments of the loadfile, i.e., the text segment, the gateway segment, and the data segment(s). Appendix A, TNS/E Native Object Files tells which sections go into each of the segments, and which permissions are assigned by the linker to each of the segments.

A loadfile with callable procedures also has a gateway segment. There are two types of gateways that the linker can create. One type is used for transitions from user mode to exec mode, and the other for transitions from exec mode to kernel mode. All the gateways within the same loadfile are of the same type. The linker creates the first type of gateway, called a user gateway, if there are any procedures with the `CALLABLE` attribute. It creates the other type of gateway, called a kernel gateway, if there are any procedures with the `KERNEL_CALLABLE` attribute. `eld` reports an error if a mixture of both of these occur among the linker's input linkfiles.

By default, the linker knows how to create the gateways.

The linker option named `-instance_data`, which takes a single parameter for which there are five possibilities, tells the linker whether to create one data segment or two

and what additional rules should be enforced, as shown in the following table. `eld` reports an error if you specify the `-instance_data` option more than once with different parameters. `eld` reports an error if you specify `-instance_data` with `-r`.

Table 2-1. Parameters to the -instance_data Option

Parameter	Meaning
data1	Create one data segment. (This is the default.)
data2 data2protected* data2hidden	These three cases tell the linker to create two data segments, i.e., the “data constant” segment and the “data variable” segment.
data1constant	It is an error to have any data that would need to go into the data variable segment if we were told to create two segments.

* The data2protected parameter is supported only on systems running J06.09 or earlier J-series RVUs and H06.20 or earlier H-series RVUs.

If the loadfile being created had no data that would go into the data variable segment then the linker sets the *data1constant* bit in the *e_flags* field of the ELF file header. Otherwise, the linker sets the bit that corresponds to which parameter was given to the `-instance_data` option. The three different values set in the *e_flags* field in these cases tell the HP NonStop operating system how to protect the data variable segment.

When the `-make_implicit_lib` option is used, `-instance_data data1constant` is imposed, and `eld` reports an error if you specify `-instance_data` with any other parameter value.

There is no special boundary between the initialized data and the uninitialized data in the data (variable) segment. If the last portion of initialized data happens to be zero, the linker considers it to be within the uninitialized data, because what is called “uninitialized” data gets initialized to zero by the operating system. That possibly makes the size of the initialized data smaller within the object file. Based on this, the linker will fill in the *p_filesz* field of the appropriate program header. On the other hand, the linker also rounds up this value to a multiple of 4 kilobytes, thus possibly making the size of the initialized data larger. This tells the amount of space occupied by the appropriate segment within the loadfile, and the portion rounded up by the linker is initialized to zero in the loadfile.

The program header of type *PT_TANDEM_RESIDENT* points at the *.restext* section, that is, the resident code. There is no special boundary between the resident and non-resident code.

For a DLL the `-t` option specifies the starting address of the text segment of the DLL in memory. If the `-t` option is not used then it is possible for the DLLs address to be determined by using a DLL registry, as covered in the sub-section below.

Note that this is a DLL registry under the user's control, not the "public DLL registry" covered in Finding and Reading The Public DLL Registry (ZREG) File on page 3-23.

If the `-t` option is not specified, and no DLL registry is specified, the DLL is placed at the address 0x78000000.

If the `-d` option is not used, the data (constant) segment of a DLL is placed immediately after the text segment at a 64KB boundary if it isn't an implicit DLL, or 128KB if it is an implicit DLL. The `-d` option may be used to tell where the data (constant) segment starts. `eld` reports an error if you specify the `-d` option for a DLL without the `-t` option.

The two segment addresses for a program are always specified independently, with the `-t` and `-d` options. The default value for `-t` is 0x70000000, and the default value for `-d` is 0x08000000.

Note: In general, the `-d` option should not be used, because it will result in the creation of a file that the operating system will refuse to load.

The values specified for the `-t` and `-d` options are rounded up, if necessary to a multiple of 128KB for an implicit DLL, or 64KB for other types of loadfiles. If rounding up is necessary, a warning message is produced.

`eld` reports an error if you specify the `-t` or `-d` option with `-r`.

Using a DLL Registry

The linker uses a DLL registry to manage DLL addresses so that the virtual addresses of some or all of the DLLs being managed by a given registry do not overlap. Note that this is a DLL registry managed by the individual user, not the "public" DLL registry covered in Finding and Reading The Public DLL Registry (ZREG) File on page 3-23. The linker can pick addresses on its own, or the registry can be modified by hand to tell the linker what to do.

The `-check_registry` and `-update_registry` options tell the linker which registry to use.

- The `-check_registry` option is used to tell the linker how the DLL must be built, giving it no choice.
- The `-update_registry` option can make suggestions to the linker, but the linker still has decisions to make, and the registry is updated as a result. If neither option is used, the linker does not use a registry.
- `eld` reports an error if these options are used when the linker is not building a DLL. `eld` reports an error if both options are used. `eld` reports an error if either the `-check_registry` or `-update_registry` option is used with the `-t` or `-d` option.

When a DLL registry is used the linker lays out data segment(s) of the DLL immediately after the text segment, at a 64KB boundary if it isn't an implicit DLL, or at a 128KB boundary if it is an implicit DLL.

It is possible to use the DLL registry with a DLL that has two data segments, or that has a gateway. In such cases, when this discussion of the DLL registry refers to the "data segment", it means a fictitious segment that is the concatenation of the data constant segment, data variable segment, and/or gateway segment (whichever of these segments exist). The "(unrounded) size" of this fictitious segment includes any space that gets added before the second or third of these segments in order to make that segment start on a 64 KB boundary.

The `-check_registry` and `-update_registry` options attempt to open an existing registry for reading. If they succeed in opening the file, they keep it open and locked until they are through with the file. The time that the registry file would be open, and therefore locked, would typically be a short time for the `-check_registry` option, and a long time for the `-update_registry` option.

If the specified file exists but cannot be opened because it is currently in use (i.e., as far as the operating system is concerned) or locked (by the linker's locking mechanism), the linker will pause for a brief time and try again. That could be helpful when several links are being done in the same place, and the other links are using `-check_registry`. If these opens don't work after several attempts, `eld` reports an error.

The registry is a text file. In a Guardian subvolume of OSS it must be a code 180 file. Blank lines are treated as comments, as are lines whose first two non-blank characters are hyphens. Each of the other lines of the registry must begin with one of the keywords explained below, and the rest of the line provides the parameters for that keyword. The keyword and parameters are separated by blanks or tab characters.

The keyword `-dllarea`, if used, must be the first keyword in the file. It takes two numerical parameters whose format is the same as the format of a <hexadecimal number> as described in The Linker Command Stream on page 1-5. The first parameter tells the starting address for the placement of DLLs and the second parameter tells the ending address. Whichever one of the two addresses is smaller determines the lower bound for DLL addresses. The other one determines the upper bound. The one called the starting address tells at which end of this region the linker begins placing DLLs, so that the linker can either work upward from the bottom of the region, or downward from the top of the region. The DLLs managed by this registry must have addresses that fit within this range, meaning that the starting address of the DLL must be at least as large as the lower bound and the starting address of the DLL plus its reserved size (as explained below) must be no larger than the upper bound.

If the `-dllarea` keyword is not used, the default is the following:

```
-dllarea 0x80000000 0x70000000
```

In other words, by default, a DLL registry manages DLLs whose addresses lie between 0x70000000 and 0x80000000, and the linker lays out DLLs starting at the higher end.

Each of the remaining lines of the registry must have the keyword `-range`. Each such line provides information about a DLL. The `-range` keyword has three parameters:

- a string, to tell the name of the DLL

- a <hexadecimal number> to tell the starting address of the DLL
- a <hexadecimal number> to tell the reserved size of the DLL

Together, the starting address and the reserved size tell the total range of addresses that this DLL, or possibly a larger version of it in the future, occupies or is intended to occupy.

Whenever the linker opens a registry, it checks for the following errors. `eld` reports an error if any of the DLL ranges listed in the registry extend outside the range of addresses that the registry allows for DLLs. It is also an error if any of the addresses or ranges in the registry are not multiples of 64KB if the linker is not building an implicit DLL, or 128KB if the linker is building an implicit DLL.

The linker determines whether the registry contains an entry for the DLL being built by comparing the name of the file being created (as specified in the `-o` option) against the names listed in the registry. `eld` reports an error if the name is found more than once in the registry.

Note that the name comparison mentioned in the previous paragraph is by an exact match. For example, if you have several DLLs that you store in sibling subdirectories, named *a, b,* ..., you might choose to have a single registry file that lists all the DLLs with names relative to that parent directory, such as `a/libdll1.so`, etc. In that case, you would have to spell the filename of each DLL that way in the `-o` option when you were creating it, which means that you'd have to be running the linker in the parent directory. Or, if the filenames were unique in all these directories, you might choose to put only the last part of the name into the registry, in which case you would have to be within the appropriate subdirectory when building the DLL. Or, even if you have all your DLLs in one place, you still need to be consistent in how you spell their names in `-o` options, for example, not saying `libdll1.so` the first time you build it and `./libdll1.so` the next time you expect to find it in the registry.

When `-check_registry` is used `eld` reports an error if a registry file of the specified name does not exist, or cannot be opened for reading, or does not have the proper format, as described above, or if the name of the DLL being built is not found in the registry. The entry for that DLL then tells the starting address to use. `eld` reports an error if the DLL does not fit within the size reserved for it in the registry. The registry is not modified.

The rest of this section describes what happens when `-update_registry` is used. In this case, a registry file of the specified name need not already exist. If it does not exist then the linker creates it. If the registry file already exists `eld` reports an error if the file does not have the proper format. The DLL name need not already be listed in the file.

If the DLL name was already listed in the file then the address listed for the DLL in the file is a suggested address. The DLL is placed at this address if the size required for it, including the necessary rounding but not including any room for growth, is less than the size reserved for it in the registry. In this case the registry is not updated. Otherwise, the linker emits a warning message, chooses an address for the DLL as

described below (after, in effect, deleting the old entry for the DLL from the registry), and updates the registry accordingly.

The `-grow_limit` option may only be specified if `-update_registry` is used. If `-grow_limit` is not specified then the linker determines the reserved size of the DLL from the following options, which are only allowed when `-update_registry` is used and `-grow_limit` is not used:

- `-grow_text_amount` the absolute amount by which the text may grow
- `-grow_data_amount` the absolute amount by which the data may grow
- `-grow_percent` the percentage amount by which the text or data may grow

The defaults for `-grow_text_amount` and `-grow_data_amount` are 0. The default for `-grow_percent` is 10. A size is calculated for each of text and data by first adding the corresponding “amount” option to the size of that segment (before rounding), or adding the percentage specified by the “percent” option to the size of that segment (before rounding), and taking the maximum of these two values. The resulting size for each segment is then rounded up to a multiple of 64KB (or, 128KB if the linker is building an implicit DLL), and the sum of these two sizes is the reserved size of the DLL.

When the linker is choosing a new place for a DLL, because it wasn’t specified in the registry before or didn’t fit where the the registry previously specified, and the `-grow_limit` option has been given, the reserved size that the linker gives to the new entry in the registry is the value specified in the `-grow_limit` option, rounded up to a multiple of 64KB (or, 128KB if the linker is building an implicit DLL). In this case `eld` reports an error if the sum of the sizes of all the segments of the DLL (including rounding) is larger then the value specified in this option (rounded up to a multiple of 64KB or 128KB, depending on whether it is an implicit DLL).

If the DLL name was not already listed in the file, or didn’t fit in the place previously listed for it, then the linker chooses the address for the DLL by looking for blocks of space that are at least as large as the reserved size for this DLL, that lie within the range of addresses that the registry allows for DLLs, and that don’t overlap the space reserved for any other DLLs in the registry. `eld` reports an error if there is no block large enough. The linker chooses such a block that is closest to the starting address for the registry. Thus, the linker searches upward from the lower bound of possible addresses, or downward from the upper bound, depending on how the bounds were specified for this registry. The registry is updated to tell the address and reserved size of the new DLL.

If the user wishes to specify an address for a new DLL in the registry, rather than letting the linker choose the address, the registry file can be edited by hand. It is necessary to specify both the starting address of the DLL and its reserved size, as multiples of 64KB. Or, if this is a registry that is being used for implicit DLLs, then they should be multiples of 128KB. It is also possible to change these values for a DLL already listed in the registry. It is permissible to edit a registry so that some DLLs have overlapping address ranges. When the linker picks an address on its own it requires that this DLL not overlap any other ones in the registry, but the linker doesn’t check

whether entries already in the registry overlap. Whenever a DLL registry is updated by the linker and the DLL had previously been mentioned in the file, the old entry is replaced by the new one so that the DLL is not mentioned more than once in the registry.

A new registry file is always created, mostly following the same rules as given earlier in The Creation of Output Object Files on page 2-5.

One difference is that the name of the work file is *ZLDARnnn* rather than *ZLDAFnnn* and the name of the intermediate file, if desired, is specified by the `-temp_r` option rather than the `-temp_o` option. Also, there is a `-must_use_rname` option, instead of `-must_use_oname`.

Also, if the linker cannot create a new DLL successfully (i.e., terminates in error before that point), then the linker does not modify the existing DLL registry. However, once the linker has succeeded in creating its output DLL, the linker will not consider any subsequent problems with the DLL registry to be errors. If the linker cannot update the registry as desired, that will only be reported as a warning.

Note that the linker must first read an existing private DLL registry before it writes out a new version of it. As explained above, if the linker can't read it, `eld` reports an error. So, the `-temp_r` and `-must_use_rname` options are only relevant to situations where the linker had permission to read the existing private DLL registry, but not delete it.

Input Object Files

TNS/E linkfiles have no object file version number associated with them.

Loadfiles have a version number stored in the *.tandem_info* section, placed there by the linker. At the present time the version number is zero. When the linker reads a loadfile for any reason the linker considers it an error if the loadfile contains a version number different from zero. This version checking is disabled by the `-no_version_check` option.

On the Guardian platform the linker does not check that object files have the proper file code. If the file was not the right kind of file, the linker would soon realize it in some other way.

How the Linker Finds Its Input Files and Creates the *.liblist* Section

The linker locates linkfiles, archives, DLLs, and import libraries based on items specified in its command stream. This section provides the rules for doing this. As part of this process, the linker also creates the *.liblist* section for its output loadfile.

See Using Archives on page 2-16 for an explanation of how the linker decides which files to use from an archive.

See Presetting Loadfiles on page 3-5 for an explanation of how the linker also finds indirect DLLs by using the *.liblist* sections of other DLLs that it has already found, and for how the linker finds the import library that represents the implicit DLLs.

See Using User Libraries on page 3-10 for an explanation of how the linker finds user libraries.

The linker does not have any built-in set of DLLs for which to look, other than the import library that represents the implicit DLLs. Although other DLLs may typically be needed for C/C++ runtime support, they must be specified explicitly in the linker's command stream. The user who invokes the linker indirectly through the C/C++ compiler may be unaware of this, because the C/C++ compiler automatically adds the appropriate command stream items when it invokes the linker.

The linker looks for import libraries the same way it looks for DLLs. Unless mentioned otherwise, when this section explains how the linker finds DLLs, the same remarks apply to import libraries. The linker accepts archives, DLLs, or both, according to the following options:

- `-b static` - only accept archives, not DLLs
- `-b dllsonly` - only accept DLLs, not archives
- `-b dynamic` - accept both archives and DLLs

These options form a three-way switch, selecting one of three modes for the linker at a given point in the command stream. These options can be specified multiple times in the command stream, each time setting the mode for subsequent items in the command stream until the mode is changed again. At the beginning of the command stream the mode is `-b dynamic`.

An item in the command stream that causes the linker to find a linkfile, archive, or DLL is one of the following three things:

- a name specified directly in the command stream
- a `-l` option whose parameter is a file identifier
- a `-l` option whose parameter is not a file identifier

The definition of file identifier is given in Filenames and The File Identifier on page 2-2.

The `-lib` option is a synonym for `-l`, and may be preferred because the `-lib` option is not case-sensitive, whereas `-l` is a different option from `-L`.

The linker uses one of the following two methods to find a file, based on the way it was specified in the command stream:

- for a name specified directly in the command stream, or for a full filename specified in a `-l` option, the linker opens the file normally.
- for a file identifier specified in a `-l` option, the linker searches for the file.

Opening a file normally means that the linker tries to open the name exactly as it is specified. This means that the name may be interpreted relative to the current directory

or subvolume, as is normally done for the corresponding host platform. `eld` reports an error if the file does not exist, or if the linker cannot open it for reading.

In the case that the name was specified directly in the command stream, the file can be a linkfile, archive, or DLL. In the case that the name was specified as a full filename in a `-l` option, the file must be an archive or DLL. In either case, if `-b static` is in effect then `eld` reports an error if the file is a DLL, and if `-b dllsonly` is in effect then `eld` reports an error if the file is an archive.

The linker searches for a file by performing several steps. Most of these steps involve looking for the file in a given directory or subvolume, although one step is a special way to look for public DLLs. When the linker is looking in a directory or subvolume there are certain filenames that it expects to find. If a desired file doesn't exist, or if it does exist but the linker cannot open it for reading, then the linker continues without warning. More details of what the linker does during the search are provided in the following sub-sections of this manual.

By default, if the linker gets to the end of its search without finding a file to satisfy a `-l` option, `eld` reports an error. However, the `-allow_missing_libs` option tells the linker that it is not an error unless this happens when the current mode is `-b static`. If it isn't an error, the linker instead emits an informational message.

When the `-r` option is specified, telling the linker to create a linkfile rather than a loadfile, the linker looks for archives and DLLs the same way as in other cases, but then the DLLs are ignored.

Informational messages tell the file names of all the archives and DLLs that were opened by the linker, saying for each one whether it is an archive, a DLL, or an import library. If it is an import library, the message tells whether the import library is complete or incomplete.

Whenever the linker finds an import library from the command stream, it checks whether the DLL name within this file is "__IMPLICIT_LIB__". If so, it is recognized to be the import library that represents the implicit libraries, and it is used as the last item in the search list, as described in Presetting Loadfiles on page 3-5. It is always an error if the linker finds a DLL, rather than an import library, whose DLL name is "__IMPLICIT_LIB__".

If the `-make_implicit_lib` option is given, `eld` reports an error if any of the DLLs or import libraries that the linker finds in the command stream do not have the `EF_TANDEM_IMPLICIT_LIB` bit set in their ELF headers. Also, when `-make_implicit_lib` is used, `eld` reports an error if the linker finds an import library whose DLL name is "__IMPLICIT_LIB__".

The linker creates the *.liblist* section of its output loadfile. The *.liblist* section contains one entry for each DLL (or import library) obtained from the command stream, regardless of the method used to find it. The entry tells the name of the DLL, as obtained from the *DT_SONAME* field of the DLLs *.dynamic* section. There is no requirement that the name found within the DLL, and therefore stored in the *.liblist* section, match the name that was used to find the DLL from the command stream.

However, to simplify build processes, the user may find it convenient for these names to be the same.

The *.liblist* section does not contain an entry corresponding to the user library, does not contain an entry for the import library that represents the implicit libraries, and does not contain entries for DLLs that the linker finds indirectly.

When the search to satisfy a `-l` option does not succeed, and this is not an error (because the `-allow_missing_libs` option was specified and the mode is not `-b static`), the *.liblist* section will still contain an entry for that `-l` option. In other words, it is assumed that the name was intended to be a DLL as opposed to an archive. Because the linker doesn't have a DLL name to put into the *.liblist* section entry, it instead will put in the string that was the parameter to the `-l` option. The linker also sets the `LL_NOT_FOUND` bit in the *.liblist* section entry to identify this as an entry for a `-l` option for which the search did not succeed.

Each *.liblist* section entry tells if it is reexported. The `-reexport` and `-no_reexport` options form a two-way switch, selecting one of two modes for the linker at a given point in the command stream. These options can be specified multiple times in the command stream, each time setting the mode for subsequent items in the command stream until the mode is changed again. When `-reexport` is in effect, the *.liblist* section entry for a DLL found in the command stream says that it is re-exported. When `-no_reexport` is in effect it is not re-exported. At the beginning of the command stream the mode is `-no_reexport.` `eld` reports an error if you use either of these options when not building a DLL.

The following rules apply to situations where the same file is found several times in the command stream:

- It is not explicitly called out as an error if the same linkfile is specified more than once in the command stream, but it may lead to the error of multiply defined symbols, as explained later in Accepting Multiply-Defined Symbols on page 3-17.
- It can be useful to specify the same archive more than once in the command stream, as explained later in Using Archives on page 2-16.
- With regard to DLLs, a requirement is that the same DLL name cannot be present more than once in the *.liblist* section. The rule is that, if the linker finds the same DLL more than once in the command stream, where "same" means that they have the same DLL name, the linker ignores all the instances after the first one. However, the linker also checks whether the DLL was found with the same export digest each time. If not, a warning message is provided.
- When a user library is used by the linker, it is treated like an additional DLL at the beginning of the command stream. However, if the DLL name within the user library matches the name of another DLL found later, that other DLL is still used, and gets a *.liblist* entry 7 , although the linker puts out a warning message about this.

The rules above, about finding the same DLL name more than once, also apply to the special DLL name "__IMPLICIT_LIB__".

Using Archives

An archive is a file that contains copies of linkfiles. The linker looks for files within the archive to be used by the link. Linkfiles in the archive are used if they define global symbols that are currently known about, but undefined, at this point in the command stream. In each case, once the linker decides to use a file from the archive, that entire file is used the same way it would be used if it had been specified directly in the command stream.

It can be meaningful to specify the same archive more than once in the command stream, because each time the linker opens the archive it only looks for linkfiles that resolve symbols that are needed at that point.

When a linkfile is brought in from an archive, that can lead to additional needed symbols. The archive is searched repeatedly to find such symbols. However, once the linker has moved on to the next token in the command stream, this archive will not be looked at again unless it is specified again in the command stream.

The `-u` option is used to specify the name of a symbol for which `eld` should look in archives in the command stream, if `eld` had not already seen a definition of this symbol before the archive was encountered. Note that this is similar to having a hypothetical linkfile at the start of the command line that declared such a symbol without defining it. However, one difference is that, when a symbol really does get declared in a linkfile, that declaration tells if the symbol is globalized or not, but the `-u` option does not imply anything about whether the symbol is globalized.

The `-all` option tells the linker to unconditionally use all linkfiles found in archives, rather than only using those that provide needed symbols. The `-none` option turns this off, so that files found in archives are used only if they provide needed symbols as described above. These options form a two-way switch, selecting one of two modes for the linker at a given point in the command stream. These options can be specified multiple times in the command stream, each time setting the mode for subsequent items in the command stream until the mode is changed again. At the beginning of the command stream the mode is `-none`. One use of `-all` is to convert an archive into a DLL by telling the linker to build a DLL that contains all the same files as are present in the given archive.

The option `-include_whole` is accepted as a synonym for `-all`, and `-no_include_whole` as a synonym for `-none`.

In general, users having 32-bit and 64-bit versions of their code are recommended to maintain two separate versions to avoid confusion. For the archives, it is recommended that the users do not mix up the `32-bit` and `64-bit` object files in the same archive. Typically, a user has two archives with similar contents, one for the `32-bit` and the other for the `64-bit` case. If the user links on to the OSS, two archives can have the same name but, if the `32-bit` archive is placed into `/lib`, `/usr/lib`, or `/usr/local/lib`, while the `64-bit` archive is placed into `/lib64`, `/usr/lib64`,

or /usr/local/lib64 then, the same -l option in the linker verifies the desired one based on the type of link created. Otherwise, the user can decide the naming convention for the archives and formulate the linker to find the right version.

Archives are a method of making loadfiles smaller, because the linker will automatically bring into the link only those members of the archive which are needed. Archives are also a way of packaging together multiple linkfiles, as an alternative to using the -r option.

The Steps in Looking for Archives and DLLs

The linker performs the following steps, in the order given, to search for an archive or DLL:

1. The linker first looks in the directories or subvolumes whose names are specified in -first_L options in the command stream, in the same order that those options occurred in the command stream.
2. The linker next looks for public libraries. However, the linker does not do this if the -r option is specified or if -b static is in effect.
3. The linker next looks in the directories or subvolumes whose names are specified in -L options in the command stream, in the same order that those options occurred in the command stream. The -libvol option is a synonym for -L, and preferred because the -libvol option is not case- sensitive, whereas -L is a different option from -l.
4. On OSS, the linker looks in a list of standard places. If the linker is building a 64-bit object, or if the -alf option is processing a 64-bit loadfile, then the first three places in this list are/lib64, /usr/lib64 and /usr/local/lib64. In all other cases, the next three places in the list are /lib, /usr/lib, and /usr/local/lib. Each of the above names is prefixed with the contents of the COMP_ROOT environment variable, if the variable is defined.
5. Finally, on Guardian or OSS, if the linker is building a 64-bit object or if the -alf option is processing a 64-bit loadfile, the linker looks into the $SYSTEM.YDLL and in all other cases the linker looks into $SYSTEM.ZDLL. However, the linker does not perform these actions if the -r option is specified or if -b static is in effect.

The following sub-section tells the rules that are used for looking for public libraries (step 2 above), and the sub-section after that tells what the linker does when it is looking through other directories or subvolumes (all the steps above other than step 2).

If the -nostdlib option is specified, steps (2), (4), and (5) in the above list are skipped, so that the linker would only look in the places specified by -first_L or -L options. The option -no_stdlib is accepted as a synonym for -nostdlib.

The same search method is used for all searches. For example, it doesn't matter whether some of the -L options came later in the command stream than the -l option for which the search is being performed.

The linker does not make use of any environment variables other than COMP_ROOT.

When eld is creating a new object file X from a set of linkfiles, or processing a loadfile X with the -alf option, it is desired that DLL's of the appropriate data model match the data model of X. Therefore, if X is neutral then it is desired that all the DLL's used are neutral, and if X is not neutral then it is desired that all the DLL's used are either neutral or same as X. During a search, if a DLL that is not desired is encountered, the search continues. If the search function later finds an archive or a DLL which is desired, the previously found DLL that was not desired is ignored. If the search does not succeed, the DLL found earlier is used, even though it is not desired. A warning is issued if a DLL of the undesired model is used.

Finding Public DLLs

This section explains how the linker looks for a public DLL based on the file identifier. (The file identifier may have been specified in a -l option in the command stream, or may have been found in the liblist of some other DLL).

As described in Finding and Reading The Public DLL Registry (ZREG) File on page 3-23, the linker may or may not have located the public DLL registry file. If the linker did not locate the public DLL registry file, it does not look for public DLLs. (It continues to look for this DLL in other ways). The rest of this section assumes that the linker did locate the public DLL registry file.

The linker verifies whether the file identifier matches with one of the public DLL filenames found in the public DLL registry. If there is no such match and the linker is either creating a 64-bit object file or the -alf option processes a 64-bit loadfile. The linker verifies whether the name added with prefix "y" and suffix "DLL", matches one of the public DLL filenames found in the public registry. Later, the linker verifies whether the name added with prefix "z" and suffix "DLL" matches the public DLL filenames found in the public DLL registry.

Note. All of these matches are case insensitive.

If the linker does not find the file in the appropriate place as described above, or finds it but cannot open it for reading, or if the file is not a DLL, eld reports an error.

If the linker is creating a 64-bit object file or the -alf option is processing a 64-bit loadfile, and if a file exists with the name lib instead of "y" prepended, and y.so instead of "DLL" appended, then linker uses this file. If a file exists, with the name lib instead of "z" prepended and .so instead of "DLL" appended, then linker uses this file. The linker considers it an error only if these files do not exist, or are not DLLs.

Note. There is also an exception which applies to platforms other than Guardian.

In other words, as a result of the special cases described above, it is possible to put a copy of the real zreg file into a location on the PC or OSS so that it lists the public DLLs with names like zcredll or ycredll, etc., but in the same location. Instead of having files named zcredll or ycredll, etc., you can rename them as: libcre.so, or libcrey.so, and so on. eld does not consider it as an error when

you enter the `-l cre` option or if it processes a DLL and as a result finds the name: `zcredll` or `ycredll` in the liblist of the other loadfile.

The Rules to Find Files

This section tells the rules that the linker uses to decide which files to try to find and open in each directory or subvolume that it is searching. This applies to all the steps followed by the linker in searching for an archive or DLL, other than the special step for the public DLLs.

This is the algorithm that the linker uses to look for a file within a directory or subvolume:

- First, the linker looks for a file with the name as it was given in the `-l` option.
- Next, if the platform is not Guardian, and this is not a directory on OSS that is a Guardian subvolume, and `-b static` is not in effect, the linker looks for a filename of the form `libx.so`, where `x` was the name specified in the `-l` option. This rule is not applicable for instances where the linker is searching for an indirect DLL. While searching for an indirect DLL, `eld` searches files with the name specified in the `.liblist` section.
- Finally, if the platform is not Guardian, and this is not a directory on OSS that is a Guardian subvolume, and `-b dllsonly` is not in effect, the linker looks for a filename of the form `libx.a`, where `x` was the name specified in the `-l` option.

For example, on OSS, if the linker is given the `-l ab` option, and it is searching through an OSS directory that is not a Guardian subvolume, it may find `ab`, `libab.a`, or `libab.so`. Or, if given the `-la.b` option, it similarly may find `a.b`, `liba.b.a`, or `liba.b.so`. On the other hand, if it is searching through a Guardian subvolume on OSS, and it is given the `-lab` option, it will only look for `ab`, not `libaba` or `libabso`. If given the `-la.b`, it won't look for any of these things in a Guardian subvolume.

When the linker opens a file under the name as it was given in the `-l` option, if `-b static` is in effect then `eld` reports an error if the file is a DLL, and if `-b dllsonly` is in effect then `eld` reports an error if the file is an archive. When the linker opens a file with the name `libx.so`, `eld` reports an error if the file isn't a DLL. When the linker opens a file with the name `libx.a`, `eld` reports an error if the file isn't an archive.

The linker does not check whether some of the directory or subvolume names are syntactically incorrect. It simply discovers that the files with the names constructed as explained above can't be opened, and then continues. For instance, the user may find it convenient to use the same search path on several platforms, where the search path contains some names valid for one host platform, and other names valid for another host platform. On each host platform, the names valid for that platform may do something meaningful, and the other ones would be ignored.

3 Binding of References

This section contains the following topics:

- Overview -an overview of symbol resolution and code relocation.
- Presetting Loadfiles - the process of resolving references to DLLs at linktime.
- To Preset or Not to Preset, and Creation of the LIC - the linker's rules for presetting.
- Handling Unresolved References - what happens if a symbol is not found in any loadfile in the linker's search list?
- Using User Libraries - introduces the libname options.
- Creating Import Libraries - three types are available.
- Ignoring Optional Libraries - a command stream toggle is available.
- Merging Symbols Found in Input Linkfiles - merges the symbol information from the input files into its output file.
- Accepting Multiply-Defined Symbols - how to accept multiple definitions for a symbol?
- Using the -cross_dll_cleanup option - reduces the total size of a program and the private DLLs that are used by a process.
- Specifying Which Symbols to Export, and Creating the Export Digest - to export global and defined symbols.
- Public Libraries and DLLs - two types, namely implicit and explicit.
- The Public Library Registry - lists all public DLLs by name.
- Finding and Reading The Public DLL Registry (ZREG) File - how the linker finds the file.

Overview

The primary job of the linker is to bind abstract (symbolic) names to real addresses. For example, as a programmer you can use the name `getfile` in one module, while the linker binds that to "a location 512 bytes from the start of module `iosys`". This is known as symbol resolution.

The other closely related function of `eld` is that of relocation. Compilers and assemblers generally create each file of object code with the program address starting at zero, an address you are unlikely to be able to use. Furthermore if a program is created from multiple subprograms, all the subprograms have to be loaded at non-overlapping addresses. These addressing problems are solved by relocation, the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses.

The functions are related because the linker can use symbol resolution to handle relocation, by assigning a symbol to the base address of each part of the program then treating the relocatable addresses as references to the base address symbols.

In the case of the HP NonStop operating system compilers; in TNS/R the addresses started at zero and incremented for each section, in TNS/E all section addresses start at zero, they don't really contain addresses but rather contain section offsets.

`eld` is concerned with cases where the compiler or assembler does not know the ultimate contents that the object file should contain for symbolic references. Such cases are listed in the relocation tables that the compiler or assembler creates in linkfiles. When `-r` is specified, the linker creates new linkfiles with the same kinds of relocation tables.

This section considers how the linker creates a loadfile.

In order to fill in the proper values for symbolic references, the linker matches up symbols across linkfiles, determines runtime addresses for the symbols defined in this loadfile, and searches other DLLs to resolve references to symbols not defined within the current object file. The loadfile that is built by the linker tells `rld` what needs further examination when the loadfile is brought into virtual memory. The addresses chosen by the linker are called the preferred addresses for this loadfile. A program is always loaded at its preferred addresses, but that is not necessarily true for a DLL.

Here are two examples of references:

- A data item that is a pointer, initialized with the address of another data item. The compiler or assembler doesn't know the final address to put in, so it creates a relocation table entry for this data item. The linker may fill in a value, but in any case it propagates the same type of relocation table entry to the loadfile for `rld` to use.
- An instruction that refers to some data item. This is different from the case above because only the linker can modify executable code, not `rld`. The compiler or assembler may generate code that looks up the address of the symbol in a data location whose address is calculated by adding a 22-bit offset to the GP register, and it creates a relocation table entry accordingly. The linker allocates that data location as part of the *.got* section and updates the code so that it contains the right 22-bit GP-relative offset to reach the corresponding *.got* section entry. The linker may also fill in a value in the *.got* section entry, but in any case it puts information into the object file so that `rld` knows how to fill in or update the *.got* section entry at load time.

There are various *relocation type*s which tell the linker how to modify a given location in code or data after it knows the address that need to be represented there. We often say that the linker "fills in the address of the symbol at the relocation site", but that isn't precisely correct. Depending on the relocation type, it may the actual address of the symbol, or it may be something else, such as the GP-relative offset of the *.got* entry that contains the address of the symbol, and so on.

The way references are fixed up across loadfile boundaries depends on the import control of the loadfile being built. This controls how the search list is created, both at link time and at load time, to find the DLLs that are needed to resolve symbols referenced in the loadfile.

There are three choices for this, set by the following options:

-b localized (this is the default) means “localized”. The searchlist for the loadfile at link time and load time is:

- loadfile itself
- If the loadfile is a program and has a user library, that user library.
- a breadth-first transitive closure of re-exported liblist-specified DLLs.
- DLLs specified in the liblist.

This is the the HP NonStop operating system default and gives you the most control on how your undefined references are resolved at runtime.

-b globalized means “globalized”. The searchlist for the loadfile at link time is the same as that of “localized” except that the transitive closure does not include re-exported DLLs. At load time the searchlist order is as follows:

- The program
- If the program has a user library, that user library.
- The liblist of the program
- a breadth-first transitive closure of liblist-specified DLLs
- Other dynamically-loaded DLLs

The key thing to remember is that a globalized loadfile can have its own definitions preempted by another loadfile. This is the UNIX default behavior.

-b semi_globalized or `-b symbolic` means “semi-globalized”. It is basically the same as -bglobalized except that the loadfile itself is at the head of the searchlist at load time. That means that its definitions cannot be preempted.

All three possibilities are allowed, whether building a program or a DLL, although for a program the semi-globalized case means the same thing as globalized. `eld` reports an error if more than one of these options is specified. If the `-make_implicit_lib` option is given, `eld` reports an error if you specify an import control other than localized.

The import control is stored in the *EF_TANDEM_IMPORT_CONTROLS* bits of the *e_flags* field of the ELF header of the loadfile being built. This manual explains how these options affect the linker’s actions, but does not explain all the details of how they affect what `rld` does.

Do not confuse these uses of `-b` with the other uses of `-b` described earlier in How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

The linker does not necessarily know the proper load time values to use for addresses, because things can change between link time and load time for reasons such as these:

- The DLL currently being built, or the other DLLs to which it refers, will not necessarily be loaded at the same (preferred) addresses as those assumed by the linker.
- The other DLLs to which the current loadfile refers may not be the same DLLs used at runtime, because they might be updated in the interim. Thus, a given DLL may no longer provide the same set of symbols, or it may provide the same symbols but at different addresses. This updating of DLLs may also change which DLLs are indirectly found.
- Although the same DLLs may be in the same places at load time as at link time, different ones may be used because `rld` looks for them with different path lists.

Also, the rules that are used for binding references at load time are not rules that could be completely implemented by the linker, even if things didn't change as described above. For instance, it is possible for a reference from a globalized or semi-globalized DLL to be resolved in the main program at load time, but the linker does not see the main program when it is building that DLL. The rules are such that, if things don't change as described above, and if the linker is creating a main program or a localized DLL, then it will search the same DLLs, and in the same order, as `rld`.

Some symbols are exported by the loadfile that contains them, and some are not. Only the exported symbols are visible to other loadfiles.

The linker fills in all references to symbols that are defined in the same loadfile, using the address of the symbol within this same loadfile.

In some cases what the linker fills in may need to be modified by `rld`, but the linker still guarantees to fill in all these references that were within the same file. `rld` may be able to decide that this much of what the linker did was correct, and therefore avoid the need to recalculate those references at load time. Or, `rld` may recalculate the references, because it wasn't sure that what the linker did was correct, and in such cases it can still be advantageous if the linker was correct.

The linker also may fill in other references, to symbols not defined within this same loadfile. In particular, the linker may do this for symbols that are not globalized symbols, and that process is called presetting. References to globalized symbols are not filled in, and do not generate errors or warnings. When the linker fills in references to undefined symbols `rld` may again be able to decide that what the linker did was correct, and therefore avoid additional work at load time.

There is a concept that goes under various names, such as "delayed binding", "lazy evaluation", etc. This refers to the situation where there can be procedure call references that are purposely not resolved at link time, or even at load time. Instead, the first time the procedure is called, the `rld` runtime support resolves its address. HP NonStop systems do not support this capability.

Presetting Loadfiles

This section discusses how the linker presets a loadfile. Except as mentioned otherwise below, import libraries are used the same way that DLLs are, and remarks made here about DLLs in the search list also apply to import libraries in the search list.

The search list begins with the loadfile itself. Next, for a program that has a user library, comes the user library. The user library mostly acts the same as any other DLL, except for its special placement at this point in the search list. After that come the entries in this loadfile's own *.liblist* section, namely, the DLLs that were obtained from the command stream, in the order they were found in the command stream.

Then, each DLL that has already been placed into the search list after the loadfile itself can specify other DLLs in its *.liblist* section. The linker finds these other DLLs by using the names in the *.liblist* section. The linker performs a breadth-first traversal of all these *.liblist* sections in order to add more entries to the search list. The names added to the search list after the *.liblist* entries of the loadfile itself are called the indirect DLLs. A DLL is only added to the search list if its DLL name is different from all of those already in the search list. If the loadfile being built is localized then the traversal to find indirect DLLs only pays attention to *.liblist* section entries that are re-exported. The linker makes use of the DLL names found in one *.liblist* section in order to search for other DLLs, but does not check that the DLL name found within a DLL matches the name that was used to search for it.

The same kinds of rules that apply to DLLs found directly in the command stream, as given in How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12, also apply to indirect DLLs. This includes the following things:

- The rule that, when the `-make_implicit_lib` option is used, all the DLLs that are found must be implicit.
- The rules about multiple DLLs with the same DLL name.
- The rules about which informational messages to write out.
- The rules about what to do if a search completes without finding a DLL.

The above rules about an indirect DLL are not enforced when the linker does not look for that DLL. More specifically:

- The linker does not look up a name found in a *.liblist* section entry if that entry is not reexported and the linker is building a localized file.
- The linker need not look up names in *.liblist* sections after it has decided that it is neither going to preset nor check for unresolved references. (But the linker is allowed to continue looking for DLLs in this case and do the usual checks on them.)

Finally, if the `-make_implicit_lib` option has not been specified, the linker tries to add the import library that represents the set of implicit libraries to the end of its search list.

There are several ways that the linker may find the import library that represents the implicit libraries. Regardless of how the import library that represents the implicit libraries is found, the linker always uses it the same way, placing it at the end of the search list.

The linker may have found the import library that represents the implicit libraries from the command stream, such as described in How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12. The linker recognizes it as the import library that represents the implicit libraries because it has the special DLL name "__IMPLICIT_LIB__". Note that it doesn't matter where it was found in the command stream, because it any case it is placed at the end of the search list.

In the previous paragraph, found "from the command stream" means that the linker found it during the same process that the linker uses to find all the other DLLs, those that come directly from the command stream and those found indirectly through entries in liblists. If, during this process, an import library that represents the implicit libraries is found more than once, the rules that are followed are the same as those followed when that situation comes up for other DLLs. In particular, it is not an error, and the first instance is the one that the linker will use, but if multiple occurrences of this library are found via non-identical export digests then a warning will appear.

If the linker has not found the import library that represents the implicit libraries from the command stream, it may also determine its name by a system call.

If the linker still has not found the import library that represents the implicit libraries, but it has found a public DLL registry file the linker next looks for a file named zimpimp in the same subvolume or directory as the public DLL registery file. On OSS, where case matters, zimpimp is lower case. If that file does not exist, or the linker can't open it for reading, the linker continues processing.

If the linker finds the import library that represents the implicit libraries via the above process, and successfully opens it, then `eld` reports an error if the file is not a DLL, or if the DLL name within the file is not "__IMPLICIT_LIB__".

If the linker has not found an import library that represents the implicit libraries by the process described above, and has gotten to this point without considering it an error, then the linker gives a warning message.

After constructing the search list the linker looks for each required symbol by searching through each DLL in the list, stopping at the first DLL that exports a symbol of that name. The address specified for the symbol in that DLL is the address that is used to resolve references to the symbol. Or, if the relocation type says that the address of the official function descriptor is required, the linker would require that the target symbol be a procedure and then would fill in the address of the official function descriptor rather than the address of the procedure itself.

To Preset or Not to Preset, and Creation of the LIC

LIC is the ‘Library Import Characterisation’, a section in the object file that encodes information about the current loadfile, plus each DLL or import library that was used to do the presetting, in the search list order, noting its export digest and whether it was used to resolve any references. The LIC is essentially a data string that characterizes the information used by a linker or loader to bind the global symbols of a particular loadfile. If the same loadfile is bound on two occasions, and its LIC has not changed, the two bindings are the same. Thus it is possible to reuse a set of bindings if it has the same LIC as that determined for this loadlfile in the presence of the other loadfiles with which it is being loaded.

By default, the linker tries to preset the loadfile that it is creating. The following are reasons why the linker will decide not to preset:

The linker will not preset if the `-no_preset` option is specified.

The linker will not preset if there is any address overlap among the DLLs or import libraries in the search list, including this file itself. If there are any overlaps then the linker emits a warning message. The linker always checks for this among the DLLs and import libraries that it has opened, even if it is not trying to preset. However, the linker does not check for overlap against the import library that represents the set of implicit libraries (because, it can be assumed that the implicit libraries have been correctly built in a separate place). Also, when some DLL has the same filename as the user library, and their memory segments overlap, the linker does not consider that a reason to stop presetting.

The linker will decide not to preset if it was unable to locate an item specified in a `-l` option in the command stream or in a *.liblist* section entry of a DLL.

The linker will decide not to preset if this is a program that has a user library, but the linker was unable to open the user library.

The linker will decide not to preset if it encountered any incomplete import libraries. The linker will also decide not to preset in certain cases if there are unresolved references, as explained in Handling Unresolved References on page 3-8.

The linker sets the *EF_TANDEM_PRESET* bit in the *e_flags* field of the ELF header of the loadfile to tell that it was preset.

When the linker presets a file it calculates the LIC and stores it in the *.lic* section. The LIC encodes information about this loadfile itself, plus each DLL or import library that was used to do the presetting, in the search list order, noting its export digest and whether it was used to resolve any references. Thus, the LIC begins with an entry for this loadfile itself, then has an entry for the user library if this is a program that has a user library, then has entries for each of the elements of the *.liblist* section, then has entries for each of the additional elements the linker found by pursuing the transitive closure of *.liblist* sections, and then has an entry for the import library that represents

the implicit libraries, if the linker found that library. If there was a DLL with the same filename as the user library, then that DLL gets no LIC entry.

Note that, for each entry in the *.lic*, there is a bit that tells whether any references were actually resolved by symbols exported from that DLL. The linker does not put out any warning messages about DLLs that were “not used”. It is actually quite difficult to decide which DLLs are “used”, because DLL A may indirectly bring in DLL B whose symbols are used, even if that wasn’t true of DLL A. And, even that doesn’t mean that DLL A was really “needed”, because there could also be other DLLs that similarly cause DLL B to get brought into the link. (For ideas on related linker features, see Ignoring Optional Libraries on page 3-14.)

The linker makes the *.lic* section larger than necessary so that it may be possible for the object file to be modified by the `-alf` option when the LIC has to increase in size.

Even when the linker is not presetting it still creates the *.lic* section in a loadfile. The section has no contents in this case, but it provides space for a subsequent `-alf` option to create a LIC.

The `-must_preset` option tells the linker that it must be able to preset, or else it is to be considered an error. `eld` reports an error if you give `-must_preset` with `-no_preset`. Otherwise, if the linker can’t preset, and the `-no_preset` option was not used, the linker generates a warning message.

Handling Unresolved References

This section discusses what the linker does when it is trying to bind a reference to a symbol and that symbol is not found in any loadfile in the linker’s search list.

If an undefined symbol is mentioned only in the symbol table, with no references to it from relocation tables, the linker continues processing. Checking for unresolved references is only concerned with symbols that have references from code or data.

The treatment of an unresolved reference depends on whether the target symbol was expected to be code or data. The linker knows this based on the symbol type listed for the symbol in the ELF symbol table in the linkfile that refers to it. `STT_OBJECT` means that it is data, and `STT_FUNC` means that it is code. No other possibilities are allowed.

The `-unres_symbols` option tells the linker how to treat unresolved references. It takes a parameter value that is one of the following:

- `error` - treat unresolved references as errors
- `warn` - put out warning messages about unresolved references
- `ignore` - be silent about unresolved references

`-error_unresolved` is supported as a synonym for `-unres_symbols error`, and `-warning_unresolved` as a synonym for `-unres_symbols warn`.

When `eld` is building a new loadfile and a public DLL registry has not been found the default for the `-unres_symbols` option is warn. On the other hand, if a public DLL

registry has been found then the default for the -unres_symbols option is error. In other words, eld only thinks it is unreasonable to have unresolved references if the public DLL registry has been found.

On the other hand, regardless of what the default would have been, and regardless of whether the -unres_symbols option actually was specified, the linker imposes -unres_symbols ignore in the following cases:

- If the -allow_missing_libs option has been specified and there were any missing DLLs.
- If the linker is building a program that has a user library and the linker has not been able to open the user library.

The linker does not check for unresolved references when the -r option is specified, and eld reports an error if you specify any of the options described above with -r.

Note that, even if the linker is not presetting, it will still try to resolve references for the purpose of producing messages about unresolved symbols.

If there are unresolved references to symbols that are not globalized, and this is not being considered an error, and the linker is presetting, then the linker must decide what to do with those references. If the reference is to a symbol that is expected to be code, the linker will look for a symbol named UNRESOLVED_PROCEDURE_CALLED_. The linker looks for this symbol by first looking for a symbol of this name that is exported by the loadfile being created, and then looks for the symbol in the usual way through DLLs. If found, and it is a procedure, then the references to the desired symbol are resolved to this symbol, and the file can still be preset. The references are still considered to be unresolved as far as putting out messages about unresolved symbols is concerned. If presetting is successful in this case, the LIC tells that there were unresolved references.

If unresolved references to symbols that are not globalized cannot be handled as described in the previous paragraph then the linker decides that the file cannot be preset. Therefore, whenever a file is preset, it means that all references to symbols that are not globalized were resolved to something.

The reason that unresolved references to code get more lenient treatment than unresolved references to data is that it is possible to handle them in a more predictable way at runtime. Specifically, the HP NonStop operating system provides an implementation of UNRESOLVED_PROCEDURE_CALLED_ that generates a SIGILL signal that can be caught but not deferred.

Whether or not the linker can preset is not affected by whether there are any unresolved references to globalized symbols. UNRESOLVED_PROCEDURE_CALLED_ is never used to handle unresolved references to globalized symbols. UNRESOLVED_PROCEDURE_CALLED_ is also not used to handle unresolved references to $n_EnterPrivN symbols from gateways.

There is also an option named -set rld_unresolved, with the same three parameter values as for -unres_symbols, and with error as the default. The linker places the value specified by this option into the RUNTIME_UNRES_CHECKING bits of

the *flags* field of the *.tandem_info* section. This tells `rld` how it should treat unresolved references at runtime. Even if this loadfile is preset, and even if `rld` can verify that the presetting is correct, `rld` will still repeat the presetting process in order to generate the error or warning messages requested by the `RUNTIME_UNRES_CHECKING` bits if the LIC says that there were unresolved references.

The bits set by `-set rld_unresolved` also provide defaults for how `-alf` treats unresolved references.

There is no option for specifying the treatment of unresolved references based on a symbol's name.

Using User Libraries

A user library is a DLL that is found in a special way by a program.

The `-set libname` option, given to the linker when it is building a program, tells the filename for the user library as it needs to be present at runtime. `eld` reports an error if you give this option when not building a program. The `-set` option is described The -set and -change Options on page 4-8.

The option `-libname` may be used as a synonym for `-set libname`. The specified user library name must always have the form *$a.b.*c, i.e., a valid Guardian file name, fully qualified up to the volume name, and not including the system name, because this is the only type of name that will work at runtime. Note that, when running the linker on OSS, one would need to do something such as putting the name in single quotation marks or preceding it with a backslash to avoid the special meaning of the dollar sign to the shell. The linker places the name specified by the `-set libnamee` option into the *.tandem_info* section of the program that it is creating. The linker also converts the name to upper case, if not done already. On Guardian APIs, single quotes do not work because they are not recognized by the HP Tandem Advanced Command Language (TACL); therefore, it is not important to specify them.

The `-local_libname` option tells the linker the filename of the user library to use at link time. Section 3, Binding of References explains how this user library is used, similarly to other DLLs, when the linker is fixing up references. It is not an error if the linker can't find the user library, but it does mean that the linker cannot preset the program that it is creating.

The locations of the `-set libname` and `-local_libname` options in the command stream are irrelevant. These options may only be used when building a program. Additional rules related to these options depend on the platform:

The PC hosted linker knows that it is building a program that uses a user library because the `-set libname` option is used. If the `-local_libname` option is not used then the linker can't find the user library. `eld` reports an error if you specify `-local_libname` without `-set libname`.

On Guardian, the linker knows that it is building a program that uses a user library because either `-set libname` or `-local_libname` is given. If only one of these is given, it provides a default for the other one. Specifically, if `-local_libname` is not given, the linker uses the parameter of `-set libname` for the value of `-local_libname`. Or, if `-set libname` is not given, the linker uses the name specified for `-local_libname`, but fully qualifying it by adding the volume and subvolume name if necessary, and omitting the system name. The volume and subvolume names used are the defaults that this instance of the linker would use for opening files with partially qualified names.

The rules on OSS are similar to those of Guardian. If `-local_libname` is not given, the linker converts the name specified in `-set libname` to the form */G/a/b/c* for use at link time (not converting it to upper case). Conversely, if `-set libname` is not given, the linker requires that the name specified for `-local_libname` be located in the Guardian name space, and then the fully qualified Guardian name of the file, not including the system name, is used for `-set libname`.

The linker is not required to open the `-local_libname` file unless it is either presetting the main program or checking for unresolved references. If it can't open the file, that is handled the same way as if `-allow_missing_libs` was specified and the linker couldn't find a DLL, including the production of a warning message.

Creating Import Libraries

There are three kinds of import libraries:

A complete import library may represent a single DLL, providing the linker all the same information at link time as if the DLL itself were present.

An incomplete import library similarly represents a DLL but with only some of the information that the linker needs at link time.

A special import library with the DLL name "__IMPLICIT_LIB__" represents the entire set of implicit libraries.

The linker can create an import library at the same time that it is creating the corresponding DLL, and it can also create an import library from one or more DLLs that already exist.

Whenever an import library is created, by default, it is a complete import library. If the `-set incomplete on` option is provided then the import library is incomplete. This doesn't necessarily change what is in the import library; it just marks it incomplete to indicate that the symbolic addresses within it are not to be considered reliable. The import library that represents the implicit libraries must always be complete (so it is an error to specify `-set incomplete on` in that case).

In all cases, the linker creates the import library file by following the same rules as given earlier in The Creation of Output Object Files on page 2-5, except that the name of the work file is ZLDAI*nnn* rather than ZLDAF*nn*n, the name of the intermediate file, if desired, is specified by the `-temp_i` option rather than the `-temp_o` option, and

there similarly is a `-must_use_iname` option instead of `-must_use_oname`. An existing file of the same name is not replaced if the linker terminates with any errors.

The `-change incomplete on` option can be used to demote an import library from complete to incomplete. It is not possible to demote the import library that represents the implicit libraries. It is not possible to “un-demote” an incomplete import library so that it becomes a complete import library.

The following subsections also tell how the linker decides whether to include DWARF symbol table information in an import library when it creates it. As described in Updating Or Stripping DWARF Symbol Table Information on page 4-14, the `-strip` option may be used to remove DWARF symbol table information from an existing loadfile or import library.

Creating an Import Library at the Same Time That a DLL is Created

The following option:

```
-import_lib <filename>
```

tells the linker to create an import library at the same time that it creates the corresponding DLL. If you use this option, and the DLL has DWARF symbols information, then so will the import library. If the DLL is being created without DWARF symbols information, through the use of the `-s` or `-x` options, then the import library will also not have DWARF symbols information.

The following option:

```
-import_lib_stripped <filename>
```

similarly tells the linker to create an import library, but in this case the import library does not contain DWARF symbol table information, even if the DLL does.

Creating Import Libraries From Existing DLLs

The following option:

```
-make_import_lib <filename>
```

tells the linker to make an import library out of one or more existing DLLs. The other filenames in the command stream are the DLLs that the import library will represent. `eld` reports an error if you specify `-make_import_lib` without specifying any DLLs in the command stream.

If all the DLLs have the `EF_TANDEM_IMPLICIT_LIB` bit set in their ELF headers, the linker creates the import library that represents the implicit libraries. In this case, the `EF_TANDEM_IMPLICIT_LIB` bit is set in the import library that the linker is creating, it is given the DLL name “__IMPLICIT_LIB__”, and `eld` reports an error if there is any overlap among the names of the symbols exported by the various input DLLs, unless all the copies of the symbol have the STO_MULTIPLE_DEF_OK bit set. It is also an error if any of the implicit DLL's is not preset. If any of the implicit DLL's is preset, but

has unresolved references, that is either an error, warning, or ignored, depending on what is specified for the `-unres_symbols` option, similar to how this option is used when building a new loadfile. If the `-unres_symbols` option was not specified, the default in this situation is error. The import library that represents the implicit DLL's never has DWARF symbol table information.

The address of the import library that represents the implicit DLL's is specified the same way as for the text segment of a DLL. In particular, this includes the use of a DLL registry. The name of the entry to use in the registry file would be the same as the filename specified in the `-make_import_lib` option.

Otherwise, not all the DLL's on the command line have the EF_TANDEM_IMPLICIT_LIB bit set in their ELF headers. In this case, it is required that there be only one DLL in the command stream, and the import library will represent it with the same DLL name. By default, the import library created will have the same DWARF symbols information as the existing DLL. However, if either the `-s` or `-x` option is given, the import library will not contain DWARF symbols information.

The following are the options allowed when creating an import library that represents a single DLL:

```
-must_use_iname
-no_banner
-no_verbose
-NS_extent_size
-NS_max_extents
-obey
-s
-set incomplete on
-stdin
-temp_i
-verbose
-vslisting

-warn
-x
```

The following are the options allowed when creating the import library that represents the implicit DLL's:

```
-check_registry
-must_use_iname
-must_use_rname
-no_banner
-no_verbose
-NS_extent_size
-NS_max_extents
-obey
-stdin
-t
```

```
-temp_i
-temp_r
-unres_symbols
-update_registry
-verbose
-vslising
-warn
```

Ignoring Optional Libraries

This section decribes the feature whereby `eld` can be told to omit certain DLL's from the liblist if they appear to be unnecessary. Note that `eld` always does this processing, regardless of whether it is presetting the file. `eld` does not fix up references to globalized symbols found in other DLL's, since the fixups would always need to be updated by `rld` anyway, but `eld` still looks up globalized symbols in DLL's in order to see if some DLL would resolve a reference to an undefined globalized symbol, for the purpose of deciding which DLL's are optional.

For every indirectly found DLL in the search list, there was one other DLL that was already in the search list and that caused this DLL to be added to the search list. If that other DLL was also an indirect DLL, then it similarly was pointed at by something else, etc. In this way, every indirectly found DLL can trace its presence in the search list back to one DLL that was explicitly given in the command stream, or to the user library. The set of all the DLL's that trace their presence back in this way to a given DLL A is called the search list addition set of A.

If a DLL that was named explicitly in the command stream was not actually used to resolve any references, and furthermore none of the DLL's in its search list addition set were used to resolve any references, then that DLL is called unnecessary, because the presence of that DLL in the command stream did not affect how the linker preset the file. For example, if that DLL had not been specified in the command stream, then it or the other DLL's in its search list addition set might instead have been found some other way, but in all cases they would have to come later in the search list then they actually did, or be absent entirely. That means that they wouldn't have had any consequences that they didn't have in the actual link: they wouldn't have resolved any references, and they wouldn't have caused other DLL's to be added to the search list. The resulting search list would still have been the same, at least with regard to the presence of the DLL's that were used to resolve references, and to the ordering of those DLL's among themselves.

The options named `-optional_lib` and `-no_optional_lib` are command stream toggles that determine whether the linker considers a DLL that is specified in the command stream to be optional. These options can be specified multiple times in the command stream, each time setting the mode for subsequent DLL's in the command stream until the mode is changed again. At the beginning of the command stream the mode is `-no_optional_lib`. If an optional DLL turns out to be unnecessary, then the linker will assume it is okay to ignore that DLL, and it will not be listed in the .liblist section. If `eld` is presetting, and therefore creating a LIC, then the LIC is created as if

the omitted .liblist entries had never been present. For example, some of the DLL's in the search list addition set of such a .liblist entry may still be present in the LIC, and some not, and if present their positions in the LIC may change, depending on other ways of getting to those DLL's indirectly.

When some elements of the .liblist section have been omitted, with corresponding changes to the LIC if present, we say that the .liblist and LIC have been abbreviated.

Note that `eld` always does this `-optional_lib` processing. It doesn't depend, for example, on whether `eld` is presetting the loadfile.

If all the same direct and indirect DLL's as were found by the linker would also have been found at runtime regardless of whether the .liblist was abbreviated, as described above, and all these DLL's would have been found in the same order and with the same export digests, then the linker's presetting would give the same results as what would happen at runtime with no such abbreviation. In this case, the -optional_lib option is just an optimization, preventing some DLL's from being loaded into a process when those DLL's would not have affected how the loadfile would be fixed up when it was loaded into memory. The abbreviated LIC, if present, corresponding to the abbreviated .liblist, will similarly lead to the conclusion that the linker fixed up all the reference sites correctly, so the file can run without being rebound at load time (except, of course, that references to globalized symbols always need to be rebound).

In other cases, the use of the `-optional_lib` option is not just an optimization, but rather can cause the loadfile to run differently. These are the cases where the DLL's that were omitted from the abbreviated .liblist or abbreviated LIC would have been found at runtime, and would have had different export digests from the ones that the linker saw, or would have been found in a different order, or would have indirectly brought in other DLL's that the linker didn't see. In such cases, the abbreviated .liblist and abbreviated LIC might lead to the conclusion that the file was fixed up correctly by the linker, and allowed to run without being rebound, even though if it were rebound at load time based on a .liblist that had not been abbreviated then some of the bindings might have been different. Or, if the abbreviated .liblist and abbreviated LIC don't lead to the conclusion that the file had been correctly fixed up by the linker, the file is rebound at load time, but the use of the abbreviated .liblist could still lead to different bindings from what would have occurred with a .liblist that was not abbreviated.

The use of the `-optional_lib` option can also affect how things are fixed up at load time with regard to globalized symbols, because `eld` does not take globalized symbols into account when it decides which DLL's are unnecessary.

Also, even if the abbreviated .liblist and LIC don't affect how the loadfile itself is fixed up at load time, the abbreviations will result in different DLL's being present in memory, or being loaded in a different order, in a way that could cause the dlsym runtime routine to give different results.

In any case, the use of the `-optional_lib` option can affect a program's semantics, so it should not be used unless the consequences are understood.

A special case is when `eld` finds a DLL on the command line that is the same file as what was specified for the user library. In such a case, an entry is still made in the

.liblist section for such a DLL. With regard to the `-optional_lib` option, this option ever causes such an entry to be removed from the .liblist. The point is that such an entry is only significant if a different user library were to be used at runtime, and `eld` does not try to analyze whether this .liblist entry would be "necessary" if a different user library had been present.

The main intended use of these options is for `-optional_lib` to be placed before the set of DLL names that a compiler may automatically place at the end of the command stream that it sends to the linker, for a set of DLL's that it thinks the user might generally need. Such DLL's would often be unnecessary, depending on which language features the program used. Since these DLL's are at the end of the command stream, they would come after all other DLL's in resolving references, except for the implicit DLL's. Therefore, this use of `-optional_lib` would have no runtime consequences, unless the user wanted to be able to get a different version of one of these libraries at runtime, with additional symbols in it that hid symbols otherwise found in the implicit libraries.

Note that this is not necessarily the most general way to define "unnecessary", nor is there necessarily any one best way to do it. For example, suppose that DLL's A and B both point at C, and C is used to resolve references, but neither A nor B themselves resolve any references. You could argue that either A or B could be considered unnecessary on its own, but you can't say that they are both unnecessary at the same time without figuring out how you are going to get to C. The definition of "unnecessary" given in this section can be applied separately to each DLL in the command stream, without having to take into account such dependencies. As it actually would work in this case, the linker could only ignore A or B if it was not the one that led to putting C into the search list. For instance, suppose A came before B in the .liblist section of the program, so that A caused C to be in the search list. Then, the linker would ignore B if the user marked B optional, but there would be no way to tell the linker to ignore A (i.e., without changing other things in the linker's command stream, such as the ordering of A and B).

Merging Symbols Found in Input Linkfiles

TNS/E linkfiles contain ELF symbol tables. The linker merges the symbol information from the input files into its output file, creating an ELF symbol table if the output file is a linkfile, or creating the *.dynsym* and *.dynsym.gblzd* sections if the output file is a loadfile.

In loadfiles, globalized symbols are placed into the *.dynsym.gblzd* section, while all other symbols are placed into the *.dynsym* section.

If the `-make_implicit_lib` option is given then `eld` reports an error if you have any globalized symbols. Also, if the `-instance_data` option is specified with a parameter value of `data2protected` or `data2hidden` then `eld` reports an error if you have any globalized symbols.

Note. The `data2protected` parameter is supported on systems running J06.09 or earlier J-series RVUs and H06.20 or earlier H-series RVUs.

In linkfiles, entries in the ELF symbol table tell the source language for each symbol.

When there are multiple entries of the same name that have been defined and allocated by the compiler or assembler, the rules followed by the linker in deciding which copy to keep are found inAccepting Multiply-Defined Symbols on page 3-17.

If there is only one entry that has been defined and allocated by the compiler or assembler, the linker keeps that one. If there are no such entries, but there is an entry for the symbol as common data, the linker will choose such an entry and allocate the symbol in the *.bss* section. If there are multiple common data entries for a symbol, the linker chooses the first one among those with the largest size. Otherwise all the entries are just external references, and the symbol remains an external reference in the output file, with the linker keeping the first one it sees.

The `-y` option tells the linker to print out information about a symbol of a given name, telling the names of the linkfiles that mentioned the symbol in their ELF symbol tables and giving the information provided about the symbol in each of those files.

Object files also contain DWARF symbol table information for the use of debuggers. See Updating Or Stripping DWARF Symbol Table Information on page 4-14.

Accepting Multiply-Defined Symbols

This section is only concerned with symbols that are defined and allocated by the compiler or assembler, not with symbols that are external references or common data. This section covers global definitions, which make it possible to have multiple definitions of the same symbol. Multiple definitions are only allowed if all the definitions are as data items, or if all the definitions are as procedures, with different rules for the two cases as described below.

Rules For Data Items

Here are the rules concerning multiple definitions of a data item:

- A symbol of a given name that occurs in one the sections named *.data*, *.sdata*, *.bss*, and *.sbss*, is allowed to come up in different sections among these four possibilities in different linkfiles. Similarly, a symbol of a given name that occurs in one of the sections named *.rdata* and *.srdata* is allowed to come up in different setions among these two possibilities in different linkfiles. A symbol that occurs in any other data section must only occur in that same named data section in all the linker's input linkfiles.
- `eld` reports an error if an initialized data item (including "zero-initialized" data items found in the .bss or .sbss sections, which in the case of C++ also includes data that was not explicitly initialized in the source) is defined in more than one C or C++ file unless the *STO_MULTIPLE_DEF_OK* bit is set in all the corresponding ELF symbol table entries. HP's definitions of C and C++ say that it is an error for users to create such situations, but the *STO_MULTIPLE_DEF_OK* bit allows the compiler to do it.

- The linker uses the *st_size* field in the ELF symbol tables of its input files to understand the sizes of data items. When there are multiple definitions of a data item `eld` reports an error if the sizes are not the same.

  It is not an error if some of the definitions have initial values (i.e., in the *.data* or *.sdata* sections) and others don't (i.e., in the *.bss* or *.sbss* sections). However, `eld` reports an error if the initial values are not the same in all the copies of the data that are initialized. Data in the .bss and .sbss sections is considered to have the initial value zero.

When there are multiple definitions of a data item, and it is not an error, the linker must choose which copy to use. The linker makes this choice by comparing copies in the following ways:

- Choose an instance that is initialized over one that isn't (i.e., choose *.data* or *.sdata* over *.bss* or *.sbss*).
- If that doesn't narrow it down to one choice, choose an instance whose language is C or C++ over one that isn't. Doing this, after the previous rule, is convenient for enforcing the rules given above about initialized definitions in C and C++.
- If that doesn't narrow it down to one choice, choose the first instance encountered.

Note that the linker does not care whether various definitions are short data (namely, the `.sdata`, `.srdata`, and `.sbss` sections) versus long data (namely, the `.data`, `.rdata`, and `.bss` sections). If the above rules cause the linker to choose a copy of a symbol that cannot be reached by 22-bit GP-relative addressing, and 22-bit GP-relative addressing is used for the symbol, it is an error.

Rules for Procedures

Here are the rules concerning multiple definitions of a procedure:

- It is an error unless either the STO_MULTIPLE_DEF_OK bit is set in all the corresponding ELF symbol table entries, or the -allow_duplicate_procs option is used.
- It is an error if any copy of the procedure has the CALLABLE or KERNEL_CALLABLE attribute.
- It is an error if the various copies do not all agree on the MAIN, SHELL, EXTENSIBLE, and COMPILED_NONSTOP attributes.

There is no requirement that multiple definitions of the same procedure contain the same code or have the same size.

When there are multiple definitions of a procedure, and it is not an error, the linker must choose which copy to use.

If two copies of the procedure agree on the *RESIDENT* attributes, the linker prefers the first one it sees.

The linker prefers one that is *RESIDENT* over one that is not *RESIDENT*. Note that the linker doesn't know why someone thought the procedure needed to be *RESIDENT*, but as long as at least one copy said so, the linker assumes it is necessary.

In all cases, after the linker chooses which copy to use, that determines the contents of the code for the procedure.

The above rules about duplicate symbols apply both when the linker is creating a loadfile as well as when the linker is creating a linkfile with the `-r` option.

When there are multiple definitions of a symbol, the one chosen by the linker is the only one still represented in the ELF symbol table of the output file (if it is a linkfile), or in the *.dynsym* or *.dynsym.gblzd* section of the output file (if it is a loadfile).
All references to a symbol of this name get fixed up to this copy of the symbol.

Note that the linker never uses DWARF information to decide if multiple definitions are allowed, or to decide which definition to keep. (The linker never uses DWARF information to decide things outside of DWARF itself, because the DWARF information can be stripped from linkfiles.) When the linker fills in addresses of symbols in the DWARF information, it fills in -1 for the address of an unused copy of a symbol. The linker does this for both linkfiles and loadfiles.

The information about unused copies of procedures is deleted from the *.procinfo* section of a linkfile, and from the stack unwinding information of a linkfile or loadfile.

The *-show_multiple_defs* option tells the linker to print out information about global symbols that are multiply defined. The result is similar to giving a *-y* option about each such symbol, except that it isn't necessary to know ahead of time which symbols they are, and the information only comes out about the linkfiles that defined the symbol, not about other linkfiles that only mentioned it as an external reference.

Using the `-cross_dll_cleanup` option

The `-cross_dll_cleanup` option is used to reduce the total size of a program and the private DLLs that are used by a process. This option affects the behavior of the `eld` program with regard to symbols `eld` finds in its input linkfiles that have all the following properties:

- The symbol is a global symbol.
- The symbol is a definition (not an external reference).
- The symbol is a procedure.
- The symbol is marked STO_MULTIPLE_DEF_OK (implying that it is a multidef or globalized symbol).

Without the `-cross_dll_cleanup` option, you can store multiple copies of a symbol that have the above properties, and eld discards the code for all except one copy of

the symbol. When the `-cross_dll_cleanup` option is specified, eld also verifies if a symbol of the same name is found in a DLL. If it finds a same name, eld deems the last copy of the procedure as "unused", and its code can be cleaned up. The references to the symbol are therefore resolved to the copy of the symbol in the DLL.

When the `-cross_dll_cleanup` option is used, it is necessary that all the input linkfiles be compiled with the `-Wglobalized` option (the same as when eld is specified to build a globalized DLL, since this new option is a form of preemption).

To get maximum benefit from the `-cross_dll_cleanup` option, list the DLLs that are directly used (on the eld command line), and also DLLs that previously were only indirectly used.

Specifying Which Symbols to Export, and Creating the Export Digest

When creating a loadfile the linker must determine which symbols to export. If no options related to this are given in the linker's command stream then its default behavior is that it exports a symbol if it is a global symbol and the *STO_EXPORT* bit is set in some ELF symbol table entry for the symbol in the linker's input files. This is the way the compiler or assembler tells the linker that certain symbols should be exported. If the *STO_EXPORT* bit is set in any ELF symbol table entry for a symbol in the linker's input linkfiles, and the linker is creating a linkfile, then this bit is set in the linker's output file.

If the *-export_all* option is used then the linker exports all defined, global symbols except for those in the following categories:

- Procedures whose names begin with the special prefixes that mark them for inclusion in the *ctor*s, *dtor*s, *init*z, or *termz* arrays.
- Symbols that are specially known to the linker and have names that are intended to only be meaningful within the loadfile being created. This includes all the predefined symbols created by the linker, such as the symbols that are the names of the *ctor*s, *dtor*s, *init*z, and *termz* arrays. However, *-export_all* does cause *_MCB* to be exported, if the linker has created that symbol.

The above rules can be overridden for specific symbols by name. If the -exported_symbol option is specified then the symbol of the specified name is unconditionally exported. If the -hidden_symbol option is specified then the symbol of the specified name is unconditionally not exported. It is an error if the same symbol is specified in both of these options. It is an error if the symbol specified in the -exported_symbol option is not a global symbol that is defined in the loadfile being created. It is a warning rather than an error if such a symbol is specified in the -hidden_symbol option.

Note that globalized symbols are not treated specially with regard to these rules. That means they are exported by default because they have the STO_EXPORT bit set, but you can avoid exporting them by using the *-hidden_symbol* option.

The option *-export* is accepted as a synonym for *-exported_symbo*l, and *-export_not* as a synonym for *-hidden_symbo*l.

Exported symbols are identified in the *.dynsym* and *.dynsym.gblzd* sections by the fact that they are marked *STB_GLOBAL* and not *SHN_UNDE*F. If a symbol that was *STB_GLOBAL* in its input linkfile is not exported, that fact is indicated by marking it as *STB_LOCAL* rather than *STB_GLOBAL* in the output loadfile.

The linker calculates the export digest based on the names and addresses of exported symbols in the *.dynsym* section, and on the GP value for the loadfile, and stores it in the *export_digest* field of the *.tandem_info* section. Globalized symbols are not included in the calculation of the export digest.

The options described in this section are not allowed if *-r* is specified.

If you are starting with code that had been a single program before, perhaps built from archives, and now you are changing it to be split up among several DLLs, you might find it useful to build the program and DLLs with the *-b globalized* and *-export_all* options, to assure that all symbolic references among the program and DLLs keep referring to a single copy of each symbol.

Processing of Code and Data Sections

A *text section* contains procedures. A *data section* contains data that is allocated at a certain address in virtual memory when the loadfile is brought into memory, as opposed to data that is dynamically allocated on the stack or in the heap.

The linker checks that the sizes of all code and data sections are multiples of 16 bytes.

Concatenating Code and Data Sections Found in the Input Linkfiles

A linkfile may have many text sections, with names beginning either *.text* or *.restex*t. When the linker is creating a new linkfile with the *-r* option, it concatenates text sections of the same names in its input files to create text sections of those same names in its output file. When section names are the same in different input linkfiles, the linker concatenates them in the same order as the linker saw those linkfiles.

However, when the linker creates a loadfile, it combines all the input sections whose names begin *.text* into a single output text section named *.tex*t, and similarly for *.restex*t. No guarantee is given as to the order in which different text sections whose names begin *.text* are combined into a single text section named *.tex*t, and similarly for *.restex*t.

With regard to data sections, again, the linker usually concatenates data sections of the same names in its input files into data sections of the same names in its output file, and the linker does this whether creating a linkfile or a loadfile. Except, when an input linkfile has a section named *.rdat*a, and it contains no relocation sites, the linker changes the name of that input section to *.rcons*t, and combines all sections named *.rconst* into a section of that name in its output file accordingly.

When creating a linkfile, the linker similarly concatenates the relocation tables that accompany the code and data sections.

The linker's input linkfiles may also contain *common data*. Such data items are defined, but not assigned addresses within sections. If a normal definition of the same symbol is found, the common data definitions are ignored. If there is no normal definition, and the linker is creating a loadfile, the linker converts the common data symbol into a data symbol that is allocated space in the *.bss* section. Common data cannot have initial values.

If you are creating an implicit DLL (with the *-make_implicit_lib* option) `eld` reports an error if you have any data that would need to go into the data variable segment (i.e., *-instance_data data1constant* is imposed).

Public Libraries and DLLs

TNS/E supports public libraries. Public libraries are a set of (DLL) libraries, available to all users of the system, and managed as part of the system software. They are mostly supplied by HP, although you and third party software providers can also provide DLLs to be added to the public DLLs. You use DSM/SCM to add your DLLs to the public libraries. Note that these must be loadfiles, not linkfiles.

Public libraries include:

- TNS/E compiler run-time libraries.
- Libraries that support connections to TNS/E communication facilities.
- Certain TNS/E tools, utilities, and the loader library (`rld`).

TNS/E compilers generate needed linkages from PIC programs and DLLs to the compilers' run-time libraries.

In addition to accessing public libraries, PIC programs and DLLs will automatically access the system and millicode libraries, without your specifying this linkage requirement. The system and millicode libraries are PIC libraries that the system loads before loading any application code, and the loader and operating system automatically link your application to these libraries as appropriate. These are known as implicit libraries because every loadfile is implicitly a user of them.

This can be contrasted with the public DLLs, which are explicit because a loadfile must explicitly ask to use a public DLL, although you need not specify where to find the public DLL. The combination of the ZREG file (the Public LIbrary Registry file) and ZREGPTR (pointer) file specifies the location.

One main user of the ZREG file (and the ZREGPTR) is the preloader. Public DLLs are preloaded during coldload, reload of a CPU or when a set of public DLLs is replaced online. The other main user is the linker (`eld`). Typically, `eld` "finds" the DLLs by finding the ZREG file that is in the same subdirectory, then searches the registry. The linker does not use the ZREGPTR pointer directly, but acquires its information from the preloader by use of a procedure call.

Each set of public libraries is installed in a separate subvolume, separate from the SYSnn subvolume and separate from any other set of public DLLs. This subvolume is on the same disk as the SYSnn subvolume.

The SYSnn subvolume also contains the imp-imp file, named zimpimp. This is the import file usable for resolving external references to the explicit libraries. The imp-imp file can be copied to the public-DLL subvolume. This renders the public-DLL subvolume portable. A portable public-DLL subvolume contains everything the linker needs to link files to use these particular public libraries. A portable subvolume can be copied for use by the linker (`eld`) on another system or another platform, such as a PC.

The Public Library Registry

The public-DLL registry file (ZREG) serves as an interface between DSM/SCM (that you use), the public-library installation tool (that HP uses), the preloader and the linker.

DSM/SCM creates an initial registry file, listing all the public DLLs by name. This is an edit file (filecode 101). Use DSM/SCM to add your public DLLs to those provided by HP.

Entries to the file consist of a series of statements. The dll statement describes a public DLL. In its simplest form, it is just a name, for example:

```
dll file ztestdll;
```

Here is another example; it contains the license attribute. A licensed DLL is one that contains priviliged code. Unless you use this attribute along with the value “1”, the default is “0”, which means the DLL is unlicensed.

```
dll license 1, file privdll;
```

There are other attributes which are created automatically, for example the timestamps that you and the tools can use for version control. Here are two examples, the link_timestamp (from when the linker first created the DLL), and the update_timestamp (from when the linker last updated the DLL, or when another tool rebases or presets it):

```
dll file zredll,
link_timestamp    2004-08-01 16:34:41.213592,
update_timestamp 2004-08-01 17:15:17.119634;
```

From these examples you can see that attributes can be in any order, attributes are separated by commas, and statements are terminated by semicolons.

Finding and Reading The Public DLL Registry (ZREG) File

The linker always tries to find the public DLL registry file whenever it is creating a loadfile. There are three ways that the linker may find the public DLL registry file:

1. If the *-public_registry* option is specified, that tells the name of the public DLL registry file. (This option could also be used to override another location of the zreg file.) If the file does not exist, or the linker cannot open the specified file for reading, `eld` reports an error.

2. If the *-public_registry* option is not specified then the linker looks for a file named *zreg* in its own directory or subvolume. If this location is a \bin or /bin, `eld` will also look in a corresponding \lib or /lib location. This supports the practice, on PCs and OSS, of putting `eld` in a bin directory and the public DLLs (together with the zreg file) in a sibling lib directory.

   On OSS, where case is significant, *zreg* is lower case. If the file exists and the linker can open it for reading, this file is deemed to be the public registry file. Otherwise, if the *-nostdlib* option has not been specified, the linker writes out a warning message.

3. If still not found, and the host platform is TNS/E, the linker uses a system call (*pubLibSpecs_get_)* to find the location of the zreg file.

If the linker finds the public DLL registry, it then determines a list of public DLL filenames. This information is used elsewhere in the linker for two purposes.

First, it is used to locate public DLLs, as discussed in Finding Public DLLs on page 2-18. Second, it is used to locate the import library that represents the implicit libraries, as discussed in Presetting Loadfiles on page 3-5.

As explained in those two sections, a filename found within the public DLL registry file may be used by the linker to search for a file on a platform where filenames are case sensitive. In that case, the linker will interpret what it finds within the public DLL registry to be lower case. So, to avoid confusion, it is strongly recommended that all filenames mentioned in the public DLL registry be written in lower case, and that when corresponding files exist on a platform where case is significant those files are given lower case filenames, and furthermore that the DLL names found within such files are also written in lower case.

4 Other eld Processing

This section contains the following information:

Adjusting Loadfiles: The -alf Option - how to repeat the presetting of a loadfile when DLLs change.

The -set and -change Options - how to set various options within the loadfile.

eld Functionality for 64-Bit - how the linker performs consistency checks.

Renaming Symbols - how the linker treats each input file.

Updating Or Stripping DWARF Symbol Table Information - from the input and output object files.

Modifying the Data Sections that Contain Stack Unwinding Information - when concatenating sections to create a new loadfile.

Creating the MCB - the Master Control Block contains key settings such as product version numbers, valid file types, language dialects, and so on.

Processing of Floating Point Versions and Data Models - more consistency checks.

Specification of the Main Entry Point - there are two ways to specify the main entry point.

Specifying Runtime Search Path Information for DLLs - `eld` tells `rld` where to find the DLLs.

Merging Source RTDUs - used with SQL/MP.

Adjusting Loadfiles: The -alf Option

The main purpose of this option is to tell the linker to repeat the process of presetting an existing loadfile. Most likely, this is done because some of the DLLs used by this loadfile have changed, and therefore resolving the references against the newer set of DLLs will allow the loadfile to load more quickly. The updating of the loadfile might happen automatically at runtime, but this option provides a way of making sure the file is updated by user request. It would also be possible to relink the file from its constituent pieces, but `-alf` makes it possible to update the references even if those pieces are no longer present and without knowing everything about how the original link was done.

It is also possible that the loadfile had not been preset before, in which case this option would be presetting it for the first time.

It is also possible to rebase a DLL while rebinding it. And, when a DLL is rebased by `-alf`, it is also possible for the text and data segments to be rebased by different amounts. If there are two data segments, they are always rebased by the same amount. The gateway segment of a DLL is rebased together with the data segment.

`-alf` stands for “Adjust LoadFile”.

`-alf` recreates the file by the usual linker mechanisms.

Certain relocation sites within a file must be set up by the linker, whether the file is preset or not, and must never become inconsistent thereafter. That is because rebinding makes use of the existing values at those sites. Such sites only need modification if a file is being rebased. Because updating in place would run the risk of terminating unexpectedly in the middle, leaving such sites inconsistent, we never update in place when rebasing is being done. This is why `-alf` always recreates the file, rather than updating it in place.

The `-alf` option can be applied to a DLL whose segments are not contiguous, i.e., when the DLL was created by using the `-d` option to place its data separate from its code.

The parameter to the `-alf` option is the name of an existing program or DLL. `eld` reports an error if this file does not exist, or if it is any other kind of file. This option performs the same version number checking on this loadfile as is described in Input Object Files on page 2-12.

When the `-alf` option is used, the names of input linkfiles cannot be specified. All the code, data, symbols information, etc., comes from the existing loadfile. Archives and DLLs cannot be specified, either directly in the command stream or with `-l` options. The linker uses the DLL names found in the *.liblist* section of the existing loadfile as if they were specified in the command stream with `-l` options.

When the `-alf` option is used, it is possible to put the name of the import library that represents the implicit libraries directly on the command line. This is an exception to the rule stated in the previous paragraph, that no filenames can be on the command line. On the other hand, it keeps the processing of the `-alf` option consistent with usual linker processing with regard to the ability to get the name of this import library from the command line. The filename specified on the command line overrides any other way the `-alf` option may have found the import library that represents the implicit libraries. `eld` reports an error if you have more than one filename on the command line with `-alf` . The `-alf` option always reset the *goldsmith_region_info* and *goldsmith_region_addr* fields of the *.tandem_info* section to zero.

The `-alf` option updates the update_timestamp if there were any other changes made to the file. It does not update the update_timestamp if that would be the only change to the file. It never updates the creation_timestamp. An informational message tells whether the file required any changes.

The `-alf` option cannot create an import library in parallel with updating a loadfile, nor can they be used to update an import library. When the `-alf` option is used to rebase a DLL, many things in the DLL change in ways that affect users of that DLL, such as the addresses of its exported symbols. If you had an import library that matched the previous version of that DLL, you would therefore probably want to run the linker with the `-make_import_lib` command on the new DLL afterward in order to create a new import library that matched it. Do not worry about this when using the `-alf` option just to rebind a DLL, because in that case the only change to the DLL that would be reflected in a corresponding import library is whether the ELF file header

said that the DLL was preset, and that bit isn't really of any importance in an import library.

The following are some things that cannot be specified with this option because they are unconditionally inherited from the existing version of the loadfile:

- It is not possible to specify import controls.
- It is not possible to say which symbols are exported.
- If it is a DLL, it is not possible to say what its DLL name is.

Only one `-alf` option can be specified in the linker command stream.

Additional rules about the `-alf` option is given in the following subsection.

Additional rules about -alf

In addition to updating the references within an existing DLL or program to other DLLs, the `-alf` option may be used to rebase an existing DLL. It is not possible to rebase a program.

To rebase a DLL with `-alf`, specify the `-t` option to tell the new starting address for the text segment. The data segment is moved by the same amount that the text segment is moved.

With the `-alf` option it is permissible to specify `-no_preset`. It is also possible for the `-alf` option to discover that it can't preset, for the same reasons as described in To Preset or Not to Preset, and Creation of the LIC on page 3-7, and the `-alf` option acts similarly in that case to what the linker does when it is creating a new loadfile and discovers that presetting is impossible.

The following are differences between the way the `-alf` option makes this decision, and the way it is described in To Preset or Not to Preset, and Creation of the LIC on page 3-7.

An additional reason that the `-alf` option cannot preset is that the new LIC would be too large to fit into the size that the *.lic* section has in the existing loadfile.

Even if the `-alf` option is neither rebasing nor presetting, it may still produce messages about unresolved symbols and it would turn off the `EF_TANDEM_PRESET` bit if it previously was on.

The way the `-alf` option looks for DLLs or import libraries is the same as the way the linker verifies for them when a file is originally linked the search path for `-alf` is the following:

- The places specified in `-first_L` options.
- The places specified by the `DT_TANDEM_RPATH_FIRST` entry of the *.dynamic* section.
- The public libraries.

- The location of the existing loadfile.
- The places specified in `-L` options.
- The places specified by the `DT_RPATH` entry of the *.dynamic* section.
- On OSS, if the existing file is a 64-bit data model: `/lib64`, `/usr/lib64`, and `/usr/local/lib64`.
- On OSS, `/lib`, `/usr/lib`, and `/usr/local/lib`.
- On Guardian or OSS, if the existing file is a 64-bit data model: `$SYSTEM.YDLL`.
- On Guardian or OSS: `$SYSTEM.ZDLL`.

The `-nostdlib` option stops the `-alf` option from looking for public libraries or in the standard places (i.e., the last two items in the above list), the same as when this option is used when the linker is creating a new loadfile.

The search path makes use of the information in `DT_TANDEM_RPATH_FIRST` and `DT_RPATH` entries of the *.dynamic* section so that its search path is similar to that of `rld`, and so that the user need not specify any `-L` or `-first_L` options. However, this can only be expected to work when `-alf` is used in the same environment as where the program will run, which means not only that it is also the HP NonStop operating system platform, but that it is the proper version of the operating system, etc. The `-L` and `-first_L` options are provided so that the `-alf` option can be used when the places specified by entries in the *.dynamic* section would not work.

It is also possible to specify the `-rld_L` and `-rld_first_L` options with `-alf`. As with the linker's normal processing of these options, you can specify them multiple times, and the linker concatenates the strings given. Unlike the linker's normal processing of these options, the `-alf` option does not concatenate these strings with the ones found in the *dt_rpath* and *dt_tandem_rpath_first* entries of the *.dynamic* sections of other DLLs that it sees. Instead, if values are specified for `-rld_L` or `-rld_first_L` with `-alf`, that is used instead of the strings found in the *dt_rpath* and *dt_tandem_rpath_first* entries of the loadfile being updated by the `-alf` option, when the `-alf` option is looking for DLLs. The contents of the *dt_rpath* and *dt_tandem_rpath_first* entries of the *.dynamic* section of the loadfile being updated by the `-alf` option are not updated by these options. The only way to update these entries within the loadfile is to relink it from scratch.

Note that the linker still looks in all the places specified above, regardless of the platform. For example, even though it may not accomplish anything, the linker does look in the places specified by the *dt_rpath* and *dt_tandem_rpath_first* entries of the *.dynamic* section, even when running on the PC platform. Colons are still used to separate names in these entries, even on the PC platform.

If the *limit_runtime_paths* bit is set in the flags field of the *.tandem_info* section of the loadfile then the `-alf` option does not look in the location of the existing loadfile.

Searches performed by the `-alf` option must always find DLLs rather than archives, as if `-b dllsonly` had been specified. It is not possible to specify any `-b` options with `-alf`.

The `-alf` option supports the `-allow_missing_libs` option. If a DLL that is listed in the *.liblist* section cannot be found, the *.liblist* entry is left alone.

When the `-alf` option is operating on a program, it uses the user library the same as when the linker created the program. By default, the library name stored within the file is used for the filename of the user library. Or, the `-local_libname` option can be specified with `-alf`, to override the name found within the file (this does not change the filename that is stored in the file). The `-set libname` option cannot be specified with `-alf`.

Note that the `-alf` option uses whatever user library name is currently within the file, or specified by the `-local_libname` option. It isn't necessarily the same as the one that was there when the file was previously preset.

The `-change libname` option cannot be specified with `-alf`. If desired, one linker invocation could be used to change the name stored within the program, by specifying `-change libname`, and then a separate linker invocation could give the `-alf` option.

If a program specifies the name of a user library, and the `-local_libname` option is not used, then on the PC this always means that the linker can't find the user library. If the linker doesn't have a name for the user library, or, on any platform, if it can't open such a file, it is treated like a missing DLL, the same as when a program is created. If the `-local_libname` option is used, and the program doesn't mention any user library name, that is an error.

By default, the `-alf` option considers it an error if the loadfile has any relocation sites in the text segment, which would indicate non-PIC code. However, if the `-update_code` option is specified, the `-alf` option does not consider this an error, and it updates those relocation sites. The only relocation sites updated in the text segment are 64-bit addresses of symbols found in the same DLL, which need to be updated what that DLL is rebased, and is an error if the `-update_code` option is specified when the `-alf` option is updating a program rather than a DLL.

By default, the way the `-alf` option treats unresolved references depends on the value of the *runtime_unres_checking* bits of the *.tandem_info* section of the loadfile. Note that, unlike the case of creating a loadfile, the default does not depend on whether the public DLL registry was found. This can be overridden by specifying the `-unres_symbols` option with the desired parameter value. When `-unres_symbols` option is specified with `-alf`, the *runtime_unres_checking* bits of the *.tandem_info* section are unchanged. The `-change` option can be used (in a separate run of the linker) to change those bits within the file.

The meanings of the three choices for how unresolved references are handled is the same for the `-alf` option as when the object file is first linked, except that the decision as to whether an unresolved symbol is expected to be code or data is made

by looking at the dynamic symbol table of the existing loadfile rather than the ELF symbol table of an input linkfile.

The following parts of the existing object file potentially need to change when the `-alf` option is used without rebasing:

- In the ELF header, the `EF_TANDEM_PRESET` bit may be changed.
- In the *.tandem_info* section, the *update_timestamp* field may be updated. The *creation_timestamp* and *tim_dat* fields are not updated. Some other fields may be reset to zero, as mentioned earlier.
- The LIC is updated. It may end up with more or less entries than before. This includes the case that the LIC now is empty, or no longer is empty, because the file now can or cannot be preset. But the *.lic* section always has the same size as before. As mentioned earlier, if this size would not be big enough to contain the LIC, then the file is not preset, so there is no LIC to worry about.
- Throughout the data segment, all the references to undefined symbols are updated as necessary.

The following additional parts of the existing object file also potentially need to change when the `-alf` option is used with rebasing:

- In the ELF header, the address of the entry point (the `e_entry` field) needs to be updated, if this is a DLL that has an entry point. It is updated to the new (rebased) address of the same procedure that was the main entry point before. The `-e` option is not allowed with `-alf`.
- In the program headers, the *p_vaddr* fields are updated.
- In the *.tandem_info* section, the *gp_valu*e, the *export_diges*t, and the four fields to tell the addresses of the *ctors* array, etc., are updated, if they are nonzero. In the *.dynamic* section, all the entries that tell addresses of sections of the loadfile, and the entry that tells the GP valu*e,* are updated.
- In the *.dynsym* and *.dynsym.gblzd* sections, the addresses of all the defined symbols are updated. If they are exported procedures, their *st_size* fields (telling the addresses of their official function descriptors) are similarly updated.
- In the *.rela.dyn* and *.rela.gblzd* sections, the addresses of all the relocation sites are updated.
- Throughout the data segment, all the references to symbols defined within the same loadfile are updated. This includes references that are explicitly identified by relocation table entries as well as things that `-alf` finds in other ways, such as the contents of the *_ctor*s, etc. arrays and the official function descriptors.
- In the ELF section headers, the *sh_addr* fields are updated, if they are nonzero.

The following additional part of the existing object file would potentially need to be updated, but only in the case when `-alf` was rebasing the code and data segments by different amounts:

- GP-relative addressing from the gateways to the text segment, specifically, when the gateway contains the 64-bit GP-relative address of the procedure for which it is the gateway.

The way that the `-alf` option updates all the items listed above depends on the type of item:

- If the target of the reference is identified by name, `-alf` updates the reference based on that name.
- If the reference site is one that is filled in by the linker, and `-alf` only knows that it needs to be updated by the amount that this DLL was rebased, `-alf` determines which segment previously contained the target of the reference and then updates the reference site by the amount that that segment moved.

`eld` reports an error if the existing file cannot be opened for reading. It is possible to specify the *-o* option to tell the name of the new output file. If the `-o option` is not specified then it defaults to the same name as the existing loadfile. Even if the existing file is a DLL, its DLL name does not influence the output file name. The output file is created by the same method as described in The Creation of Output Object Files on page 2-5, including the use of the `-temp_o` option or the `-must_use_oname` option.

Here is a complete list of the user options that are allowed with the`-alf`option:

```
-allow_missing_libs
-first_L
-L
-local_libname
-must_use_oname
-must_preset
-no_preset
-no_verbose
-no_version_check
-nostdlib
-o
-obey
-public_registry
-rld_first_L
-rld_L
-stdin
-t -temp_o
-unres_symbols
-verbose
-warn
```

For further information on `-alf` see Appendix , How -alf Updates DWARF for a discussion of the debug.reloc section of the object file.

The -set and -change Options

The `-set` option is used to set certain items in the file being created. These items are called “file attributes”. The following chart lists the file attributes and tells what they mean, usually telling the name of the section of this document that provides more explanation. Each of the attributes is specified together with a parameter to tell the value of the attribute.

Table 4-1. The -set and -change Options

Attribute	Attribute Meaning
`data_model`	For this attribute, allowed values are: ilp32, lp64, and neutral. See eld Functionality for 64-Bit on page 4-12.
`systype`	Specifies the target platform personality. See Target Platforms on page 2-2.
`libname`	Specifies the user library name.

`highpin`
`highrequestors`
`oktosettype`
`runnamed`
`saveabend`
`user_buffers`
	For these attributes the possible values are “ON” or “OFF”. These attributes correspond to flag bits in the *.tandem_info* section. The default is ON for highpin and highrequestors, and OFF for runnamed, saveabend, and oktosettype. There are several synonyms for “highrequestors”.

`oktosettype`	For this attribute the possible values are "ON" and "OFF". For compatibility with the past, `eld` accepts this attribute and checks its syntax, but otherwise this attribute is ignored.
`inspect`	For this attribute the possible values are “ON” and “OFF”. This determines which debugging facility is made available. (Internally the corresponding value within the *flags* field of the *.tandem_info* section of the file is *INSPECT_SUBSYSTEM* or *GARTH*, respectively.) With a TNS/E native file, “ON” will either start a Visual Inspect session or pass the file to an already established session. “OFF” will start Native Inspect.

Table 4-1. The -set and -change Options

Attribute	Attribute Meaning

`heap_max`
`mainstack_max`
`process_subtype`
	These attributes have numerical values. These attributes correspond to fields in the *.tandem_info* section. If not specified, the default value in each case is 0.

`space_guarantee`	See the *Guardian Procedure Calls Manual* for more information about how to use the heap_max, mainstack_max, and space_guarantee attributes. More information about process_subtype is provided in the notes following this table.
`floattype` `float_lib_overrule`	See Processing of Floating Point Versions and Data Models on page 4-16.
`CPlusPlusDialect`	See eld Functionality for 64-Bit on page 4-12.
`pfssize`	This attribute has a numerical value. For compatibility with the past, the linker accepts this attribute and checks its syntax, but otherwise this attribute is ignored.
`rld_unresolved`	The setting of this attribute tells the *-alf* option, as well as *rl*d, how to treat unresolved symbols. See Handling Unresolved References on page 3-8.
`incomplete`	For this attribute the only allowed (and therefore required) parameter is “ON”. If this attribute is specified, it means that the import library being created by the linker is incomplete. The incomplete attribute only applies to import libraries. The -change flag can be used to turn it ON, but cannot turn it off. See Creating Import Libraries on page 3-11.

The only attributes of the `-set` option which can be specified with the `-r` option are `data_model`, `floattype`, and `process_subtype`.

The linker produces a warning message if it is creating a loadfile with `highpin on` and any of the DLLs or import libraries that it has opened have `highpin off`.

The `-libname` option is a synonym for `-set libname`.

When `eld` builds a new object file from a set of input linkfiles it combines the values of the `process_subtype` attribute found in those input linkfiles. If an input linkfile has a *.tandem_info* section that is abbreviated to four bytes then its `process_subtype` is considered to be zero. A non-zero value in one input file is accepted in preference to a zero value in another input file, but two distinct non-zero values among the input files

are considered inconsistent. If `-set process_subtype` is specified then the value specified is placed into the output file and if any of the input linkfiles had a nonzero process subtype different from that then a warning message is produced. On the other hand, if `-set process_subtype` is not specified then it is an error if the input linkfiles were inconsistent, and if they are consistent then the value from the input linkfiles is placed into the output file.

The `-set incomplete on` option is only allowed when the linker is creating an import library that represents a single DLL. It tells the linker to create an incomplete import library, rather than a complete one. This could either be when the linker is creating the import library at the same time as the linker is creating the corresponding DLL, or when the linker is creating the import library for a DLL that already existed. In either case, all the other things that are controlled by the `-set` option are the same in the import library as in the corresponding DLL.

All the other attributes listed above, except for `libname`, can be specified whether building a program or a DLL, even though some of them may be meaningless for DLL's. The `libname` attribute can only be specified when building a program.

The `-change` option is used to alter the same attributes that can be set by the `-set` option. The difference is that `-change` modifies an existing file, whereas `-set` is used when a new file is being created. The `-change` option takes the same parameters as `-set`, plus the name of the existing file.

The `-change` option can be used with a linkfile, loadfile, or import library, with the following rules:

- The `-change` attributes that are allowed with a linkfile are the `data_model` and `floattype` attributes.
- The `-change libname` attribute is only allowed with programs. The linker places the name specified by the`-change libname` option into the *.tandem_info* section of the program that it is creating. The linker also converts the name to upper case, if not done already. On Guardian APIs,single quotes do not work because they are not recognized by TACL; therefore, it is not important to specify them.
- The `incomplete` attribute of the `-change` option is only allowed with import libraries. This tells the linker to demote the import library to an incomplete one, if it was a complete one before. There is no way to “un-demote” an incomplete import library.
- The `data_model` attribute of the `-change` option is only allowed with DLLs and import libraries.

It is possible to give an empty string, or a string consisting just of a single double-quote character ("), as the user library name in the `-change libname` option. This makes it possible to remove the user library name from an existing program. From the PC or OSS command line you can use two double-quote characters ("") to pass an empty string. However, on the TACL command line, when you use two double-quote characters (""), a single double-quote character (") gets passed to the process. That is

why `eld` also allows a single double-quote character (") to be given with `-change libname`, and interprets it as an empty string. Note that this is only allowed with `-change libname`, not with `-set libname`.

Except for the items listed above, there are no restrictions on which attributes can be specified. Note that some attributes can be specified for DLLs even though they are meaningless for DLLs.

Note that it is permissible to change any of the attributes of an import library, even though only a few of these attributes are significant in an import library. (The ones that are “significant” are the ones that the linker might look at when building a client of that import library in order to check for consistency, namely, the `highpin` bit, the floating point type, and the C++ dialect, as well as the bit that tells whether the import library is complete.)

The `-change` option performs the same version number checking on a loadfile or import library as is described in Input Object Files on page 2-12.

In all cases, the `-change` option only changes the specific information specified, and has no other consequences. For example, when `-change libname` is specified for a program, it only changes the user library name that is shown in the program. If the program was previously preset, using a different user library, that fact is unchanged. The newly specified user library would be used in the future when this program is run. At that time, if the user library found with that name was not equivalent to (i.e., contained the same export digest as) the user library that had been used when the program was preset, then the program would not have been preset correctly for its runtime environment and would need to be updated accordingly. That is the same as what would happen if the same user library filename were still being used, but the contents of that user library file had changed (i.e., so that its export digest was different).

The `-change` option overwrites the existing file without recreating it. `eld` reports an error if the file cannot be opened for update. If an error occurs the specified change may or may not have taken place, but the file will not have been modified in any other way.

It is possible for the command stream to have multiple `-change` options. Each one is executed independently, if the previous one succeeds, even if they refer to the same filename. For example, two `-change` options may both specify the same attribute, such as `-change highpin`, and they don’t have to agree on the value to which this attribute is being set, whereas such a consistency check would occur if `-set highpin` were specified more than once. The only other options allowed with the `-change` option are `-no_verbose`, `-obey`, `-stdin`, `-verbose`, and `-warn`.

Note: There are no `-set` or *`-change`* options that affect the procedure attributes that are found in the *.procinfo* section of a linkfile or the stack unwinding information of a loadfile. However, the `-e` option can be used to turn on the *MAIN* attribute in the *.procinfo* section of a linkfile that is being created by the linker.

eld Functionality for 64-Bit

Object files can be `32-bit`, `64-bit`, or `neutral`. When `eld` is creating a new object file out of a set of linkfiles, the desired data model for the output file is specified by the following option:

```
-set data_model [value]
```

where `value` can be `ilp32`, `lp64`, or `neutral`.

For `lp64` input linkfiles, it is considered as an error, if `-set data_model ilp32`" is specified. Conversely, for `ilp32` input linkfiles, it is considered as an error, if `lp64` is specified. Specifying neutral, does not verify the data models for the input linkfiles and the output file which is marked as neutral. The neutral option is allowed only while creating a linkfile or a DLL, and not for a program.

If the input linkfiles contains a combination of `ilp32` and `lp64` linkfiles, it is considered as an error if the `-set data_model` option is not specified. If this error does not occur, either the output file is marked with the data models (`ilp32` or `lp64`) which occur within the input linkfiles, or is marked as neutral if all the input linkfiles are neutral.

The `-change data_model neutral` option marks an object file neutral, regardless of how it was marked before. This is allowed for linkfiles and DLL's, but not for programs.

The searching that is done for `-l` options or for liblist entries depends on the data model of the file that is being created or processed by the `-alf` option, as discussed in this manual.

When `eld` creates an import library that represents a single (non-implicit) DLL, the import library is given the same data model as that DLL.

The import library that represents the implicit DLL's is always neutral.

Checking the C++ Language Dialect

To build a new linkfile or loadfile, the linker performs consistency checks of C++ language dialects.

In a linkfile, the *st_other* fields of ELF symbol table entries tell which C++ language dialects are used by that object file, if any. If more than one C++ dialect occurs among all the input linkfiles, that is an error.

By default, if some dialect of C++ occurred in the input linkfiles, then that value is placed into the *.tandem_info* section when `eld` is building a loadfile. If none of the input files used C++ then the value `cppneutral` is placed into the *.tandem_info* section.

CPlusPlusDialect is an attribute accepted by the `-set` option, and the only allowed value of the attribute is `cppneutral`. Also, this option is only allowed when creating a loadfile. If specified, it means that the loadfile will be marked `cppneutral`, even if the

input files contained C++. An informational message is generated if the input files contained C++.

When `eld` is creating a loadfile it also performs C++ dialect checking against all the DLL's that it sees as part of that load. The check is that no DLL has the opposite C++ dialect from that seen in an input linkfile.
Note that the use of `-set CplusPlusDialect cppneutral` does not affect this check. If this check fails a warning message is generated; it is not considered an error because different DLL's might be used at runtime.

Renaming Symbols

The `-rename` option affects how the linker treats each input linkfile as it is reading it in. This option takes two parameters, *symbol-1* and *symbol*-2. The option has no effect unless this linkfile defines a global symbol named *symbol*-1. `eld` reports an error if this same linkfile also defines a global symbol named *symbol*-2. The linkfile that defined *symbol-1* is now considered to define a symbol named *symbol*-2, with the same properties that *symbol-1* had. The linkfile is still considered to declare a symbol named *symbol*-1, with the same properties as before, except that it is now an external reference rather than a definition.

As a result, if the object file being built would have defined a symbol named *symbol*-1, it will now define *symbol-2* instead. If there were references to *symbol*-1, those will still refer to *symbol*-1, and therefore they could not be resolved within this same object file. If a new linkfile is being built with the `-r` option, a future link could combine this linkfile with another one that supplied a definition of *symbol*-1, to satisfy such references. Or, in general, the references could be satisfied in other DLLs.

One purpose of this option is to allow one copy of *symbol-1* to be replaced by another one, without having to recompile the source that created the linkfile containing *symbol*-1. For example, the given object file that defines *symbol-1* may be put through the linker with the `-r` and `-rename` options to create another linkfile that no longer defines *symbo*1, and then that linkfile could be linked with another one that provides a different definition of *symbol*-1. For this purpose it is not necessary that the name *symbol-2* ever be referenced. However, when a procedure is renamed, the new version of *symbol-1* might wish to do the same things that the older *symbol-1* would have done, plus some new things. It could do this by calling *symbol-2* to do the old stuff, and having additional code for the new stuff.

Any number of `-rename` options may be in the command stream. They all apply to all the input linkfiles, regardless of where they occur in the command stream. As each input linkfile is processed, the linker does the processing described above for each `-rename` option, in the order that the `-rename` options appeared in the command stream.

Creating Linker-Defined Symbols

Predefined Symbols on page A-14 lists the symbols that are automatically defined by the linker when it creates a loadfile, and what they mean. The linker resolves references to these symbols by using the value of the symbol as if it was an address.

As mentioned in Creating the MCB on page 4-15, `eld` reports an error if you define a symbol named _*MC*B. For other linker-defined symbols the linker puts out a warning if they are defined by the user. Note that, if such a symbol is defined by the user, then the linker uses the symbol defined in the linkfile in the usual way, rather than creating its own symbol of this name with a special meaning.

Updating Or Stripping DWARF Symbol Table Information

The linker concatenates the DWARF symbol tables of its input linkfiles into the DWARF symbol table of its output file.

The linker fills in the addresses of symbols found in the DWARF symbol table information. Such relocation sites are described in linkfiles by the same kinds of relocation table entries as any other data, so that the linker doesn't need to understand the format of the DWARF symbol table to do this. In loadfiles, the addresses in the DWARF symbol table are the usual addresses, not segment-relative addresses.

When there are multiple copies of a symbol, the DWARF addresses for the unused copies are set to -1, as discussed in Accepting Multiply-Defined Symbols on page 3-17.

The *stripping* of symbols information means that the linker does not create the DWARF symbol table information in its output file(s). The reason to do this is to make object files smaller, when the user is not concerned with debugging. For compatibility with the past, this can be done with either the `-s` or `-x` option. In our TNS/R implementations, `-s` strips all the symbols information, whereas `-x` strips only the symbols information that the linker doesn't need, but the distinction has disappeared with the segregation of the debugging information into the DWARF symbol table. In other words, the TNS/E linker does not depend on information in the DWARF symbol table for any of its other activities.

When the linker is creating a linkfile and stripping the DWARF symbol table the linker also omits the corresponding relocation table sections.

When the linker is creating an import library, that import library may or may not contain DWARF symbol table information. See Creating Import Libraries on page 3-11.

The `-strip` option tells the linker to remove the DWARF symbol table from a loadfile or import library that already exists. This option creates a new file in place of the old file.

It is possible for the command stream to have multiple `-strip` options. Each one is executed independently, if the previous one succeeds. The only other options allowed with the `-strip` option are `-must_use_oname`, `-no_verbose`, `-no_version_check`, `-obey`, `-stdin`, `-temp_o`, `-verbose`, and `-warn`. If

there are multiple `-strip` options, the temporary filename specified by `-temp_o` applies to all of them.

The ELF symbol table can never be stripped from a linkfile, and the *.dynsym* and *.dynsym.gblzd* sections can never be stripped from a loadfile, because they are needed by the linker and/or runtime loader.

Modifying the Data Sections that Contain Stack Unwinding Information

Linkfiles may contain an unwind function section, an unwind information section, and a relocation table section for that unwind function section. The linker concatenates these sections when creating another linkfile. The linker similarly concatenates the information in *.procinfo* and *.procnames* sections.

When building a loadfile, the linker creates an unwind function section, unwind information section, and string space. The format of the unwind function section in a loadfile is different from the format that it has in a linkfile, containing additional information that the linker obtains from the *.procinfo* and *.procnames* sections of its input linkfiles.

If there were multiple copies of procedures, the information about unused copies is omitted from the *.procinfo* and unwind function sections of linkfiles, and from the unwind function sections of loadfiles. In the linkfile case, that also means doing something about the corresponding relocation table entries.

Creating the MCB

The linker creates the *MCB* (“master control block”) as additional data in the *.data* section. The internal name of the MCB is *_MCB* (one underscore), so this name can be used to reference the *MCB*. The linker only creates the MCB if it is building a program, and only if there is a reference to the symbol named *_MCB*. `eld` reports an error if a symbol named *_MCB* is defined in any of the linker’s input object files, regardless of the type of file the linker is creating.

Two linker options that specifically affect the value placed into the MCB are `-ansistreams` and `-nostdfiles`. The value that the linker has decided to place into the ELF header to describe the floating point type is also reflected in the MCB.

The TNS/E linker considers it an error if either of the above options is specified when the linker is not creating a program. The option `-no_stdfiles` is accepted as a synonym for `-nostdfiles`.

Further information on the MCB may be found in The MCB (Master Control Block) on page A-14.

Processing of Floating Point Versions and Data Models

When the linker builds a new linkfile or loadfile it performs consistency checks of floating point versions. These checks use the floating point bits that are found in the *e_flags* field of the ELF header.

The linker determines whether the input linkfiles are consistent, meaning that they do not contain a mixture of both the `tandem` and `ieee` floating point types. If they are consistent then the result type is determined to be whichever of `tandem` or `ieee` occurred, or `neutral` if neither occurred.

If the `-set floattype` option is used, that determines the value that is placed into the output linkfile or loadfile. The three values that can be specified with the `-set floattype` option are `ieee`, `neutral`, and `tandem`, and they have the synonyms `ieee_float`, `neutral_float`, and `tandem_float`, respectively. An informational message is generated if the input files were not consistent, or if they were consistent but with a result type different from what the option specified.

If a program is marked `neutral`, the HP NonStop operating system and CRE consider that to mean the same thing as `tandem`. If the linker is creating a program and `-set_floattype neutral` has been specified then the linker will put out a warning message to say that floating point type `neutral` actually means `tandem` to the HP NonStop operating system and CRE.

If the `-set floattype` option is not used, `eld` reports an error if the input files are not consistent. If they are consistent, the result type is placed into the output file.

When the linker is building a loadfile, the `-set float_lib_overrule` option may be used to turn on the *EF_TANDEM_FLOAT_LIB_OVERRULE* bit in the ELF header of the loadfile. The linker allows this bit to be set in both programs and DLLs.

The linker performs additional floating point checks with regard to DLLs when it is creating a new loadfile, as described in the following paragraphs. The same checks are performed by the `-alf` option when they are updating an existing loadfile. These checks result in warning messages, not errors, because different DLLs may be used at runtime.

If the linker is building a program, or updating it with `-alf`, the intention is to be consistent with the checks that the HP NonStop operating system will do. So, there is no checking against DLLs if the linker has been told to turn on the *FLOAT_LIB_OVERRULE* bit in the program. If this bit is not to be turned on then the linker checks the program's floating point type against all the DLLs that it has opened. All DLLs must either be neutral or have the same floating point type as the one indicated in the program. As mentioned above, if the program is marked `neutral`, the linker considers that to mean `tandem`.

If the linker is building or updating a DLL then it checks that all the DLLs seen by the linker, including the DLL that is being created or updated, do not contain a mixture of both `tandem` and `ieee`.

Checks are only performed against DLLs that the linker actually saw. For example, the linker is not required to search for indirect DLLs if it is not presetting and not checking for unresolved symbols, so in that case it is not required to perform floating point consistency checking against such indirect DLLs.

Whenever the linker builds a new linkfile or loadfile it checks for the consistency of the data model among its input linkfiles, because the two data models cannot be intermixed in the same loadfile, and the data model of the input linkfiles determines the data model for the output file. More specifically, at the present time, the linker does not allow the 64-bit data model at all, but in the future that will be relaxed. There is no checking for consistency against DLLs, because loadfiles with different data models are allowed in the same process.

Specification of the Main Entry Point

There are two ways to specify the main entry point of a loadfile. One is to use the `-e` option, specifying the name of a procedure. The other way is for a procedure to have the MAIN attribute.

In a program, the *e_entry* field is filled in with the address of the real code for the main entry point. In a DLL, if it has a main entry point, then that procedure must have an official function descriptor, and the *e_entry* field contains the address of that official function descriptor.

When the `-e` option is specified the linker checks that it is the name of a procedure that is defined in this object file (otherwise it is an error). `eld` reports an error if this procedure has the *CALLABLE* or *KERNEL_CALLABLE* attribute. Additional rules depend on the type of file that the linker is building:

- If the linker is building a program then `eld` reports an error if no main entry point has been specified by either of the above methods. The linker will put out a warning if the `-e` option is used and the program has a procedure with the *MAIN* attribute, unless the same procedure is indicated in both cases. The linker uses the `-e` option to determine the main entry point, overriding the fact that a procedure had the *MAIN* attribute.
- If the linker is building a DLL then `eld` reports an error if any procedure has the *MAIN* attribute. However, the `-e` option is allowed. If the `-e` option is not provided then the *e_entry* field contains zero.
- If the linker is building a linkfile then the `-e` option has a different purpose. Namely, it turns on the *MAIN* attribute in the *.procinfo* entry for the specified name. The *e_entry* field of the ELF header in a linkfile always contains zero.

When the linker is building a loadfile `eld` reports an error if more than one procedure has the *MAIN* attribute, unless the `-allow_multiple_mains` option is used. When there are multiple procedures with the *MAIN* attribute, and this is not an error, the linker pays attention to the *MAIN* attribute on the first procedure that it sees with it. This ordering is implied by the order in which the linker finds linkfiles from the command stream and the ordering of the entries in the *.procinfo* sections of the linkfiles.

When building a program in C or C++, and using the standard runtime library support provided by these compilers, the usual method is to place in the linker command stream an object file that is also supplied with these compilers and contains a procedure that has the *MAIN* attribute. Users who invoke the linker through the compiler may not realize this, because the compiler automatically adds the file name to the linker command stream when it invokes the linker. Thus, for most users of C or C++, the main procedure is not a procedure they've written themselves, and there is no need to use the `-e` option. If you use the `-e` option and specify the wrong procedure (for instance, the procedure named *main* that you've written) as the main entry point, and you ignore the linker warning about this, then you will probably build a program that will not run correctly because it will not start at the proper point.

Specifying Runtime Search Path Information for DLLs

The linker fills in the *DT_RPATH* and *DT_TANDEM_RPATH_FIRST* entries of the *.dynamic* section to tell `rld` where to find DLLs at load time. Each of these fields is a list of places to look, where a colon separates the names in the list. The individual names cannot contain colons. The commands used for filling in these entries, and the purposes of these entries, are as follows:

- *DT_RPATH* tells places to look after looking for public DLLs. The `-rld_L` option specifies a string to place into *DT_RPAT*H. `-rpath` is accepted as a synonym for `-rld_L`.
- *DT_TANDEM_RPATH_FIRST* tells places to look before looking for public DLLs. The `-rld_first_L` option specifies a string to place into *DT_TANDEM_RPATH_FIRS*T.

Each of the `-rld_L` and `-rld_first_L` options may be specified multiple times in the command stream. The linker concatenates the information provided, in the order the options were given. The linker understands that names are separated by colons, and removes duplicate names from the list. `eld` reports an error if you specify these options with `-r`.

Note that the linker does not try to detect names that are "invalid" for any reason and remove them from the list.

Note that the `-rld_L` and `-rld_first_L` options are allowed whether building a program or a DLL. `rld` only uses this information found in programs. However, the linker looks at this information in each DLL that it uses in a link that is creating a loadfile. The linker concatenates this information from DLLs the same as if the corresponding values had been given in the command stream, considering them to come at the end of the command stream and in the same order as the DLLs were found by the linker.

The `-limit_runtime_paths` option is used to tell `rld` that its algorithm for looking for DLLs is not to be concerned with the location of the program nor with any runtime path specifications. When this option is specified the linker turns on the *LIMIT_RUNTIME_PATHS* bit in the *flags* field of the *.tandem_info* section.

The *DT_RPATH* and *DT_TANDEM_RPATH_FIRST* entries in the *.dynamic* section are also used by `-alf` to decide where to look, although in this case they can be overridden by giving the `-rld_L` or `-rld_first_L`options with `-alf`.

Merging Source RTDUs

Each input linkfile may contain a set of source RTDUs, which are used in the implementation of SQL/MP. If so, then the linker creates source RTDUs in its output file. This mostly involves concatenating the corresponding sections, although there are also pointers into RTDU string spaces that need to be updated appropriately.

5 Summary of Linker Options

This section lists all the options supported by the TNS/E linker. For each one the complete syntax is shown, a brief statement of its function is given, and a hyperlinked reference is given to the main discussion of it elsewhere in this manual.

`-alf <filename>`

Rebase and/or rebind an existing loadfile, recreating the file.
See Additional rules about -alf on page 4-3.

`-all`

Use all members from archives. See Using Archives on page 2-16.

`-allow_duplicate_procs`

Do not consider it an error if there are multiple definitions of procedures with the same name. See Accepting Multiply-Defined Symbols on page 3-17.

`-allow_missing_libs`

Do not consider it an error if a `-l` option cannot be resolved, except in situations where `-b static` is in effect. See How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

`-allow_multiple_mains`

Do not consider it an error if more than one procedure has the MAIN attribute.
See Specification of the Main Entry Point on page 4-17.

`-ansistreams`

At runtime, the program will use the ANSI version of C I/O.
See Creating the MCB on page 4-15.

`-b { dllsonly | dynamic | static }`

These options specify whether the linker accepts DLLs and/or archives.
See How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

`-b { globalized | localized | semi_globalized | symbolic }`

These options affect how references are resolved across loadfiles.
See Overview on page 3-1.

`-call_shared`

Create a program. See Output Object Files on page 2-4.

`-change <attribute> <value> <filename>`

Change the parts of an existing object file corresponding to things that the `-set` option would set up. The <attribute> and <value> have the same possibilities as for the `-set` option shown below. See The -set and -change Options on page 4-8.

`-check_registry <filename>`

Use the specified DLL registry to tell where the DLL being built must be placed in memory. See Using a DLL Registry on page 2-8.

`-cross_dll_cleanup`

Discard a procedure if found in another DLL. For more information, see Using the -cross_dll_cleanup option on page 3-19.

`-d <hexadecimal address>`

Use the specified value as the starting address of the data (constant) segment. See Creating Segments of the Output Loadfile on page 2-6.

`-data_resident`

This is a special option that may be used when building a “proto-process”, also known as a “sysgen process”.

`-dll`

synonym for `-shared`.

`-dllname`

synonym for `-soname`.

`-e <symbol name>`

Use the address of the specified procedure as the main entry point. See Specification of the Main Entry Point on page 4-17.

`-error_unresolved`

synonym for `-unres_symbols error`.

`-export`

synonym for `-exported_symbol`.

`-export_all`

Export all symbols that one might normally want to have exported without naming them explicitly. See Specifying Which Symbols to Export, and Creating the Export Digest on page 3-20.

`-exported_symbol <symbol name>`

Export the specified symbol from the loadfile being created. See Specifying Which Symbols to Export, and Creating the Export Digest on page 3-20.

`-export_not`

synonym for `-hidden_symbol`.

`-first_L <location>`

The specified directory or subvolume is one of the places where the linker will look for DLLs and archives before it looks for public DLLs. See The Steps in Looking for Archives and DLLs on page 2-17.

`-FL`

synonym for `-obey`.

`-grow_data_amount <number>`

Leave the specified amount of slack space in virtual memory for the data of this DLL. See Using a DLL Registry on page 2-8.

`-grow_limit <number>`

Use the specified value as the total amount of memory reserved for this DLL. See Using a DLL Registry on page 2-8.

`-grow_percent <number>`

Leave the specified percentage of slack space in virtual memory for each of the text and data segments of this DLL. See Using a DLL Registry on page 2-8.

`-grow_text_amount <number>`

Leave the specified amount of slack space in virtual memory for the text of this DLL. See Using a DLL Registry on page 2-8.

`-hidden_symbol <symbol name>`

Do not export the specified symbol. See Specifying Which Symbols to Export, and Creating the Export Digest on page 3-20

`-import_lib <filename>`

Build a complete or incomplete import library with the specified filename in addition to creating a new DLL. See Creating Import Libraries on page 3-11.

`-import_lib_stripped <filename>`

Build a complete or incomplete import library with the specified filename in addition to creating a new DLL, and strip the DWARF symbol table from the import library. See Updating Or Stripping DWARF Symbol Table Information on page 4-14.

`-include_whole`

synonym for `-all`.

`-instance_data { data1 | data2 | data2protected | data2hidden | data1constant }`

This tells the linker whether to create one or two data segments, and whether to require that the loadfile have no data that would need to go into the data variable segment if two segments were created. See Creating Segments of the Output Loadfile on page 2-6.

`-l <filename>`

Use the specified filename to locate a DLL or archive. The "`-l`" must be specified in lower case. See How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

`-L <location>`

The specified directory or subvolume is one of the places where the linker will look for DLLs and archives, after it looks for public DLLs. The "`-L`" must be specified in upper case. See The Steps in Looking for Archives and DLLs on page 2-17.

`-lib`

synonym for `-l`. This usage may be preferred because it is not case sensitive and therefore cannot be confused with `-L`.

`-libname`

synonym for `-set libname`.

`-libvol`

synonym for `-L`.

`-limit_runtime_paths`

If this is specified then `rld` will not permit the user to override the places specified at link time for where DLLs may be found.
See Specifying Runtime Search Path Information for DLLs on page 4-18.

`-local_libname <filename>`

Use the specified filename as the name of the user library that can be used to resolve references in this program at link time.
See Using User Libraries on page 3-10.

`-m`

synonym for `-map`.

`-make_implicit_lib`

Mark the DLL being created as an implicit library.
See Output Object Files on page 2-4.

`-make_import_lib <filename>`

Create a complete or incomplete import library with the specified filename, to represent the other DLL or DLLs whose filename(s) are found in the command stream. See Creating Import Libraries on page 3-11.

`-map`

Produce a map showing how memory has been laid out. See Creating Segments of the Output Loadfile on page 2-6.

`-must_preset`

Consider it an error if presetting fails. See To Preset or Not to Preset, and Creation of the LIC on page 3-7.

`-must_use_iname`

`eld` reports an error if the linker isn't able to delete an existing file of the same name when creating an import library. See Creating Import Libraries on page 3-11.

`-must_use_oname`

`eld` reports an error if the linker isn't able to delete an existing file of the same name when creating its main output object file. See The Creation of Output Object Files on page 2-5.

`-must_use_rname`

`eld` reports an error if the linker isn't able to delete an existing file of the same name when it is recreating a private DLL registry. See Using a DLL Registry on page 2-8.

`-no_include_whole`

synonym for `-none`.

`-none`

Only include archive members in the link if they satisfy needed references. See Using Archives on page 2-16.

`-no_optional_lib`

Do not consider later DLLs in the command stream to be optional. See Ignoring Optional Libraries on page 3-14.

`-no_preset`

Do not preset the loadfile being created. See To Preset or Not to Preset, and Creation of the LIC on page 3-7.

`-no_reexport`

Do not reexport DLLs found after this point in the command stream. See How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

`-nostdfiles`

At runtime, do not automatically open the standard C I/O files.
See Creating the MCB on page 4-15.

`-no_stdfiles`

synonym for `-nostdfiles`.

`-nostdlib`

Do not look in the standard places for DLLs and archives. See The Steps in Looking for Archives and DLLs on page 2-17.

`-no_stdlib`

synonym for `-nostdlib`.

`-noverbose`

synonym for `-no_verbose`.

`-no_verbose`

Do not show warnings or informational messages unless they are requested by a linker option. See General Information on page 6-1.

`-no_version_check`

Do not perform the object file version check. See Input Object Files on page 2-12.

`-NS_extent_size` *`extent-size`*

Note. For Guardian environment only.

Changes the extent size from the default (32 pages) to the specified size.

`extent-size`

is an even number in the range 2 to 65534, inclusive.

A page has 2048 bytes.

eld uses *`extent-size`* for both primary and secondary extents.

`-NS_max_extents` *`max_extents`*

Note. For Guardian environment only.

Changes the maximum number of extents from the default (900) to the specified number.

`max_extents`

is a number in the range 16 to 900, inclusive.

Note. The *Guardian Procedure Calls Reference Manual* recommends that *`max-extents`* not exceed 500.

`-o <filename>`

Use this as the name of the output object file. See The Creation of Output Object Files on page 2-5.

`-obey <filename>`

Use the specified file as an obey file. See Obey Files and the Use of Standard Input on page 1-7.

`-optional_lib`

Consider later DLLs in the command stream to be optional. See Ignoring Optional Libraries on page 3-14.

`-public_registry <filename>`

Use the specified file as the public DLL registry file. See Finding Public DLLs on page 2-18

`-r`

Create a linkfile rather than a loadfile. See Output Object Files on page 2-4.

`-reexport`

Re-export DLLs found after this point in the command stream. See How the Linker Finds Its Input Files and Creates the .liblist Section on page 2-12.

`-rename <symbol name> <symbol name>`

Change the name of a symbol while creating a new file. See Renaming Symbols on page 4-13.

`-rld_first_L <path>`

The string specified by <path> should be a list of directories and/or subvolumes separated by colons. At runtime, the specified directories and/or subvolumes are

places where `rld` will look for DLLs before it looks for public DLLs. See Specifying Runtime Search Path Information for DLLs on page 4-18.

`-rld_L <path>`

The string specified by <path> should be a list of directories and/or subvolumes separated by colons. At runtime, the specified directories and/or subvolumes are places where `rld` will look for DLLs after it looks for public DLLs. See Specifying Runtime Search Path Information for DLLs on page 4-18.

`-rpath`

synonym for `-rld_L`.

`-s`

Omit the DWARF symbol table when creating the output file. See Creating Import Libraries on page 3-11.

`-set <attribute> <value>`

Set the specified attribute to have the specified value in the loadfile being created. The following chart lists the attributes, their possible values, their defaults if not specified, and the section of this document to look at for more information.

Table 5-1. Set Attributes

Attribute Name	Allowable Values	Default	See Section for Details
CPPDialect | CPlusPlusDialect	neutral | cppneutral	These two names and values are synonymous. Use one only. The value comes from the input linkfiles	eld Functionality for 64-Bit on page 4-12
data_model	ilp32 | lp64 |neutral	ilp32	See eld Functionality for 64-Bit on page 4-12
floattype	ieee | neutral | tandem	The value comes from the input linkfiles	Processing of Floating Point Versions and Data Models on page 4-16
float_lib_overrule on | off		off	Processing of Floating Point Versions and Data Models on page 4-16

Table 5-1. Set Attributes

Attribute Name	Allowable Values	Default	See Section for Details
heap_max <	number>	0	The -set and -change Options on page 4-8
highpin on	| off	on	The -set and -change Options on page 4-8
highrequester | highrequesters | highrequestor | highrequestors	on | off	on	The -set and -change Options on page 4-8
incomplete	on (note: only one allowable value, so it is therefore also required)	If not specified, and an import library is being created, it is a complete import library.	Creating Import Libraries on page 3-11
inspect	on | off	on	The -set and -change Options on page 4-8
interpose_user_library	on | off	off	The -set and -change Options on page 4-8
libname	<filename>	If not specified, there may be no user library, or the name may be derived from what is specified for the -*local_libname* option.	Using User Libraries on page 3-10
mainstack_max <	number>	0	The -set and -change Options on page 4-8
oktosettype on	| off	off	The -set and -change Options on page 4-8
pfs | pfssize	<number>	Option is a no-op.	The -set and -change Options on page 4-8
process_subtype | subtype	<number>	0	The -set and -change Options on page 4-8

Table 5-1. Set Attributes

Attribute Name	Allowable Values	Default	See Section for Details
rld_unresolved	error | warn | ignore	error	Handling Unresolved References on page 3-8
runnamed	on | off	off	The -set and -change Options on page 4-8
saveabend	on | off	off	The -set and -change Options on page 4-8
space_guarantee	< number>	0	The -set and -change Options on page 4-8
systype	guardian | oss	(depends on the platform)	Target Platforms on page 2-2
User_buffers	on | off	off	The -set and -change Options on page 4-8

`-shared`

Create a DLL. See Output Object Files on page 2-4.

`-show_multiple_defs`

Put information into the listing about symbols that are defined in more than one of the input linkfiles. See Accepting Multiply-Defined Symbols on page 3-17

`-soname <filename>`

Specify the DLL name for the DLL being created. See Output Object Files on page 2-4.

`-stdin`

Use the standard input file as an obey file. See Obey Files and the Use of Standard Input on page 1-7.

`-strip <filename>`

Remove the DWARF symbol table from an existing loadfile or import library. See Updating Or Stripping DWARF Symbol Table Information on page 4-14.

`-t <hexadecimal number>`

Use the specified value as the starting address of the text segment of the loadfile being built. See Creating Segments of the Output Loadfile on page 2-6.

`-temp_i <filename>`

Use the specified filename as the name of the intermediate file during the creation of an import library. See Creating Import Libraries on page 3-11.

`-temp_o <filename>`

Use the specified filename as the name of the intermediate file during the creation of the linker's main output object file. See The Creation of Output Object Files on page 2-5.

`-temp_r <filename>`

Use the specified filename as the name of the intermediate file during the recreation of a DLL registry. Using a DLL Registry on page 2-8.

`-u <symbol name>`

Consider the specified symbol to be needed when deciding which files to take from archives. Using Archives on page 2-16

`-ul`

Create a user library. Actually, this option is a synonym for `-shared` plus `-export_all`. See Output Object Files on page 2-4.

`-unres_symbols { error | ignore | warn }`

Handle unresolved references in the way specified. See Handling Unresolved References on page 3-8

`-update_registry <filename>`

Use the specified DLL registry to suggest where the DLL being built may be placed in memory and update it with the location chosen. See Using a DLL Registry on page 2-8.

`-verbose`

Show all messages. See Output Listings and Error Handling on page 6-1.

`-warn`

Show all error and warning messages. See Output Listings and Error Handling on page 6-1.

`-warn_common`

Warn when a common symbol is combined with another common symbol of the same name but of different size.

`-warning_unresolved`

synonym for `-unres_symbols warn`.

`-x`

Omit the DWARF symbol table when creating the output file. See Creating Import Libraries on page 3-11.

`-y <symbol name>`

Provide information about how this symbol is mentioned in the ELF symbol tables of the linker's input files. See Merging Symbols Found in Input Linkfiles on page 3-16.

6

Output Listings and Error Handling

This section contains the following topics:

General Information - when and how messages are created.

Error Messages - individual cause, effect and recovery information.

Glossary of Errors - further detail on some of the error message text .

General Information

The linker creates a listing file that is written to the C standard output file. `eld` does not override whatever default rules are provided for the standard output file by the C runtime on each platform. For example, if it is not possible to open the standard output file, and the default behavior of the C runtime is to write out a message on standard error in such a situation, then that is what will happen. Also, note that when you put standard output into a file on Guardian, it gets appended to what was in that file before, rather than overwriting the file (this is "normal" Guardian behavior; for example, FUP does the same thing).

Messages that appear in the output listing fall into the following four severity levels:

Table 6-1. Completion Codes - The Severity Levels of Messages

Severity Level	Type of Message	Meaning
3	fatal error	The linker cannot do what was requested of it and the linker immediately stops. This includes all cases of command stream syntax errors, I/O errors, and memory allocation errors.
2	error	The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping.
1	warning	The linker can do what was requested of it, but the linker isn't sure that this is what the user really wanted.
0	information	This is not indicative of a problem.

On Guardian `eld`'s return code is equal to the highest severity level that occurred during the link, i.e., 0, 1, 2, or 3. That is done because this is a Guardian standard. On the other hand, on the PC and OSS, users are accustomed to just checking whether the value is 0 or 1, and considering 1 to be an error, and not caring about warnings. So, on the PC and OSS, `eld` calls exit with a parameter value of 0 if the highest severity was 0 or 1, and with a parameter value of 1 if the highest severity level was 2 or 3.

Normally, eld cleans up any output files and flushes the standard output file. However, if eld terminates unexpectedly, incomplete output files may still be in existence and the standard output file may not be flushed.

It is also possible for the standard output file to disappear while eld is running. For example, this could happen on Guardian if the standard output file is a process, and that process is killed while eld is running. In that case, eld still goes on with what it was doing, producing the same return code as described above, to tell if the output file could be correctly created. The only difference is that there is no standard output file.

Versions of eld running on TNS/R and TNS/E are built (by the corresponding TNS/R or TNS/E tools) with -set saveabend off, so by default these versions of eld do not generate a saveabend file if they terminate in error. You may use the normal HP NonStop OS command line mechanisms to request that eld generate a saveabend file if it terminates in error.

Messages are numbered and tell their severity level. Messages can be understood in conjunction with the other material in this document, including the following section that lists the messages. It is not intended that people depend on the specific numbers or contents of messages, since they are always subject to change. Messages related to I/O errors contain the message text for the error returned from the C runtime.

Some messages are requested by specific linker options. Such messages always appear in the output listing. The following are the options that do this:

```
-map
-show_multiple_defs
-unres_symbols (warn or error)
-y
```

The appearance of other messages in the output listing is controlled by the following options:

-verbose

show all messages (error, warning, and informational)

-warn

show all error and warning messages (but not informational messages)

-no_verbose

show error messages (but not warnings or informational messages)

It is an error if more than one of these options is specified. Since all messages created by eld during command stream processing are errors, there is no need for a toggle.

If none of the above options is given, the default is -no_verbose on OSS and the PC, and verbose on Guardian.

The option -noverbose is a synonym for -no_verbose.

In those cases where the default would be -verbose, the listing always begins with banner information. In other cases, there is banner information at the top of the output file if -verbose is explicitly specified or if any messages need to be written. The banner information includes the following items:

- the name of eld itself (i.e., as provided to it by the argv [0] string)
- information that tells when this version of eld was built
- a copyright statement
- a statement about the GNU license, as required by that license

However, this banner information is omitted if either the -vslisting or -no_banner option is given. If eld encounters a fatal error while processing the command line then these options need to be earlier on the command line in order to have an effect.

The purpose of the -vslisting option is to guarantee that the format of the eld listing satisfies the requirements of ETK, the Enterprise Tool Kit, as explained below. This is a tool that causes eld to be invoked and also specifies the -vslisting option.

The purpose of the -no_banner option is for c89 or c99 to use it at least in cases where eld is in a directory that has a space in its name. In such a case, c89 or c99 does not need to set up argv [0] correctly for eld. Instead, c89 or c99 itself may print out the correct name of eld and use the -no_banner option to stop eld from printing out the incorrect string that c89 or c99 passed to it.

The next thing in the listing is a copy of the command line, without -obey options expanded.

After that come all the messages in the four categories shown above.

If the listing is not otherwise empty then it concludes with summary information. The first summary line tells whether the link succeeded and what type of file was created or updated. The next summary line tells the date when the file was created, corresponding to the tim_dat field of the .tandem_info section.

After that there are counts of the number of errors, number of warnings, and number of informational messages. This also tells how many warning and informational messages were suppressed if the mode is not -verbose. It also tells the elapsed time. This portion of the listing is omitted if the -vslisting option is given.

Each message that eld writes to the listing file occupies some number of lines. The first line of the message does not begin with white space, while subsequent lines of a message begin with three blanks. The first line of a numbered message has the format:

```
**** <type> **** [number]:
```

where <type> is one of "FATAL ERROR", "ERROR", "WARNING", or "INFORMATIONAL MESSAGE". When a message needs to extend over several lines,

eld breaks the message at a space if possible. When a message is broken at a space, the space is left at the end of the broken line.

eld demangles C++ symbol names in listing messages, as follows. Any name that contains two consecutive underscores is assumed to be a C++ name, and eld tries to demangle it, using a demangling algorithm that tries to be consistent with the mangling done by the C++ compiler. If a name cannot be successfully demangled, eld writes out the name as it found it. If a name can be demangled, eld writes out both the original and the demangled form.

Error Messages

This section contains detailed error messages with cause, effect and recovery information where necessary.

The table shows all the messages that could normally appear during eld's operation. There is no particular significance to any of the message numbers. Following the table there is a Glossary of Errors on page 6-126 that provides more help based directly on the words that appear in the messages.

It is also possible that you may see a message that begins with the string "Internal error" or "Bad input file", not listed in this section. In some cases, the message may help you resolve the problem at hand. If you cannot resolve such a problem, or if any other message appears that is not listed in the table below, or if eld terminates abnormally, that is probably something to be reported as a possible eld bug to your HP representative. If eld complained about an input file, that might also indicate a bug in some other tool that created that input file.

1000 Cannot open file <filename>.

Cause. This message may appear in one of two scenarios. One is that you gave the -make_import_lib option, specifying the names of one or more existing DLLs on the command line, and the filename mentioned in the message is one of those names. The other is that you gave the -change option, and the filename mentioned in the message is the name of the existing file that you intend to update. In either case, the specified file doesn't exist or you don't have permission to read it.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you really intended to specify a file of the indicated name, that you spelled it correctly, and that you have permission to read it.

1004 The addresses of the two memory areas overlap: the text memory area goes from <address> to <address>, and the data memory area goes from <address> to <address>.

Cause. You gave options such as `-t` and `-d` to specifically provide the starting addresses for the code and data segments of the program or DLL that you are building. However, with these starting addresses, the segments overlapped.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you need to give these options in the first place? There is usually no reason to use these options when building a program, and there is usually no reason to use the `-d` option at all. But, if this is some special case where you do need to give these options, and you know why you are doing it, then you should give different values, so that the segments don't overlap. You can tell from the error message how big each segment was.

1005 Unresolved reference to <symbol name>.

Cause. .You gave the `-alf` option, to repeat the process of fixing up references in an existing program or DLL, but the symbol named in the message did not exist in that same program or DLL, nor was `eld` able to find it by looking into other DLL's. This may occur for many reasons, such as problems with DLL's that other people are supposed to provide to you, which either they didn't provide or you didn't pass along to `eld` when you first built this program or DLL, or "standard" things not set up correctly in your installation. You also gave an option such as `-unres_symbols warn`, to say that `eld` should not consider this an error situation.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. You don't necessarily need to do anything. A program can run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, look at the names of the symbols that `eld` said it couldn't find, and see if they exist somewhere. They may be in DLLs, for example, that `eld` wasn't using, so you may need to relink your program or DLL again, supplying the names of those DLLs. `eld` will print out informational messages about all the DLLs that it used if you supply the `-verbose` option. A symbol in a DLL also needs to be exported from that DLL for `eld` to find it. The `-unres_symbols` option specifies whether `eld` should consider unresolved references to be errors, warnings, or neither.

1006 Unresolved reference to <symbol name>.

Cause. You gave the `-alf` option, to repeat the process of fixing up references in an existing program or DLL, but the symbol named in the message did not exist in that same program or DLL, nor was `eld` able to find it by looking into other DLL's. This may occur for many reasons, such as problems with DLL's that other people are supposed to provide to you, which either they didn't provide or you didn't pass along to `eld` when you first built this program or DLL, or "standard" things not set up correctly in your installation. You did not give an option such as `-unres_symbols warn`, so by default `eld` considered this an error situation.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. A program can run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, you need to look at the names of the symbols that `eld` said that it couldn't find, and see if they exist somewhere. They may be in DLLs, for example, that `eld` wasn't using, so you may need to relink your program or DLL again, supplying the names of those DLLs. `eld` will print out informational messages about all the DLLs that it used if you supply the `-verbose` option. A symbol in a DLL also needs to be exported from that DLL for `eld` to find it. The `-unres_symbols` option specifies whether `eld` should consider unresolved references to be errors, warnings, or neither.

1007 A user library name can only be specified for programs.

Cause. You gave the `-libname`, `-set libname`, or `-change libname` option. The first two of these (which are synonyms) are used to tell the Guardian filename that the user library will have when you run the program that you are buildling. The `-change libname` option tells `eld` how to update that Guardian filename within an existing program. However, along with either `-libname` or -set libname you have also given the `-dll`, `-shared`, or `-ul` option to tell `eld` to build a DLL, rather than a program, or you have given the `-r` option, to tell `eld` to build a "linkfile", i.e., an object file that can be used as `eld` input again, rather than a program. Or, in the case that you gave the `-change` option, the filename that you specified with the `-change` option is not a program but instead is either a DLL or a "linkfile", such as is created by a compilation.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program then don't specify options for creating something that isn't a program, such as the `-dll`, `-shared`, or `-ul` options that tell `eld` to create a DLL, or the `-r` option that tells `eld` to create another object file that can be used again as `eld` input. If you intended to create one of these other types of object files, rather than a program, then don't give the `-libname` or `-set libname` option. If you intended to update the user library name within an existing program, then figure out why the filename that you gave to `eld` was not the name of a program. There is no user library name in any other kind of object file.

1008 The 'systype' attribute is not allowed with the -r option.

Cause. You gave the `-set systype` option, to specify the system type (Guardian or OSS) for the program or DLL that you are creating, and you also used the `-r` option, to tell `eld` to create another object file that can be used as linker input, rather than a program or DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used again as `eld` input, then don't specify the `-set systype` option.

1009 The 'float_lib_overrule' attribute is not allowed with the -r option.

Cause. You gave the `-set float_lib_overrule` option, to specify the type of floating point consistency checking to perform at runtime for the program or DLL that you are creating, and you also gave the `-r` option, to tell `eld` to create an object file that can be used again as linker input, rather than a program or DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used again as `eld` input, then don't specify the `-set float_lib_overrule` option.

1010 The 'incomplete' attribute is only allowed if you are creating an import library or using the -change option to update in import library.

Cause. This can occur in one of two scenarios. In one case, you gave the `-set incomplete` option, which is used to say that the import library that you are creating should be marked "incompete", but you did not give either the `-import_lib` or `-import_lib_stripped` option to say that you wanted to make an import library at the same time that you were making a DLL. The other case is that that you gave the `-change incomplete` option, but the filename specified for the `-change` option was not an import library. The 'incomplete' attribute only applies to import libraries.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create an import library at the same time that you are creating a DLL, you must specify the `-import_lib` or `-import_lib_stripped` option. If you are not creating an import library, do not specify the `-set incomplete` option. Or, if you intended to update an existing import library to say that it was incomplete, then figure out why the filename that you gave to `eld` as not the name of an import library. You cannot set the incomplete attribute in any other kind of object file.

1011 Input file <filename> specifies that the address of <symbol name> is supposed to be placed into a 32-bit pointer, but it doesn't fit into 32 bits.

Cause. You gave options such as the -t and `-d` options to specify the starting virtual addresses for the code and data segments of the program or DLL that you are creating, and you specified addresses that wouldn't fit into 32 bits. Or, as part of this

link, eld looked at some other DLL whose addresses didn't fit into 32 bits. In either case, the program or DLL that you are trying to build contains 32-bit pointers that need to be initialized with the addresses of symbols, in the file being built or in some other DLL, whose addresses can't be represented in 32 bits.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Did you need to use the -t and -d options? There usually is no reason to use these options when building a program, and there usually is no reason to use the -d option at all. But, if you want to assign addresses to this program or DLL that don't fit into 32 bits, or make references to some other DLL that has a range of addresses that doesn't fit into 32 bits, then change your source code so that it doesn't try to use 32-bit pointers in those cases.

1012 Input file <filename> specifies that the address of <symbol name>, plus <number>, is supposed to be placed into a 32-bit pointer, but it doesn't fit into 32 bits.

Cause. You gave options such as the -t and -d options to specify the starting virtual addresses for the code and data segments of the program or DLL that you are creating, and you specified addresses that wouldn't fit into 32 bits. Or, as part of this link, eld looked at some other DLL whose addresses didn't fit into 32 bits. In either case, the program or DLL that you are trying to build contains 32-bit pointers that need to be initialized with the addresses of symbols, in the file being built or in some other DLL, whose addresses can't be represented in 32 bits, or at least not when the offset mentioned in the message is added to their addresses.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Did you need to use the -t and -d options? There usually is no reason to use these options when building a program, and there usually is no reason to use the -d option at all. But, if you want to assign addresses to this program or DLL that don't fit into 32 bits, or make references to some other DLL that has a range of addresses that doesn't fit into 32 bits, then change your source code so that it doesn't try to use 32-bit pointers in those cases.

1013 Input file <filename> specifies that the address of <symbol name> is supposed to be placed into a 32-bit procedure pointer, but it doesn't fit into 32 bits.

Cause. You gave options such as the -t and -d options to specify the starting virtual addresses for the code and data segments of the program or DLL that you are creating, and you specified addresses that wouldn't fit into 32 bits. Or, as part of this link, eld looked at some other DLL whose addresses didn't fit into 32 bits. In either case, the program or DLL that you are trying to build contains 32-bit procedure pointers

that need to be initialized with the addresses of procedures, in the file you are building or some other DLL, whose addresses can't be represented in 32 bits.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Did you need to use the `-t` and `-d` options? There usually is no reason to use these options when building a program, and there usually is no reason to use the `-d` option at all. But, if you want to assign addresses to this program or DLL that don't fit into 32 bits, or make references to some other DLL that has a range of addresses that doesn't fit into 32 bits, then change your source code so that it doesn't try to use 32-bit procedure pointers in those cases.

```
1014 The output program or DLL has highpin on, although DLL
<filename> has highpin off.
```

Cause. You are building a program or DLL that has the HIGHPIN attribute set ON, which is the default, but during the link one or more of the DLLs that `eld` looked at had the HIGHPIN attribute set OFF.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. This is only considered a warning by `eld` because what matters is which DLLs you will be using at runtime. Determine which of your programs and DLLs should have HIGHPIN ON, and which should have it OFF. The rule is that a program or DLL with HIGHPIN ON cannot be a client of a DLL with HIGHPIN OFF. Your program must satisfy this rule or NSK will not allow it to run. If you also satisfy this rule at link time, your link won't produce this warning message.

```
1017 Using complete import library <filename>.
```

Cause. `eld` prints out informational messages about some of the files that it used and what types of files they were. In this case, it is telling you about a complete import library.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1018 Using incomplete import library <filename>.
```

Cause. `eld` prints out informational messages about some of the files that it used and what types of files they were. In this case, it is telling you about an incomplete import library.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1019 Using DLL <filename>.

Cause. eld prints out informational messages about some of the files that it used and what types of files they were. In this case, it is telling you about a DLL.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1021 Bad input file: the symbol named _MCB is defined in <filename>, and that is not allowed.

Cause. The symbol _MCB is a special symbol that eld creates within a program if the program makes a reference to it. User code is only allowed to make external references to it, not to define it, but there was a definition of this symbol in your code.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to make a reference to the symbol named _MCB, change your “definition” into an “external reference”. The syntax for that depends on your source language.

1024 Error reading file <filename>.

Cause. This message can appear under various circumstances, when eld had successfully opened a file but later had trouble reading it. That probably indicates that there was something wrong with the contents of the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If the input file is bad, that may or may not be a problem you can fix yourself. This might indicate an error in some other tool that created the input file. The problem may need to be referred to your HP representative.

1025 When multiple filenames are given on the command line in addition to the -make_import_lib option, they must all be filenames of implicit DLL's, but <filename> is not an implicit DLL.

Cause. You gave the -make_import_lib option to tell the name of the import library that you want to create, and you also gave more than one other filename on the command line. That is only allowed if you are creating the zimpimp file that represents the implicit DLLs that constitute system library. But, one of those other filenames on the command line was not an implicit DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to create a zimpimp file, then the other filenames on the command line should be implicit DLLs. If your intention is to create an import library to represent an ordinary DLL (not one of the implicit DLLs), then correct your command line syntax, because there are more filenames present than is allowed. If it is not your intention to create any kind of import library then don't give the -make_import_lib option.

```
1026 The -make_import_lib option was given, so the other
filename on the command line should be a DLL, but <filename>
is not a DLL.
```

Cause. You specified the -make_import_lib option, telling the name of the import library that you want to create, and you also specified one other filename on the command line. That means that you are creating an import library to represent some other (ordinary) DLL. The other filename you put on the command line should be the DLL to be represented. It wasn't a DLL, so that's an error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create an import library to represent a single (ordinary) DLL, then the other filename on the command line should be that DLL. If you are not trying to create an import library, then don't specify the -make_import_lib option.

```
1041 The symbol named <symbol name> was found both in
<filename> and in <filename>.
```

Cause. You specified the -make_import_lib option, in order to create the zimpimp file that represents the multiple implicit DLLs that constitute system library, and you also specified those implicit DLLs on the command line. There are some symbols that legitimately exist in more than one of the implicit DLLs, because they are specially marked by the C++ compiler, but other symbols may only come up in one of the implicit DLLs. The indicated symbol was illegally present in more than one of the implicit DLLs, so this is an error.

Effect. Error (The linker immediately stops).

Recovery. This indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1042 The -make_import_lib option was specified, but there
were no other filenames on the command line to tell the
DLL(s) to use as inputs.
```

Cause. You specified the -make_import_lib option, telling the name of the import library that you want to create, but you didn't specify any other filenames on the command line. That is an error.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to create a zimpimp file, which represents the multiple implicit DLLs that constitute system library, or if you are trying to create an import file to represent another (ordinary) DLL, then the names of one or more DLLs are required on the command line. If you are not trying to do either of these things, you should not specify the -make_import_lib option.

1043 Cannot open output workfile <filename>.

Cause. You are trying to create an import library. eld tries to create a workfile in the same location (OSS directory, Guardian subvolume, or PC folder) as the place where you specified that the import library should be created. eld could not create that workfile and open it for writing.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you have permission to create files in the indicated location, and that it isn't a Guardian subvolume that is full.

1044 Error seeking in output file <filename>.

Cause. You are trying to create an import library. eld first creates a workfile in the same location (OSS directory, Guardian subvolume, or PC folder) as the place where you specified that the import library should be created. For some reason, eld had a problem writing to the workfile, after it had been successfully created.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that the indicated location isn't in a Guardian subvolume that is full. If that doesn't explain the problem, it probably needs to be reported to your HP representative.

1045 Error writing output file <filename>.

Cause. You are trying to create an import library. eld first creates a workfile in the same location (OSS directory, Guardian subvolume, or PC folder) as the place where you specified that the import library should be created. For some reason, eld had a problem writing to the workfile, after it had been successfully created.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that the indicated location isn't in a Guardian subvolume that is full. If that doesn't explain the problem, it probably needs to be reported to your HP representative.

```
1046 Could not set file code; deleting import library
<filename>.
```

Cause. You are trying to create an import library on Guardian, and after creating the import library `eld` tried to set its file code to 800, but was unable to do so.

Effect. Error (The linker cannot do what was requested of it and stops, deleting the file that had been created up to that point).

Recovery. If you are not able to set the file code of a file, that is an NSK question that is beyond the scope of this document.

```
1047 Error attempting to delete file <filename>.
```

Cause. This message comes out after message 1046 (see above). It means that, after being unable to set the file code of the file, and then trying to delete the file, `eld` was also unable to delete the file.

Effect. Fatal error (`eld` immediately stops, and the output file has not been deleted).

Recovery. If you are not able to delete a file that you just created, that is an operating system question that is beyond the scope of this document.

```
1048 Cannot create -temp_i file <filename>.
```

Cause. When you specify the `-temp_i` option, `eld` still first creates a temporary file in another place, and when that file is created `eld` then tries to rename it to the filename specified in the `-temp_i` option. That renaming failed.

Effect. Warning (`eld` still creates the import library, but not using the file you specified with the `-temp_i` option as an intermediate file).

Recovery. If you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1049 Cannot create -make_import_lib file <filename>; object
file name: <filename>.
```

Cause. You are trying to create an import library. `eld` first creates it in a temporary location, deletes any file that previously existed with the name specified for the import library, and then renames the temporary file to the final location. That process failed. The file has instead been left in the place that the message calls the “object file name”.

Effect. Warning (`eld` produces an output file, but not with the filename you intended).

Recovery. If there already was a file with the same name as the file you wanted to create, and you didn't have permission to delete it, either find some other way to delete that old file, or specify a different filename for the import library that you want to create. If there was no file of that name already, and you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1065 Floating point type inconsistency among input linkfiles.
File <filename> specifies 'tandem'.  File <filename>
specifies 'ieee'.
```

Cause. As shown in the message, one of the input object files said it wanted the "Tandem" type of floating point, and another one specified "IEEE", and you didn't specify which one you wanted to take precedence on the `eld` command line. That is an error.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If these files really do need to use the respective types of floating point, then that is impossible, and you'll have to change your source code to use one type of floating point consistently. More often, one or both files specifies a floating point type unnecessarily, because the compilers may do that by default. In that case, use the -set floattype option to specify the type of floating point that `eld` should assume is really needed, or "neutral" if neither type is required.

```
1081 Cannot open <filename>: <reason>.
```

Cause. A filename was specified directly on the command line, for `eld` to open, but either that file doesn't exist or you don't have permission to read it.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Check that you really intended to specify a file of the indicated name, that you spelled it correctly, and that you do have permission to read it.

```
1082 Cannot find <name>.
```

Cause. You specified the -l option, to tell `eld` to search for a DLL or archive based on the string given as the parameter to the -l option, and `eld` could not find that DLL or archive, and you also specified the -allow_missing_libs option, to say that it was not an error if a DLL could not be found.

Effect. Information (This is not indicative of a problem).

Recovery. No action required (assuming you did specify the -allow_missing_libs option on purpose, for a good reason).

```
1083 Cannot find <name>.
```

Cause. You specified the -l option, to tell eld to search for a DLL or archive based on the string given as the parameter to the -l option, and eld cound not find that DLL or archive.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. The rules for searching for DLLs and archives are complicated. For example, the DLL or archive may be present in your current location (Guardian subvolume, OSS directory, or PC folder), but that is not a place where eld looks by default. You can tell eld to look there with the appropriate -L option. More generally, you may need to review all the rules by which eld does the search, to determine where the DLL or archive should be placed, and how eld should be told to look there. You might decide it's easier to just put the fully qualified name of the DLL or archive directly on the command line, without using the -l option at all. You might also decide that you didn't need that DLL or archive anyway. Or, you may be specifying everything as you should, to tell eld how to find the DLL or archive in the location you expect it to be, but the DLL or archive is not there, or you don't have permission to read it.

```
1098 <filename> is a DLL, but the -b static option was in
effect.
```

Cause. eld has found a file, specified on the command line, possibly through a -l option, and opened it, and found that it was a DLL, but the -b static option was in effect at this point on the command line, which says that it is an error if eld finds a DLL as opposed to an archive.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really don't want to find a DLL at this point on the command line, possibly through a -l option, then you need to figure out why eld did find a DLL, rather than an archive. If the DLL was found through a -l option that did a search, you may need to review all the rules for how eld does that search, which is complicated. You might decide it's easier to just put the fully qualified name of the archive directly on the command line, without using the -l option at all. On the other hand, if you really do want to find a DLL, then don't specify the -b static option, or at least not at this place on the command line.

```
1099 <filename> is a DLL, but the filename ends in '.a'.
```

Cause. eld has found a file, specified on the command line, and opened it, and found that it was a DLL, but the name of the file ends in ".a". eld considers that an error, since the convention is to use such filenames for achives rather than DLLs.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. You shouldn't have a DLL whose filename ends in ".a", so you need to fix your build procedure.

```
1100 <filename> is a linkfile, but it was found for a -l
option.
```

Cause. You specified the -l option, to tell `eld` to search for a DLL or archive based on the string given as the parameter to the -l option, and `eld` was able to find and open a file, but the file was a linkfile, such as an object file produced from a compilation, not a DLL or archive. `eld` considers that an error, since the convention is to only use -l options to search for DLLs and archives, not linkfiles.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Perhaps there is a DLL or archive that you wanted `eld` to find, but instead `eld` first found a linkfile of the same name. The rules for searching for DLLs and archives are complicated. You may need to review all the rules by which `eld` does the search, to determine where the DLL or archive should be placed, and how `eld` should be told to look there. You might decide it's easier to just put the fully qualified name of the DLL or archive directly on the command line, without using the -l option at all. If you really did intend to use the linkfile that was found, the easiest thing is to just put its fully qualified name on the command line.

```
1101 <filename> is a linkfile, but it was found as an
indirect DLL.
```

Cause. `eld` was searching for a DLL based on a liblist entry of some other previously opened DLL, to indirectly bring another DLL into this link, and `eld` was able to find and open a file, but the file was a linkfile, such as an object file produced from a compilation, not a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Perhaps there is a DLL that you expected `eld` to find, but instead `eld` first found a linkfile of the same name. The rules for searching for DLLs are complicated. You may need to review all the rules by which `eld` does the search, to determine where the DLL should be placed, and how `eld` should be told to look there.

```
1102 Making an implicit DLL, but <filename> isn't implicit.
```

Cause. You specified the -make_implicit_lib option, to build one of the implicit DLLs that contain the contents of system library. Such implicit DLLs may refer to other implicit DLLs, but not to any other DLLs. However, during this link, `eld` was given another DLL to use, where that other DLL wasn't marked as an implicit DLL, so that's an error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you specified the -make_implicit_lib option for no reason, stop doing it. Otherwise, this indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1103 <filename> is an archive, but it was found as an
indirect DLL.
```

Cause. eld was searching for a DLL based on a liblist entry of some other already opened DLL, to indirectly bring another DLL into this link, and eld was able to find and open a file, but the file was an archive, not a DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Perhaps there is a DLL that you expected eld to find, but instead eld first found an archive of the same name. The rules for searching for DLLs are complicated. You may need to review all the rules by which eld does the search, to determine where the DLL should be placed, and how eld should be told to look there.

```
1104 <filename> is an archive, but the -b dllsonly option was
in effect.
```

Cause. eld has found a file, specified on the command line, possibly through a -l option, and opened it, and found that it was an archive, but the -b dllsonly option was in effect at this point on the command line, which says that it is an error if eld finds an archive as opposed to a DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really don't want to find an archive at this point on the command line, then you need to figure out why eld did find an archive, rather than a DLL. If the archive was found through a -l option that did a search, you may need to review all the rules for how eld does that search, which is complicated. You might decide it's easier to just put the fully qualified name of the DLL directly on the command line, without using the -l option at all. On the other hand, if you really do want to find an archive, then don't specify the -b dllsonly option, or at least not at this place on the command line.

```
1105 <filename> is an archive, but the filename ends '.so'.
```

Cause. eld has found a file, specified on the command line, possibly through a -l option, and opened it, and found that it was an archvie, but the name of the file ends in ".so". eld considers that an error, since the convention is to use such filenames for DLLs rather than archives.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. You shouldn't have an archive whose filename ends in ".so", so you need to fix your build procedure.

```
1106 Using archive: <filename>.
```

Cause. eld prints out informational messages about some of the files that it used and what types of files they were. In this case, it is telling you about an archive.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1107 <archive filename>: member <member name> in archive is
not an object file.
```

Cause. eld used the archive mentioned in the message, and in more detail also tried to use the member of the archive specified in the message, but that member was not a valid TNS/E object file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Perhaps there is something wrong with the archive, so that it looked like a TNS/E archive, but didn't have TNS/E object files in it. In that case, the process that created the archive needs to be examined. Or, perhaps the archive has a mixture of different types of object files in it, and you used the -all option, which tells eld to try to use all the object files in the archive. In that case, you should build a different archive that only contains TNS/E object files.

```
1112 Cannot open output file <filename>: <reason>.
```

Cause. eld tries to create a workfile in the same location (OSS directory, Guardian subvolume, or PC folder) as the place where you specified that the output file should be created. For some reason, eld could not create that workfile and open it for writing.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you have permission to create files in the indicated location, and that it isn't a Guardian subvolume that is full.

```
1117 Cannot open <filename>, the file specified to be the
zimpimp file on the -alf command line.
```

Cause. You have given the -alf option, and in addition to the filename that was the parameter for the -alf option you have also specified another filename on the command line, which tells eld to use that file as the zimpimp file during this -alf option, for resolving references to system library. However, either the file specified as the zimpimp file doesn't exist, or you don't have permission to read it.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Did you really intend to specify a special version of the zimpimp file? If you did, check that you spelled it correctly, and that you do have permission to read it.

1129 No main entry point was specified when creating a program.

Cause. You have asked eld to create a main program. However, you have not specified the -e option, and no procedure within the program has the MAIN attribute. So, eld doesn't know what the main entry point of the program should be, and that's an error.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your program contains C or C++ code then a certain object file is supposed to be included in the link, and if you ask the compiler to invoke eld for you then the compiler will supply that file automatically. If you invoke eld yourself, you need to specify the same object file that the compiler would. For these languages, you should not specify the -e option. You also should not specify the -e option for Cobol. If your program only contains pTAL code then you can either put the MAIN attribute on the main entry point in your pTAL source code, or you can specify the name of that procedure to eld with the -e option. I

1131 Name specified in the -e option not found: <symbol name>.

Cause. You used the -e option to tell eld the name of the procedure that is supposed to be the main entry point for the program that you are building, but there is no symbol of that name defined within the program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your program contains C, C++, or Cobol code, you shouldn't be using the -e option at all. If your program only contains pTAL code then you can use the -e option to tell the name of the pTAL procedure that should be the main entry point. Did you spell the name wrong? Note that the pTAL compiler converts all names to upper case, so you need to specify it as upper case to eld, regardless of how you spelled it in the pTAL source code. Also note that, if you misspell some other option beginning with an "e", then eld will interpret that as a "-e" option, possibly leading to this message. There are several such options, as described elsewhere in this manual, and you have to spell them exactly as shown in this manual, except for case. Note that there is no option spelled "-elf" in eld, although there was in nld. If you say "-elf" to eld, eld will believe that you are saying that "lf" is the name of the main entry point. So, don't do that.

```
1132 Input file < filename> cannot be linked into a
globalized DLL; all of the code that went into that file must
be recompiled with the 'globalized' option.
```

Cause. Using the `-b globalized` option requires that all the input object files be compiled with `–Wglobalized` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Compile the input object files with the `–Wglobalized` option.

```
1133 Procedure <symbol name> with the 'main' attribute was
overridden by an -e option for <symbol name>.
```

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If your program contains C, C++, or Cobol code, you shouldn't be using the -e option at all. In that case, this "warning" is probably indicative of a serious error, and you should not specify the -e option. If your program only contains pTAL code then this is okay, if that's really what you meant to do. In other words, one of the procedures in your source code has the MAIN attribute, but you are telling `eld` that you want execution to begin someplace else.

```
1136 Conflicting values given for 'interpose_user_library'.
```

Cause. You gave the -set interpose_user_library option more than once on the command line, with different attribute values. (The possible values are "on" and "off".) You can give this option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

```
1138 The 'interpose_user_library' attribute is only allowed
if you are creating a DLL.
```

Cause. You used the -set interpose_user_library option, which is an attribute you can set when you build a DLL, but you are building some other type of object file rather than a DLL.

Effect. Error (The linker cannot do was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a DLL to be used as an interpose user library, then specify an appropriate option, such as `-dll`, `-shared`, or `-ul`, to build a DLL. If your intention is to build some other type of object file, not a DLL, then don't specify the -set interpose_user_library option.

```
1141 Floating point type inconsistency.  The -set floattype
option specifies <string>, but the input linkfiles imply
<string>.
```

Cause. The input object files consistently used one type of floating point, i.e., either "Tandem" or "IEEE". On the `eld` command line, you specified the -set floattype option, either saying that the file was the opposite type, or that it was neutral. That's okay, but `eld` puts out an informational message about it.

Effect. Information (This is not indicative of a problem).

Recovery. None required, assuming you know that this object file really should be marked the way you indicated, because you know that is the type of floating point it will really need at runtime.

```
1142 Floating point type inconsistency among input linkfiles.
File <filename> specifies 'tandem'.  File <filename>
specifies 'ieee'.
```

Cause. The input object files contained a mixture of both types of floating point, i.e., "Tandem" and "IEEE". On the `eld` command line, you specified the -set floattype option, to say which type the file really was, or to say it was neutral. That's okay, but `eld` puts out an informational message about it.

Effect. Information (This is not indicative of a problem).

Recovery. None required, assuming you know that this object file really should be marked the way you indicated, because that is the type of floating point it will really need at runtime.

```
1143 Floating  point type inconsistency among input DLL's.
File <filename> specifies 'tandem'.  File <filename>
specifies 'ieee'.
```

Cause. You are building a DLL, and during this link `eld` is also looking at various other DLLs, and among those other DLLs `eld` sees that at least one says that it requires the "Tandem" version of floating point at runtime, while another says that it requires the "IEEE" version of floating point at runtime.

Effect. Warning (`eld` produces an output file, but it thinks that you might want to see this warning about what might go wrong at runtime with other DLLs).

Recovery. None required, because even though the DLLs say they require a specific type of floating point at runtime, that might not really be true. You might want to look into this further, though, with people who are familiar with what those DLLs actually do need.

```
1144 Floating point type inconsistency.  The DLL being
created specifies 'tandem'.  DLL <filename> specifies 'ieee'.
```

Cause. You are building a DLL, and it says that it needs the “Tandem” type of floating point at runtime, but during this link `eld` is also looking at various other DLLs, and among those other DLLs `eld` sees that at least one says that it requires the “IEEE” version of floating point at runtime.

Effect. Warning (`eld` produces an output file, but it thinks that you might want to see this warning about what might go wrong at runtime).

Recovery. If the DLL being built does not actually need the “Tandem” type of floating point, it would be nicer to use the -set floattype neutral option, to say so. However, even if this DLL does require the “Tandem” type of floating point at runtime, there isn’t necessarily a problem here. Even though another DLL says that it requires the “IEEE” version of floating point, that might not really be true. You might want to look into this further, though, with people who are familiar with what that other DLL actually does need.

```
1145 Floating point type inconsistency.  The DLL being
created specifies 'ieee'.  DLL <filename> specifies 'tandem'.
```

Cause. You are building a DLL, and it says that it needs the “IEEE” type of floating point at runtime, but during this link `eld` is also looking at various other DLLs, and among those other DLLs `eld` sees that at least one says that it requires the “Tandem” version of floating point at runtime.

Effect. Warning (`eld` produces an output file, but it thinks that you might want to see this warning about what might go wrong at runtime).

Recovery. If the DLL being built does not actually need the “IEEE” type of floating point, it would be nicer to use the -set floattype neutral option, to say so. However, even if this DLL does require the “IEEE” type of floating point at runtime, there isn’t necessarily a problem here. Even though another DLL says that it requires the “Tandem” version of floating point, that might not really be true. You might want to look into this further, though, with people who are familiar with what that other DLL actually does need.

```
1146 The program being created is floating point type 'ieee'.
DLL <filename> is 'tandem'.
```

Cause. You are building a program, and it says that it needs the “IEEE” type of floating point at runtime, but during this link `eld` is also looking at various other DLLs, and among those other DLLs `eld` sees that at least one says that it requires the “Tandem” version of floating point at runtime.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If the version of the DLL that is used at runtime still says that it requires the “Tandem” type of floating point, your program will not be allowed to run. If the program that you are building does not really require the “IEEE” type of floating point, it would be better for you to specify -set floattype neutral when you build the program. If your program really does need the “IEEE” type of floating point, though, there still may not be a problem here. Even though the DLL says that it requires the “Tandem” version of floating point, that may not really be true. You might want to look into this with people who are familiar with what the DLL actually does need. You can avoid the runtime check and make it possible to run your program despite the apparent inconsistency by specifying the “-set float_lib_overrule on” option.

```
1147 The program being created is floating point type
'tandem' or 'neutral'.  DLL <filename> is 'ieee'.
```

Cause. You are building a program, and it either says that it needs the “Tandem” type of floating point at runtime, or that it is “neutral”, but during this link `eld` is also looking at various other DLLs, and among those other DLLs `eld` sees that at least one says that it requires the “IEEE” version of floating point at runtime.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If the version of the DLL that is used at runtime still says that it requires the “IEEE” type of floating point, your program will not be allowed to run. Even though your program may call itself “neutral”, that still is interpreted as meaning “Tandem” at runtime, and NSK considers this to be inconsistent with what the DLL says. If you know that all the DLLs used by the program say that they are either neutral or “IEEE”, then you could lie by specifying the -set floattype ieee option to say that your program also wants “IEEE”, and be able to run. But even if your program really does need the “Tandem” type of floating point, that still may not mean there is a real problem here, because even though a DLL says that it needs the “IEEE” type of floating point, that may not be true. You might want to look into this with people who are familiar with what the DLL actually does need. If you believe that it does make sense to run your program, you can avoid the runtime check and make it possible to run your program by specifying the “-set float_lib_overrule on” option.

```
1148 The implicit DLL <filename> has not been preset.
```

Cause. You specified the -make_import_lib option, in order to create the zimpimp file that represents the multiple implicit DLLs that constitute system library, and you also specified those implicit DLLs on the command line. One of the checks that `eld`

performs on these implicit DLLs is that they are “preset”, which would have to be true if that implicit DLL was correctly fixed up at the time it was linked. But, the implicit DLL mentioned in the message was not preset.

Effect. Fatal error (The linker cannot do what was requested of it and the linker immediately stops).

Recovery. This indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1150 File <filename> is C++ dialect v2; file <filename> is
C++ dialect v3.
```

Cause. As shown in the message, one of the input object files said it was using C++ dialect v2, and another said it was using dialect v3. Such a mixture is not allowed.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Some of the source files need to be compiled, so that they are all using the same dialect of C++.

```
1151 The -set cplusplusdialect option specified <string>, but
the input linkfiles implied <string>.
```

Cause. The input files use C++, which means that they are either dialect v2 or v3, as shown in the message, and you specified the -set cppdialect neutral option, to indicate that the program or DLL that you are building should say that it doesn’t use C++. That is okay, but `eld` puts out an informational message about it.

Effect. Information (This is not indicative of a problem).

Recovery. No action required, assuming you know that it would be okay for this program or DLL to be included in a process where the program or other DLLs were using the other dialect of C++. `eld` did not see any DLLs of the opposite C++ dialect during this link, or else `eld` would have put out a warning message about that, but `eld` doesn’t necessarily see all other DLLs that will be in the process at runtime, and if you are building a DLL then `eld` similarly doesn’t see the program that will be in the process.

```
1152 The loadfile being built has C++ dialect v2; DLL
<filename> has C++ dialect v3.
```

Cause. The input files use the v2 dialect of C++, and during the link `eld` saw a DLL that was using the v3 dialect of C++.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. In special cases, different loadfiles within the same process may be able to use different dialects of C++. You may wish to check with someone who is familiar with the DLL mentioned in the message, to see if there might be problems at runtime due to this dialect inconsistency.

```
1153 The loadfile being built has C++ dialect v3; DLL
<filename> has C++ dialect v2.
```

Cause. The input files use the v3 dialect of C++, and during the link `eld` saw a DLL that was using the v2 dialect of C++.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. In special cases, different loadfiles within the same process may be able to use different dialects of C++. You may wish to check with someone who is familiar with the DLL mentioned in the message, to see if there might be problems at runtime due to this dialect inconsistency.

```
1155 Could not create workfile of the form <string>.
```

Cause. Before `eld` creates an output file, it first writes to a temporary workfile that is in the same location (Guardian subvolume, OSS directory, or PC folder) as the final file to be created. `eld` chooses a filename for the workfile of a certain pattern, as shown in the message, where three of the characters will be filled in with a number from 000 to 999. `eld` was not able to create any file whose name matched this pattern, perhaps because 1000 files with names matching the pattern already existed. They might exist because previous runs of `eld` failed and left these files behind. `eld` doesn't delete such files that previously existed, it only tries to find an unused name.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Check that you have permission to create files in the indicated location, and that it isn't a Guardian subvolume that is full, and also see whether there already are 1000 files in existence, with names matching the pattern. If so, delete some of them.

```
1156 No input files.
```

Cause. Based on the options you specified, you are trying to use `eld` to create a new object file from a set of existing object files. However, no existing object files were specified on the command line, to be included in the output file. Perhaps you specified DLLs on the command line, but DLLs are only referenced during the link, not included in the output file. Perhaps you specified archives on the command line, but the object files within an archive are only included in the link if the -all option is in effect, or if a previous object file already in the link refers to a symbol that an archive member can provide.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to combine one or more existing object files into a new object file, put their names on the command line. If you want to do something else, such as stripping the symbols from an existing object file (for example), you need to put the appropriate option for that purpose on the command line, and of course you need to spell it correctly.

1174 Output file <filename> and temporary file <filename> must be in the same directory/subvolume (and spelled the same way on the eld *command line).*

Cause. You specified one of the -temp_i, -temp_o, or -temp_r options, to tell the name that eld should use for an intermediate file during the process of creating some other file, as shown in the message. You also specified the name of the temporary file as a qualified name (not a simple name). When you do that, the name that you specified for the temporary file is required to be in the same location (Guardian subvolume, OSS directory, or PC folder) as the final file that you are trying to create, and you must spell them both exactly the same way for eld to understand this. But, the locations didn't match.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really want to use the -temp_* option, there is no need to specify a fully qualified name. You can just specify a simple name, and then eld will put it in the right place. But you probably also have no need for the -temp_* option at all, so you can just omit it.

1176 The -rpath or -rld_L option is not allowed with the -r option.

Cause. You used the -rpath or -rld_L option, to specify a place where NSK will look for DLLs at runtime, and you also used the -r option, to tell eld to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the -r option. If your intention is to use the -r option to create an object file that can be used again as eld input, then don't specify the -rpath or -rld_L option.

1177 The -rld_first_L option is not allowed with the -r option.

Cause. You used the -r_first_L option, to specify a place where NSK will look for DLLs at runtime, and you also used the `-r` option, to tell `eld` to build a another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -rld_first_L option.

1178 The -limit_runtime_paths option is not allowed with the -r option.

Cause. You used the -limit_runtime_paths option, which affects the way NSK looks for DLLs after it loads the program you are building, and you also used the -r option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program, then don't specify the -r option. If your intention is to use the -r option to create another object file that can be used as `eld` input, then don't specify the -limit_runtime_paths option.

1179 The -soname or -dllname option is only allowed when you are building a DLL.

Cause. You used the -soname or `-dllname` option, which tells the "DLL name" to be placed inside the DLL that you are creating, but you did not specify any option to tell `eld` to build a DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a DLL, then specify an option to do that, such as the `-dll`, `-shared`, or `-ul` option. If that is not your intention, then don't specify the -soname or `-dllname` option.

1180 The -set cplusplusdialect option is not allowed with the -r option.

Cause. You used the -set cppdialect neutral option, which say that the program or DLL that you are building does not need either dialect of C++, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set cppdialect option.

1181 The -must_use_iname option is only allowed when you are making an import library.

Cause. You used the -must_use_iname option, which tells `eld` that it should be an error if it cannot create an import library with the name specified for it, but you did not specify any option to tell `eld` to build an import library.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create an import library, specify an appropriate option to do that. If that is not your intention, don't specify the -must_use_iname option.

1182 The -temp_i option is only allowed when you are making an import library.

Cause. You used the `-temp_i option`, to tell the name that `eld` should use for an intermediate file during the process of creating an import library, but you did not specify any option to tell `eld` to build an import library.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create an import library, specify an appropriate option to do that. If that is not your intention, don't specify the `-temp_i` option.

1183 You can only specify a user library name for a program.

Cause. You used the `-libname` or `-set libname` option, to tell `eld` the name of the user library to place within the program that you are creating, but you have also used some other option to tell `eld` to build another object file that can be used as linker input, or a DLL, but not a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program, which happens by default, don't specify an option such as `-r`, `-dll`, or `-ul` that would tell `eld` to do something else. If you are not creating a program, don't specify the `-libname` or `-set libname` option.

1184 The 'highpin' attribute is not allowed with the -r option.

Cause. You used the -set highpin option, which is an attribute you can set when you build a program or DLL, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set highpin option.

```
1185 The 'highrequestors' attribute is not allowed with the -
r option.
```

Cause. You used the -set highrequestors option, which is an attribute you can set when you build a program or DLL, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set highrequestors option.

```
1186 Cannot rebase the two segments by different amounts
because this file contains PC-relative relocations that go
between the code and data segments.
```

Cause. You have specified the `-alf` option, together with the `-t` and `-d` options, to rebase a DLL and move the two segments (the code segment and data segment) by different amounts. Within this DLL, there are cases where, in one segment, a word contains a value that is a self-relative offset into the other segment. Our TNS/E compilers never generate such code, but if somehow you get such code into your DLL, that can work. However, such words would need updating if the two segments were rebased by different amounts, and the `-alf` option does not support that, so it reports this as an error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you compile code with some other compiler, and manage to get it to go through `eld` to create this DLL? We don't recommend doing that. In any case, no matter how you created this DLL, you cannot rebase the two segments by different amounts with the `-alf` option. Relink the DLL, correctly specifying the two segment addresses that you want the first time, instead of using the `-alf` option to change them.

1187 The 'runnamed' attribute is not allowed with the -r option.

Cause. You used the -set runnamed option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set runnamed option.

1188 The 'saveabend' attribute is not allowed with the -r option.

Cause. You used the -set saveabend option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set saveabend option.

1189 The 'inspect' attribute is not allowed with the -r option.

Cause. You used the -set inspect option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set inspect option.

1190 The 'heap_max' attribute is not allowed with the -r option.

Cause. You used the -set heap_max option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set heap_max option.

1191 The 'mainstack_max' attribute is not allowed with the -r option.

Cause. You used the -set mainstack_max option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set mainstack_max option.

1192 Both the -import_lib and -import_lib_stripped options were specified.

Cause. You have specified the -import_lib and the -import_lib_stripped options. The -import_lib option tells `eld` to make an import library for a DLL, at the same time `eld` is building that DLL, and not to strip the DWARF symbols from the import library. The -import_lib_stripped option also says to make an import library, but to strip the DWARF symbols. `eld` will only make one import library, and considers the -import_lib and -import_lib_stripped options to be inconsistent with each other.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Specify only one of -import_lib and -import_lib_stripped, depending on what you want to do.

1193 The 'space_guarantee' attribute is not allowed with the -r option.

Cause. You used the -set space_guarantee option, which is an attribute you can set when you build a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set space_guarantee option.

1194 The 'rld_unresolved' attribute is not allowed with the -r option.

Cause. You used the -set rld_unresolved option, which is an attribute you can set when you build a program or DLL, to tell later invocations of the `-alf` option, and rld, how they should handle unresolved references in that program or DLL, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the -set rld_unresolved option.

1195 Specified the -t and/or -d option with the -r option.

Cause. You used the `-t` or `-d` option, when building a program or DLL, to tell the starting addresses of its segments, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to create another object file that can be used as `eld` input, then don't specify the `-t` or `-d` option.

1198 The value for the -t option was rounded up to <number>.

Cause. You used the `-t` option, which specifies the starting address of the code segment of the program or DLL being built. It was rounded up to a multiple of 64K bytes (or, 128K bytes if you are building an implicit DLL).

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. The starting address of the code segment of a program or DLL is required to have the indicated alignment. No action is required if you understand that and are satisfied with the rounding, although it would be cleaner if you specified a number with the right alignment in the first place. If this doesn't make sense to you, because you

don't understand the purpose of the -t option, read the documentation or contact HP for more detailed advice. Perhaps you intended to make some code section come out at a particular location, but there is no direct way to do that.

1199 The value for the -d option was rounded up to <number>.

Cause. You used the -d option, which specifies the starting address of the data segment of the program or DLL being built. It was rounded up to a multiple of 64K bytes (or, 128K bytes if you are building an implicit DLL).

Effect. Warning (eld produces an output file, but it might not be what you intended).

Recovery. The starting address of the data segment of a pjrogram or DLL is required to have the indicated alignment. No action is required if you understand that and are satisfied with the rounding, although it would be cleaner if you specified a number with the right alignment in the first place. More likely, there was no reason for you to use this option in the first place. If this doesn't make sense to you, because you don't understand the purpose of the -d option, read the documentation or contact HP for more detailed advice. Perhaps you intended to make some data section come out at a particular location, but there is no direct way to do that.

1200 Specified the -ansistreams option, but not building a program.

Cause. You used the -ansistreams option, which affects how the program that you are building will do its I/O, but the file that you have told eld to create is not a program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to create a program, specify that correctly. For example, don't specify the -dll, -shared, or -ul option, which means that you are telling eld to build a DLL, rather than a program. And don't specify the -r option, which tells eld that you are building another object file that can be used as input to eld, rather than a program. Or, if you don't intend to create a program, then don't specify the -ansistreams option.

1201 Specified the -nostdfiles option, but not building a program.

Cause. You used the -nostdfiles option, which affects how the program that you are building will do its I/O, but the file that you have told eld to create is not a program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to create a program, specify that correctly. For example, don't specify the -dll, -shared, or -ul option, which means that you are telling eld to build a DLL, rather than a program. And don't specify the -r option, which tells eld that you are building another object file that can be used as input to eld, rather than a

program. Or, if you don't intend to create a program, then don't specify the -nostdfiles option.

1202 The -make_implicit_lib option is only allowed when you are creating a localized DLL.

Cause. You specified the `-dll`, `-shared`, or `-ul` option, together with the -b globalized or -b semiglobalized option, to build a globalized or semiglobalized DLL, and you also specified the -make_implicit_lib option, to tell `eld` to make one of the implicit DLLs that constitute system library. That's not allowed, because the implicit DLLs are never created as globlaized or semiglobalized DLLs.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you are not building one of the implicit DLLs that constitute system library, then don't specify the -make_implicit_lib option. If you are, then don't specify the -b globalized or -b semiglobalized option.

1206 The -shared or -ul option is not allowed with the -r option.

Cause. You specified the `-shared`, `-dll`, or `-ul` option, which are all ways of telling `eld` to create a DLL. And you also specified the `-r` option, which tells `eld` to create another object file that can be used as linker input, not a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to build a DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to build.anoter object file that can be used as input to `eld`, then don't specify the `-dll`, `-shared`, or `-ul` option.

1208 The 'interpose_user_library' attribute is only allowed for DLLs.

Cause. You gave the `-change interpose_user_library` option, to tell `eld` that the DLL you are updating is to be marked as an interpose user library, but the filename specified with the `-change` option is not a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to update a file that is not a DLL then don't specify the -set interpose_user_library option. If you intention is to update a DLL then you must specify the name of a DLL.

1210 <filename>: unresolved reference to <symbol name>.

Cause. eld is building a program or DLL, and the program or DLL makes a reference to the symbol mentioned in the message, but eld was unable to find a copy of that symbol, either in the program or DLL being built, or in any other DLL that was looked at during the link. This may occur for many reasons, ranging from spelling errors in your source code, or things that you still need to write that you don't yet have in your source code, to problems with files that other people are supposed to provide to you, which either they didn't provide or you didn't pass along for eld to use, or "standard" things not set up correctly in your installation.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. You don't necessarily need to resolve all references at link time. A program may run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, you need to look at the names of the symbols that eld said it couldn't find, and see if they exist somewhere. The message also told you the name of the input object file that had the reference to the symbol, and the name of a code or data section within that object file where the reference occurred, and the offset of the reference within that code or data section. That symbol may be in a DLL, for example, that eld wasn't using, so you need to supply those DLLs to eld. eld will print out informational messages about all the DLLs that it used if you supply the -verbose option. A symbol in a DLL also needs to be exported from that DLL for eld to find it. A symbol might also be a member of an archive, but the archive needs to come later on the command line than the reference to the symbol to guarantee that eld finds the symbol in the archive. If you have unresolved references, to get an error-free link you need to specify either -unres_symbols warn (to change these messages into just warning messages) or -unres_symbols ignore (to not get any messages at all).

```
1211 <filename>: unresolved reference to <symbol name>.
```

Cause. eld is building a program or DLL, and the program or DLL makes a reference to the symbol mentioned in the message, but eld was unable to find a copy of that symbol, either in the program or DLL being built, or in any other DLL that was looked at during the link. This may occur for many reasons, ranging from spelling errors in your source code, or things that you still need to write that you don't yet have in your source code, to problems with files that other people are supposed to provide to you, which either they didn't provide or you didn't pass along for eld to use, or "standard" things not set up correctly in your installation.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. You don't necessarily need to resolve all references at link time. A program may run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, you need to look at the names of the symbols that eld said it couldn't find, and see if they exist somewhere. The message also told you the name of the input object file that had the reference to the symbol.

That symbol may be in a DLL, for example, that eld wasn't using, so you need to supply those DLLs to eld. eld will print out informational messages about all the DLLs that it used if you supply the -verbose option. A symbol in a DLL also needs to be exported from that DLL for eld to find it. A symbol might also be a member of an archive, but the archive needs to come later on the command line than the reference to the symbol to guarantee that eld finds the symbol in the archive. If you have unresolved references, to get an error-free link you need to specify either -unres_symbols warn (to change these messages into just warning messages) or -unres_symbols ignore (to not get any messages at all)

```
1212 Cannot create -temp_o file <filename>.
```

Cause. You are creating a new object file out of a set of input files. eld creates a temporary output file, before creating the real output file. When you specify the -temp_o option, eld still first creates the temporary file in another place, and when that file is created eld then tries to rename it to the filename specified in the -temp_o option. That renaming failed. The temporary file that was created, and the filename that you specifed in the -temp_o option, are both in the same location (Guardian subvolume, OSS direction, or PC folder) as the new object file that you are trying to create.

Effect. Warning (eld still creates the output object file, but not using the file you specified with the -temp_o option as an intermediate file).

Recovery. If you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1213 Cannot create -o file <filename> by renaming the
temporary file <filename>.
```

Cause. eld first creates the output object file in a temporary location, deletes any file that previously existed with the name specified for the output object file, and then renames the temporary file to the final location. That process failed. The file has instead been left in the place that the message calls the "temporary file".

Effect. Warning (eld produces an output file, but not with the filename you intended).

Recovery. If there already was a file with the same name as the file you wanted to create, and you didn't have permission to delete it, either find some other way to delete that old file, or specify a different filename for the object file that you want to create. If there was no file of that name already, and you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1214 The implicit DLL <filename> has unresolved references.
```

Cause. You specified the -make_import_lib option, in order to create the zimpimp file that represents the multiple implicit DLLs that constitute system library, and you also specified those implicit DLLs on the command line. One of the checks that eld performs on these implicit DLLs is that they don't have unresolved references, which would have to be true if that implicit DLL was correctly fixed up at the time it was linked. But, the implicit DLL mentioned in the message had unresolved references.

Effect. Fatal error (The linker cannot do what was requested of it and the linker immediately stops).

Recovery. This indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1215 The implicit DLL <filename> has unresolved references.
```

Cause. You specified the -make_import_lib option, in order to create the zimpimp file that represents the multiple implicit DLLs that constitute system library, and you also specified those implicit DLLs on the command line. One of the checks that eld performs on these implicit DLLs is that they don't have unresolved references, which would have to be true if that implicit DLL was correctly fixed up at the time it was linked. But, the implicit DLL mentioned in the message had unresolved references. It is only a warning message, not an error message, because the -unres_symbols warn option was also specified.

Effect. Warning (eld produces an output file, but it might not be what you intended).

Recovery. This issue involves the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1227 Multiple definition of <symbol name>: first definition
found in <filename>, second definition found in <filename>.
```

Cause. You specified the -show_multiple_def option, to tell eld to put information into the listing about symbols that were defined more than once. This message reports the first two definitions that eld saw for the symbol named in the message.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1228 Multiple definition of <symbol name>: third or later
definition found in <filename>.
```

Cause. You specified the -show_multiple_def option, to tell eld to put information into the listing about symbols that were defined more than once. There was an instance of message number 1227 earlier, which reported the first two definitions that eld saw for

the symbol named in this message. After that, there is an instance of this message number 1228 for each additional definition that `eld` sees for the symbol.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1229 Definition of <symbol name> from file <filename> is
overriding common of the same name.
```

Cause. `eld` encountered global symbols of the same name, of which one is common data and the other is a definition. The definition overrides the common data.

Effect. The application does not execute properly. See warning 1657 on page 6-124.

Recovery. Ensure that the application does not include duplicate global symbols of different sizes. See warning 1230 on page 6-38 to know the name of the common data and the file it is present in.

```
1230 Common <symbol name> is in file <filename>.
```

Cause. Multiple global symbols of the same name <*symbol name*> are present in the link files, of which common data is present in <*filename*>.

Effect. For informational purposes only. (might indicate a problem, see warning 1657 on page 6-124).

Recovery. See the recovery step of warning 1229 on page 6-38.

```
1231 Common of <symbol name> in file <file name> is
overridden by definition.
```

Cause. `eld` encountered global symbols of the same name, of which one is common data and the other is a definition. The definition overrides the common data.

Effect. The application does not execute properly. See warning 1657 on page 6-124.

Recovery. Ensure that the application does not include duplicate global symbols of different sizes. For more information about the name of the common data and its location, see warning 1232 on page 6-38.

```
1232 Common of <symbol name> is defined in file <file name>.
```

Cause. Multiple global symbols of the same name <*symbol name*> are present in the link files, of which common data is present in <*filename*>.

Effect. For informational purposes only (might indicate a problem, see warning 1657 on page 6-124).

Recovery. See the Recovery step of warning 1231 on page 6-38.

```
1233 <file name>: common of <symbol name> overridden by
larger common.
```

Cause. `eld` encountered global symbols of the same name, <*symbol name*>, both of which are common data items and are of different size. The symbol of the larger size is retained, overriding the smaller common data item.

Effect. Application might not function as expected.

Recovery. Correct the application to not include duplicate global symbols of different sizes. To know the name of the file in which the larger common data is present, see Warning 1234 on page 6-39.

```
1234 <file name>: larger common is here.
```

Cause. Multiple global symbols of the same name are present in the link files, of which larger common data is located in <*file name*>.

Effect. Information (might indicate a problem, see warning 1233).

Recovery. See Recovery step of Warning 1233 on page 6-39.

```
1235 <file name>: common of <symbol name> overriding smaller
common.
```

Cause. `eld` encountered global symbols of the same name, <*symbol name*>, both of which are common data items and are of different sizes. The symbol of the larger size is retained, overriding the smaller common data item.

Effect. Application might not function as expected.

Recovery. Correct the application to not include duplicate global symbols of different sizes. To know the name of the file in which the larger common data is present, see Warning 1234 on page 6-39.

```
1236 <file name>: smaller common is here.
```

Cause. Multiple global symbols of the same name are present in the link files, of which smaller common data is located in <*file name*>.

Effect. Information (might indicate a problem, see Warning 1235 on page 6-39).

Recovery. See Recovery step of Warning 1235 on page 6-39.

1254 <filename>: unresolved reference to <symbol name>.

Cause. eld is building a program or DLL, and the program or DLL makes a reference to the symbol mentioned in the message, but eld was unable to find a copy of that symbol, either in the program or DLL being built, or in any other DLL that was looked at during the link. This might occur for many reasons, ranging from spelling errors in your source code, or things that you still need to write that you do not yet have in your source code, to problems with files that other people are supposed to provide to you, which either they did not provide or you did not pass along for eld to use, or "standard" things not set up correctly in your installation.

Effect. Warning (eld produces an output file, but it might not be what you intended).

Recovery. You don't necessarily need to do anything. A program may run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, you need to look at the names of the symbols that eld said it couldn't find, and see if they exist somewhere. The message also told you the name of the input object file that had the reference to the symbol, and the name of a code or data section within that object file where the reference occurred, and the offset of the reference within that code or data section. That symbol may be in a DLL, for example, that eld wasn't using, so you need to supply those DLLs to eld. eld will print out informational messages about all the DLLs that it used if you supply the -verbose option. A symbol in a DLL also needs to be exported from that DLL for eld to find it. A symbol might also be a member of an archive, but the archive needs to come later on the command line than the reference to the symbol to guarantee that eld finds the symbol in the archive. Depending on the situation, you may be able to use the -unres_symbols option to specify whether eld should consider unresolved references to be errors, warnings, or neither.

1255 <filename>: unresolved reference to <symbol name>.

Cause. eld is building a program or DLL, and the program or DLL makes a reference to the symbol mentioned in the message, but eld was unable to find a copy of that symbol, either in the program or DLL being built, or in any other DLL that was looked at during the link. This may occur for many reasons, ranging from spelling errors in your source code, or things that you still need to write that you don't yet have in your source code, to problems with files that other people are supposed to provide to you, which either they didn't provide or you didn't pass along for eld to use, or "standard" things not set up correctly in your installation.

Effect. Warning (eld produces an output file, but it might not be what you intended).

Recovery. You don't necessarily need to do anything. A program may run correctly, even if it has unresolved references at link time. But, you may prefer that your link be clean. In that case, you need to look at the names of the symbols that eld said it couldn't find, and see if they exist somewhere. The message also told you the name of the input object file that had the reference to the symbol. That symbol may be in a DLL, for example, that eld wasn't using, so you need to supply those DLLs to eld.

eld will print out informational messages about all the DLLs that it used if you supply the -verbose option. A symbol in a DLL also needs to be exported from that DLL for eld to find it. A symbol might also be a member of an archive, but the archive needs to come later on the command line than the reference to the symbol to guarantee that eld finds the symbol in the archive. Depending on the situation, you may be able to use the -unres_symbols option to specify whether eld should consider unresolved references to be errors, warnings, or neither.

1266 <filename>: reference to <symbol name>.

Cause. You used the -y option, to ask eld to provide information about the symbol mentioned in the message, and eld is telling you that there is a reference to that symbol in the indicated file.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1267 <filename>: definition of <symbol name>.

Cause. You used the -y option, to ask eld to provide information about the symbol mentioned in the message, and eld is telling you that there is a definition of that symbol in the indicated file.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1273 Bad input file: relocation table entry <number> for the <section name> section of input file <filename> is a local reference to the <section name> section. That is an error because the <section name> section has been omitted from the link, since it only contains an unneeded duplicate copy of a procedure.

Cause. Procedures are contained within code “sections”. The C++ compiler may create duplicate copies of procedures, and eld can remove the duplicates if the compiler gives permission. When this is done, the entire section containing the unneeded copy of the procedure is removed. There may be references to the procedure, but those references should be done by giving the name of the procedure, and they will be fixed up to another copy of the procedure that was not removed. It is also possible to make references to something within a code section by telling, not the name of a symbol, but rather the name of the section and the offset within the section. If there are such references into a section, the C++ compiler should not give eld permission to remove it, because there would be no way to fix up the references. That is what has happened. The message tells what input file this happened in, and where

the reference was (in terms of its index in a table of relocation sites for some input section), and which section it was pointing at (i.e., the section that was removed).

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming this file was created by the C++ compiler, that indicates a compiler error, to be reported to HP, or perhaps incorrect usage of the compiler. ENOFT can provide more information about the parts of the object file mentioned in the message.

```
1276 <filename> contains the same DLL name as <filename>, and
is therefore being ignored.
```

Cause. While doing this link, and looking at other DLLs to resolve references, `eld` has come upon the same DLL twice, where "same" means that it contains the same "DLL name" inside it. These DLLs may have been found from items placed on the command line, or indirectly through the liblist entries of other DLLs, or a combination of these things. When looking for indirect DLLs, it is perfectly reasonable to find the same DLL more than once, and usually `eld` is silent about that. In this case, `eld` put out a warning message, because `eld` has not found the same file twice, nor two files that look like copies of each other. In other words, it looks like there are two different DLLs that just happen to have the same DLL name inside them, which is probably not something you meant to do. The determination of whether two DLLs "look like copies of each other" is based on whether they contain the same "export digest" within them. Having the same export digest means that `eld` could use either copy of the DLL and the resulting fixups of the file being created would come out exactly the same.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. You should figure out what these two DLLs are and decide which one you probably wanted `eld` to find, and which one you didn't want `eld` to find, and change your build procedure accordingly. You may want to use both of them, and to do that you need to give them different DLL names, using the -soname option when you build those DLLs.

```
1280 Can't open obey file <filename>.
```

Cause. You used the -obey option, but `eld` could not open the filename you specified.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Check that you really intended to specify a file of the indicated name, that you spelled it correctly, and that you do have permission to read it.

```
1281 Unmatched double quotes in obey file.
```

Cause. You used the -obey option, and `eld` was reading that obey file, and `eld` found a quotation mark (double quote character) within that obey file, either at the beginning of a line or after white space in the middle of a line. `eld` assumes that a token begins

after this quotation mark, and it terminated by the next quotation mark on the line, but there was no additional quotation mark on the line.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Find the offending quotation mark in the obey file. What did you intend to have there? You may need to review the rules in the eld manual about how eld treats quotation marks in obey files.

1282 A string starting with an equal sign is not allowed as the parameter for the <option name> option.

Cause. You specified an option such as -L or -rpath, where the parameter is supposed to be the name of a Guardian subvolume, OSS directory, or PC folder, or you specified an option such as -soname, where the parameter is supposed to be a DLL name. The parameter you specified began with an equal sign, but eld does not allow parameters to begin with equal signs in these cases.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really intended to give that kind of a name in these contexts, you can't. Choose another name that doesn't begin with an equal sign.

1283 A string starting with an equal sign is only allowed as the filename parameter for the <option name> option in the Guardian version of eld*.*

Cause. You specified an option with a parameter, where the parameter began with an equal sign. In the Guardian case this would be allowed, where eld would treat the parameter as a "Guardian DEFINE", and expand it to a filename But, you are running eld on the PC or on OSS, and on these platforms this type of parameter is not allowed.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really wanted to use a name that begins with an equal sign in this context, you can't. On the PC or OSS, the Guardian DEFINE mechanism is not present, so you need to directly specify the intended filename.

1284 The string <string> could not be expanded as a MAP DEFINE.

Cause. You specified an option with a parameter, where the parameter began with an equal sign, and you are running eld on Guardian. In this situation, eld tries to treat the parameter as a "Guardian DEFINE". However, when eld tried to look up the name as a Guardian DEFINE, that failed.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you have issued a MAP DEFINE prior to running `eld`, where this MAP DEFINE equated that string that began with an equal sign to a filename. Also check that DEFMODE is ON.

```
1285 The MAP DEFINE <string> was expanded to <string>.
```

Cause. You specified a string that began with an equal sign on the command line, and you are running the Guardian version of `eld`. `eld` treated the string as a “Guardian DEFINE”, and succeeded in expanding it to a filename. The message shows you how it was expanded.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1286 Parameter required for <option name>.
```

Cause. You specified the option named in the message, and this option requires a keyword parameter, but the option itself was the last thing on the command line, with no keyword parameter after it.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1287 Keyword parameter required for <option name>.
```

Cause. You specified the option named in the message, and this option requires a keyword parameter, but the next token on the command line began with a hyphen, indicating another option, not the parameter for this one.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1288 Number required for <option name>.
```

Cause. You specified the option named in the message, and this option requires a number as a parameter, but the option itself was the last thing on the command line, with no parameter after it. Or, it is also possible to get this message if the next thing on the command line was a string such as “0x”, which is the prefix signifying the start of a hexadecimal number, but immediately after the “0x” there was just a space, or the end of the command line.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1289 Badly formed number for <option name>.
```

Cause. You specified the option named in the message, and this option requires a number as a parameter, but the next token following the option contained characters other than the allowed decimal or hexadecimal digits, depending on the situation.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1291 A user library name must be of the form
'$vol.subvol.file'.
```

Cause. You used the `-libname`, `-set libname`, or `-change libname` option to specify the name of a user library. A user library name must always have the form $a.b.c, but either the name you specified didn't start with a dollar sign, or it didn't contain two periods.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Are you sure you want to specify a user library name? If so, and you entered the user library name incorrectly, fix it. Note that it must not contain a system name.

```
1292 Each of the volume, subvolume, and file name portions of
the user library name must have at most eight characters.
```

Cause. You used the `-libname`, `-set libname`, or `-change libname` option to specify the name of a user library. A user library name must always be a Guardian filename specified in the form $a.b.c. `eld` has complained because at least one of "a", "b", and "c" was longer than 8 characters.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Are you sure you want to specify a user library name? If so, and you entered the user library name incorrectly, fix it.

```
1293 Invalid value for <attribute name>; 'on' or 'off' is
required.
```

Cause. You gave a -set highpin, -set highrequestors, -set runnamed, -set inspect, -set saveabend, -set float_lib_overrule, -set oktosettype, -set interpose_user_library, or -set user_buffers option. For each of these cases, the next token on the command line must be either "on" or "off", to tell the value for this -set attribute. However, the next token on the command line was something other than these two possibilities.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1294 Invalid value <string> given for 'cplusplusdialect': .
```

Cause. You gave a -set cppdialect option. The next token on the command line must be either “neutral” or (synonymously) “cppneutral”, to tell the (only allowed) value for this -set attribute. However, the next token on the command line was something other than these two possibilities.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

```
1295 Duplicate procedure <symbol name> in <filename> is
callable.
```

Cause. There are multiple copies of the indicated procedure, and in particular at least the copy of the procedure in the indicated file has the CALLABLE or KERNEL_CALLABLE attribute. These attributes are not allowed for procedures that have multiple copies. Procedures with these attributes are used to write system code, and we believe it is better to only have one copy of such a procedure.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you didn’t really intend to write a CALLABLE (or KERNEL_CALLABLE) procedure, remove that attribute from your source code. If you do want to write such a procedure, make sure you only have one copy of it.

```
1296 When a program is marked floating point neutral, it
really means tandem floating point at runtime.
```

Cause. You gave the -set floattype neutral option when building a program. That is allowed, but questionable. In fact, at runtime, a program is never “neutral”. It is always set up to use either the Tandem or IEEE form of floating point. So, it would probably be better for you to explicitly specify which one you want to have. As the message says, you said “neutral”, but that really means “Tandem” at runtime.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If some DLL that is used at runtime says that it requires the “IEEE” type of floating point, your program will not be allowed to run (unless you also specify “-set float_lib_overrule on”). But there may not be a real problem here, because even though a DLL says that it needs the “IEEE” type of floating point, that may not be true. You might want to look into this with people who are familiar with what the DLL actually does need. As already mentioned, this runtime check will be avoided if you also specify the “-set float_lib_overrule on” option. Or, if you know that other DLLs used by the program say that they are “IEEE”, or neutral, and none of them say “Tandem”, then you will be able to run if you say -set floattype ieee, rather than -set floattype neutral.

1297 Invalid value for 'floattype': <string>.

Cause. You gave a -set floattype or -change floattype option. The next token on the command line must be either “tandem”, “ieee”, or “neutral” (or the longer synonyms “tandem_float”, etc.), to tell the value for this attribute. However, the next token on the command line was something other than these possibilities.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

1298 Conflicting values given for 'floattype'.

Cause. You gave the -set floattype option more than once on the command line, with different attribute values. (The possible values are “tandem”, “ieee”, and “neutral”, or the longer synonyms “tandem_float”, etc.) You can give the option more than once, but only if you specify the same (or a synonymous) value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1299 Conflicting values given for 'float_lib_overrule'.

Cause. You gave the -set float_lib_overrule option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1300 Conflicting values given for 'highpin'.

Cause. You gave the -set highpin option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

```
1301 Conflicting values given for 'highrequestors'.
```

Cause. You gave the -set highrequestors option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

```
1302 Conflicting values given for 'inspect'.
```

Cause. You gave the -set inspect option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

```
1304 Conflicting values given for 'runnamed'.
```

Cause. You gave the -set runnamed option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

```
1305 Conflicting values given for 'saveabend'.
```

Cause. You gave the -set saveabend option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1306 Conflicting values given for 'heap_max'.

Cause. You gave the -set heap_max option more than once on the command line, with different numbers specified as the attribute value. You can give the option more than once, but the numerical value that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1307 Conflicting values given for 'mainstack_max'.

Cause. You gave the -set mainstack_max option more than once on the command line, with different numbers specified as the attribute value. You can give the option more than once, but the numerical value that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1308 Conflicting values given for 'space_guarantee'.

Cause. You gave the -set space_guarantee option more than once on the command line, with different numbers specified as the attribute value. You can give the option more than once, but the numerical value that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1309 Only 'on' is valid with 'incomplete'.

Cause. You gave a -set incomplete or `-change` incomplete option. The next token on the command line must be “on”, to tell the (only allowed) value for this attribute. However, the next token on the command line was something other than this.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

1310 Conflicting values given for 'libname'.

Cause. You gave the `-libname` or `-set libname` option(s) more than once on the command line, with different filenames specified for the user library. You can give these options more than once, but the name that you specify must be the same each time (except for lower case versus upper case).

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide what name you want to specify, and only specify that name.

1311 Conflicting values given for 'process_subtype'.

Cause. You gave the -set process_subtype option more than once on the command line, with different numbers specified as the attribute value. You can give the option more than once, but the numerical value that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1312 Invalid value for 'rld_unresolved': <string>.

Cause. You gave a `-set rld_unresolved` or `-change rld_unresoloved` option. The next token on the command line must be either “error”, “warn”, or “ignore”, to tell the value for this attribute. However, the next token on the command line was something other than these possibilities.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

1313 Conflicting values given for 'rld_unresolved'.

Cause. You gave the `-set rld_unresolved` option more than once on the command line, with different attribute values. (The possible values are “error”, “warn”, and “ignore”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1314 Invalid value for 'systype': <string>.

Cause. You gave a `-set systype` or `-change systype` option. The next token on the command line must be either “guardian” or “oss”, to tell the value for this attribute. However, the next token on the command line was something other than these possibilities.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

1315 Conflicting values given for 'systype'.

Cause. You gave the -set systype option more than once on the command line, with different attribute values. (The possible values are “guardian” and “oss”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1316 Invalid attribute for <option name> option: <string>.

Cause. You gave the -set or `-change` option on the command line. The next token on the command line must be one of the possible attributes that you can set or change with this option, but `eld` didn’t recognize the attribute specified.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the spelling of the attribute. Case doesn’t matter, but other than that it must be exactly as shown in the manual.

1317 Multiple, inconsistent specifications for import control.

Cause. You gave more than one of the options named -b localized, -b globalized, and -b semi_globalized (or, -b symbolic, a synonym for -b semi_globalized). You can give the same option more than once if you wish, including synonyms for the same option, but otherwise you must not specify more than one of these options.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify one of these options, decide which one you want to specify, and only specify that one, not any other ones along with it.

1318 The -alf option was given more than once.

Cause. You specified the `-alf` option more than once on the command line. That is not allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to run multiple `-alf` options, do each one in a separate `eld` command.

1320 Illegal duplicate definition of the data item <symbol name> in <filename> and <filename> because they are different sizes.

Cause. Each of the two files mentioned in the message defined data items of the same name, as shown in the message. Various rules apply to such a situation. One rule is that the two copies of the data item must have the same size (or, they can each have size 0, which is a way for the compiler to not actually tell what the size is). But, in this case, the sizes were different.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define data items with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you are using the same data item in more than one place, only one of those places needs to be a definition, and the other places can just be external references to that definition. Review the rules for what makes a declaration a definition, depending on the source language that you are using, because the rules are different for each language. If you really do intend to have two definitions of this data item, visible across separate compilations, then the sizes must be the same. If the sizes are different because the two files were created from different versions of the source code, or by using different compiler options, repeat the compilations doing things more consistently. If the two copies of the symbol are in different source languages, you may need to review the rules for how different language compilers lay out data, to get them to both give the data item the same size.

1321 Unrecognized option: <string>.

Cause. You specified the indicated option, but `eld` didn't recognize it.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the spelling of the option. Case doesn't matter, but other than that it must be exactly as shown in the manual.

1322 Unrecognized parameter to the -b option.

Cause. You gave the -b option on the command line. The next token on the command line must be one of the possible keyword parameters that you can give with this option,

but eld didn't recognize the parameter specified. (The possibilities are "localized", "globalized", "semi_globalized", "symbolic", "static", "dynamic", and "dllsonly".)

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Fix the spelling of the parameter. Case doesn't matter, but other than that it must be exactly as shown in the manual.

1323 One or more options were given that are not allowed with the -change option.

Cause. You gave the -change option. Along with a -change option, the only other options that are allowed are -no_banner, -no_verbose, -obey, -stdin, -verbose, -vslisting, and -warn. But, you specified some other option besides these.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Remove the offending option.

1324 You specified the name of a DLL registry, but that is only allowed if you are (a) building a DLL, (b) updating a DLL with the -alf option, or (c) building the zimpimp file with the -make_import_lib option.

Cause. You gave the -update_registry or -check_registry option, telling the name of a private DLL registry to use during the link. A private DLL registry is used to calculate the address at which to place a DLL. It is only allowed when you are doing one of the things listed in the message, and you weren't doing any of those things.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to do one of the things listed in the message, provide the appropriate option to specify that. If you are not doing any of those things, do not specify the -update_registry or -check_registry option.

1325 The -map option is not allowed with the -r option.

Cause. You used the -map option, which provides information about addresses in the program or DLL that you are creating, and you also used the -r option, to tell eld to build a another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or a DLL, then don't specify the -r option. If your intention is to use the -r option to create another object file that can be used as eld input, then don't specify the -map option.

1326 Multiple specifications of -soname or -dllname options with different filenames.

Cause. You gave the `-dllname` option (or its synonym, the -soname option) more than once on the command line. This option tells the "DLL name" to be placed inside the DLL that you are creating, but you specified different DLL names. You can give this option more than once, but only if you specify the same DLL name. Also note that case is significant for this check.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which DLL name you want to put inside the DLL, and only specify that name.

1327 On OSS, if you specify -local_libname, you must either also specify -set libname, or else the string specified for -local_libname must be in the Guardian namespace, to be used as the name of the user library at runtime.

Cause. You are using the OSS version of `eld`, and you are building a program that uses a user library. You specified the `-local_libname` option, to tell `eld` where the user library currently exists, for `eld` to fix up refernces to it during the link. It is also necessary for `eld` to place a Guardian file name for the user library within the program being built, to tell where the user library will exist in the Guardian namespace at runtime. You could specify this with the `-libname` option, but you didn't. In this case, `eld` tries to use the name you specified for the `-local_libname` option, converting it to the form of a Guardian filename. But, for that to work, the name you specify with the `-local_libname` option needs to be a Guardian file (i.e., it needs to be a name such as /G/a/b/c). But, it wasn't, so that's the error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you don't intend for your program to use a user library, don't specify the `-local_libname` option. If you are using a user library, you must decide what its Guardian filename will be at runtime. If you move the file there now, and specify it with the format `-local_libname` /G/a/b/c, that will work. Or, if the file isn't there now, you can tell `eld` where it will be in the future with an option of the form `-libname` $a.b.c, in addition to still giving the `-local_libname` option, to tell `eld` where the file is now.

1328 The specified input file <filename> is a program.

Cause. The options that you gave to `eld` said that you wanted to build a new object file out of existing object files. Those object files would be object files, such as those created by a compilation, perhaps found in archives, and you can also tell `eld` about DLLs to look at during this process. But, the file mentioned in the message, whose name you put on the command line, is none of those things, but rather is a program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to build a new object file out of existing files, to be able to use this new object file as eld input, then don't specify the name of a program on the command line. You can't add more stuff to an existing program, you can only rebuild a different program from scratch. The options that you can apply to an existing program are -alf, -change, and -strip. If you want to do one of these things, specify the appropriate option.

1329 The parameter of the -local_libname option could not be converted to Guardian format.

Cause. You are using the Guardian version of eld, and you are building a program that uses a user library. You specified the -local_libname option, to tell eld where the user library currently exists, for eld to fix up refernces to it during the link. However, the name that you specified for the -local_libname option was not a valid Guardian filename. (This particular message comes out when you specify the -local_libname option and you don't also specify the -libname option. Otherwise, a different message would appear, saying that eld could not open the file specified for the -local_libname option, although the underlying problem would be the same.)

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you want to specify this option then you must enter a valid Guardian filename for its parameter.

1330 The -e option was given multiple times with different procedure names: <symbol name> and <symbol name>.

Cause. You gave the -e option more than once on the command line. This option tells eld the name of the procedure that should be where execution begins for the program you are building, but you specified different names at different places on the command line. You can give this option more than once, but only if you specify the same name each time. Also note that case is significant for this check.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Are you sure you want to specify the -e option at all? There usually is no need to give this option, and specifying it is a mistake. You can validly use it to tell the main entry point for a pTAL program, if you didn't put the MAIN attribute into the source code. But you must only specify one such name with -e options. Also note that, if you misspell some other option beginning with an "e", then eld will interpret that as a "-e" option, possibly leading to this message. There are several such options, as described elsewhere in this manual, and you must spell them exactly as described in this manual, except for case. Note that there is no option spelled "-elf" in eld, although there was

in nld. If you say “-elf” to `eld`, `eld` will believe that you are saying that “lf” is the name of the main entry point. So, don’t do that.

1331 <attribute name> has been changed to <attribute value> in file <filename>.

Cause. You specified a `-change` option, and this message tells you that it succeeded, changing a certain attribute within a certain file to a certain value, as shown in the message.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1332 The -t and -d options cannot be given when using a DLL registry.

Cause. You specified the `-t` or `-d` option, to explicitly tell `eld` what the starting address should be for the text or data segment. You also specified the -check_registry or -update_registry option, to tell `eld` that it should figure out what address to give the DLL that you are creating based on what a private DLL registry says. You can’t do both at the same time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to use a private DLL registry to decide what the DLLs address should be, don’t specify the `-t` or `-d` option. If you want to explicitly tell `eld` where to place the new DLL, without using a private DLL registry, then the `-t` or `-d` options can be used to do that, although there is rarely any reason to use the `-d` option. If you want to build a DLL, and explicitly say where it should go, and also have a private DLL registry updated with that information, you can use the `-t` option to build the DLL, and separately put that information into the private DLL registry file by hand. Or, you can put the information by hand into the private DLL registry first, and then tell `eld` to do what the registry says by using the -check_registry option.

1333 The <option name> option can only be given when you are updating a DLL registry.

Cause. You specified either the -temp_r option, the -must_use_rname option, or one of the options whose names begin “-grow”. These options relate to how `eld` chooses an address for a DLL and updates a private DLL registry accordingly. However, you did not specify the -update_registry option, which tells `eld` about a private DLL registry that it should update.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to have `eld` create or update a private DLL registry with information about the DLL that you are building, specify the -update_registry option. If

you are not trying to do that, then don't specify the -temp_r option, the -must_use_rname option, or any option whose name begins "-grow".

```
1334 The -check_registry and -update_registry options cannot
both be given.
```

Cause. You specified both the -check_registry and -update_registry options. The -check _registry option tells eld that it must assign a DLL an address as specified in a private DLL registry, and the registry is unchanged. The -update_registry option tells eld to use information in the registry to decide what address to use for the DLL, and then update the registry accordingly. You can't do both at the same time.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you want to use a private DLL registry, decide which of the two ways you want to be using it, and specify the correct option accordingly.

```
1335 Cannot specify -grow_limit with other -grow* options.
```

Cause. You have specified the -update_registry option, to tell eld to calculate the address for a DLL and store it in a private DLL registry, also allocating a certain amount of space for future growth of that DLL. You specified the -grow_limit option, which tells eld what that future size should be. But you also specified the -grow_data_amount, -grow_text_amount, or -grow_percent option, and these options are used to tell eld which formula to use to calculate the future growth if the -grow_limit option is not given. You can't give the -grow_limit option at the same time as any of these other options whose names begin "-grow".

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Decide what formula you want eld to use, to calculate the amount of future growth of this DLL, and specify the proper options accordingly, as explained elsewhere in this manual.

```
1336 <filename> is a linkfile.  Only -change floattype is
valid for a linkfile.
```

Cause. You gave the -change option, telling the name of the file that you want to modify, and the name of the attribute that you want to modify inside that file. The file that you specified is a linkfile, such as a file created by a compilation, not a program or a DLL. The only attribute that you are allowed to modify in a linkfile is "floattype", but you specified a different attribute from that.

Effect. Fatal error (eld immediately stops without modifying the file).

Recovery. The attribute that you specified cannot set or modified in linkfiles. To set up that attribute value in a program or DLL, you must give it in the eld command that creates that program or DLL.

```
1337 Multiple values specified for the -instance_data option.
```

Cause. You gave the `-instance_data` option more than once on the command line, and you specified a different value for it each time. You can give this option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without modifying the file).

Recovery. The `-instance_data` option is a speical option that should only be used when you know why you are using it. If you need to use it, decide which value you need to specify, and specify only that value.

```
1338 Error opening DLL registry <filename>.
```

Cause. You specified the -check_registry or -update_registry option, and the file that you specified as the private DLL registry does exist, but you don't have permission to open it for reading. The problem is not that the file was in use by another link, because `eld` would give a different message about that.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Check that you spelled the name correctly and that you do have permission to read it.

```
1339 The <option name> option would be allowed if you were
making a zimpimp file, but because the DLL on the command
line isn't an implicit DLL, you are instead making an
ordinary import library, in which case this option isn't
allowed.
```

Cause. You specified the -make_import_lib option, which means that you are either making an import library to represent a single DLL, or that you are making the zimpimp file that represents all the implicit DLLs that constitute system library. You also specified either the -temp_r, -must_use_rname, -check_registry, -update_registry, or `-t` option, and these are options that are allowed when you are creating the zimpimp file, but not when creating other kinds of import libraries. On the other hand, you also put the name of a DLL on the command line, and this DLL isn't an implicit DLL, but only implicit DLLs are allowed when you are creating the zimpimp file. So, this set of conditions is inconsistent.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a zimpimp file, then the other filenames on the command line should be implicit DLLs. If you are trying to create an import library to represent a single (ordinary) DLL, then don't use the option that was mentioned in the message. If you are not trying to create an import library, don't specify the -make_import_lib option.

1340 The <option name> option would be allowed if you were making an ordinary import library, but because the DLL's on the command line are implicit DLL's, you are instead making a zimpimp file, in which case this option isn't allowed.

Cause. You specified the -make_import_lib option, which means that you are either making an import library to represent a single DLL, or that you are making the zimpimp file that represents all the implicit DLLs that constitute system library. You also specified either the -s, -x, or -set incomplete option, and these are options that are only allowed when you are creating some other kind of import library, not the zimpimp file. On the other hand, the DLLs that you listed on the command line are implicit DLLs, and that's only allowed when you are creating the zimpimp file. So, this set of conditions is inconsistent.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a zimpimp file, then don't use the option that was mentioned in the message. If you are trying to create an import library to represent a single DLL, then that should be the only DLL listed on the command line, and it should not be an implicit DLL. If you are not trying to create an import library, don't specify the -make_import_lib option.

1341 <filename>: <message about why this input file contains something that would lead to the existence of variable data in the output file>, which is not allowed when the -make_implicit_lib option is specified.

Cause. You have specified the -make_implicit_lib option, which means that you are building one of the DLLs that constitute system library. These DLLs are not allowed to contain writeable data. However, the input file mentioned in the message contained writeable data.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming you really do want to specify the -make_implicit_lib option, you need to examine your input files to determine why they contain variable data, modify them, and recompile them, to fix that.

1342 Unrecognized parameter to the -instance_data option.

Cause. You gave the `-instance_data` option on the command line. The next token on the command line must be one of the parameters that you can specify for this option (the possibilities are "data1", "data2", "data2protected", "data2hidden", and "data1constant"), but `eld` didn't recognize the parameter specified.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. The `-instance_data` option is a speical option that should only be used when you know why you are using it. If you need to use it, decide which value you need to specify, and spell it correctly. Case doesn't matter, but other than that it must be exactly as shown in the manual.

```
1343 <filename> is neither a linkfile, DLL, or program.
```

Cause. You gave the `-change` option, but the file that you specified for this option is not any kind of valid TNS/E object file.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Use the correct object file.

```
1345 Multiple specifications of the -local_libname option
with different filenames.
```

Cause. You gave the `-local_libname` option more than once on the command line, and you specified a different filename parameter for it each time. You can give this option more than once, but only if you specify the same filename each time.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. The `-local_libname` option tells the name of a user library that `eld` will use during this link to fix up references from the program that it is creating. There can only be one user library. If you have a user library, specify its name on the command line, and no other name. If you use this option to specify the name more than once, it must be written exactly the same way each time.

```
1346 Multiple specifications of the <option name> option with
different filenames.
```

Cause. You gave the -import_lib, the -import_lib stripped, or the -make_import_lib option more than once on the command line. Each of these options takes a filename as a parameter, and you specified different names each time that option was used. You can give each of these options more than once, but only if you specify the same filename each time for that option.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. These are three different options that tell `eld` to create an import library, and the parameter tells the name of that import library. Decide what name you want to give the import library, and specify just that name on the command line.

1348 DLL registry <filename> not found.

Cause. You gave the -check_registry option, to tell eld the name of an existing private DLL registry for eld to use during the link, but no such file exists.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you spelled the name of the private DLL registry correctly.

1349 You are building a program or a DLL, and you have both resident code and writeable data. To build such a file you are required to specify the -instance_data data2 option. Except, if you are building a 'proto-process' (also known as a 'sysgen'ed process'), then instead you should specify the -data_resident option.

Cause. The program or DLL that you are building contains writeable data and you have not specified the -instance_data option with an appropriate parameter value to cause eld to separate the writeable data from other data so that NSK can give them different types of protection. Also, you have resident code in your program or DLL, and you have not specified the -data_resident option, to tell NSK that the data for this program needs to be resident. That combination of conditions is considered an error.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Yes, this is tricky. Most users who have both writeable data and resident code should specify the -instance_data data2 option to make things work. It is beyond the scope of this manual to explain when you might instead prefer to specify -instance_data data2protected or -instance_data data2hidden, which is the other possible choice for the -instance_data option that also makes eld create two data segments. And, none of this is the right thing to do for certain special programs that are called “proto-processes” or “sysgen’ed processes”. It is similarly beyond the scope of this manual to explain what these things are, but if you are creating a proto-process then you should know that you are doing that. In that case, do not specify the -instance_data option, but rather the -data_resident option.

1350 A zimpimp file (<filename>) is not allowed when you are building an implicit DLL.

Cause. You have specified the -make_implicit_lib option, to make one of the implicit DLLs that constitute system library. When an implicit DLL is created, eld does not use the zimpimp file, which is the import library that represents the entire set of implicit DLLs after they have all been created. So, when you are making an implicit DLL, eld doesn’t try to find the zimpimp file on its own. However, one of the files that eld found, when it was looking for normal DLLs, turned out to be a zimpimp file. That’s the error.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. This indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

1351 Conflicting verbosity options: <string> and <string>.

Cause. The -verbose, -warn, and -no_verbose options are mutually exclusive, and you specified more than one of them. You can specify the same one of these options more than once if you wish, but you cannot specify different ones as part of the same `eld` invocation.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. These options tell `eld` which messages you want to see, other than error messages, which you cannot turn off. Decide which option you want and only specify that one.

1353 DLL registry <filename> is in use. Giving up.

Cause. You specified the -check_registry or -update_registry option, to tell `eld` to use a private DLL registry. Such a file did exist, and the first thing that `eld` tried to do with it was to open the file for reading. However, `eld` could not do that, because the file was being used exclusively by another process, perhaps by another link that was using the -update_registry option. In this situation, `eld` waits and tries again, and that should work. If that doesn't work after a certain number of tries, then `eld` gives up, resulting in this message.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Look into the status of the file that you specified. Perhaps someone has it opened for an extended period of time in an editing session. Run `eld` again when it is possible for `eld` to read that file.

1355 Multiple specifications of the -o option with different filenames.

Cause. You gave the -o option more than once on the command line, specifying different filenames. You can give this option more than once, but only if you specify the same filename each time.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. This option tells the name for the main object file that `eld` is creating Decide what name you want, and specify just that name on the command line. Also note that, if you misspell some other option beginning with an "o", then `eld` will interpret that as a "-o" option, possibly leading to this message. The only other valid

option names beginning with an “o” are “-obey” and “-optional_lib”, and they must be spelled exactly that way, except for case.

1356 Options to specify how unresolved references should be handled would be allowed if you were making a zimpimp file, but because the DLL on the command line is not an implicit DLL, you are instead making an ordinary import library, in which case such options are not allowed.

Cause. You specified the -make_import_lib option, which means that you are either making an import library to represent a single DLL, or that you are making the zimpimp file that represents all the implicit DLLs that constitute system library. You also specified either the -unres_symbols, -error_unresolved, or -warning_unresolved option, and these are options that are allowed when you are creating the zimpimp file, but not when creating other kinds of import libraries. On the other hand, you also put the name of a DLL on the command line, and this DLL isn’t an implicit DLL, but only implicit DLLs are allowed when you are creating the zimpimp file. So, this set of conditions is inconsistent.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a zimpimp file, then the other filenames on the command line should be implicit DLLs. If you are trying to create an import library to represent a single (ordinary) DLL, then don’t use the option that was mentioned in the message. If you are not trying to create an import library, don’t specify the -make_import_lib option.

1357 <filename>: <message about why this input file contains something that would lead to the existence of variable data in the output file>, which is not allowed when -instance_data data1constant is specified.

Cause. You have specified the `-instance_data data1constant` option, which means that you want `eld` to consider it an error if the program or DLL that you are building contains any writeable data. And, indeed, the input file mentioned in the message contains writeable data.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming you really don’t want to have any writeable data, you need to examine your input files to determine why they contain variable data, modify them, and recompile them, to fix that.

1358 The -instance_data option cannot be specified with the -r option.

Cause. You used the `-instance_data` option, which affects the layout of the program or DLL that you are creating, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL, then don't specify the `-r` option. If your intention is to use the `-r` option to tell `eld` to make another object file that can be used as `eld` input, then don't specify the `-instance_data` option.

1359 DLL registry line <number>: encountered end of file when expecting a number.

Cause. While reading the private DLL registry that was specified for this link in the -check_registry or -update_registry option, `eld` ran off the end of the file when it was expecting that the next thing in the file would be a number. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

1360 Two symbol names are required for the -rename option.

Cause. You gave the `-rename` option, which requires two symbol name parameters. However, the next two things on the command line were not symbol name parameters. Either there were no symbol name parameters, or there was only one, and then it was either the end of the command line or the next token on the command line started with a hyphen, indicating another option rather than another parameter to this one.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, fix the syntax.

1361 DLL registry line <number>: number expected but not found.

Cause. While reading the private DLL registry that was specified for this link in the -check_registry or -update_registry option, at the line indicated in the message, `eld` expected to see a number, but the next thing in the file was not a number with the correct format. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

1362 DLL registry line <number>: encountered a bad character within a number.

Cause. While reading the private DLL registry that was specified for this link in the -check_registry or -update_registry option, at the line indicated in the message, `eld` expected to see a number, and the next thing in the file looked like the beginning of a number, but then invalid characters were seen. A number needs to be followed by white space to show where the number ends. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

1363 The same symbol, <symbol name>, was specified twice in the same -rename option.

Cause. You gave the `-rename` option, to tell `eld` how it should change the definition of a symbol of one name into the definition of a symbol of another name, but you specified the same symbol name twice.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, the two symbol names must be different.

1365 A lone filename (<filename>) is not allowed on the command line with the -strip option.

Cause. You gave the `-strip` option, to tell `eld` to remove the DWARF symbols from an existing program or DLL. The parameter to the `-strip` option is the name of the file to be stripped. When you give the `-strip` option, you can also give the -o, -obey, and -temp_o options, which also have filenames as parameters. However, except for these reasons, you are not allowed to have any other filenames on the command line.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Why did you specify the filename mentioned in the message? Assuming it was for one of the possible reasons listed above, provide the proper syntax.

1369 Multiple specifications of the -temp_i option with different filenames.

Cause. You gave the -temp_i option more than once on the command line. This option tells the name of a file to use as a temporary file while creating an import library. You can give this option more than once, but only if you specify the same filename.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, only specify one filename.

```
1370 Multiple specifications of the -temp_o option with
different filenames.
```

Cause. You gave the -temp_o option more than once on the command line. This option tells the name of a file to use as a temporary file while creating the output file. You can give this option more than once, but only if you specify the same filename.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, only specify one filename

```
1371 With -make_implicit_lib, -instance_data must be
data1constant.
```

Cause. You have specified the -make_implicit_lib option, which means that you are building one of the DLLs that constitute system library. These DLLs are not allowed to contain writeable data. In other words, the `-instance_data data1constant` option is imposed, to say that the file cannot contain any writeable data. You can specify the `-instance_data` option if you wish, but only if you specify data1constant as the parameter, and you specified something else.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming you really do want to specify the -make_implicit_lib option, do not specify the `-instance_data` option with a parameter value other than data1constant.

```
1372 Cannot create -o file <filename>.
```

Cause. You are trying to create an object file. `eld` first creates it in a temporary location, deletes any file that previously existed with the name specified for the object file, and then renames the temporary file to the final location. That process failed. In such a situation, `eld` would usually leave the file in another location and tell you about it. However, you also specified the -must_use_oname option, to say that it should be considered an error if the file could not be created in the named location. So, that is what happened.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If there already was a file with the same name as the file you wanted to create, and you didn't have permission to delete it, either find some other way to delete that old file, or specify a different filename for the object file that you want to create. If there was no file of that name already, and you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document..

1373 Unrecognized parameter to the -unres_symbols option.

Cause. You gave the -unres_symbols option on the command line. The next token on the command line must be one of the possible keyword parameters that you can give with this option, but `eld` didn't recognize the parameter specified. (The possibilities are "error", "warn", and "ignore".)

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the spelling of the parameter. Case doesn't matter, but other than that it must be exactly as shown in the manual.

1374 Multiple specifications of the -temp_r option with different filenames.

Cause. You gave the -temp_r option more than once on the command line. This option tells the name of a file to use as a temporary file while creating a private DLL registry. You can give this option more than once, but only if you specify the same filename.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, only specify one filename.

1375 Multiple specifications of the <option name> option with different values.

Cause. The option mentioned in the message takes a numerical parameter. You gave this option more than once on the command line, with different numbers specified as the parameter value. You can give the option more than once, but the numerical value that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1376 Multiple specifications of the -update_registry option with different values.

Cause. You gave the -update_registry option more than once on the command line, with different names given for the parameter, to tell the name of the private DLL registry file. You can give the option more than once, but the filename that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which filename you want to specify, and only specify that one.

```
1377 Multiple specifications of the -check_registry option
with different values.
```

Cause. You gave the -check_registry option more than once on the command line, with different names given for the parameter, to tell the name of the private DLL registry file. You can give the option more than once, but the filename that you specify must be the same each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which filename you want to specify, and only specify that one.

```
1380 DLL registry line <number>: two -dllarea commands in the
registry file.
```

Cause. The private DLL registry that was specified for this link in the -check_registry or -update_registry option contained more than one `-dllarea` command. That is not allowed. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

```
1381 The -make_implicit_lib option is only allowed when
creating a new DLL.
```

Cause. You did not give an option such as `-dll`, `-shared`, or `-ul`, which would tell `eld` that you want to make a DLL, but you specified the -make_implicit_lib option, which tells `eld` more specifically what kind of DLL to make, namely, one of the implicit DLLs that constitute system library.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is not to build one of the implicit DLLs that constitute system library, then don't specify the -make_implicit_lib option. If you are trying to do that, then this indicates some problem with the procedure for building and installing the NSK operating system, which is beyond the scope of this document.

```
1382 The <option name> option is not allowed with the
-make_import_lib option.
```

Cause. The *-make_import_lib* option creates an import library to represent one or more other DLLs that already exist. Only certain other options are allowed in conjunction with the -make_import_lib option, and the one mentioned in the error message is not one of them.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you are not trying to create an import library to represent one or more other DLLs that already exist, don't specify the -make_import_lib option. If you are trying to do that, don't specify the other option mentioned in the error message.

```
1389 The symbol named <symbol name> is a cross_dll_cleanup
symbol, but that's not allowed because <reason>.
```

Cause. The symbol <*symbol name*> is a cross_dll_cleanup symbol but is not allowed because of any of the following reasons:

- It is an initerm symbol
- It has the CALLABLE attribute
- It has the MAIN attribute
- Not all copies of the <*symbol name*> are marked STO_MULTIPLE_DEF_OK.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that the attributes of the <*symbol name*> are as required. If not, update the attributes to comply with the rules that are mandated by the -cross_dll_cleanup option. If the attributes are not as required, the -cross_dll_cleanup option cannot be used.

```
1391 Input file <file name> was not compiled with the -
Wglobalized option, but all input object files must be
compiled with this option when the -cross_dll_cleanup option
is used.
```

Cause. Using the -cross_dll_cleanup option requires that all the input object files be compiled with -Wglobalized option.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Compile the input object files with the -Wglobalized option.

```
1393 Multiple procedures with the 'main' attribute: <symbol
name> and <symbol name>.
```

Cause. The MAIN attribute is one of the attributes that can be assigned to a procedure within an object file. That can be done by using the appropriate syntax in pTAL or

Cobol source code, and can also be done by a link a link step that uses both the -e and `-r` options. The procedure so marked is the one where execution begins, when your object file is linked into a program. `eld` found two different procedures that were each marked this way, as indicated in the message, and that is an error.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If you were linking together two different pTAL object files, each of which had a procedure with the MAIN attribute, are you sure you wanted to do that? It usually makes sense to have one of these, if you are writing pTAL, but not more than one. If you used a link with the -e and `-r` options to create an object file with a procedure marked with the MAIN attribute, are you sure you wanted to do that? There usually is no reason to do that. Did you include more than one copy of the special object file that C and C++ include when creating a main program? If so, get rid of the extra copies of that file. Did you include an object file created by pTAL, containing a procedure with the MAIN attribute, when you were building a program whose main language was C, C++, or Cobol? If so, don't do that. Or, if you really do want to link together the object files you specified, despite the fact that they contain more than one procedure with the MAIN attribute, you can do that with the -allow_mulitple_mains option. In that case, `eld` will choose one of those procedures to be the one where execution begins, so you would need to check that `eld` chose the one you wanted. `eld` will choose the "first" one it sees, which you can control by changing the order of object files on the command line, assuming the different procedures with the MAIN attribute are in different object files.

```
1394 A procedure with the 'main' attribute cannot be included
in a DLL: <symbol name>.
```

Cause. The MAIN attribute is one of the attributes that can be assigned to a procedure within an object file. For example, this can be done by using the appropriate syntax in pTAL or Cobol source code. The procedure so marked is the one where execution begins, when your object file is linked into a program. But you also specified, with an option such as `-dll`, `-shared`, or `-ul`, that you wanted `eld` to build a DLL, not a program. That is considered an error.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a DLL, then you cannot have any procedure with the MAIN attribute in your input object files. If you really want those object files in your DLL, rebuild them so that they don't contain any procedures with the MAIN attribute. Perhaps, when building your DLL, you incorrectly included the special object file that C, C++, and Cobol include when creating a main program? If so, leave out that object file. Or, if you want to build a program, rather than a DLL, don't specify an option such as `-dll`, `-shared`, or `-ul`, that tells `eld` to create a DLL.

```
1395 <filename>: <procedure name> is in <section name>, which
is not a code section.
```

Cause. This message indicates a problem with the contents of the specified input filename, so that `eld` refuses to process the file.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. This is a bug that needs to be reported to HP. Was the specified input file created by a compilation, or by a previous link step? That would indicate which tool created the bad file, and therefore had a bug that needs to be fixed.

```
1447 <output filename>: the range of data that requires GP-
relative addressing is too large (<number> >= 0x400000).
```

Cause. The data in the program or DLL that you are creating is divided into various data "sections". Some of the sections are called "GP-relative", meaning that the data is to be addressed by adding an offset to the contents of the GP register. A single region of data is reserved for this purpose, and all the GP-relative data sections need to fit within it. Some of these GP-relative sections come directly from compilations, because the compiler decides that the data can be referenced by GP-relative addressing. There also are tables created by the linker, to be referenced by GP-relative addressing, related to the number of different symbols to which you have references. Also, if you used the `-instance_data` option, specifying two data segments, that causes data to be rearranged in a way that requires more data to fit in the area reserved for GP-relative addressing. In any case, at most 4 megabytes of data can be in the area reserved for GP-relative addressing, and your program or DLL has exceeded that.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. You need to make your program or DLL smaller, so that it contains less data, or makes references to fewer symbols, or perhaps you don't need to use the `-instance_data` option (if you specified that option). Or, you might decide to split a single large DLL into multiple DLLs. Each program and DLL gets its own region of GP-addressable data.

```
1448 <output filename>: the value specified for the GP
register, due to the definition of a symbol named __gp, does
not make it possible to reach all the data that requires
GP-relative addressing.
```

Cause. Some of the data in a program or DLL is called "GP-relative", meaning that it is to be addressed by adding an offset to the contents of the GP register. A single region of data is reserved for this purpose, and all the GP-relative data needs to fit within it. `eld` chooses the value for the GP register so that it can reach all the data in this region, if possible. However, your program or DLL also included a symbol named

__gp. That tells eld to lay out this symbol in the usual way, like any other data item, but then use its address as the value to put in the GP register. After doing this, all the GP-relative data could not be reached from that address.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Do you really need to DEFINE a symbol named __gp? If you want to write code that figures out the value that in the GP register, you can do that just be having a REFERENCE to the symbol named __gp, not a DEFINITION of it. The rules for what makes the declaration of a symbol a "definition", versus a "reference", depend on the source language, and that is beyond the scope of this manual. If you really do need to define the symbol named __gp, that would be for some very special reason that is also beyond the scope of this manual.

1462 Illegal duplicate definition of the data item <symbol name> in <filename> and <filename> because they have different initial values.

Cause. Each of the two files mentioned in the message defined data items of the same name, as shown in the message. Various rules apply to such a situation. One rule is that, if the two copies of the data item are initialized, then they must have the same initial value. However, these two copies of the data item had different initial values. It is also possible, in C++, for a data item to appear to be uninitialized, but in that case it is treated as being initialized with the value 0.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Did you really intend to define data items with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you are using the same data item in more than one place, only one of those places needs to be a definition, and the other places can just be external references to that definition. Review the rules for what makes a declaration a definition, depending on the source language that you are using, because the rules are different for each language. If you really do intend to have two definitions of this data item, visible across separate compilations, then the initial values must be the same. Or, if you are not writing in C++, it would also be possible for some copies of the data item to be uninitialized. If the initial values are different because the two files were created from different versions of the source code, or by using different compiler options, repeat the compilations doing things more consistently.

1463 Illegal duplicate definition of the data item <symbol name> because it is in the <section name> section in <filename> and the <section name> section in <filename>.

Cause. Each of the two files mentioned in the message defined data items of the same name, as shown in the message. Various rules apply to such a situation. One rule is that the two data items must look like the same "kind" of data, which eld judges

according to the name of the “data section” in which the data was placed by the compiler. eld considers the data sections named “.data”, “.data1”, “.sdata”, “.sdata1”, “.bss”, and “.sbss” to all be “standard, writeable” data sections, and different copies of a data item can be in different sections of these names without getting a complaint. The same goes for the “standard, readonly” data sections, which are named “.rdata”, “.srdata”, “.rodata”, “.srodata”, and “.rconst”. Any other data sections are considered special by eld, and if a data item is in such a section then all copies of that data item must be in a section of that same name. Similarly, different copies of a data item can’t disagree about whether they are “writeable” or “readonly”. These rules were violated, as shown in the message.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Did you really intend to define data items with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you really do intend to have two definitions of this data item, visible across separate compilations, then they must be the same kind of data, as explained above. If they are different kinds of data because the two files were created from different versions of the source code, or by using different compiler options, repeat the compilations doing things more consistently.

1493 Cannot open <filename>.

Cause. A filename was specified for eld to open, but either that file doesn’t exist or you don’t have permission to read it. This particular message can come out about the existing program or DLL to be updated by the -alf or -strip option, or about the file specified for the -gateway_template option, which is a special option that few people would ever use.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you spelled the name of the file correctly and that you have permission to read it.

1495 Specified file <filename> is not an ELF object file.

Cause. A filename was specified for eld to read, and the file was supposed to be a valid TNS/E object file of some sort, but it wasn’t. In particular, this message is saying that the file was not the ELF format, which is the format used by TNS/E object files. This particular message can come out about the existing object file to be updated by the -alf, -change, or -strip option, or about the file specified for the -gateway_template file, which is a special option that few people would ever use.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you spelled the name of the file correctly and that it is a valid TNS/E object file of the type appropriate for the command that you gave.

1496 Specified file <filename> is not a TNS/E object file.

Cause. A filename was specified for `eld` to read, and the file was supposed to be a valid TNS/E object file of some sort, but it wasn't. In particular, this message is saying that the file was not the 64-bit ELF format, and more specifically not the format used for Intel's 64-bit IPF implementation, which is the format used by TNS/E object files. This particular message can come out about the existing object file to be updated by the `-alf`, `-change`, or `-strip` option, or about the file specified for the -gateway_template file, which is a special option that few people would ever use.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Check that you spelled the name of the file correctly and that it is a valid TNS/E object file of the type appropriate for the command that you gave.

1497 Specified file <filename> is neither a program nor a DLL.

Cause. You speciifed either the `-alf` or `-strip` option, followed by the name of an existing file. For these options, that file must be a program or DLL. However, the file that was specified was instead an object file, such as a file produced by a compilation, not a program or a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to use the `-alf` or `-strip` option, then the file that you specify must be a program or DLL. A similar stripping operation on object files, such as those produced by a compilation, rather than a program or a DLL, can be accomplished by using the -s and `-r` options together.

1498 For the -alf option you can't specify the -d option without the -t option.

Cause. You gave the `-alf` option, to repeat the fixup process on an existing program or DLL. Along with the `-alf` option, you can also specify the `-t` option, to provide a new address for the code segment of the program or DLL. If you just do that, then the data segment moves by the same amount as the text segment. Or, if you give the `-t` option, then you can also give the `-d` option, to independently tell where the data segment should move. However, you cannot give the `-alf` option with the `-d` option if you don't also give the `-t` option, and that is what you did.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to repeat the fixups without changing the address of the existing program or DLL, don't specify the `-d` option. If you want to move the text and data segments by the same amount (including the most usual case, where the data segment is immediately after the text segment), then specify the `-t` option, not the `-d`

option. If you want to independently specify where each segment should move, then specify both the `-t` and `-d` options.

1499 The value of the -t option was rounded up to <number>.

Cause. You specified the `-alf` option with the `-t` option, to repeat the fixup process on an existing DLL while specifying a new address for its code segment. The address you specified was rounded up to a multiple of 64K bytes (or, 128K bytes if you are doing this to an implicit DLL).

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. The starting address of the code segment of a DLL is required to have the indicated alignment. No action is required if you understand that and are satisfied with the rounding, although it would be cleaner if you specified a number with the right alignment in the first place. If this doesn't make sense to you, because you don't understand the purpose of the `-t` option, read the documentation or contact HP for more detailed advice. Perhaps you intended to make some code section come out at a particular location, but there is no direct way to do that.

1500 The value of the -d option was rounded up to <number>.

Cause. You specified the `-alf` option with the `-d` option, to repeat the fixup process on an existing DLL while specifying a new address for its data segment. The address you specified was rounded up to a multiple of 64K bytes (or, 128K bytes if you are doing this to an implicit DLL).

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. The starting address of the data segment of a program or DLL is required to have the indicated alignment. No action is required if you understand that and are satisfied with the rounding, although it would be cleaner if you specified a number with the right alignment in the first place. More likely, there was no reason for you to use this combination of options in the first place. If this doesn't make sense to you, because you don't understand the purpose of the `-d` option, read the documentation or contact HP for more detailed advice. Perhaps you intended to make some data section come out at a particular location, but there is no direct way to do that.

1502 <name of a relocation table section> entry <number> is not in the data segment.

Cause. You gave the `-alf` option to repeat the fixup process on an existing program or DLL. Normally, this only updates places in the data segment, that need to be filled in with the addresses of symbols found in this same program or DLL, or in other DLLs. Those places are listed in "relocation tables" within the program or DLL. However, one of the relocation table entries indicated an address to be fixed up that was not within the data segment of the program or DLL. Possibly the program or DLL is bad, which

would indicate a bug with eld when it created that program or DLL, and should be reported to HP. It is also possible that the relocation table entry indicates an address in the code segment, but you did not specify the -update_code option, to tell the -alf option that it was okay to update such places. There are two relocation tables, named ".rela.dyn" and ".rela.gblzd", and the message tells you which table had the bad entry in it, and which entry it was. You may be able to get more information, such as the name of the symbol being referred to, by dumping out this relocation table section with ENOFT.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. The -update_code option is a special option that is only intended to be used when running -alf on the millicode DLL that is part of system library. If that is the case you are doing, specify this option. Other than that case, there is probably nothing you can do about this error message except to report it to HP as a possible bug with eld when it created this program or DLL.

1506 Cannot create workfile <filename>.

Cause. This message can come out either when you give the -alf option, or the -strip option, or the -r option when there is exactly one input object file. eld tries to create a workfile in the same location (OSS directory, Guardian subvolume, or PC folder) as the place where you specified that the new version of the file should be created. eld could not create that workfile and open it for writing.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Check that you have permission to create files in the indicated location, and that it isn't a Guardian subvolume that is full.

1508 Cannot create -temp_o file <filename>.

Cause. This message can come out either when you give the -alf option, or the -strip option, or the -r option when there is exactly one input object file. eld creates a temporary output file, before creating the real output file. When you specify the -temp_o option, eld still first creates the temporary file in another place, and when that file is created eld then tries to rename it to the filename specified in the -temp_o option. That renaming failed. The temporary file that was created, and the file that you specified in the -temp_o option, are both in the same location (Guardian subvolume, OSS directory, or PC folder) as the place that you specified for the new version of the file to be created.

Effect. Warning (eld still creates the output object file, but not using the file you specified with the -temp_o option as an intermediate file).

Recovery. If you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1509 Cannot (re-)create <filename>; naming the output file
<filename> instead.
```

Cause. This message can come out either when you give the `-alf` option, or the `-strip` option, or the `-r` option when there is exactly one input object file. `eld` first creates the new version of the object file in a temporary location, deletes any file that previously existed with the name that the new version of the file is to have, and then renames the temporary file to the final name. That process failed. The file has instead been left in the temporary location as given in the message.

Effect. Warning (`eld` produces an output file, but not with the filename you intended).

Recovery. If there already was a file with the same name as the file you wanted to create, which may in fact be the old version of the file that you are updating with the `-alf`, `-strip`, or `-r` option, and you did not have permission to delete it, specify a different filename for the new version of the object file. If there was no file of that name already, and you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document.

```
1510 eld encountered the file <filename> which has the lp64
data model, and eld applies the rules for the lp64 data model
henceforth. These rules have an effect on how the search is
performed for –l option and how the warning messages are
produced for the data models of DLL's. This link has already
performed one or both of these options by following the rules
for the default ilp32 data model and the results are not what
is desired. Henceforth to avoid this scenario, enter the –set
data_model lp64 option on the command line.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. This is a warning message and `eld` continues with the link. However, the results are not as desired because, `eld` performed the search for `–l` option. The search according to the `ilp32` rule has produced a warning message that, it has found the `lp64` DLL before obtaining the `lp64` linkfile and thereafter started following the `lp64` rule. If you are performing a `lp64` build and want to ensure that `eld` is consistent throughout the link then, enter the `–set data_model lp64` option in the command line.

```
1511 The -t and -d options are not allowed when running the -
alf option on a program.
```

Cause. You gave the `-alf` option, to repeat the fixup process on an existing program. Along with the `-alf` option, you also specified the `-t` or `-d` option. These options can be used with the `-alf` option when it is operating on a DLL, to tell it to also change the

address of the DLL. However, you are not allowed to change the address of a program with the `-alf` option, so it is wrong to use these options.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to repeat the fixups on a program, do not specify the `-t` or `-d` option. If you want to change the address of a program, you can not. You must rebuild the program to specify a different address for it, and usually there is no reason to do that. Perhaps you meant to run the `-alf` option on a DLL, but the filename that you specified was a program, not a DLL.

1512 Multiple, inconsistent specifications for the handling of unresolved references.

Cause. You gave more than one of the options named -unres_symbols error (or its synonym, -error_unresolved), -unres_symbols warn (or its synonym, -warning_unresolved), and -unres_symbols ignore. You can give the same option more than once if you wish, including synonyms for the same option, but otherwise you must not specify more than one of these options.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify one of these options, decide which one you want to specify, and only specify that one.

1513 The -d option was specified twice with different values.

Cause. You gave the `-d` option more than once, specifying different numerical values each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. There is rarely any reason to specify the `-d` option. But, if you want to specify this option, decide which value you want and only specify that one.

1514 The -t option was specified twice with different values.

Cause. The `-t` option tells the starting virtual address for the code segment of the DLL or program you are building. But, you gave the `-t` option more than once, specifying different numerical values each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. There is rarely any reason to specify the `-t` option for a program, but it is reasonable to do so when you are building a DLL. If you want to give this option, decide which value you want and only specify that one.

1515 Multiple specifications of the -gateway_template option with different filenames.

Cause. The -gateway_template option tells the name of a template file that is used for overriding the standard gateway format when you are building a DLL that contains callable procedures. You gave this option more than once, specifying different filenames each time.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Few people will ever need to use this option. But, if you have a template file that you wish to give to eld with this option, just specify the option once, giving the name of that template file.

1516 <gateway template filename> is not a TNS/E linkfile.

Cause. You gave the -gateway_template option, specifying a filename as a parameter. That file should be a TNS/E object file that can be used as linker input (usually, in this case, produced by the TNS/E assembler), but the file that you specified for this option is not that type of file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Few people will ever need to use this option. If you really are such a person, then you need to create the template file, typically by assembling some code that is a modfication of the assembler code used for this purpose before.

1517 Bad format for <gateway template filename>, <reason>.

Cause. You gave the -gateway_template option, specifying a filename as a parameter. That file is required to meet various conditions, in order to describe to eld the type of gateways that it should create for callable procedures, to override the default gateway formats. The message tells which condition the file did not meet.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Few people will ever need to use this option. If you really are such a person, then you need to create the template file, following the detailed rules for it, which is beyond the scope of this manual.

1518 The .procinfo section of <filename> gives a bad number of parameters (<number>) for <symbol name>.It should be between 0 and 32 for a user callable procedure.

Cause. You wrote a callable procedure, and the object file produced by the compiler is supposed to tell how many parameters it has, which should be a number between 0 and 32. However, the object file said something else.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you wrote a callable procedure with more than 32 parameters, you need to write it a different way, because a callable procedure cannot have more than 32 parameters. The exact definition of what constitutes “one parameter” may depend on the language and is beyond the scope of this manual. In any case, the fact that the compiler created such a bad file is also a bug that should be reported to HP.

```
1519 The <option name> option is not allowed with the -r
option.
```

Cause. You used the <*option name*> option, to provide instructions to `eld` about how to create the DLL or program that you are building, and you also used the *-r* option, to specify `eld` to build another object file that can be used as a linker input, rather than a program or a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program or a DLL, do not specify the `-r` option. If your intention is to combine several existing object files into a new object file that can be processed again by `eld` then do not specify the <*option name*> option.

```
1520 A main entry point (<symbol name>) is not allowed to be
callable.
```

Cause. The procedure named in the message has the CALLABLE (or KERNEL_CALLABLE) attribute. And, you have specified that this procedure be the main procedure where execution begins for the program that you are building, either by using the -e option or by the fact that the procedure is marked with the MAIN attribute in its object file. However, that combination is not allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is that this procedure be the place where execution begins for your program, then it cannot be marked CALLABLE or KERNEL_CALLABLE. If you want your program to begin execution in privileged mode, you can accomplish that purpose by instead giving the procedure the PRIV attribute. If you specified that this procedure be the main entry point by giving the -e option, are you sure you wanted to do that? In pTAL, you could do that if you forgot to put the MAIN attribute on the main procedure, but it would probably be better for you to change the source that way, at the same time that you change it to saying PRIV rather than CALLABLE. It is usually a mistake to give the -e option when building a program in any other language. For example, in C, the procedure that you call “main” is not really where execution begins, so you should not say “-e main”. Since it isn’t where execution really begins, it is okay

for your “main” procedure in C to have the CALLABLE attribute, and that is a perfectly normal way to make your program run in privileged mode (although certain initialization activities will have taken place before it became privileged).

```
1521 The main entry point(<symbol name>)is not in a code
section.
```

Cause. You used the -e option to specify the main entry point for the program you are building, and the parameter to the -e option is supposed to be the name of a procedure. However, the name that you specified is instead the name of a data item (at least, it does not exist in a section that has the right name for a code section).

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your program contains C, C++, or Cobol code, you shouldn’t be using the -e option at all. If your program only contains pTAL code then yoyu can use the -e option to tell the name of the pTAL procedure that should be the main entry point. Did you spell the name wrong? You must specify the name of a procedure, not a data item.

```
1522 For entry <number> in <relocation table name>, the
updated value <number> no longer fits into 32 bits.
```

Cause. You specified the `-t` and/or `-d` options, along with the `-alf` option, to tell `eld` to move a DLL to different addresses, and you specified addresses that woudn’t fit into 32 bits. Or, you are using the `-alf` option to repeat the fixups on a program or DLL, and you are pointing at some other DLL that uses addresses that don’t fit into 32 bits. In either case, the program or DLL that you are fixing up contains 32-bit pointers, that it is using to point at symbols, either in the same DLL or in another DLL, whose addresses can’t be represented in 32 bits. So, that’s impossible to do.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If you really want to move this DLL to a range of addresses that don’t fit into 32 bits, or make references to some other DLL that has a range of addresses that don’t fit into 32 bits, then you need to change your source code so that it doesn’t try to use 32-bit pointers in those cases.

```
1523 The <option name> option is not allowed with the -alf
option.
```

Cause. You gave the `-alf` option. Along with an `-alf` option, only certain other options are allowed. However, you gave the option mentioned in the message, which is not one of the options allowed with the `-alf` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Remove the offending option.

1524 You can put one filename on the command line with the -alf option, to be the zimpimp file, but more than one filename is an error.

Cause. You specified the `-alf` option, telling the name of the program or DLL that you want to update. Various other options are allowed with the `-alf` option, and some of these other options also have filenames as parameters. But, in addition to all these filenames, you also had two other filenames on the command line by themselves, not as parameters of options. It is possible to have one such filename, to tell the name of the zimpimp file that represents the DLLs that constitute system library. However, it is never legal to have more than one such filename.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Perhaps you need to review the section of the manual that describes the `-alf` option. The only filenames that should be on the command line are as described above.

1526 Can't specify the -no_preset or -must_preset option with the -r option.

Cause. You used the -no_preset or -must_preset option, which affect the `eld` program's behavior when it is creating a program or DLL, and you also used the `-r` option, to tell `eld` to build another object file that could be used as `eld` input, rather than a program or DLL. That's inconsistent.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program or DLL, don't use the `-r` option. If your intention is to create another object file that can be used as `eld` input, rather than a program or DLL, then don't use the -no_preset or -must_present option.

1527 Can't specify the -must_preset option with the -no_preset option.

Cause. The -no_preset option tells `eld` not to mark the file "preset", because you know that you don't have the right linker inputs to fix up references correctly. The -must_preset option tells `eld` that you believe you do have the right linker inputs to fix up references correctly, and it should be an error if `eld` can not do that. These are contradictory.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Based on what these two options do, decide which one you want to give, or neither.

```
1528 Cannot preset because <reason>.
```

Cause. You gave the -must_preset option, to say that eld should consider it an error if it couldn't correctly perform fixups, and that is what happened. There are many reasons why fixups might not have been possible, and the message tells the reason that occurred.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. It is possible to run a program even though it could not be fixed up correctly at link time. For example, one reason that fixups can't be performed at link time is if eld is told to look at various DLLs that have overlapping addresses. This may be true of your link environment, simply because it doesn't have copies of the DLLs for which people have tried to get the addresses sensible. And even if you are doing a link on the same machine where you can run, pointing at the real DLLs that you will use, it is still possible that the DLLs have overlapping addresses, so that fixups can't be performed at link time. The addresses of the DLLs can be changed at runtime, making it possible to run correctly. So, you might just want to omit the -must_preset option and not worry about this. Or, if you want to make sure that the fixups can be correctly performed at link time, which is a requirement for some special programs, and which is probably why you specified -must_preset in the first place, then you will need to investigate the reason why fixups couldn't be done, as given in the message.

```
1529 <filename> is a <type of file>, but the -alf option
requires DLL's for the liblist entries in <existing program
or DLL name>.
```

Cause. You have given the -alf option, to update an existing program or DLL. When that program or DLL was originally built, it specified certain other DLLs to be used. Those DLLs contained "DLL names", which were saved in the program or DLL being built at the time. During the -alf option, eld uses those DLL names to search for those other DLLs again. eld found such a file, and opened it, but instead of being a DLL it was a different kind of file, as explained in the message. The message might have called it a "linkfile", which means, for example, that it might be a file produced by a compilation, or the message might have said it was an "archive", as produced by the "ar" tool. But, in any case, it wasn't a DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. The simplest approach is that, when you build a DLL, give it a simple name, and let its DLL name be that same simple name. And then, leave that DLL where it is, while building another program or DLL that uses it, and then using the -alf option on that other program or DLL. Then you shouldn't get into this kind of error situation. Somehow, by having a DLL name that doesn't match the filename, or by having other files with the same names as the names of DLLs, things have gotten confused. Start over, and don't do that.

```
1530 Using the zimpimp file <filename>.
```

Cause. eld normally looks for a “zimpimp” file, based on which it gets information about the symbols in system library, and this message is just telling you where eld found this file.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

```
1532 User_callable and kernel_callable procedures are both
present.
```

Cause. Procedures may be given the CALLABLE attribute, which means that code goes into privileged mode when such procedures are called. Obviously, this attribute is only used by a limited number of people. There is also an attribute called KERNEL_CALLABLE, which is even more restricted, only occuring in the millicode DLL that is part of system library. A single DLL is not allowed to contain a mixture of procedures with both of these attributes, but both of these attributes were used somewhere among the input files that you gave to eld.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you are one of the persons who should be using these attributes in the first place, and having this problem, then it is a problem with building the NSK operating system that is beyond the scope of this manual.

```
1533 The 'incomplete' attribute is not allowed for implicit
DLL's.
```

Cause. You specified the -change incomplete option, to mark an existing import library as an incomplete import library. However, the file name that you specified for the -change option is in fact a zimpimp file, the special import library that represents the procedures in system library, and you are not allowed to mark that file incomplete.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention was to mark some other import library as incomplete, specify its name correctly. You cannot mark the zimpimp file incomplete.

```
1534 Globalized symbols (first one: <symbol name>) are not
allowed in an implicit DLL.
```

Cause. The C++ compiler marks some symbols as “globalized” symbols, because the fixups to such symbols need to follow special rules at runtime. You have globalized symbols in the files that you are linking together, as indicated in the message. You

have also given the `-dll` and -make_implicit_dll options, to tell `eld` to make one of the implicit DLLs that constitute system library. System library is not allowed to contain globalized symbols.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming you do want to make an implicit DLL, you probably need to look over your C++ source code to figure out why it has globalized symbols in it, and change the code to avoid that. It is beyond the scope of this manual to explain the rules that the C++ compiler uses in deciding which globalized symbols to create.

```
1535 Copies of symbol <symbol name>, in <filename> and
<filename>, don't agree on whether it is globalized.
```

Cause. The C++ compiler marks some symbols as “globalized” symbols, because the fixups to such symbols need to follow special rules at runtime. Each of the two files mentioned in the message contain symbols of the same name, as shown in the message. One of the copies was globalized and one wasn’t, which is an error. It doesn’t matter if the declarations of the symbols are definitions or just external references, they still need to agree on whether the symbol is globalized.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define symbols with the same name in each of these two files, and have both symbols visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is defined and only visible within its own compilation. If you are using the same symbol in more than one place, all the places must agree on whether that symbol is globalized. If the symbols disagree about this because the two files were created from different versions of the source code, or by using different compiler options, repeat the compilations doing things more consistently.

```
1536 Filename <filename> was specified on the command line
with the -alf option, but that means that this file would be
the zimpimp file, and the existing file <filename> is an
implicit DLL, and an implicit DLL cannot use a zimpimp file,
so this is an error.
```

Cause. You specified the `-alf` option, telling the name of the DLL that you want to update, and this was one of the implicit DLLs that constitute system library. Various other options are allowed with the `-alf` option, and some of these other options also have filenames as parameters. But, in addition to all these filenames, you also had another filename on the command line by itself, not as the parameter of an option. In general, with the `-alf` option, it is possible to have one such filename, to tell the name of the zimpimp file that represents the DLLs that constitute system library. However, it is not possible to specify a zimpimp file when in fact you are now working on one of the implicit DLLs themselves.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. This indicates some problem with the procedure for building the NSK operating system, which is beyond the scope of this manual.

```
1537 The file named <filename>, which should be a zimpimp
file, is not a zimpimp file.
```

Cause. The zimpimp file is a file that tells eld about the symbols in system library. eld has various methods of locating this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zimpimp”. eld did find a file by these methods, but the file turned out not to have the proper structure for a zimpimp file. This particular message comes out when the file did have the structure of a DLL, but did not have the proper “DLL name” within it, to indicate that it was the zimpimp file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. There is something wrong with your installation. The message told the name of the file that eld thought should be the zimpimp file, but wasn’t. Perhaps that will help you figure out what is wrong. A proper zimpimp file should be built as part of the process of creating the operating system, which is beyond the scope of this manual.

```
1538 <filename> is a linkfile but a zimpimp file was
expected.
```

Cause. The zimpimp file is a file that tells eld about the symbols in system library. eld has various methods of locating this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zimpimp”. eld did find a file by these methods, but the file turned out not to have the proper structure for a zimpimp file. This particular message comes out when the file was not a DLL, which is what the zimpimp file looks like, but rather was a linkfile, such as an object file created by a compilation.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. There is something wrong with your installation. The message told the name of the file that eld thought should be the zimpimp file, but wasn’t. Perhaps that will help you figure out what is wrong. A proper zimpimp file should be built as part of the process of creating the operating system, which is beyond the scope of this manual.

```
1539 Option -public_registry specified multiple times with
different filenames <filename> and <filename>.
```

Cause. The -public_registry option specifies the name of a public DLL registry file, which eld uses to look up information about the operating system and other standard DLLs. You gave this option twice on the command line, and you specified different names each time, which is an error. Note that case is significant for this check.

Effect. Fatal error (eld immediately stops without creating an output file).

Cause. Are you sure you want to specify the -public_registry option at all? There usually is no need to give this option, because eld should be able to find the official version automatically. If you specify this option, you can do it more that once, but only if you specify the same filename each time.

```
1540 Cannot open public DLL registry file <filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zreg”. Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. However eld was told where the public DLL registry file was, either a file of that name didn’t exist, or you don’t have permission to read it.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, check that you spelled the name correctly, and that you do have permission to read it. If eld could not find it on its own, and you didn’t specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what’s wrong.

```
1541 Unexpected characters <string> found on line <number> of
<filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zreg”. Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into “statements”, which in turn can have “attributes”, and this particular message comes out when eld found unexpected characters between

two statements or between two attributes. The message tells the line number at which this occurred in the file.

Effect. Warning. It is possible that the format of the official public DLL registry file changed, and that is why eld cannot read it, but still eld tries to continue with the link. Because eld could not read the public DLL registry file, however, that means that eld will not be able to automatically find any of the standard DLLs.

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong

```
1542 Statement starting at line <number> of the public DLL
registry file <filename> went off the end of file.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", and there are many statements that eld simply ignores, and this particular message comes out when eld was trying to skip over a statement by searching for the final semicolon but ran off the end of the file instead. The message tells the line number at which this statement began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

1543 While looking for the next attribute in a statement, starting at line <number> of the public DLL registry file <filename>, went off the end of file.

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", and this particular message comes out when eld was looking for the next attribute within a statement, but ran off the end of the file instead. The message tells the line number at which this statement began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

1544 Attribute starting at line <number> of the public DLL registry file <filename> went off the end of file.

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", and this particular message comes out when eld was trying to read through to the end of an attribute but ran off the end of the file instead. The message tells the line number at which this attribute began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find

the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

```
1545 Bad syntax for the file attribute on line <number> of
<filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", and this particular message comes out when eld detected a problem parsing the "file" attribute after seeing the filename. The message tells the line number at which this attribute began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

```
1546 Filename expected for 'file' attribute on line <number>
of <filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", and this particular message comes out when eld detected a problem parsing the "file" attribute

before seeing the filename. The message tells the line number at which this attribute began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

```
1547 Cannot specify the -public_registry option with the -r
option.
```

Cause. The -public_registry option specifies the name of a public DLL registry file, which eld uses to look up information about the operating system and other standard DLLs when eld is creating a program or a DLL. You also used the -r option, to tell eld to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to create a program or DLL, then don't specify the -r option. If our intention is to use the -r option to create a new object file that can be used as eld input, then don't specify the -public_registry option.

```
1548 More than one file attribute in the same 'dll'
statement, at line <number> in <filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", but the same attribute can't occur twice in the same statement, and this particular message comes out when eld detected two "file" attributes in the same "dll" statement. The message tells the line number at which this occurred in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

```
1549 No file attribute in the 'dll' statement that started at
line <number> in <filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld parses the contents of the file into "statements", which in turn can have "attributes", and a "dll" statement should always contain a "file" attribute, but this particular message comes out when eld saw a "dll" statement that did not contain a "file" attribute. The message tells the line number at which this statement began in the file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn't specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what's wrong.

```
1550 More than one 'dll' statement with name <string> in
<filename>.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. eld did find a file by these methods, but the file turned out not to have the proper structure for a public DLL registry file. eld

parses the contents of the file into “statements”, which in turn can have “attributes”, and a “dll” statement always contains a “file” attribute, which tells the DLLs name. This particular message comes out when eld saw two different “dll” statements with the same name.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you specified the -public_registry option, are you sure you need to do that? There usually is no need to give this option, because eld should be able to find the official version of the public DLL registry automatically. If you specify this option, it is your responsibility to provide eld with a correct public DLL registry file, and it is beyond the scope of this manual to describe how this file should be created. If eld could not find it on its own, and you didn’t specify the -public_registry option, then there is something wrong with your installation. The message told the name of the file that eld thought was the public DLL registry file. Perhaps that will help you figure out what’s wrong.

```
1551 This link is not using any zimpimp file.
```

Cause. The zimpimp file is a file that tells eld about the symbols in system library. eld has various methods of locating this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zimpimp”. Or, you can simply put the name of the zimpimp file directly on the eld command line. However, none of these methods specified a zimpimp file to eld.

Effect. Warning. eld will continue to produce an output file, but it will not be able to fix up references to system library.

Recovery. It is not necessary to have references fixed up at link time. However, normally, eld should find a zimpimp file. So, this may indicate that there is something wrong with your installation. A proper zimpimp file should be built as part of the process of creating the operating system, and then installed in the proper location for eld to find it. The process of creating the operating system is beyond the scope of this manual.

```
1552 This link is not using any public DLL registry file.
```

Cause. eld uses the public DLL registry file to look up information about the operating system and other standard DLLs. There are various ways that eld may find this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name “zreg”. Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. However, none of these methods specified a public DLL registry file to eld.

Effect. Warning. eld will continue to produce an output file, but it will not be able to fix up references to standard DLLs.

Recovery. It is not necessary to have references fixed up at link time. However, normally, eld should find the public DLL registry file. So, this may indicate that there is something wrong with your installation. The process of creating and installing the public DLL registry file is beyond the scope of this manual.

```
1553 <filename> is an archive, but eld was expecting a
zimpimp file.
```

Cause. The zimpimp file is a file that tells eld about the symbols in system library. eld has various methods of locating this file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zimpimp". eld did find a file by these methods, but the file turned out not to have the proper structure for a zimpimp file. This particular message comes out when the file was not a DLL, which is what the zimpimp file looks like, but rather was a archive, as would be created by the "ar" tool.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. There is something wrong with your installation. The message told the name of the file that eld thought should be the zimpimp file, but wasn't. Perhaps that will help you figure out what is wrong. A proper zimpimp file should be built as part of the process of creating the operating system, which is beyond the scope of this manual.

```
1554 Can't open public DLL file named <filename>. Such a file
should be in the same place as the zreg file.
```

Cause. eld is searching for a DLL, and you are using eld on Guardian. eld found the name mentioned in the message in the public DLL registry file, meaning that this DLL is one of the standard DLLs and is supposed to exist in a certain location. Specifically, it is required that there be a file named "zxxxdll" in the same location (Guardian subvolume, OSS directory, or PC folder) as the public DLL registry file. But, either such a file does not exist, or you don't have permission to read it.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. There are various ways that eld may find the public DLL registry file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". The first place eld looks is its own location (Guardian subvolume, OSS directory, or PC folder), and if there is no "zreg" file there then it will also look in a sibling directory or folder whose name ends "lib". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. The file mentioned in the message was supposed to exist in the same place as that public DLL registry file, but wasn't. If you gave the -public_registry option, then you are responsible for setting up all the files correctly. Otherwise, there is something wrong with your

installation. The procedure for creating and installing a public DLL registry file is beyond the scope of this manual.

```
1555 <filename> is an archive, but it should have been a
public DLL.
```

Cause. You gave a -l option, to tell `eld` to search for a DLL based on the parameter to the -l option. Based on that, `eld` decided that this was one of the standard DLLs, found it in the standard place, and opened it. The file, however, was not a DLL, but instead was an archive, as would be created by the "ar" tool.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. There is something wrong with your installation. Files that are not DLLs have been placed in the location for the standard ("public") DLLs. The process of installing the standard DLLs in the standard location is beyond the scope of this manual.

```
1556 <filename> is a linkfile, but it should have been a
public DLL.
```

Cause. You gave a -l option, to tell `eld` to search for a DLL based on the parameter to the -l option. Based on that, `eld` decided that this was one of the standard DLLs, found it in the standard place, and opened it. The file, however, was not a DLL, but instead was a linkfile, i.e., an object file created by a compilation or by running `eld` with the `-r` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. There is something wrong with your installation. Files that are not DLLs have been placed in the location for the standard ("public") DLLs. The process of installing the standard DLLs in the standard location is beyond the scope of this manual.

```
1557 The -set data_model neutral option is not allowed when a
program is being created.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Fatal error. `eld` stops immediately without creating an output file.

```
1558 The DLL <filename> contains the same DLL name as the
user library <filename> but is still being used in the link.
```

Cause. You are building a program, and you are using a user library. A user library is a DLL, and like any other DLL has a "DLL name" inside it. `eld` has also opened another DLL, whose filename is shown in the message, and found that it contained the

same DLL name as the user library, but it is not the same file, nor does it look like a copy of the same file.

Effect. Warning (eld produces an output file, but it might not be what you intended).

Recovery. This is not necessarily an error, although it may indicate that you didn't do what you intended to do. The DLL mentioned in the message will be used by eld during this link. For example, you may have specified this DLL to eld with a -l option, and the program will remember that fact, so the operating system will look for this DLL again at runtime. That's probably okay. It's just a possible source of confusion that the user library has the same DLL name as this DLL. You probably should avoid this situation. The DLL name inside the DLL should match the name that the DLL will have at runtime. You should either build it with that filename to start with, which by default makes its DLL name the same as that file name, or you should use the -soname option when you build the DLL, to tell the filename to which you intend to rename the DLL later. And you should build the user library in a different place, so that it has a different DLL name inside it.

```
1559 The DLL <filename> was found with the same export digest
as the user library.
```

Cause. You are building a program, and you are using a user library. A user library is a DLL, and like any other DLL has a "DLL name" inside it. eld has also opened another DLL, whose filename is shown in the message, and found that it contained the same DLL name as the user library. Also, it probably is the same file, or a copy of the same file, because it exports all the same symbols with the same addresses as the user library.

Effect. Warning (eld produces an output file, but it might not be what you intended).

Cause. This is not necessarily an error, although it may indicate that you didn't do what you intended to do. The DLL mentioned in the message will be ignored by eld during this link, because it is redundant with the user library as far as doing fixups is concerned. For example, you may have specified this DLL to eld with a -l option, but the program will not remember that fact, so the operating system will not know to look for this DLL at runtime, even if a different user library is used then. It's hard to say what all the consequences of this might be. You probably should avoid this situation. The DLL name inside the DLL should match the name that the DLL will have at runtime. You should either build it with that filename to start with, which by default makes its DLL name the same as that file name, or you should use the -soname option when you build the DLL, to tell the filename to which you intend to rename the DLL later. And you should build the user library in a different place, so that it has a different DLL name inside it.

```
1560 The -local_libname option is only allowed when you are
building a program.
```

Cause. You used the `-local_libname` option, which tells `eld` where to find a copy of the user library for the program that you are building, but the file that you have told `eld` to create is not a program.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program, specify that correctly. For example, don't specify the `-dll`, `-shared`, or `-ul` options, which mean that you are telling `eld` to build a DLL, rather than a program. And don't specify the `-r` option, which tells `eld` that you are building another object file that can be used as input to `eld`, rather than a program. Or, if you don't intend to create a program, then don't specify the `-local_libname` option.

```
1561 On the PC, if you specify the -local_libname option, you
must also specify the -set libname option, to tell the
Guardian name of the user library for runtime.
```

Cause. You are running `eld` on the PC and creating a program that has a user library. The `-set libname` option tells the Guardian filename that the user library will have at runtime, to be stored in the file. The `-local_libname` option tells `eld` where to find a copy of the user library during the link. `eld` needs both pieces of information. You have given the `-local_libname` option, but not the `-set libname` option. If you run `eld` on NSK, and the name given for `-local_libname` is in the Guardian namespace, then it's okay to omit the `-set libname` option, and `eld` will assume that the user library will be in the same location at runtime. However, on the PC, it is an error to give `-local_libname` without `-set libname`.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you are using a user library, then you must decide the Guardian filename that the user library will have at runtime, and specify that with the `-set libname` option.

```
1562 File <filename>, specified as the user library, is an
archive.
```

Cause. You are creating a program that has a user library, and you have given `eld` the filename for a copy of the user library, by specifying the `-local_libname`, `-libname`, or `-set libname` option. A user library is a DLL. However, `eld` opened the user library and found that it wasn't a DLL but instead was an archive, such as is created by the "ar" tool.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to create a program that uses a user library, you must create the user library as a DLL by running `eld` and specifying an option such as `-dll`, `-shared`, or `-ul`. Then you can give the name of that user library to `eld` when you create the program.

```
1563 File <filename>, specified as the user library, is a
linkfile.
```

Cause. You are creating a program that has a user library, and you have given `eld` the filename for a copy of the user library, by specifying the `-local_libname`, `-libname`, or `-set libname` option. A user library is a DLL. However, `eld` opened the user library and found that it wasn't a DLL but instead was a linkfile, i.e., an object file produced by the compiler or by running `eld` with the `-r` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to create a program that uses a user library, you must create the user library as a DLL by running `eld` and specifying an option such as `-dll`, `-shared`, or `-ul`. Then you can give the name of that user library to `eld` when you create the program.

```
1564 Both the -exported_symbol and -hidden_symbol options
were specified for the same symbol <symbol name>.
```

Cause. The -exported_symbol option (or its synonym, the -export option) tells `eld` that the named symbol should be exported from the program or DLL being created. The -hidden_symbol option (or its synonym, the -export_not option) tells `eld` the opposite. You first specifed that this symbol should be exported, and then later on the command line you said it shouldn't be exported.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Decide whether you want this symbol to be exported and specify the proper option.

```
1565 Both the -exported_symbol and -hidden_symbol options
were specified for the same symbol <symbol name>.
```

Cause. The -exported_symbol option (or its synonym, the -export option) tells `eld` that the named symbol should be exported from the program or DLL being created. The -hidden_symbol option (or its synonym, the -export_not option) tells `eld` the opposite. You first specifed that this symbol should not be exported, and then later on the command line you said it should be exported.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Decide whether you want this symbol to be exported and specify the proper option.

```
1566 The -export_all and -ul options are not allowed with the
-r option.
```

Cause. You used the -export_all option, to tell `eld` that all normal global symbols should be exported from the program or DLL being created. Or, you may have used the `-ul` option, which implies the -export_all option. But, you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used as `eld` input then don't specify the -export_all or `-ul` option.

1567 The -hidden_symbol option is not allowed with the -r option.

Cause. You used the -hidden_symbol option (or its synonym, the -export_not option), to tell `eld` not to export a certain symbol from the program or DLL being created. You also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used as `eld` input then don't specify the -hidden_symbol option.

1568 The -exported_symbol option is not allowed with the -r option.

Cause. You used the -exported_symbol option (or its synonym, the -export option), to tell `eld` to export a certain symbol from the program or DLL being created. You also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program or DLL.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. If your intention is to create a program or DLL then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used as `eld` input then don't specify the -export option.

1569 The -exported_symbol <symbol name> is not present.

Cause. You gave the -exported_symbol option (or its synonym, the -export option) to say that a certain symbol should be exported. However, only symbols that are defined

in the input object files, and are visible outside their own compilations, can be exported, and such a symbol of this name did not exist.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. Check that you spelled the name of the symbol correctly. In pTAL or Cobol, for example, it must be given in upper case. In C or C++, symbol names are case sensitive. Is the symbol defined (not just an external reference), and visible outside its compilation? The rules for which symbols are defined by a compilation, and visible outside that compilation, depend on the source language, and are beyond the scope of this manual.

```
1570 The -exported_symbol <symbol name> is not defined in any
of the input linkfiles.
```

Cause. You gave the -exported_symbol option (or its synonym, the -export option) to say that a certain symbol should be exported. In fact, `eld` has seen such a symbol, perhaps in a DLL, or perhaps as an external reference from the program or DLL being built. However, only symbols that are defined in the input object files can be exported, and such a symbol of this name did not exist.

Effect. Error (The linker cannot do what was requested of it and will eventually stop, but may continue for the purpose of detecting additional errors before stopping).

Recovery. The symbol must be defined (not just an external reference). The rules for which symbols are defined by a compilation depend on the source language, and are beyond the scope of this manual.

```
1571 The -hidden_symbol <symbol name> is not present.
```

Cause. You gave the -hidden_symbol option (or its synonym, the -export_not option) to say that a certain symbol should not be exported. However, only symbols that are defined in the input object files, and are visible outside their own compilations, can be exported, and such a symbol of this name did not exist.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. Saying that a symbol should not be exported, and no such symbol existed in the first place, is not necessarily an error, but you probably want to make your link clean. If no such symbol is present, don't give this option. If you think that there should be such a symbol, check that you spelled the name of the symbol correctly. In pTAL or Cobol, for example, it must be given in upper case. In C or C++, symbol names are case sensitive. Is the symbol defined (not just an external reference), and visible outside its compilation? The rules for which symbols are defined by a compilation, and visible outside that compilation, depend on the source language, and are beyond the scope of this manual.

1572 The -hidden_symbol <symbol name> is not defined in any of the input linkfiles.

Cause. You gave the -hidden_symbol option (or its synonym, the -export_not option) to say that a certain symbol should not be exported. In fact, `eld` has seen such a symbol, perhaps in a DLL, or perhaps as an external reference from the program or DLL being built. However, only symbols that are defined in the input object files can be exported, and such a symbol of this name did not exist.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. Saying that a symbol should not be exported, and no such symbol existed in the first place, is not necessarily an error, but you probably want to make your link clean. If no such symbol is present, don't give this option. If you think that there should be such a symbol, note that the symbol must be defined (not just an external reference). The rules for which symbols are defined by a compilation depend on the source language, and are beyond the scope of this manual.

1573 The user library file <filename> could not be opened.

Cause. You are creating a program, or using the `-alf` option to repeat the fixups on a program, that has a user library. You have either told `eld` the filename for a copy of the user library, by specifying the `-local_libname`, `-libname`, or `-set libname` option, or in the case of using `-alf` on an existing program the name for the user library could have come from the existing program itself. In any case, either the file mentioned in the error message does not exist or you do not have permission to read it.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. Even though `eld` could not open the user library, it will still continue doing the link to create or update the program. In this case, `eld` will not be able to do fixups on the program. In the `-alf` case, this invocation of `eld` may accomplish very little, since `eld` could not repeat the fixup process, which was probably the reason to give the `-alf` option in the first place, but again it won't fail. These things are not considered errors because it is possible to run a program even though it could not be fixed up at link time. Still, you probably want to make your link clean. To do that, find a copy of the user library that will be available at runtime, put it in the location where `eld` expects to find it, as shown in the error message, and make sure it is readable.

1574 The -local_libname option is not allowed when updating a DLL.

Cause. You gave the `-alf` option together with the `-local_libname` option, which means that you are repeating the fixup process on an existing program and telling `eld`

the filename for a copy of the user library that the program uses. However, the file that you specified in the -alf option is a DLL, not a program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to use the -alf option to update an existing DLL, then don't specify the -local_libname option. If your intention was to update an existing program, then you specified the wrong name in the -alf option, because that was a DLL, not a program.

```
1575 The -local_libname option was specified but the existing
program does not have a user library.
```

Cause. You gave the -alf option to tell eld to repeat the fixup process on an existing program. You also gave the -local_libname option to tell eld the filename for a copy of the user library that the program uses, but the existing program says that it doesn't use a user library.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you don't intend for the program to use a user library, don't give the -local_libname option. If you do intend for the program to use a user library, you should first run eld with the -change libname option, to update the program to tell the Guardian location where the user library will exist at runtime. Then eld will let you use the -alf option to update the fixups on the program. If you really want to repeat the fixups, but not have the program mention a user library name afterward, you can then run eld with the -change libname option again, after doing the -alf option, to remove the user library name from the program.

```
1576 Cannot specify the -temp_o option with the
-must_use_oname option.
```

Cause. The -temp_o option tells the name of a temporary file that you want eld to use to save the object file that it is creating, if it can't create it with the name you preferred, as specified for example by the -o option. The -must_use_oname option says that it should be an error if eld can't create the object file with the name you preferred. So, these are inconsistent.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If eld cannot create the object file with the name you prefer, do you want that to be an error? Or, do you want to tell eld the name of a different place to put it? Depending on what you want to do, specify at most one of the -temp_o and -must_use_oname options.

```
1577 Cannot specify the -temp_i option with the
-must_use_iname option.
```

Cause. The -temp_i option tells the name of a temporary file that you want `eld` to use to save the import library that it is creating, if it can't create it with the name you preferred, as specified by the -import_lib or -make_import_lib option. The -must_use_iname option says that it should be an error if `eld` can't create the import library with the name you preferred. So, these are inconsistent.

Effect. Fatal error (`eld` immediately stops without creating any output file).

Recovery. If `eld` cannot create the import library with the name you prefer, do you want that to be an error? Or, do you want to tell `eld` the name of a different place to put it? Depending on what you want to do, specify at most one of the -temp_i and -must_use_iname options.

```
1578 Cannot specify the -temp_r option with the
-must_use_rname option.
```

Cause. The -temp_r option tells the name of a temporary file that you want `eld` to use to save the private DLL registry that it is creating or updating, if it can't (re-)create it with the name you preferred, as specified by the -update_registry option. The -must_use_rname option says that it should be an error if `eld` can't (re-)create the private DLL registry with the name you preferred. So, these are inconsistent.

Effect. Fatal error (`eld` immediately stops without creating any output file).

Recovery. If `eld` cannot (re-)create the private DLL registry with the name you prefer, do you want that to be an error? Or, do you want to tell `eld` the name of a different place to put it? Depending on what you want to do, specify at most one of the -temp_r and -must_use_rname options.

```
1579 Cannot (re-)create <filename>.
```

Cause. This message can come out either when you give the `-alf` option, or the `-strip` option, or the `-r` option when there is exactly one input object file. `eld` was unable to create the output file in the specified location and you gave the -must_use_oname option, to say that `eld` should consider that an error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If there already was a file with the same name as the file you wanted to create, which may in fact be the old version of the file that you are updating with the `-alf`, `-strip`, or `-r` option, and you didn't have permission to delete it, specify a different filename for the new version of the object file. Or, if there is some reason why you cannot rename one filename to another filename in the target location (Guardian subvolume, OSS directory, or PC folder), that could also lead to this message, but that is an operating system question that is beyond the scope of this document.

```
1580 Cannot create -make_import_lib file <filename>.
```

Cause. You gave an option to ask `eld` to create an import library, and `eld` was unable to create the import library in the specified location, and you also gave the -must_use_iname option to say that `eld` should consider that failure an error.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If there already was a file with the same name as the file you wanted to create, and you didn't have permission to delete it, specify a different filename for the new version of the object file. Or, if there is some reason why you cannot rename one filename to another filename in the target location (Guardian subvolume, OSS directory, or PC folder), that could also lead to this message, but that is an operating system question that is beyond the scope of this document.

```
1581 Globalized symbols (first one: <symbol name>) are not
allowed when the -instance_data option is specified with the
data2hidden or data2protected parameter value.
```

Cause. The C++ compiler marks some symbols as "globalized" symbols, because the fixups to such symbols need to follow special rules at runtime. You have globalized symbols in the files that you are linking together, as indicated in the message. You have also given the `-instance_data` option with a parameter value of data2hidden or data2protected. That means that you are creating a program or DLL that has special requirements on how data is protected, and in such cases globalized symbols are not allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Assuming you do need to make the special kind of program or DLL that requires that `-instance_data` option, you probably need to look over your C++ source code to figure out why it has globalized symbols in it, and change the code to avoid that. It is beyond the scope of this manual to explain the rules that the C++ compiler uses in deciding which globalized symbols to create.

```
1584 The same symbol, <symbol name>, occurs in two -rename
options.
```

Cause. You have specified two different `-rename` options that have a symbol name in common. In other words, perhaps you are trying to rename the same symbol to two other names, or you are trying to rename two different symbols to the same name, or you are trying to rename "A" to "B" and "B" to "C". None of these possibilities is allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. These possibilities are not allowed because they are too complicated. Are you sure you didn't misspell the names of some symbols? If you really want to do these kinds of things, in a situation where it would make sense, you can probably accomplish that goal using multiple link steps, doing one of the renamings each time,

giving a subset of the input files to eld each time, and then there would be less confusion about what is actually going to happen.

1585 In <filename>, <symbol name> is defined, and is renamed to <symbol name>, but that symbol is also already defined in this file.

Cause. You gave a -rename option, specifying that a symbol should be renamed. A definition of that symbol (the first symbol name mentioned in the message) did occur in the file mentioned in the message, so eld wanted to rename it (to the second symbol name mentioned in the message), but there already was a definition of a symbol of that name in this same input file.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. You can't have two different symbols of the same name in a linker input file, and similarly you can't use this kind of -rename option to make things look that way. Once you have compiled two different symbols into the same object file, you can't make the linker think that they are the same symbol. If that was what you wanted to do, change your source code to use the same name throughout.

1586 Cannot use a DLL registry when running the -alf option on a program.

Cause. You have given the -alf option, and the file that you specified with this option is a program. You have also specified the -update_registry or -check_registry option, which would only be possible if you were updating a DLL, not a program.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If your intention is to use the -alf option to repeat the fixup process on a program then don't specify the -check_registry or -update_registry option. If your intention was to use the -alf option on a DLL, then you specified the wrong file to eld, because the file that you specified was a program, not a DLL.

1587 Cannot use a DLL registry with the -alf option when the code and data of the existing DLL are not next to each other.

Cause. You specified the -alf option, to update a DLL. You also specified the -check_registry or -update_registry option, to use a private DLL registry to choose new addresses for the DLL. By default, when eld builds a DLL, it puts the data segment right after the code segment. But, when this DLL was built, the -d option was used, to make the code and data segments come out at some other addresses, so that the data segment was not immediately after the code segment. On the other hand, private DLL registries can only be used to manage DLLs that have their data segments immediately after their code segments.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really want to specify the `-d` option when you (previously) built this DLL? There usually is no reason to do that. If you really need to separately specify the code and data segment addresses for this DLL, and now you want to change them, you can do that by specifying the `-t` and `-d` options with the `-alf` option, rather than trying to do this with a private DLL registry. You cannot make a private DLL registry store information about a DLL whose code and data segments are not next to each other.

1588 In the DLL registry, the entry for <filename> reserves it a size of <number>, but it needs <number>.

Cause. You have given either the `-alf` option, to update an existing DLL, or you have given the -make_import_lib option, to create the zimpimp file that represents the implicit DLLs that constitute system library. In either case, you have also given the -check_registry option, to say that the file being created must fit in the memory space assigned to it by a private DLL registry. But, the total size of the segments of the DLL or import library, including the required rounding up to certain alignments, was larger than the size specified in the private DLL registry.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you were using the `-alf` option then the size of the DLL would be the same as it was before. But, that didn't agree with the size of the DLL as listed in the private DLL registry. So, that means you are using a private DLL registry that is not the same as what might have been used when the DLL was previously created. Perhaps the private DLL registry file had been edited by hand, putting in smaller sizes than it had before.

Or, as part of building the NSK operating system, perhaps you were using a private DLL registry to keep track of the address of the zimpimp file, so that you always rebuild it at the same address as before, and so that it will be an error if it grows larger than the space allocated for it. So, that has happened.

In either case, assuming you wanted to be doing this, and want to keep doing it, you now need to make new decisions about where the DLL or zimpimp file should go. You could do that by editing the private DLL registry file by hand, or by deleting its entry from the private DLL registry and letting `eld` choose a new location for it with the -update_registry option.

1590 The -update_code option can only be specified with the -alf option.

Cause. The `-alf` option is used to repeat the fixups on an existing program or DLL. Usually, the `-alf` option only updates fixups within the data segment, because there shouldn't be any in the code segment. However, the millicode DLL within system library is a special case that has fixups to be done in the code segment. The `-alf`

option does this if you specify the -update_code option. But you specified the -update_code option without the `-alf` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to run the `-alf` option on the millicode DLL then you should have specified the -update_code option. In any other case, you should not specify the -update_code option.

1591 The -update_code option is only allowed when rebasing a DLL.

Cause. You gave the `-alf` option, and the file that you gave with it was a program. Usually, the `-alf` option only updates fixups within the data segment of a program or DLL, because there shouldn't be any in the code segment. However, the millicode DLL within system library is a special case that has fixups to be done in the code segment. The `-alf` option does this if you specify the -update_code option. But, since it is only intended to be used in this special case, it is considered an error if you specify the -update_code option and the file given to the `-alf` option was a program rather than a DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to run the `-alf` option on the millicode DLL then you should have correctly specified the name of the millicode DLL. In any other case, you should not specify the -update_code option

1595 Illegal duplicate definition of <symbol name>: the definition in <filename> is <this type of symbol> and the definition in <filename> is <this type of symbol>.

Cause. Each of the two files mentioned in the message defined symbols items of the same name, as shown in the message. However, they were different types of symbols, as shown in the message, and that is not allowed. For instance, the message may say that one of the symbols was "code", meaning a procedure, whereas the other symbol was "data".

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define symbols with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. You cannot have a procedure and a data item of the same name.

1596 Illegal duplicate definition of the procedure <symbol name> in <filename> and <filename>.

Cause. Each of the two files mentioned in the message defined procedures of the same name, as shown in the message. There are procedures that are specially created by the C++ compiler, where the compiler marks them to say that duplicates are okay, but at least one of the copies of this procedure was not so marked. Also, you did not specify the -allow_duplicate_procs option. So, the duplicates are not allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define procedures with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you really did intend to do this, you can use the -allow_multiple_procs option to tell `eld` it is okay. In that case, `eld` will pick one copy to use, so you must be sure that is the one you really want to use. If one copy of the procedure has the resident attribute, `eld` will pick that one. Otherwise, `eld` will pick the first one it sees, so you can affect that by the order in which you specify the input object files on the command line. If the procedures were created by the C++ compiler, but not marked to say that duplicates are okay, it could be that you need to write your C++ code differently or use different compiler options. The details of the C++ rules are beyond the scope of this manual.

1597 Illegal duplicate definition of the initialized data item <symbol name> in C and/or C++, occurring in <filename> and <filename>.

Cause. Each of the two files mentioned in the message defined data items of the same name, as shown in the message, and each of these files was written in C or C++, and in each case the compiler called it an “initialized” data item. As a rule, it is illegal in C or C++ to have duplicate initialized data items. However, there also are data items that are specially created by the C++ compiler, where the compiler marks them to say that duplicates are okay, but at least one of the copies of this data item was not so marked. So, the duplicates are not allowed.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define data items with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you are using the same data item in more than one place, only one of those places needs to be a definition, and the other places can just be external references to that definition. Review the rules for what makes a declaration a definition, depending on the source language that you are using, because the rules are different for each language. If you really do intend to have two definitions of this data item, visible across separate compilations, and the languages involved are C or C++, then at most one copy of the data item is allowed to be initialized. In C++, data items are always initialized, but in C that is not necessarily true. In any case, that also is something that involves the rules of the language. Modify your source code to get past these rules. Or, it also is possible that the data

items were created by the C++ compiler, and not marked to say that duplicates are okay, but you need to write your C++ code differently or use different compiler options so that the compiler does say that the duplicates are okay. The details of these C++ rules are also beyond the scope of this manual.

1598 Illegal duplicate definition of the procedure <symbol name> in <filename> and <filename> because of a procedure attribute mismatch.

Cause. Each of the two files mentioned in the message defined procedures of the same name, as shown in the message, but the procedures differ in certain of their "attributes", and for that reason the duplicates are not allowed. The attributes on which all the copies are required to agree, or else this message would appear, are "main", "shell", extensible", and "compiled_nonstop". The reasons that procedures acquire these attributes depend on the source language and are beyond the scope of this manual.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Did you really intend to define procedures with the same name in each of these two files, and have both definitions visible across separate compilations? If not, change the name of one of them, or change the declaration of one of them so that it is only visible within its own compilation. If you are using the same procedure in more than one place, only one of those places needs to include a copy of the code, and the other places can just be external references to that definition. If you really do intend to have two definitions of this procedure, visible across separate compilations, then the attributes listed above must be the same. If the attributes are different because the two files were created from different versions of the source code, or by using different compiler options, repeat the compilations doing things more consistently.

1600 Different values were specified for the process subtype in <filename> and <filename>, and no -set process_subtype option was given.

Cause. It is possible to use the `-r` option to build a new object file that can again be used as `eld` input, and when you do that you can also use the -set process_subtype option to assign a numerical subtype to that object file. When `eld` is then given that object file in a subsequent link, it copies over the subtype to its output object file again. However, in the present link, two of the input files had subtypes assigned to them by previous links, and the numerical values assigned in those previous links were different, and the present link did not use the -set process_subtype option to resolve the ambiguity.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you didn't intend for your previous links to specify inconsistent process subtypes, you should clean that up. If you do want various object files to have different process subtypes, and sometimes to be linked together, then when you link them

together you must specify the -set process_subtype option again to resolve the ambiguity.

1601 Different values were specified for the process subtype in <filename> and <filename>; the value given in the -set process_subtype option was used.

Cause. It is possible to use the `-r` option to build a new object file that can again be used as `eld` input, and when you do that you can also use the -set process_subtype option to assign a numerical subtype to that object file. When `eld` is then given that object file in a subsequent link, it copies over the subtype to its output object file again. However, in the present link, two of the input files had subtypes assigned to them by previous links, and the numerical values assigned in those previous links were different, and the present link used the -set process_subtype option to resolve the ambiguity.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If your previous links specified inconsistent process subtypes, and you didn't intend to do that, you should clean that up. If you do want various object files to have different process subtypes, and sometimes to be linked together, where you specify a -set process_subtype option to resolve the ambiguity, then you will have to live with this warning message about it.

1602 The value given in the -set process_subtype option was used, but a different value was found in <filename>.

Cause. It is possible to use the `-r` option to build a new object file that can again be used as `eld` input, and when you do that you can also use the -set process_subtype option to assign a numerical subtype to that object file. When `eld` is then given that object file in a subsequent link, it copies over the subtype to its output object file again. However, in the present link, some of the input files had subtypes assigned to them by previous links, and all those values were the same, but a different value was specified by a -set process_subtype option in the present link.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. If there was no need for the input object files to say they had a different process subtype than what you wanted to specify for the present link, you should clean that up. If you do want various object files to have process subtypes, and sometimes to be linked together, where you specify a -set process_subtype option with a value that doesn't agree with what the input object files said, then you will have to live with this warning message about it.

1603 Bad input file <filename>; this file is invalid because the code section named <section name> is larger than 16 megabytes.

Cause. As part of the rules for the TNS/E software architecture, there is a restriction on the types of object files that can be given to the linker. An exact statement of the restriction is hard to give, but here is the general idea. The code within an object file can be divided into multiple "sections". You will probably run into the limitation if any one of these code sections is close to 16 megabytes in size. Our compilers know to divide the code into multiple code sections so that no single code section gets too large. But, as the message indicates, one of the files did have a code section that was larger than 16 megabytes. `eld` doesn't wait to see if this would actually run into the restriction, but just relies on the heuristic that a code section this large is bad.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. This might indicate a compiler problem, because we shouldn't create this type of object file in a compilation. If the compiler always creates one code section, with the same name every time, then splitting your source into multiple compilations may not help much, because if all the compilations have code sections of the same name then you won't be able to put them through `eld` with the `-r` option, to create a new object file that could be used as `eld` input again, because `eld` will refuse to combine these object files into a new object file that would end up with a code section of that same name that was too big. You could split your code up into separate compilations and then give them all to `eld` without the `-r` option, to directly build a program or DLL out of them, because there is no restriction on the final program or DLL having a code section of any size. But you probably want to report this to HP, so that the reason this came up can be analyzed.

1604 The -r option cannot be used for this link because the output code section named <section name> would come out larger than 16 megabytes. An alternative is to put your object files into an archive, say 'x.a', and then build your program or DLL by saying '-all x.a' on the `eld` *command line.*

Cause. As part of the rules for the TNS/E software architecture, there is a restriction on the types of object files that can be given to the linker. An exact statement of the restriction is hard to give, but here is the general idea. The code within an object file can be divided into multiple "sections". You will probably run into the limitation if any one of these code sections is close to 16 megabytes in size. You used the `-r` option of `eld` to tell it to create a new object file that can be used as linker input again, so the limitation applies to this output file. `eld` will combine code sections that have the same name in its input files into a single code section of the same name in the output file. Our compilers know to divide the code into multiple code sections so that no single code section gets too large, even when later combined by `eld` for the `-r` option. But, as the message indicates, one of the code sections would come out larger than 16 megabytes in the `eld` output file. So, to avoid possible future problems in using this file, `eld` refuses to create it.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. This might indicate a compiler problem, because we shouldn't create code sections in a way that leads to this problem. You could avoid using the `-r` option, and

instead directly build a program or DLL out of your input files, because there is no restriction on the final program or DLL having a code section of any size. But you probably want to report this to HP, so that the reason this came up can be analyzed.

```
1605 The -import_lib or -import_lib_stripped option is only
allowed when creating a new DLL.
```

Cause. You did not specify an option such as `-dll`, `-shared`, or `-ul`, which would tell `eld` that you want to make a DLL, but you specified the -import_lib or -import_lib_stripped option, which tells `eld` that it should make an import library along with making the DLL that the import library would represent.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is not to make an import library, then don't specify the -import_lib or -import_lib_stripped option. If you are trying to create a DLL as well as an import library to represent it, then you need to specify the proper options to create a DLL, including `-dll`, `-shared`, or `-ul`. If you want to create an import library for a DLL that already exists then the option that you should be using is -make_import_lib.

```
1606 The -make_implicit_lib option is not allowed with the
-import_lib or -import_lib_stripped option.
```

Cause. The -make_implicit_lib option is used to create one of the implicit DLLs that constitute system library. The -import_lib or -import_lib_stripped option is used to make an import library to represet a DLL at the same time that you create that DLL. However, you cannot create an import library to represent one of the implicit DLLs. Instead, the set of implicit DLLs is represented by the zimpimp file.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to make an ordinary DLL, not one of the implicit DLLs, then do not specify the -make_implicit_lib option. If your intention is to create one of the implicit DLLs, as part of the process of building the operating system, then do not specify the -import_lib or -import_lib_stripped option. If your intention is to create a zimpimp file, the process is to first create all the implicit DLLs and then create the zimpimp file using the -make_import_lib option.

```
1607 Cannot open <filename>, the file expected to be the
zimpimp file.
```

Cause. `eld` uses the zimpimp file to know the addresses for symbols in the operating system. You are running `eld` on a TNS/E machine, and the operating system has told `eld` where the standard zimpimp file is, but `eld` was unable to open it.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. This indicates a problem with your installation. The process for creating the zimpimp file, installing it in the right place, and providing the system call that `eld` uses to find the zimpimp file, is beyond the scope of this document.

1608 The <option name> option is not allowed with the -strip option.

Cause. The `-strip` option is used to remove the DWARF symbols from an existing program or DLL. When you use this option, only certain other options are allowed. However, you specified the option mentioned in the message, which is not one of the options allowed with the `-strip` option.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to use the `-strip` option to remove the DWARF symbols from an existing program or DLL, don't specify the other option mentioned in the message. If your intention is not to do this, then don't specify the `-strip` option.

1609 DWARF symbols have been removed from <filename>.

Cause. You gave the `-strip` option, to tell `eld` to remove the DWARF symbols from an existing program or DLL, and `eld` did that.

Effect. Information (This is not indicative of a problem).

Recovery. No action required.

1610 There were no DWARF symbols in <filename>, so the file was not modified.

Cause. You gave the `-strip` option, to tell `eld` to remove the DWARF symbols from an existing program or DLL, but that file contained no DWARF symbols information. Either it was built from object files, none of which themselves had any DWARF symbols information, or the file had already been stripped before.

Effect. Information (This is not indicative of a problem). `eld` has not done anything.

Recovery. No action required.

1611 The value given in the -set process_subtype option was used, but a different value was found in the input file.

Cause. You gave the `-r` and `-set _process_subtype` options, to tell `eld` to create a new object file that can be used as `eld` input again, and that specifies what the process subtype should be when a program is later built from that object file. Also, you provided just one input object file to `eld`, and that file was also previously created by `eld`, using the `-r` and `-set_process_subtype` options, but with a different value

specified for the process subtype. The new file that `eld` creates will have the process subtype that you specified this time, not the value that was in the previous file.

Effect. Warning (`eld` produces an output file, but it might not be what you intended).

Recovery. Assuming you gave the file the wrong process subtype before, or had some other reason to change it now, no action is required.

1612 The floating point type of the input file, which was <string>, has been changed to <string>.

Cause. You gave a `-r` option, and exactly one input object file, to tell `eld` to create another object file that could be used as `eld` input again, and you also gave the `-set floattype option`, to tell `eld` which floating point type to indicate in the output object file, and it was different from what the input file contained.

Effect. Information (This is not indicative of a problem).

Recovery. Assuming you gave the file the wrong floating point type before, or had some other reason to change it now, no action is required.

1613 The -set user_buffers option is not allowed with the -r option.

Cause. You used the `-set user_buffers` option, which affects how a program performs I/O, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program, then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used as `eld` input, then don't specify the `-set user_buffers` option.

1614 DLL registry, line <number>: end of file in the middle of a -range command.

Cause. While reading the private DLL registry that was specified for this link in the `-check_registry` or `-update_registry` option, `eld` ran off the end of the file when it was expecting to see the name of a DLL after the `-range` keyword. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

1615 DLL registry, line <number>: the new DLL is listed multiple times in the registry.

Cause. The private DLL registry that was specified for this link in the `-check_registry` or `-update_registry` option contained more than one `-range` command for the DLL that is being created by this link. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual.

1616 DLL registry, line <number>: unrecognized command in the registry file.

Cause. While reading the private DLL registry that was specified for this link in the `-check_registry` or `-update_registry` option, `eld` encounted an invalid command on the line shown in the error message. Presumably, the format of the file is bad because it was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file, as explained elsewhere in this manual. The two possible commands are "`-dllarea`" and "`-range`".

1617 DLL registry, line <number>: DLL address is not a mulitiple of the page size.

Cause. The addresses listed for DLLs in a private DLL registry must be multiples of 64KB if the linker is not building an implicit DLL, or 128KB if the linker is building an implicit DLL (i.e., a component of system library). However, that rule was violated by the `-range` command on the indicated line number of the file. That could be because the file was incorrectly edited by hand. Or, if the address is a multiple of 64K, but needed to be a multiple of 128K, that could be because this private DLL registry was previously used to build a DLL that wasn't an implicit DLL, but now is being used to build an implicit DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file so that the address is on the proper boundary. A private DLL registry that is used to build ordinary DLLs should not also be used to build implicit DLLs.

1618 DLL registry, line <number>: DLL size is not a multiple of the page size.

Cause. The sizes listed for DLLs in a private DLL registry must be multiples of 64KB if the linker is not building an implicit DLL, or 128KB if the linker is building an implicit DLL (i.e., a component of system library). However, that rule was violated by the `-range` command on the indicated line number of the file. That could be because the file was incorrectly edited by hand. Or, if the address is a multiple of 64K, but needed to be a multiple of 128K, that could be because this private DLL registry was previously used to build a DLL that wasn't an implicit DLL, but now is being used to build an implicit DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file so that the size is a multiple of 64K or 128K, as appropriate. A private DLL registry that is used to build ordinary DLLs should not also be used to build implicit DLLs.

1619 DLL registry, line <number>: DLL address is below the lower limit.

Cause. The addresses listed for DLLs in a private DLL registry must be at least as large as the lower bound of the address range given by the `-dllarea` command, or 0x70000000 if an explicit `-dllarea` command is not present. However, that rule was violated by the `-range` command on the indicated line number of the file. Presumably, that was because the file was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file so that the addresses of all DLLs are at least as large as the applicable lower bound.

1620 DLL registry, line <number>: DLL extends above the upper limit.

Cause. The ending addresses listed for DLLs in a private DLL registry, as calculated by adding their starting addresses to their reserved sizes, must not exceed the upper bound of the address range given by the `-dllarea` command, or 0x80000000 if an explicit `-dllarea` command is not present. However, that rule was violated by the `-range` command on the indicated line number of the file. Presumably, that was because the file was incorrectly edited by hand.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. Fix the format of the file so that the ending addresses of all DLLs do not exceed the applicable upper bound.

1621 No entry for DLL <filename> in the given DLL registry.

Cause. You gave the `-check_registry` command, to specify that the starting address for the DLL being created by this link is to be taken from the indicated private DLL registry file. However, that DLL was not listed in the private DLL registry file.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to use a private DLL registry file to specify the address for the DLL you are building, the registry must contain a range command giving the same filename as the DLL you are building, such as you might specify with the `-o` option. You can use the `-update_registry` command to create such a registry file, or you can edit one by hand. If you just want to record the decision that `eld` made for the DLLs address in a registry file not specify the address unconditionally with the registry file, then you should use the `-update_registry` option rather than the `-check_registry` option.

1622 DLL size exceeds the value specified by the -grow_limit option.

Cause. You have given the `-grow_limit` option, which specifies a maximum size for the code and data of the DLL you are creating, and the DLL was larger than that.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. You wanted it to be an error if your DLL grew beyond a certain size, and now it has. Do whatever you planned to do at such a time, such as changing your decisions as to what addresses to assign to DLLs. Or, if you would now like to allow a larger size, specify a larger size for the `-grow_limit` option. Note that the size that you specify is rounded up to a multiple of 64K if you are creating an ordinary DLL, or 128K if you are making one of the implicit DLLs that constitute system library. The DLL has a code segment and a data segment, each of which has a size that is similarly rounded up, and the sum of those two sizes is what is being compared to the value you specify.

1623 No available address range in the DLL registry for this DLL.

Cause. You gave the `-update_registry` option to tell `eld` to choose an address for the DLL that you are creating by using a private DLL registry. `eld` has determined the amount of space it needs to reserve for the DLL, based on its size and the other options that you have given. The private DLL registry specifies a range of addresses that can be assigned to DLLs, and tells which subranges are already occupied by other DLLs. There was no available block of addresses large enough for this new DLL.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. You wanted it to be an error if `eld` couldn't find a block of space large enough, and it happened. Do whatever you planned to do at such a time, such as reorganizing your DLLs and deciding which ones can share memory addresses and

which can't, or maybe just allowing them to take up more space. Perhaps there are entries in the private DLL registry that you don't need. You can just delete those lines of the registry by hand, or you could start over with a new registry and relink all the DLLs you really need, so you only have entries for them in the new registry. Note that the registry keeps track of the filenames that you create. So, for example, if you keep using the same registry, and keep building the "same" DLL, except for giving it a different filename with the `-o` option each time, then each of those builds will put another line into the registry, using up more of its space. That isn't something you want to do.

1624 DLL size exceeds the size taken from the DLL registry.

Cause. You have given either the `-alf` option, to update an existing DLL, or you have given the `-make_import_lib` option, to create the zimpimp file that represents the implicit DLLs that constitute system library. In either case, you have also given the `-check_registry` option, to say that the file being created or updated must fit in the memory space assigned to it by a private DLL registry. But, the total size of the segments of the DLL or import library, including the required rounding up to certain alignments, was larger than the size specified in the private DLL registry.

Effect. Fatal error (`eld` immediately stops without creating or updating the output file).

Recovery. The use of the `-check_registry` option means that you have been using a private DLL registry to keep track of memory addresses of DLLs, so that you always rebuild them at the same addresses as before, and so that it will be an error if one of them grows larger than the space allocated for it. Now that has happened. Assuming you wanted to be doing this, and want to keep doing it, you now need to make new decisions about where the DLLs should go. You could do that by editing the private DLL registry file by hand, or by deleting some entries from the private DLL registry and letting `eld` choose new locations for those DLLs with the `-update_registry` option.

1625 Error updating the DLL registry. The registry is unchanged.

Cause. You gave the -update_registry option, and `eld` was trying to update the private DLL registry that you specified. `eld` does this by first creating a temporary file in the same location (Guardian subvolume, OSS directory, or PC folder) as the existing registry. `eld` was either unable to create this temporary file, or else had a problem writing to it after it was created.

Effect. Warning. `eld` will still create the DLL that you wanted to create, but the private DLL registry itself has not been modified.

Recovery. Check that you have permission to create files in the same location as the existing private DLL registry, and that it is not on a Guardian subvolume that is full.

```
1626 Cannot create the DLL registry <filename>; using this
name instead: <filename>.
```

Cause. You are trying to update a private DLL registry. eld first makes a new copy of it in a temporary location, deletes the previous copy of the registry, and then renames the temporary file to the final location. That process failed. The file has instead been left with a different name, as shown in the message.

Effect. Warning (eld produces the registry, but not with the filename you intended).

Recovery. If you don't have permission to delete the previous copy of the registry, specify a different filename for the new copy. If eld succeeded in deleting the old copy, but then could not rename the new copy, that is an operating system question that is beyond the scope of this document.

```
1627 The DLL registry -range entry for this DLL is changing
from '<address> <size>' to '<address> <size>'.
```

Cause. You used the -update_registry option to tell eld the private DLL registry to use for deciding the address of the DLL that you are creating. The private DLL registry had an entry for that DLL already. However, the size that eld wants to reserve in the registry for the new version of the DLL, which depends on its actual size and the other options that you have given, is larger than the amount of space that the registry previously reserved for this DLL. There was a block of space in the registry that was large enough for the DLL, and eld has put it there. This might or might not be the same starting address as before, but it definitely is a larger size than before.

Effect. Warning (eld produces the DLL, and updates the registry, but the size and maybe also the address of the DLL has changed).

Recovery. If the new address or size is satisfactory, you don't need to do anything.

```
1628 Cannot create DLL registry file <filename>.
```

Cause. You are trying to update a private DLL registry. eld first makes a new copy of it in a temporary location, deletes the previous copy of the registry, and then renames the temporary file to the final location. That process failed. You also gave the -must_use_rname option to say that eld should consider that failure an error.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you don't have permission to delete the existing registry, specify a different filename for the new version of it. Or, if there is some reason why you cannot rename one filename to another filename in the target location (Guardian subvolume, OSS directory, or PC folder), that could also lead to this message, but that is an operating system question that is beyond the scope of this document..

1629 Different values specified with multiple -set user_buffers options.

Cause. You gave the `-set user_buffers` option more than once on the command line, with different attribute values. (The possible values are “on” and “off”.) You can give the option more than once, but only if you specify the same value each time.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If you want to specify this option, decide which value you want to specify, and only specify that value.

1630 Addresses go beyond 0x80000000; the segment that starts at <number> has size <number>, so it goes up to <number>.

Cause. You are building a program or a DLL, so there is a code segment and a data segment. `eld` has chosen addresses where each of the segments begins, perhaps based on options that you gave, and `eld` has determined how large each of the sections is. One of the sections has a starting address below 0x80000000 and a size large enough so that it extends beyond 0x80000000. That is an error, because segments with such addresses are not supported by the operating system.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If the problem occurred because you specified a value for the `-t` or `-d` option that was close to 0x80000000, don’t do that. By default, `eld` will start a DLL at 0x78000000, which gives 256 megabytes for its code and data before reaching the 0x80000000 boundary. If you file is really larger than that, it probably is too large for the operating system to handle in any case. Splitting it into multiple DLLs may also not help, if they all need to be in memory at the same time. You probably must find some way to make your code or data smaller.

1631 This loadfile has kernel-callable procedures but $n_MillicodeCheckRV is not present.

Cause. One of the attributes that can be given to a procedure is the `kernel_callable` attribute, but this is a special attribute that is only supposed to be used by procedures in the millicode DLL that is part of system library. In particular, the use of this attribute can only work if the same DLL also contains a symbol named $n_MillicodeCheckRV, which the millicode DLL is supposed to contain. However, you have asked `eld` to build a DLL that contains `kernel_callable` procedures but does not contain $n_MIllicodeCheckRV.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. This is a problem with building the operating system, which is beyond the scope of this manual.

```
1632 Can't open public DLL file named <filename> or
<filename>.  Such a file should be in the same place as the
zreg file, <filename>.
```

Cause. eld was searching for a DLL, and you were using either the PC or OSS version of eld, rather than the Guardian version. The public DLL registry file, called the "zreg file" in the error message, contained the name "zxxxdll" that eld was looking for, which means that this is one of the standard DLLs that is supposed to exist in a certain location. Specifically, it is required that there be a file named either "zxxxdll" or "libxxx.so" in the same location (OSS directory,or PC folder) as the public DLL registry file. But, either such a file does not exist, or you don't have permission to read it.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. There are various ways that eld may find the public DLL registry file. For example, if you are running eld on TNS/E then the operating system tells eld where the file is. In other cases, eld looks for it in an appropriate place, expecting it to have the name "zreg". The first place eld looks is its own location (Guardian subvolume, OSS directory, or PC folder), and if there is no "zreg" file there then it will also look in a sibling directory or folder whose name ends "lib". Or, you can override these methods by explicitly telling eld where it is with the -public_registry option. One of the files mentioned in the message was supposed to exist in the same place as that public DLL registry file, but wasn't. If you gave the -public_registry option, then you are responsible for setting up all the files correctly. Otherwise, there is something wrong with your installation. The procedure for creating and installing a public DLL registry file is beyond the scope of this manual

```
1633 <filename> is an archive, but the filename has the form
reserved for public DLL's.
```

Cause. eld searched for the name specified in the message, and found it in the public DLL registry file, meaning that this is one of the standard DLLs. And, eld found a file of an appropriate name (of the form zxxxdll or libxxx.so) in the standard place, but when eld opened it eld found that the file was an archive rather than a DLL.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. This indicates that something is wrong with your installation. The method creating the public DLL registry file, and making sure that the standard DLLs with the right names are located in the same place (Guardian subvolume, OSS directory, or PC folder) as that registry file, are beyond the scope of this manual.

```
1634 Cannot give the -call_shared option with the -r option.
```

Cause. You used the -call_shared option, which tells `eld` to create a program, and you also used the `-r` option, to tell `eld` to build another object file that can be used as linker input, rather than a program.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program, then don't specify the `-r` option. If your intention is to use the `-r` option to create a new object file that can be used as `eld` input, then don't specify the `-call_shared` option. Actually, there is never any reason to specify the `-call_shared` option, because it is the default.

```
1635 Cannot give the -call_shared option with the -dll
option.
```

Cause. You used the `-call_shared` option, which tells `eld` to create a progrm, and you also used the `-dll`, `-shared`, or `-ul` option, to tell `eld` to build a DLL, rather than a program.

Effect. Fatal error (`eld` immediately stops without creating an output file).

Recovery. If your intention is to create a program, then don't specify the `-dll`, `-shared`, or `-ul` option. If your intention is to create a DLL, then do specify one of these options. There is never any reason to specify the `-call_shared` option, because it is the default.

```
1638 Cannot create -temp_r file <filename>; -temp_r option
ignored.
```

Cause. When you specify the `-temp_r` option, `eld` still first creates a temporary file in another place, and when that file is created `eld` then tries to rename it to the filename specified in the `-temp_r` option. That renaming failed.

Effect. Warning (`eld` still (re-)creates the private DLL registry, but not using the file you specified with the `-temp_r` option as an intermediate file).

Recovery. If you are not able to rename a file to another name in the same location (Guardian subvolume, OSS directory, or PC folder), that is an operating system question that is beyond the scope of this document

```
1639 No <type of parameter required> was specified for the
<option name> option.
```

Cause. You gave the option shown in the message, and that option requires a certain type of parameter, such as the name of a file or the name of a symbol, as also shown in the message. However, there was no parameter at all. Either the option occurred at the end of the command line, or the next token on the command line began with a hyphen, meaning that it was the next option, not a parameter for the current option.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Assuming you really do want to specify this option, give the correct parameter(s) to it.

1640 No filename was specified for the libname attribute of the <option name> option.

Cause. You gave the -set or -change option, with the "libname" attribute, and this requires that the next parameter on the command line be the Guardian filename that the user library will have at runtime. However, there was no parameter at all. Either the option occurred at the end of the command line, or the next token on the command line began with a hyphen, meaning that it was the next option, not a parameter for the current option.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. Assuming you really do want to specify this option, give the correct parameter(s) to it.

1641 A string starting with an equal sign is only allowed as the libname attribute of the <option name> option in the Guardian version of eld*.*

Cause. You specified the -set libname or -change libname option, where the parameter that you specified for the name of the user library began with an equal sign. In the Guardian case this would be allowed, where eld would treat the parameter as a "Guardian DEFINE" and expand it to the name to use for the user library name. But, you are running eld on the PC or on OSS, and on these platforms this type of parameter is not allowed.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really wanted to use a name that begins with an equal sign for a user library name, you can't. On the PC or OSS, the Guardian DEFINE mechanism is not present, so you need to directly specify the intended user library name.

1642 A string starting with an equal sign is only allowed as a command line filename in the Guardian version of eld*.*

Cause. You specified a string directly on the command line, not as the parameter of some option, and the string began with an equal sign. In the Guardian case this would be allowed, where eld would treat the parameter as a "Guardian DEFINE", and expand it to a filename. But, you are running eld on the PC or on OSS, and on these platforms this type of command line string is not allowed.

Effect. Fatal error (eld immediately stops without creating an output file).

Recovery. If you really wanted to use a name that begins with an equal sign in this context, you can't. On the PC or OSS, the Guardian DEFINE mechanism is not present, so you need to directly specify the intended filename.

```
1657 Common data is larger in size, possible loss of data.
```

Cause. Multiple global symbols are present in the link files. The common data file is larger in size than the defined data file.

Effect. The application might not work because of possible data loss.

Recovery. Ensure that the application does not include duplicate global symbols of different sizes.

```
1659 < filename>: the resident procedure <procedure name> is
in <section name>, which is not a resident code section.
```

Cause. The contents of the specified input filename might not be correct. As a result, `eld` does not process the file.

Effect. Fatal error. `eld` stops immediately without creating an output file.

Recovery. Contact your HP representative.

```
1660 <filename>: the procedure <procedure name> is in
<section name>, which is a resident code section, but is not
a RESIDENT procedure.
```

Cause. The contents of the specified input filename might not be correct. As a result, `eld` does not process the file.

Effect. Fatal error. `eld` stops immediately without creating an output file.

Recovery. Contact your HP representative.

```
1665 The -set data_model neutral option is not allowed when a
program is being created.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file..

```
1666 Bad value given for the -change data_model option; the
only allowed value is 'neutral'.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file.

```
1667 The -change data_model option is not allowed because the
existing file <filename> is a program.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file.

```
1668 Using <filename>, even though it does not have the
desired data model.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Warning. (`eld` (re-)creates the private DLL registry without using the file you specified with the `-temp_r` option as an intermediate file).

```
1669 Bad value given for the -set data_model option; the
allowed values are ‘ilp32’, ‘lp64’, and ‘neutral’.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file.

```
1670 The -set data_model option was given multiple times with
different values specified.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file.

```
1672 Input files contain a mixture of 32-bit and 64-bit data
models. The first 32-bit is <filename>. The first 64-bit is
<filename>. To allow this, use the –set data_model neutral
option.
```

For details on **Cause** and **Recovery**, see eld Functionality for 64-Bit on page 4-12.

Effect. Fatal error. `eld` stops immediately without creating an output file.

Glossary of Errors

This glossary of errors provides more information about the words that appear in `eld` error messages. Some glossary entries may also help you understand other glossary entries.

There is also a main Glossary at the end of this reference manual that contains explanations of the major concepts in linking and loading files on a TNS/E system.

Archive. An archive is a file that contains within it one or more files, called the members of the archive. In general, the members of an archive can be any kinds of files, but the members should probably only be TNS/E linkfiles if the archive is intended to be used with `eld`. `eld` can read an archive and use some of its linkfiles as inputs for a link.

Bad input file. When `eld` says this, it means that `eld` believes there is something wrong with the named input file, and the next step is to see where that file came from and why it is bad. For example, that file may have been an object file created by a compiler, or by a previous `eld` command with the `-r` option.

Callable procedure. A callable procedure is a procedure that has the "callable" attribute, which means that code enters privileged mode when that procedure is called. The linker creates "gateways" to make that happen. There is a special type of callable procedure, called a "kernel callable" procedure, that is only used in one of the special system DLL's, and that DLL must also contain the procedure named $n_MillicodeCheckRV. All other callable procedures are called "user callable" procedures, and they are required to have from 0 to 32 parameters.

Create. When `eld` says that it cannot create a file, or cannot open a workfile that will eventually become some file, that may mean that the name you specified for such a file was invalid, or that `eld` didn't have permission to create a file in that location.

Data segment. A data segment is that portion of a program or DLL that contains writeable data that is allocated once before the process begins execution (i.e., as opposed to data that is a runtime stack or heap).

Definition. The definition of a symbol means that the symbol is present in the given object file, such as a procedure whose code is present, or a data item for which space has been allocated. It does not mean a symbol that may be declared/referenced here, but actually exists elsewhere. Each language has its own rules for which syntactical elements define, or don't define, symbols.

Delete. When `eld` is told to create an output file, and a file of that name already exists, `eld` tries to delete it. If `eld` says that it cannot delete a file, that can indicate that `eld` does not have permission to delete that previously existing file.

DLL. A DLL (dynamically linked library) is an object file that is loaded into memory as part of a process. Each process that you run contains exactly one program and any number of DLL's. `eld` builds DLL's out of linkfiles. `eld` also uses DLL's to resolve references when it it building a new program or DLL, or when the `-alf` option is used. Options such as the `-l` option are used to tell `eld` which DLL's to use to resolve references.

DLL name. A DLL name is an identifying string found within a DLL, and is not necessarily the same as the DLL's filename. DLL names are stored in the liblist portion of a loadfile.

DLL registry. This is a file that can be used to tell `eld` which addresses to use when creating a DLL, and/or to record the choices `eld` made about such addresse.

DWARF. This is the name for the industry-standard format of the symbol table information that is used for debugging (not for most linking activities).

ELF. This is the name for the industry-standard object file format upon which our object files are based.

Export. Symbols that exist in a DLL are not visible outside that DLL, to resolve references in other programs and DLL's, unless they are exported. Some symbols are marked for export by the compiler, and others can be specified to be exported by using `eld` options such as `-export` or `-export_all`.

Filecode. On Guardian, files have "filecodes". TNS/E object files are always given file code 800.

Find. See "Search".

Floating point. Guardian supports two types of floating point formats, named "`tandem`" and "`ieee`". Normally, you should compile all your files the same way so as not to mix the two types.

Gateway. This is code that `eld` creates for callable procedures to perform appropriate steps to enter privileged mode.

GP-relative addressing. This refers to a method of addressing data that adds a 22-bit value to the contents of a specific register called the GP register. Therefore, all the data that is addressed this way, within a given program or DLL, must lie within a range of 4 MB, as laid out by `eld`, or else `eld` reports an error. Data that requires GP-relative addressing includes data so specified by compilers (often called "short data") and various tables created by `eld`. If you are building a loadfile that runs into this error situation you must try to link it a different way, perhaps splitting it up into multiple DLL's rather than making it a single program or DLL. If an input file defines a symbol named "__gp", that affects the value that is in the GP register and therefore can lead to this error situation, even if the amount of data that requires GP-relative addressing is not large.

Implicit DLL's. These are the DLL's that collectively form "system library", and are collectively represented by the special import library called the zimpimp file.

Import controls. When a program or DLL is built, you specify whether it will be "localized", "globalized", or "semi-globalized" by using the `-b localized`, `-b globalized`, or `-b semiglobalized` options. The default is `localized`. Also, when you build a DLL, and it refers to other DLL's by use of `-l` options, you specify whether those other DLL's are "reexported" by using the `-reexport` option. By default, DLL's are not reexported. If you are building a program or a DLL, and there is a reference to a symbol, and the symbol is not present in the same file, then `eld` will look for it at least in the other DLL's that you specify on the command line, such as with `-l` options. `eld` will also look in other DLL's ("indirect DLL's") pointed at by DLL's it has already opened, but only if you are building a file that is globalized or semi-globalized, or if the DLL's that point at such indirect DLL's also reexport them.

Import library. An import library is a file that is like a DLL but only with the header information. It can be used by `eld` to resolve references when building another program or DLL, but cannot be loaded into memory in place of the real DLL at runtime. An import library can be marked "incomplete", such as by the `-set incompete on` option of `eld`. When an import library is incomplete, `eld` may still use it to see if there are unresolved references, but not to preset.

Internal error. An internal error is something that is never supposed to happen and therefore should be reported to HP.

Liblist. This is the name of the section of a DLL that tells the names of the other DLL's at which it points.

Linkfile. A linkfile is an object file that is created by a compilation or by an invocation of `eld` with the `-r` option. Linkfiles themselves cannot be run from a command line, but can be provided as inputs to `eld` so that `eld` can create a new object file (linkfile or loadfile) from them.

Loadfile. A loadfile means a program or a DLL as opposed to a linkfile.

Main entry point. A program needs a main entry point. In pTAL, you specify this by putting the "main" attribute on a procedure. Or, if you forget to do that, you can use the `-e` option of the linker to tell the name of the main program. In C or C++ you should not use the `-e` option, but rather a special linkfile containing the main program (not the procedure in your own file named "main") needs to be linked together with the files you have compiled. When you have the C or C++ compiler invoke the linker for you, it should do that automatically. In Cobol you also should not use the -e option.

MAP DEFINE. This is a feature, available only on Guardian, that allows a string that begins with an equal sign to represent a filename on the `eld` command line.

Member. A file within an archive.

Memory. If the linker says it is "out of memory", that should mean that there wasn't enough memory available for the linker to do its job. If possible, try again where more memory is available. Or, this may be a problem that needs to be reported to HP.

Memory area. For a program, the text memory area consists of the text segment plus the gateway segment, and the data memory area consists of the data segments. For a DLL, the text memory area consists of the text segment, and the data memory area consists of the data segments plus the gateway segment. You can use the `-t` and `-d` options to tell `eld` which starting addresses to use for the two memory areas. Most users will not to this, or perhaps will only use the `-t` option when building a DLL. By default, for a DLL, the data memory area comes immediately after the text memory area, and another way of specifying this combined area for a DLL is with a DLL registry.

Obey file. This is a file that provides additional command line input to `eld`, as specified by the `-obey` option.

Open. If `eld` says that it cannot open a file it is possible that you specified the wrong file name, or that you shouldn't have specified any file name in the first place. If you

correctly specified a filename that eld needed to open, but eld says it can't open it, then you need to figure out why that is so. It may be that the file does not exist, or it may be that the file exists but you do not have permission to open it. If it is a file that eld only needs to read, then you would generally only need permission to read it, not write it.

Option names. eld has some options whose names are one letter long, and they take parameters that are not keywords, and a space is not required between the name of the option and the parameter. The examples are -d, -e, -l, -L, -o, -t, and -u. If you intended (correctly or not) to specify some other option whose name began with one of these letters, and what you entered was not the name of an option, eld will assume you meant to give one of these one-letter options with a parameter, and provide an error message accordingly. For example, eld has no such option as -elf, so if you specify -elf, eld will think you are specifying the -e option with the parameter lf. Many options have synonyms and the name shown in an eld message does not necessarily match the way it was written on the command line.

Preset. eld is said to "preset" a program or DLL when eld has filled in addresses for all references and created additional information within the file so that, at runtime, the operating system can realize that eld has already done this and possibly avoid the work of redoing it. eld will only mark a file "preset" when there is a chance that the addresses could be correct at runtime.

Procinfo. This is the name of the section of a linkfile that provides information about its procedures.

Program. A program is an object file that contains the main code for a process. eld builds programs out of linkfiles.

Public DLL registry. This is a file that should automatically be found by eld. It tells eld how to look for public DLL's in a standard location, in addition to other ways that eld would look for DLL's. Public DLL's are DLL's that are usually distributed by HP, rather than created by other users.

Read. If eld says that it cannot read a file, it is possible that you specified the wrong file name, or that eld did not have permission to open and read the file.

Rebase. The -alf option always updates the references within an existing loadfile, that is, it fills in pointers within this loadfile that contain the addresses of symbols that are not in this same loadfile but rather are found in other DLL's. In addition, if you tell the -alf option to change the address of the existing DLL itself, that is called "rebasing" that DLL.

Reference. A reference to a symbol means that the address of the symbol is needed in this place for some purpose. That symbol itself may or may not be present in the same object file.

Relocation table. This is the name a section of an object file that tells the locations of references and the symbols to which they refer.

Search. When the -l option is used, and its parameter is a simple (unqualified) filename, that tells eld to search for an archive or DLL based on that name. Some of

the places eld looks for the archive or DLL are the directories or subvolumes specified by -L options. It is usually an error if the search does not find something, but you can override that with the -allow_missing_libs option.

Section. An object file contains "sections", which have names, and sometimes the names imply properties of the section, such as whether the section contains code or data. There is a complicated architectural restriction on the maximum size of a code section in a linkfile, but a general statement is that the limit is probably reached at a size just under 16 megabytes. It is not expected that this limit will be reached, because we use many small code sections, rather than large ones.

SQL. This is a database language whose statements can be embedded in the code of other languages.

Symbol name. A "symbol" can be either a procedure or a data item. The "name" of a symbol means the name as it is represented within the object files that eld sees, which might not be the same as how the name appears in your source code, depending on what name manipulations were done by the compiler. For example, in C++, names may be "mangled". eld may try to show both the mangled and unmangled forms of C++ names in its error messages, but when a symbol name is provided as input to eld it must be the mangled C++ form.

Systype. A process runs either as a Guardian process or as an OSS process. You tell eld which type of process you are creating by giving the -set systype option while building a program.

Text segment. The text segment is that portion of a program or DLL that contains all the executable code (other than gateways) along with various header information and readonly data.

TNS/E ("Tandem Non-Stop/E"). This is the name of the HP operating system for which eld processes object files.

Unresolved reference. You can have an unresolved reference when eld is building a program or a DLL, or when eld is doing the -alf option. That means that the file you were building or processing needed to have the address of some named symbol filled in, and that symbol did not exist in that same loadfile, nor was eld able to find the symbol by looking into other DLL's. This may occur for many reasons, ranging from spelling errors in your source code, or things that you still need to write that you don't yet have in your source code, to problems with linkfiles or DLL's that other people are supposed to provide to you, which either they didn't provide or you didn't pass along for eld to use, or "standard" things not set up correctly in your installation. eld tells you the name of the symbol that it couldn't find. When the -verbose option is used, eld lists the DLL's that it found and used to look for symbols. If you think that eld should have looked at a certain DLL, and it didn't, review your use of -l options as well as your import controls. If you think that eld looked at a DLL, and didn't find a symbol there, but the symbol is there, note that the symbol also needs to be exported from the DLL for eld to find it. If you are using archives, you may also need to review the rules for which object files eld will use from the archive. The -unres_symbols option

controls whether eld considers unresolved references to be errors, warnings, or neither.

User library. A user library is a DLL. A program is allowed to specify at most one user library that it will need at runtime, and the user library is specified by storing its Guardian filename within the program, through the -set libname option of eld, rather than by telling eld to use a DLL the usual way, such as by a -l option.

Variable data. This refers to data that can be modified after the process has started running. This is different from data that is constant, or that contains addresses that may need to be modified when the process begins running but cannot be modified thereafter.

Workfile. eld first creates workfiles for its output files, and only at the end does it rename the workfiles to the final output filenames.

Write. If an error occurs when writing a file, it may indicate that there is no space left on that disk.

Zimpimp file. This is a special type of import library that represents all the symbols that exist in the system library.

A TNS/E Native Object Files

This appendix contains the following information:

The Object File Format - the types of object files and their content.

Code and Data Sections - the "ordinary" code and data sections that come from application source code, possibly with additions by the compiler or linker.

Relocation Tables - when code is relocated, who resolves the address and prepares relocation tables?

Finding Information About Procedures and Subprocedures in Linkfiles - an introduction to the .procinfo and .procnames sections of linkfiles.

The DWARF Symbol Table - this table contains information used by debuggers and the Cobol compiler.

Archives - contains an extension of material covered in a previous section of this manual.

Tools That Work With Object Files - a quick look at which HP NonStop operating system tools use object files.

The Object File Format

Some of this general information may also be found in Sections One and Two of this manual, but this appendix provides much more detail.

Basic Properties of Object Files

User versions of TNS/E tools may run in the following places:

- All TNS/E versions of the HP NonStop operating system, including both the Guardian and OSS environments.
- Some TNS/R versions of the HP NonStop operating system, at least in the Guardian environment.
- Appropriate versions of the Windows operating system on PC's.

TNS/E object files only run on TNS/E.

On Guardian and OSS, object files are unstructured files that are "odd unstructured", the same as in TNS/R.

On Guardian and OSS, TNS/E object files have the file code 800.

TNS/E object files use the 64-bit version of the ELF file format.

TNS/E object files are big endian. This means that all their data is big endian.

Types of TNS/E Object Files

There are the following four types of TNS/E object files.

Table A-1. Types of TNS/E Object Files

Type of Object File	Description
Linkfile	This is the term for the object files that are produced by a compiler or by the assembler, and can be given as input to the linker. It is also possible for the linker to produce a linkfile as output when run with the *-r* option.
Program	This is the term for a main program. There is one program in every process.
DLL	This stands for dynamic-link library. It is an object file that is not a program but can also be part of a process. A process can contain any number of DLLs. DLLs are also used by the linker when building other programs or DLLs.
Import Library	This is a file that contains just the part of a DLL that is needed at link time to build other programs or DLLs.

Collectively, programs and DLLs are called loadfiles. Loadfiles and import libraries are built by the linker.

This appendix describes all four types of object files. The main distinctions occur between linkfiles and loadfiles. There is little difference between a program and a DLL as far as the file format is concerned, and an import library is a subset of what is in a DLL.

A loadfile may refer by name to symbols that exist in other loadfiles in the same process. Such references are resolved when the loadfiles are brought into memory by the runtime loader, which is named `rld`, or by the runtime procedure named `dlopen`. When the loadfile was originally built by the linker it is also possible that the linker tried to resolve such references. A loadfile whose references have been resolved by the linker is said to be preset.

A process can also use one user library. A user library is a DLL. Nothing within a user library distinguishes it from other DLLs, and a DLL that serves as the user library for one program can also be used like any other DLL by other programs. The only difference between the user library and other DLLs is in the way the program identifies the user library that it uses. For a DLL to be used as a user library at runtime its filename must be in the Guardian namespace.

An import library can take the place of a DLL at link time. One use of import libraries is to save space. Another use is for security, when it is necessary for the linker to read the header information but it is not desirable for others to be able to see the code. Import libraries are further categorized as complete or incomplete. The difference is that an incomplete import library need not contain the correct addresses for symbols. A complete import library can be used by the linker when presetting a loadfile. The

linker can use an incomplete import library to check for unresolved references, but not to preset.

DLLs and import libraries can also be used at compile time by the COBOL compiler to find out information about procedure call interfaces.

Some DLLs are called public libraries because they are provided as part of the TNS/E implementation and are found in a special way by the linker and runtime loader. A public library has the same format as any other DLL, and can have an import library to represent it.

Some of the public libraries are called implicit libraries because they are used at link time and run time without explicit mention on the part of the user. There are several implicit libraries, and there is a bit in a DLL that tells if it is an implicit library. A single implicit library never has an import library to represent it to the linker. Rather, at link time, when building a loadfile that is not an implicit library, a single import library represents the entire set of implicit libraries. That is called the import library that represents the implicit libraries, and it is always a complete import library.

How to Distinguish the Different Types of Object Files

The first four bytes of an ELF file (in the ELF header) identify the file as an ELF file.

The fifth byte, named e_ident [EI_CLASS], tells if it is the 32-bit or 64-bit version of ELF. This distinguishes between TNS/R and TNS/E object files.

The e_machine field of the ELF header identifies the target platform. This also distinguishes between TNS/R and TNS/E object files.

The e_type field of the ELF header distinguishes among the four types of TNS/E object files described in this section, except that the same value, ET_DYN, is used both for DLLs and import libraries.

When e_type = ET_DYN, the EF_TANDEM_IMPORT_LIB bit of the e_flags field tells if it is a DLL or an import library. When it is an import library, the EF_TANDEM_IMP_LIB_COMPLETE bit tells if it is complete or incomplete.

The EF_TANDEM_IMPLICIT_LIB bit of the e_flags field tells if this DLL is one of the implicit libraries, and is also set in the import library that represents the implicit libraries. The import library that represents the implicit libraries is also identified by the DLL name “__IMPLICIT_LIB__” found in the DT_SONAME record of the *.dynamic* section.

Summary of the Contents of an Object File

This appendix does not specify the ordering of sections within linkfiles. Compilers and the assembler are free to arrange sections as they wish, and so can the linker when it creates a linkfile with the `-r` option. The following is a list of the things that may exist in linkfiles:

ELF Header

Stack Unwinding Information (.IA_64.unwind and .IA_64.unwind_info)

Text Sections (sections whose names begin *.text* or *.restext*)

User Data Sections (*.data and .data1, .sdata and .sdata1, .bss, .sbss*, *.rdata and .rodata*, *.srdata and .srodata*, and *.rconst*)

A *.tandem_info* section (possibly abbreviated to four bytes)

The *.procinfo* and *.procnames* Sections

DWARF Symbol Table Sections

Relocation Table Sections (*.rela.x*, where *.x* could be any of the section names listed above)

ELF Symbol Table Sections (*.symtab* and *.strtab*)

Source RTDU Sections (*.rtdu.index, .rtdu.names,* and *.rtdu.data*)

ELF Section Headers and the *.shstrtab* Section

This appendix does, however, specify the ordering of sections within loadfiles and import libraries, as shown in Contents of a Loadfile or Import Library on page A-5.

It is also possible for the compilers or assembler to create sections of names not listed here. The characteristics of such sections, as listed in their ELF section headers, would tell the linker what to do with them, and they would be propagated by the linker into its output file.

Linkfiles also contain a section named *.comment*, but it is discarded by the linker.

In a loadfile, some of the sections are organized into segments. There is always a text segment, which comes at the beginning of the file. There may be a gateway segment. There may be either one or two data segments. When there are two data segments, they are called the *data constant* segment, followed by the *data variable* segment.

The first column in the table on the following page lists the items that may be found in a loadfile, in the order they would exist. Note that the text segment is always the first segment in the file, and that there may be one or two data segments. Note that the placement of the *.gateway* section (equivalent to the gateway segment) depends on whether the loadfile is a program or a DLL, and that the placement of the *.data* section depends on whether there are one or two data segments. The last two columns have an “X” next to those sections that may be referenced with 22-bit global pointer (GP) - relative addressing, or that may be found in import libraries, respectively.

The segments are loaded into virtual memory. The layout in virtual memory is the same as in the file within each segment, but there are choices for where each segment is placed into virtual memory.

The *.sbss* and *.bss* sections don’t actually take up any space in loadfiles. The table only shows where they would be placed in virtual memory.

Table A-2. Contents of a Loadfile or Import Library

Loadfile Contents	GP-Relative	Import LIbrary
ELF Header		X
ELF Program Headers		X
.tandem_info		X
.lic		
.dynamic		X
.liblist		X
.dynsym.gblzd		X
.hash.gblzd		X
.hashval.gblzd		
.rela.gblzd		
.dynstr2		X
.IA_64.unwind		
.IA_64.unwind_info		
.IA_64.unwind.strings		
.rconst		
.plt		
.restext		
.text		
.hash		X
.dynsym		X
.dynstr		X
.hashval		
.rela.dyn		
.gateway - for a program		
.data - can have more than one data segment		
.rdata		
.fptr		
.srdata	X	
.got	X	
.IA_64.pltoff	X	
.sdata	X	
.sbss	X	
.bss		

Table A-2. Contents of a Loadfile or Import Library

Loadfile Contents		GP-Relative	Import LIbrary
.gateway	(for a DLL)		
DWARF Symbol Table Sections			X
.source.rdtu	(if present, there are three of them.)		
.object.rdtu	(if present, there are three of them.)		
.shstrtab			X
ELF Section Headers			X

Note that the sections from *.got* through *.sbss* are purposely kept together as much as possible, because they are all referenced with GP-relative addressing. However, when there are two data segments, the *.data* section is allowed to intrude among these sections.

Both the data constant segment and data variable segment can have data that requires modification by `rld` when loaded into memory. The difference is that the data constant segment cannot be modified thereafter, while the data variable segment can.

The following is a brief description of each of the items that can occur in a linkfile or loadfile. Unless otherwise stated, a section is not required to be present if, based on its description, it would not contain any useful information for a given object file.

`ELF Header`

This contains header information for the entire file. It is always found at the start of an ELF file.

`ELF Program Headers`

These contain information that summarizes the main parts of the object file required for loading into memory. Program headers are required in loadfiles and import libraries.

`.tandem_info` `Section`

This contains more information of interest to the operating system. It is required in loadfiles and import libraries. It also exists in linkfiles because some of its fi`eld`s are also meaningful there.

`.lic` `Section`

This contains information about the DLLs that were used to preset this loadfile. It is required in a loadfile, as a placeholder even if the loadfile is not preset.

`.dynamic` `Section`

This contains information needed by the runtime loader, such as the addresses of the *.liblist* through *rela.dyn* sections. It is required in loadfiles and import libraries.

.liblist Section

In a loadfile, this tells the names of the DLLs that were in the linker command stream when the linker built this loadfile. In an import library that represents a single DLL it contains the same information as in that DLL.

.dynsym.gblzd Section

This is a symbol table section, similar to the *.dynsym* section (see below), but just for globalized symbols. It may be present in loadfiles and import libraries.

.hash.gblzd Section

This is a hash table section, similar to the *.hash* section (see below), but for looking up symbols in the *.dynsym.gblzd* section.

.hashval.gblzd Section

This is similar to the *.hashval* section (see below), but providing information about the symbols in the *.dynsym.gblzd* section.

.rela.gblzd Section

This is similar to the *.rela.dyn* section (see below), but for the relocation sites whose targets are globalized symbols.

.dynstr2 Section

This is a string space that is pointed at from the *.dynamic*, *.liblist*, and *.dynsym.gblzd* sections.

Stack Unwinding Sections

These contain information for stack unwinding. Note that there are two such sections in a linkfile (not counting the relocation table section named *.rela.IA_64.unwind*), and three such sections in a loadfile.

.rconst Section

This contains application-defined initialized data that does not get modified at runtime, and does not contain addresses that might need modification when the loadfile is first brought into memory. This may never be created by a compilation or assembly, but when the linker sees an input section named *.rdata* that contains no relocation sites it renames the section to *.rconst*.

.plt Section

This section contains *import stubs*. An import stub is created by the linker in a loadfile when the linker cannot guarantee that the target of an IP-relative procedure call is resolved within the same loadfile.

Text Sections

Text sections contain application-defined executable code (procedures). The object file design also allows them to contain data, but that is not expected to happen. In linkfiles, there can be any number of text sections. Their names must begin either *.text* or *.restext*, corresponding to whether they contain non-resident or resident text, respectively. In loadfiles, all the sections that had names beginning *.text* are combined into a single section named *.text*, and similarly for *.restext*, and the *.restext* section (if it exists) comes before the *.text* section. A text section is required in a program, because there must be a main entry point. Text sections in a loadfile can contain *branch stubs*, which are generated by the linker when a procedure call would need to jump farther than its instruction format allows.

.hash Section

This is a hash table for looking up symbols in the *.dynsym* section. It is required in loadfiles and import libraries.

.dynsym Section

This is the dynamic symbol table. It contains information about symbols referenced in this loadfile or exported from this loadfile, other than globalized symbols. It is required in loadfiles and import libraries.

.dynstr Section

This is a string space that is pointed at from the *.dynsym* section. It is required in loadfiles and import libraries.

.hashval Section

This contains precomputed hash values for the symbols listed in the *.dynsym* section. It is required in loadfiles.

.rela.dyn Section

This is the dynamic relocation table. It contains descriptions of the relocation sites within this loadfile whose targets are the symbols listed in the *.dynsym* section.

.gateway Section

This contains gateways. A gateway is created for each procedure entry point that has the *CALLABLE* or *KERNEL_CALLABLE* attribute.

.data Section

This contains application-defined initialized data, but doesn't have either of the restrictions that make it possible to put data into the *.rdata* or *.sdata* section.

.rdata Section

This contains application-defined initialized data that does not get modified at runtime (although the initial values may be addresses that need modification when the loadfile is first brought into memory).

.fptr Section

This section contains official function descriptors. An official function descriptor contains the address and GP value for a procedure that exists in this loadfile. Procedure pointers point at official function descriptors. An official function descriptor is only created for a procedure if the address of that procedure is taken in the same loadfile, or if the procedure is exported from the loadfile.

.srdata Section

This contains application-defined initialized data that does not get modified at runtime (although the initial values may be addresses that need modification when the loadfile is first brought into memory), and that furthermore is “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference is in the same loadfile.

.got Section

This is the global offset table, which contains addresses of data items that are referenced indirectly, as well as the addresses of official function descriptors and EnterPriv labels. The linker creates entries in the *.got* section as necessary. The entries in the *.got* section are found by 22-bit GP-relative addressing.

.IA_64.pltoff Section

This section contains local function descriptors. A local function descriptor contains the address and GP value for a procedure that is referenced from this loadfile. Direct procedure calls (i.e., not involving procedure pointers) use these local function descriptors. The linker creates entries in the *.IA_64.pltoff* section as necessary. The entries in the *.IA_64.pltoff* section are found by 22-bit GP-relative addressing.

.sdata Section

This contains application-defined initialized “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference is in the same loadfile.

.sbss Section

This contains application-defined uninitialized “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference is in the same loadfile. This section occupies no space in an object file, but rather reserves memory space that is automatically initialized

to zero. The object file design supports such sections, although compilers might not use them.

.bss Section

This contains application-defined uninitialized data, but this section doesn't have the restriction that makes it possible to put data into the *.sbss* section. It occupies no space in an object file, but rather reserves memory space that is automatically initialized to zero. The object file design supports such sections, although compilers might not use them. The linker allocates *.bss* sections in loadfiles to contain what the compiler called common data.

.rela.x Sections

These sections describe relocation sites within linkfiles. Relocation sites can be within code or data sections, including unwind function sections, the *.procinfo* section, and the DWARF sections. A *.rela.x* section is required in a linkfile for each section named *.x* that has relocation sites. For example, *rela.data* describes the relocation sites in the *.data* section.

.symtab Section

This is the ELF symbol table. It is required in linkfiles. It contains information about symbols whose names are meaningful to the linker.

.strtab Section

This is a string space that is pointed at from the *.symtab* section. It is required in linkfiles.

.procinfo Section

This section provides information about procedures and subprocedures.

.procnames Section

This is a string space pointed at by the *.procinfo* section.

DWARF Symbol Table Sections

These sections contain information for the debugger and for the COBOL compiler. There are several sections that collectively form the DWARF symbol table.

Source RTDU Sections

These are sections that represent source RTDU's, which are part of the SQL/MP implementation. These can exist only in linkfiles and programs. In a linkfile that is created by compiling a source file with embedded SQL/MP, the set of source RTDU's is represented by three sections. In a program, the set of source RTDU's is also represented by three sections, although not with the same section names as in a linkfile.

`Object RTDU Sections`

An object RTDU, which is part of the SQL/MP implementation, can be placed into a program by a tool named SQLCOMP. The object RTDU is represented by three sections.

`.shstrtab Section`

This is a string space that is pointed at from the ELF section headers. It is required.

`ELF Section Headers`

These contain header information to describe everything in the object file, except for the ELF header, the program headers, the section headers themselves, and possibly unused space within the object file.

A general principle behind the loadfile design is that the sections up through *.dynstr2* are expected to be small, and it can therefore be more efficient to have them all near the front. That is the reason that the *.dynstr2* section was invented, i.e., to segregate out the strings needed by other small sections near the front of the file.

Another general principle is that, after all the things that are “small”, all the things that might need to be resident come next. More specifically, the *.restext* section needs to be resident (by definition), and if it is present then some other sections also need to be resident, and some don’t. All the other things that would also need to be resident are placed before the *.restext* section, so that the *.restext* section (if present) marks the end of the portion of the text segment that needs to be resident.

The HP NonStop operating system has also invented the *.rela.gblzd* section to handle globalized symbols in the implementation of C++. Other implementations take different approaches, not just to handle this specific feature of C++ but with regard to the issue of preemption in general. This invention of the *.rela.gblzd* section again follows the same strategy for the HP NonStop operating system of segregating relocation table entries based on the target symbol, not based on the address of the relocation site.

The reason that Intel and HP separate out the relocation table entries for the *.IA_64.pltoff* section is related to the feature of “lazy evaluation”, which the HP NonStop operating system does not support (and which is not described in this appendix).

Code and Data Sections

This subsection discusses the "ordinary" code and data sections that come from application source code, possibly with some things added by the compiler or linker. Special types of data sections, such as the stack unwinding information, the .procinfo and .procnames sections, the DWARF sections, and the various linker-created sections in loadfiles, are not detailed here.

User Code

In linkfiles there can be many text sections. The sections whose names begin .text contain procedures and subprocedures that are not resident. The sections whose names begin .restext contain procedures and subprocedures that are resident.

When the linker is building a new linkfile it concatenates each of the text sections from the various input files into a section of the same name in the output file. On the other hand, in loadfiles, all the non-resident code is combined into a single .text section, and all the resident code into a single .restext section. The text sections of a loadfile may also contain linker-generated branch stubs, which are not present in linkfiles.

Some procedures are global, which means their names are meaningful across separate compilations. All references to global procedures must be marked with relocation table entries. When there are duplicate copies of global procedures, the linker picks one to use, and the relocation table entries are used by the linker to make sure all references go to the copy that was picked.

If a procedure is in a section whose name begins either ".text." or ".restext.", and the rest of the name is the same as that of the procedure, this is an indication by the compiler that, if this is an unused copy of the procedure, then in fact the entire section containing it may be ignored by the linker. In that case, the linker ignores that input section, thus making the resulting code space smaller.

The .procnames and .procinfo sections provide additional information about procedures and subprocedures in linkfiles. See Finding Information About Procedures and Subprocedures in Linkfiles on page A-26 for further information.

The size of executable code is always a multiple of 16 bytes, because instructions are grouped into 128-bit bundles. Actually, the HP NonStop operating system compilers usually say that text sections must be aligned on 32-byte boundaries, and similarly each procedure within a code section starts at an offset within that section that is a multiple of 32 bytes. Larger alignments can also be specified in assembler source files. When space is wasted between procedures, the assembler fills that space with no-ops.

The total size of a text section in any linkfile must not exceed 16 MB, so that the linker can add branch stubs to the section if necessary. Also, it is suggested that compilers not put all the code of a compilation into one code section, but rather divide it into multiple code sections, such as by putting each procedure into its own section. That is a way to avoid running into the 16 MB limit, either directly as the result of a compilation, or later after the linker has combined many separate compilations into a single linkfile with the -r option, because the linker will concatenate input sections that have the same name.

Certain procedures may be included just to identify an object file. Such a procedure is called a VPROC ("version procedure"). The names of such procedures would always be found in the .procinfo section of a linkfile or in the stack unwinding information of a loadfile. Depending on whether a VPROC was visible outside its compilation, or

exported from its loadfile, it might also be found in the ELF symbol table of a linkfile, or the dynamic symbol table of a loadfile or import library.

User Data

The .data (and .sdata) sections are for initialized data, while .bss (and .sbss) are for uninitialized data. However, if a data item is initialized to all zeros, the compiler may treat it as uninitialized data. That is possible, because all uninitialized data is automatically initialized to zeroes by the HP NonStop operating system.

When the linker combines a set of linkfiles into a new file it usually concatenates each of the user's data sections from the various input files into a section of the same name in the output file. For example, some of the input files may have a section named .data, and then the output file would also have a section named .data, and it would be the concatenation of the .data sections that existed in the input files. The names of typical user data sections, and what each one means, were listed near the beginning of this document. Like text sections, sizes of data sections must be multiples of 16 bytes.

The exception to the general rule given in the previous paragraph is that, if an input section has the name .rdata, but doesn't contain any relocation sites, then the linker changes its name to .rconst for the output file. Note that a similar optimization is not done for .srdata because that is GP-addressable.

The sections named .sdata, .srdata, and .sbss are called small data sections with the meaning that the compiler might choose to put "small" data items into them (i.e., data items whose sizes are no larger than 8 bytes). However, these sections actually have no such requirement. The real meaning of these sections is that the items placed here can be referenced directly by 22-bit GP-relative addressing, rather than getting their addresses out of the .got section. That is only correct to do if the compiler or assembler programmer can guarantee that the symbol cannot be preempted.

Linkfiles also have common data, which has not been allocated to any section. When the linker builds a loadfile it allocates common data in the .bss section.

The following is how the C compiler works:

Data that is global or large, and initialized to a non-zero value, is placed into .data.

Data that is global or large, and initialized to a zero value, is placed into .bss.

Data that is local and small, and initialized to a non-zero value, is placed into .sdata.

Data that is local and small, and initialized to a zero value, is placed into .sbss.

Data that is uninitialized is called common data.

Character strings are called local data items and placed into .rdata.

The MCB (Master Control Block)

The linker adds the MCB to the .data section of a program (or creates a section of this name if there was none before). The MCB is a data item that can be referenced by the name _MCB within the program. The linker only creates the MCB in programs (not DLL's), and only if the program makes a reference to the symbol named _MCB.

This is a description of the fields that are nonzero in the MCB of an object file.

The Check_quad field is an 8-byte string, where the first two and last two bytes each contain the value 0xAA and the middle four bytes contain the ASCII string "MCB ".

The Version_item field currently contains 0, but presumably could contain a different value in the future.

The Standard_C_streams bit is set to 1, rather than 0, to indicate that the program should use code 180 files for C text files, rather than code 101 files. The linker sets this bit to 1 when it creates the program if the -ansistreams option is specified or if the target platform is OSS.

The C_std_files_open bit is set to 1, rather than 0, to indicate that this program should automatically open the standard C/C++ I/O files. This linker sets this bit to 1 if the program contains a main procedure that is written in C or C++ and the -nostdfiles option is not specified.

The FP_format field is set to indicate the floating point type assumed by this program, repeating the information also found in the file header. 0 indicates that the Tandem floating point is required. 1 indicates that the IEEE floating point is required. 2 indicates neutral.

Predefined Symbols

There are certain symbols (other than the MCB and symbols whose names are the same as names of sections of object files) that the linker also creates, but only in loadfiles, and only if they are referenced from the loadfile. The following table tells the names of these symbols, their meanings, and the values that are placed into the st_scndx fields of their dynamic symbol table entries.

Table A-3. Additional Predefined Symbols Optionally Created By The Linker In Loadfiles

Name	Meaning	Value of st_shndx
_BASE_ADDRESS	The address of the text segment.	The index of the .text section.
_DYNAMIC	The address of the .dynamic section.	The index of the .dynamic section.
_unwind	The start of the .IA_64.unwind section.	The index of the .IA_64.unwind section, or SHN_ABS (see the explanation below).
_unwind_size	The number of entries in the .IA_64.unwind section.	SHN_ABS
etext or _etext	The end of the text segment.	The index of the .text section.
fdata or _fdata	The start of the data (constant) segment.	The index of the .data section.
_GLOBAL_OFFSET_TABLE	The address of the .got section.	The index of the .got section.
__gp	The GP value.	The index of the .got section.
edata or _edata	The end of the initialized data, which is also the start of the uninitialized data.	The index of the .data section.
end or _end	The end of all the data.	The index of the .data section.

Note that the sections listed above do not imply that the address of the symbol is in that section. The value in st_scndx doesn't really matter, because the st_value field gives the symbol's value, not its offset within a section. But something needs to be filled in for the st_shndx field, and it needs to be a real section, not "absolute", so that rld and the -alf option of the linker know that it should be updated for rebasing. Accordingly, the section indices shown above have been chosen. The linker will always create sections named .text and .data, even if they would be empty, so that their indices can be used in this way. Note that_unwind_size is absolute. Also, if the loadfile has no code, and therefore has no .IA_64.unwind section, then the symbol named _unwind is made absolute, and in this case both _unwind and _unwind_size have the value 0.

Relocation Tables

It is possible that the contents of one place in the code or data of an object file need to be filled in with the address of another place in the code or data, or in some other way based on such an address. If the compiler or assembler knows what needs to go there, without later modification by the linker or runtime loader, then that's the end of the story. But, if the linker or runtime loader will need to be involved, the compiler or assembler must indicate that location accordingly, by creating relocation tables in linkfiles to provide such information. Similarly, the linker must put relocation tables into loadfiles if there is still work for the runtime loader to do.

The place that needs to be filled in is called the relocation site. It would either be an operand within an executable instruction, which come in various sizes, or a data item, which would be a 32-bit or 64-bit integer. The place whose address needs to be calculated is called the target of the relocation. The relocation site is also said to be a reference to the target symbol.

The target of a relocation site is described by giving an offset relative to a symbol that is listed either in the .symtab section (in the case of a linkfile) or the .dynsym or .dynsym.gblzd section (in the case of a loadfile). If the symbol is of type STB_LOCAL then it must be defined with an address in this object file, and that is the address that is used for the symbol. If the symbol is of type STB_GLOBAL then the definition of the symbol that is used to resolve the reference might exist in this object file or in another object file.

The process of figuring out the target address is called resolving the reference. After a reference has been resolved, the proper way to fill in the contents of the relocation site depends on the site's relocation type.

The relocation types that can occur in linkfiles and loadfiles are different, and the names of the relocation table sections are different. In linkfiles, for each code or data section named .x that contains relocation sites there is a relocation table section named .rela.x that describes the relocation sites in that section. This also includes relocation tables needed to describe relocation sites in the .procinfo section, the unwind function sections, and the DWARF symbol table sections. In loadfiles there are relocation table sections named .rela.dyn and .rela.gblzd that describe all the relocation sites in the data segment of the loadfile. Loadfiles never have relocation sites in the text segment. The entries in .rela.dyn are for relocation sites whose target symbols are in .dynsym, while the entries in .rela.gblzd are for relocation sites whose target symbols are the globalized symbols listed in .dynsym.gblzd.

The format of the relocation information is the same in all cases. The ELF section type is SHT_RELA, and the format of a relocation table entry is the following:

```
typedef struct ELF64_Rela {
                          ELF64_Addr          r_offset;
                          ELF64_Xword         r_info;
                          ELF64_Xword         r_addend;
}Elf64_Rela
```

The size of this structure is 24 bytes.

In linkfiles, relocation table entries always completely describe what needs to be filled in at the corresponding relocation sites. So, it doesn't matter what is actually in the operands at the relocation sites. In fact, what is there should be zero, with the following two exceptions:

The value "-1" is filled in for relocation sites that point from DWARF information at executable code, when they correspond to unused copies of procedures.

Relocation sites that point from one DWARF section into another, i.e., giving a section offset rather than an address, are also fixed up in linkfiles created by the linker.

For loadfiles the relocation types whose names begin R_IA_64_REL make use of the contents of the relocation site, rather than pointing at a target symbol These relocation table entries say that the contents of the relocation site need to be updated at runtime, or by the -alf option of the linker, based on how much the segment pointed at by the relocation site is rebased.

In loadfiles, the relocation sites whose targets were STB_LOCAL would only need to be updated if the loadfile was rebased.

In loadfiles, the elements of the .rela and .rela.gblzd sections are sorted by target symbol index. In particular, that means that all the entries with the same target symbol are consecutive. This includes the case of relocation types whose names begin R_IA_64_REL, which don't have a target symbol, so that the target symbol index is 0.

r_offset

This tells the location of the relocation site. In a linkfile r_offset is the offset into the section, as the name implies. In a loadfile r_offset tells the (preferred) virtual address of the relocation site, so that the name r_offset is a misnomer.

In either case, depending on the relocation type, the relocation site is understood to be either an operand of an instruction or a data item.

For a relocation site that is a data item, r_offset tells the address of the first byte in that set of bytes. In other words, since data is big endian, it is the address of the high order byte of the value. The relocation type tells whether the relocation site contains 4 bytes or 8 bytes of data.

For a relocation site that is an operand of an instruction, the r_offset field identifies both the bundle and the instruction slot within it. If you zero out the last four bits of r_offset then the resulting value, which is a multiple of 16, is the address of the bundle. The value in the last four bits of r_offset must be 0, 1, or 2, to identify the instruction slot within the bundle.

r_info

The 64-bit r_info field combines together two different pieces of information, to tell the target symbol and the relocation type. The low order 32 bits of the r_info field tell the relocation type and the high order 32 bits tell the symbol. The following #define's

extract the target and relocation type from the r_info field, or reconstruct the r_info field from its two pieces:

```
#define ELF64_R_SYM (i)      ((i) >> 32)
#define ELF64_R_TYPE (i)    ((i) & 0xffffffff)
#define ELF64_R_INFO (s,t)   (((Elf64_Xword)(s) << 32) + (Elf64_Xword)(t))
```

The target symbol is an index into the .symtab section in the case of a linkfile, or into the .dynsym or .dynsym.gblzd section in the case of a loadfile.

r_addend

This specifies a constant that is added to the address of the target symbol to obtain the address that is used to fill in the relocation site.

An alternative to the Elf64_Rela relocation table structure is Elf64_Rel. The difference is that Elf64_Rel does not contain the r_addend field. Instead, the relocation site itself contains the addend, to be added to the address of the target symbol. One reason that may be given for using the Elf64_Rela structure is that it allows the addend to be larger than the number of bits available at the relocation site. The HP NonStop operating system prefers Elf64_Rela because it means that the process of filling in the relocation site has less dependence on the contents of the site, so there are fewer situations in which we need to be concerned about what happened to the relocation site if the process of filling in relocation sites got interrupted for an unpredictable reason.

Relocation Types

The following table lists the relocation types. The ones whose names end in "MSB" are the ones that treat the relocation site as data, where the "MSB" means that the data is considered big endian, and there may be 32-bit and 64-bit varieties of these. These relocation sites are not necessarily well aligned.

The linker treats all addresses internally as 64-bit quantities. When an address is placed in a 32-bit container, the requirement is that the address must fit into 32-bits as a signed quantity. In other words, the high order 32 bits of the address must all be the same as the highest order bit in the lower 32 bits of the address, and then the lower 32 bits of the address are used to fill in the relocation site.

Table A-4. Relocation Types

Name	Value	Description
R_IA_64_NONE	0x00	No-op.
R_IA_64_IMM64	0x23	Virtual address of a symbol in code.
R_IA_64_DIR32MSB	0x24	Virtual address of a data item in data.
R_IA_64_DIR64MSB	0x26	

Table A-4. Relocation Types

Name	Value	Description
R_IA_64_GPREL22	0x2a	22-bit GP-relative address of a data item.
R_IA_64_GPREL64I	0x2b	64-bit GP-relative address of a data item.
R_IA_64_LTOFF22	0x32	22-bit GP-relative address of a .got section entry.
R_IA_64_LTOFF64I	0x33	64-bit GP-relative address of a .got section entry.
R_IA_64_PLTOFF22	0x3a	22-bit GP-relative address of a local function descriptor.
R_IA_64_FPTR32MSB	0x44	Virtual address of an official function descriptor.
R_IA_64_FPTR64MSB	0x46	
R_IA_64_PCREL60B	0x48	IP-relative address.
R_IA_64_PCREL21B	0x49	
R_IA_64_PCREL21M	0x4A	
R_IA_64_PCREL21F	0x4B	
R_IA_64_PCREL32MSB	0x4C	
R_IA_64_PCREL64MSB	0x4E	
R_IA_64_LTOFF_FPTR22	0x52	22-bit GP-relative address of the .got section entry for an official function descriptor.
R_IA_64_SEGREL64MSB	0x5e	Segment-relative address.
R_IA_64_SECREL32MSB	0x64	Section-relative address.
R_IA_64_SECREL64MSB	0x66	
R_IA_64_REL32MSB	0x6c	Runtime rebasing address.
R_IA_64_REL64MSB	0x6e	
R_IA_64_IPLTMSB	0x80	Local function descriptor.
R_IA_64_LTOFF22X	0x86	Optimizable 22-bit GP-relative .got section access.
R_IA_64_LDXMOV	0x87	Goes along with R_IA_64_LTOFF22X.

This rest of this section describes what the relocation types in the above table mean and how they are used.

R_IA_64_NONE

This is a no-op. When the relocation type is R_IA_64_NONE, the entire relocation table entry is zero.

R_IA_64_IMM64 -- Virtual Address of a Symbol in Code

This applies to the situation when the compiler or assembler knew that it was compiling code that would not be moved at runtime and that the referenced symbol would also be within the same loadfile and could not be preempted at runtime. In this case, the compiler or assembler can generate (non-PIC) code where the virtual address of a symbol is placed directly into the code. The linker would fill in the relocation site with the virtual address, and the same relocation table entry would also appear in the loadfile, so that the site could be updated by the -alf option of the linker if necessary.

R_IA_64_DIRx -- Virtual Address of a Data Item in Data (and a few other things)

This usually applies to the situation when a data item is initialized with the address of another data item. The compiler or assembler would create this kind of relocation table entry to describe such a relocation site, and the linker would propagate it into loadfiles, so it could be modified at runtime.

The object file format does not prohibit a data item from being initialized with the real address of a procedure, as opposed to the address of the official function descriptor for the procedure, and in that case this same relocation type would be used.

The R_IA_64_DIR64MSB version of this type of relocation table entry (i.e., the 64-bit variety) is also created by the linker in loadfiles to tell what address should be in a .got section entry at runtime, when that .got entry is to contain the address of a data item or the address of an EnterPriv label.

The R_IA_64_DIRx relocation types are also used in linkfiles to identify addresses that are stored in the DWARF symbol table sections.

R_IA_64_GPREL22 -- 22-Bit GP-Relative Address of a Data Item

This applies to the situation when the code has a reference to a data item, and the compiler or assembler knows that the data item will have to be found within the same loadfile, and the compiler or assembler has allocated the data item in the .sdata, .srdata, or .sbss section, and refers to it by 22-bit GP-relative addressing. It is an error if the linker cannot guarantee that the target symbol is resolved in the same loadfile, or that its address cannot be reached by 22-bit GP-relative addressing.

The compiler or assembler would generate code that calculates the address of the data item by adding the signed 22-bit offset to the GP-register. The linker would fill in that 22-bit operand by subtracting the value of the GP register for this loadfile from the address of the data item. The relocation table entry would not exist in the resulting loadfile.

R_IA_64_GPREL64I -- 64-Bit GP-Relative Address of a Data Item

This applies to the situation when the code has a reference to a data item, and the compiler or assembler knows that the data item will have to be found within the same loadfile, and the compiler or assembler refers to it by 64-bit GP-relative addressing. It is an error if the linker cannot guarantee that the target symbol is resolved in the same loadfile.

In more detail, the reference must also be within the data segment, not the text segment, of the loadfile. This is a requirement because the text and data segment could get rebased by different amounts at runtime, changing the GP-relative addresses of items in the text segment, but such references can't be updated at load time. Although the HP NonStop operating system does not rebase the text and data segments by different amounts at load time, we still obey this rule about 64-bit GP-relative addressing, and it is important to the linker's -alf option, which can rebase the text and data segments by different amounts.

\When using this relocation type, the compiler or assembler would generate code that loads a 64-bit value into a register and adds it to the GP register. The linker would fill in that 64-bit operand by subtracting the value of the GP register for this loadfile from the address of the data item. The relocation table entry would not exist in the resulting loadfile.

R_IA_64_GPREL64I is also used in gateway code, to calculate the 64-bit GP-relative address of the procedure for which this is the gateway. Then the address of the procedure is calculated by adding this value to the GP register.

R_IA_64_LTOFF22 -- 22-Bit GP-Relative Address of a .got Section Entry

This applies to the situation where there is a reference to a data item in code and the compiler or assembler does not do it directly by GP-relative addressing, as described in the previous two items. Here, the compiler or assembler generates code that adds a signed 22-bit offset to the GP register in order to get the address of an entry in the .got section, which in turn contains the address of the data item. The linker allocates that .got section entry and fills in the 22-bit operand by subtracting the value of the GP register for this loadfile from the address of this .got entry. This relocation table entry is not present in loadfiles. The linker would instead generate a relocation table entry of type R_IA_64_DIR64MSB to describe the .got entry.

It is also possible to use this relocation type to get the address of a procedure from the .got. That is not a common thing to do, because you generally need the address of its official function descriptor, which is done instead with R_IA_64_LTOFF_FPTR22. However, gateways use R_IA_64_LTOFF22 to get the real address of the EnterPriv labels from the .got section. Then that .got entry is used to branch to the EnterPriv label. A function descriptor is not used in this case, because the gateway does not need to set up the GP value for the EnterPriv label.

R_IA_64_LTOFF64I -- 64-Bit GP-Relative Address of a .got Section Entry

This may also be used in gateways, like R_IA_64_LTOFF22, to calculate the 64-bit GP-relative address of the .got entry for an EnterPriv label.

R_IA_64_PLTOFF22 -- 22-Bit GP-Relative Address of a Local Function Descriptor

This applies to the situation where there is a direct procedure call and the compiler or assembler has generated code that adds a signed 22-bit offset to the GP register in order to get the address of the local function descriptor for the target procedure. The linker allocates the local function descriptor and fills in the 22-bit operand by

subtracting the value of the GP register for this loadfile from the address of this local function descriptor.

This relocation table entry is not present in loadfiles. The linker would instead generate a relocation table entry of type R_IA_64_IPLTMSB to describe the local function descriptor.

R_IA_64_FPTRx -- Virtual Address of an Official Function Descriptor

This applies to the situation when a data item is initialized with the address of a procedure. This is a procedure pointer, so it must contain the address of the official function descriptor for that procedure. The compiler or assembler would create this kind of relocation table entry to describe such a relocation site, and the linker would propagate it into loadfiles, so it could be modified at runtime.

The R_IA_64_FPTR64MSB version of this type of relocation table entry (i.e., the 64-bit variety) is also created by the linker in loadfiles to tell what address should be in a .got section entry at runtime, when that .got entry is to contain the address of an official function descriptor.

R_IA_64_PCRELx -- IP-Relative Address

This applies to the situation when there is a direct procedure call (or another type of branch) and the compiler or assembler has generated an IP-relative procedure call instruction, which uses an operand within the instruction as an offset to the target procedure relative to the location of the bundle containing the instruction itself. The four variations of this relocation type correspond to four instruction varieties that can be used for this. In all cases, the target address is the address of a bundle, which is a multiple of 16, so four 0's are appended to the end of the operand. The 21-bit forms are signed quantities that can therefore cover a region of 225 bytes, or 16 megabytes forward or backward, while the 60-bit form can reach any 64-bit address.

It is only correct to use IP-relative addressing in a loadfile if the target address can be guaranteed to be in the same loadfile. Also, we do not use IP-relative addressing to make a call on a gateway in a DLL, because the -alf option can separately rebase the code and the gateway segment. The linker does one of two different things, depending upon whether it can make these two guarantees.

If the linker can guarantee that IP-relative addressing will work, it fills in the operand so that, when four 0's are attached to it, it will equal the address of the target minus the address of the bundle containing the current instruction. In this case, no relocation table entry remains in the loadfile.

If the linker cannot guarantee this, then the linker will allocate an import stub in the .plt section. The import stub will make the procedure call, using a local function descriptor similar to what was described above when the compiler or assembler generated the R_IA_64_PLTOFF22 type of relocation table entry. The linker updates the operand in the instruction so that, when four 0's are attached to it, it will equal the address of the import stub minus the address of the bundle containing the current instruction. As described above under R_IA_64_PLTOFF22, the loadfile would contain a relocation table entry of type R_IA_64_IPLTMSB to describe the local function descriptor.

In both cases, if the IP-relative branch was a 21-bit variety, it might not be possible for it to reach its destination. In such a case the linker would allocate a branch stub that was close enough. The branch stub would use a code sequence that enabled it to reach the target procedure or import stub, and the original instruction would be updated to reach the branch stub.

R_IA_64_LTOFF_FPTR22 --
22-Bit GP-Relative Address of the .got Section Entry for an Official Function Descriptor.

This applies to the situation when code assigns a value to a procedure pointer. The compiler or assembler generates code that adds a signed 22-bit offset to the GP register in order to get the address of an entry in the .got section, which contains the address of the official function descriptor. The linker would allocate the .got entry and fill in the 22-bit operand by subtracting the value of the GP register for this loadfile from the address of that .got entry. This relocation table entry is not present in loadfiles. The linker would instead generate a relocation table entry of type R_IA_64_FPTR64MSB to describe the .got entry.

R_IA_64_SEGREL64MSB -- Segment-Relative Address

This indicates a value that is an offset from the start of the same segment of the same loadfile.

This is the relocation type that is used for the fields in the stack unwinding sections of linkfiles. Actually, the relocation type is not important for this, because the linker will change the format of these sections.

This (or its 32-bit variation) is also the relocation type that the linker internally uses for filling in addresses in the DWARF symbol table sections, although they are marked as R_IA_64_DIRx in linkfiles.

This relocation type does not occur in loadfiles because, by definition, such relocation sites would never need updating at load time or by the -alf option of the linker.

R_IA_64_SECREL* -- Section-Relative Address

According to the standard, this indicates a value that is an offset from the start of the same section of the loadfile. The only expected use of this is when one DWARF section points at another DWARF section.

If someone wants to use SECREL in the future, to point between different code and data sections, it won't work as specified. It will put in the distance from the start of the section containing the reference site, rather than the start of the section containing the target symbol.

R_IA_64_RELx -- Runtime Rebasing Address

In a DLL, this indicates an address that needs to be updated by the amount that this DLL is rebased in memory at load time, or an address that needs to be updated by the amount that the segment containing the address is moved by the -alf option of the linker. There is no target symbol, so the target symbol index is always 0.

R_IA_64_REL64MSB implicitly applies to all the non-zero entries in the initialization and termination routines created by the linker, whose addresses are indicated by fields in the .tandem_info section, and to each half of each official function descriptor found in the .fptr section.

Note that, when this type of relocation occurs, the runtime loader and the -alf option of the linker are only capable of updating the relocation site for rebasing, not for calculating its contents from scratch. So, in all the situations mentioned above, the linker must always fill in the correct values at the relocation sites, even if the linker is not presetting the output loadfile.

R_IA_64_IPLTMSB -- Local Function Descriptor

This only occurs in loadfiles. When the linker creates a local function descriptor in a loadfile it puts this relocation table entry into the loadfile to tell the runtime loader about it. It indicates that the local function descriptor needs to be filled in with the address of the indicated procedure and the GP value for the loadfile that contains it.

R_IA_64_LTOFF22X and R_IA_64_LDXMOV --
Optimizable 22-Bit GP-Relative .got Section Access

This pair of relocation types applies to the same situation as when the compiler or assembler might use R_IA_64_LTOFF22, as explained above. The difference is that the use of R_IA_64_LTOFF22X on that instruction, and the use of R_IA_64_LDXMOV on other related instructions, gives the linker permission to do an optimization.

Specifically, the R_IA_64_LTOFF22X relocation table entry would be on an instruction with the format of addl, to compute the address of a .got entry by adding a 22-bit offset to the GP register. The R_IA_64_LDXMOV relocation table entry would be on every related instruction that used the address set up by the addl in one register to load the contents of that .got entry into another register.

If the linker doesn't do this optimization then it treats the R_IA_64_LTOFF22X as synonymous with R_IA_64_LTOFF22, and ignores all the corresponding R_IA_64_LDXMOV entries.

How -alf Updates DWARF

The .debug_relocs section is only found in DLL's. Its purpose is to provide information so that the -alf option of the linker can update the places within DWARF that contain code and data addresses. The -alf option does this when it rebases a DLL. The .debug_relocs section is not needed in programs, because the -alf option does not rebase programs. The .debug_relocs section is not needed in import libraries, because the -alf option does not work on import libraries.

In the section header for the .debug_relocs section, the type is SHT_PROGBITS, the sh_addralign field is 8, and the sh_entsize field is 8.

The .debug_relocs section is an array of 64-bit entries, one for each code or data address found in other DWARF sections. Within the 64-bit entry, the high-order byte

contains one of the following two values, which match the values for relocation types used elsewhere in object files:

```
#define R_IA_64_REL32MSB0x6c
#define R_IA_64_REL64MSB0x6e
```

If REL32MSB is present, that means it is a 32-bit address. If REL64MSB is present, that means it is a 64-bit address.

The remaining 56 bits of the 64-bit entry tell the file offset of that address, from the beginning of the object file.

The following macros show how to pull apart or put together the two parts of each 64-bit entry:

```
#define ELF64_TANDEM_DW_TYPE (i) ((i) >> 56)
#define ELF64_TANDEM_DW_OFFSET (i)((Elf64_Xword)(i) & 0xffffffffffffff)
#define ELF64_TANDEM_DW_INFO (t, o)(((Elf64_Xword)(t) << 56) + o)
```

There are no rules concerning the ordering of the elements of the .debug_relocs section.

Additional Notes About Relocation Types

Note 1:

It is possible for the compiler or assembler to use IP-relative addressing for items that it puts into the same text section and can't be preempted, such as static procedures, literals, jump tables, etc. Since the linker doesn't rearrange the contents within an individual text section of a linkfile, the compiler or assembler knows the final IP-relative offsets to use. These kinds of calculations do not require any relocation table entries.

Note 2:

When there is a relocation table entry of type R_IA_64_FPTRx or R_IA_64_LTOFF_FPTR22, which are the ones that require the address of an official function descriptor, and the target procedure exists in the same loadfile, the linker will also allocate the official function descriptor for that procedure. The linker similarly allocates such official function descriptors for all exported procedures, because relocation table entries of these types might exist in other loadfiles, requiring the address of such an official function descriptor.

Note 3:

Note that there are different relocation types to say that one wants the real address of something, versus the address of the official function descriptor for a procedure. However, there is a runtime procedure named dlsym, which can be asked for the address of a symbol, given only the name of the symbol. After dlsym finds the symbol's dynamic symbol table entry, it looks at that entry to see if the symbol is code or data, and based on that it decides to either return the real address of the symbol (if it is data) or the address of its official function descriptor (if it is a procedure).

Finding Information About Procedures and Subprocedures in Linkfiles

The linker obtains information about procedures and subprocedures from the *.procinfo* and *.procnames* sections of linkfiles. These sections tell the linker what all the procedures and subprocedures are, giving the following information about each one:

- its name
- its location
- its attributes
- how it is nested in other procedures

In addition, the *.procinfo* and *.procnames* sections tell the linker about alternate entry points that have the *CALLABLE* or *KERNEL_CALLABLE* attribute.

When creating a linkfile, the linker creates the *.procinfo* and *.procnames* sections of that linkfile from the sections of the same names in its input files. This is mostly done by concatenation, in the same order as the linker saw those linkfiles, except that the pointers from the *.procinfo* section to the *.procnames* section need the appropriate updating, and entries for unneeded copies of procedures are omitted. (This ordering is potentially important. For example, it can affect which procedure is chosen as the main entry point, when the *-allow_multiple_mains* option tells the linker that it is okay to have more than one procedure with the *MAIN* attribute.)

The DWARF Symbol Table

The DWARF symbol table contains information used by debuggers and by the COBOL compiler, whereas the .symtab, .dynsym, and .dynsym.gblzd sections contain information used by the linker and runtime loader.

The DWARF symbol table information in an import library that represents a single DLL is the same as the DWARF symbol table information that is present in the corresponding DLL. There is no DWARF symbol table information in the import library that represents the implicit libraries.

A file may be "stripped", meaning that it doesn't have debugging information in it. This means that the DWARF symbol table is not present. Note that it is even possible for a linkfile to be stripped. In other words, even after being stripped, a linkfile can still be processed by the linker, because the DWARF symbol table does not contain any information that is required by the linker. An import library can be stripped even if the corresponding DLL is not stripped.

DWARF Object File Sections

Here is a summary of the purposes of the DWARF sections that the HP NonStop operating system uses:

.debug_info

This is the main section of DWARF information. It is a tree of nodes, each node contains various attributes.

.debug_abbrev

This section provides additional information required to decode the information in the .debug_info section, including information about implementation-defined material.

.debug_line

This section contains information that tells how to map things to source line numbers.

.debug_line_nsk

This has a format similar to .debug_line, but to represent EDIT line numbers rather than sequential line numbers.

.debug_relocs

This section describes the places in DWARF sections of DLL's that contain code and data addresses, so that they can be updated by the -alf option of the linker when that option is used to rebase the DLL.

Archives

An archive is a single file that contains within it copies of other files, called the "members" of the archive. Archives are created by the tool named ar. An archive may be used for various purposes, one of which is to be an input for the linker. The linker uses archives as a source of linkfiles. Archives are not used at load time.

The format described here, used for TNS/E archives differs in various ways from what was used in the TNS/R implementation.

An archive contains "symbol table" information that tells which linkfile within the archive, if any, provides a definition for a given symbol. These would be the symbols defined in that linkfile and visible outside, i.e., their binding is STB_GLOBAL and their st_scndx field is not SHN_UNDEF in the ELF symbol table.

The first eight bytes of an archive contain the string "!<arch>", followed by a newline character (ASCII LF). This identifies the file as an archive. After that the archive is a concatenation of "pieces", each of which contains the following items, which always begin at file offsets that are multiples of 2 bytes.

an ar_hdr structure

the contents of this piece

The first one or two pieces of the archive may be special. The first special piece is the archive symbol table, which is present if the archive contains any linkfiles. The other special piece is the "long member name string space", which is present if any of the names of the members of the archive are longer than 16 characters. The contents of the remaining pieces are the members of the archive.

Here is the declaration for the ar_hdr structure:

```
typedef struct ar_hdr {
char      ar_name [16];
char      ar_date [12];
char      ar_uid [6];
char      ar_gid [6];
char      ar_mode [8];
char      ar_size [10];
char      ar_fmag [2];
} ar_hdr;
```

The size of this structure is 60 bytes.

The ar_size field tells the size of the contents of this piece of the archive, and the ar_name field tells its name. When the name is less than 16 characters long, the rest of the field is filled with blanks. The other fields of the ar_hdr are all readable ASCII character fields.

In the ar_hdr for the symbol table piece, the ar_name is a single slash ("/").

The contents of the symbol table piece are the following (in this order):

a four-byte integer that tells the number of symbols in the symbol table piece
an array of four-byte integers
a string space (see below)

The integers mentioned above are binary integers (big endian).

The string space is a concatenation of strings, telling the names of the symbols in the symbol table piece. Each name is terminated by a zero byte. If the total size is odd, an extra zero byte at the end makes it even. These strings are in the same order as the previous array of four-byte integers. For each name, the corresponding four-byte integer tells the file offset within the archive for the ar_hdr of the member that defines that symbol. Symbols are only listed in the symbol table if they are defined somewhere. A symbol may be defined in more than one member, but the symbol table only points at one place.

In the ar_hdr for the long member name string space, the ar_name is two slashes ("//").

The long member name string space is a concatenation of strings, telling the names of the members whose names are longer than 16 bytes. Each name is terminated by a slash ("/") and a newline character. If the total size is odd, an extra newline character at the end makes it even.

In the ar_hdr for an archive member, the ar_name tells the name of the file that was placed into the archive. If the name is longer than 16 bytes then it is stored instead in the long member name string space and the ar_name field for the member consists of a slash ("/") followed by an ASCII string for the integer value that is the byte offset of the member's name in the long member name string space. Leading zeroes are removed from this string, and it is blank filled on the right.

The following is a summary of what is in an archive. Horizontal lines separate pieces of the archive. This example shows the case when there is a symbol table and a long member name string space.

!<arch>

_____

ar_hdr for the symbol table
the number of symbols in the symbol table

file offset for the member that defines the first symbol
file offset for the member that defines the second symbol
...

_____

name of the first symbol
name of the second symbol
...

_____

ar_hdr for the long member name string space
the string space of long member names

_____

ar_hdr for the first member
contents of the first member

_____

ar_hdr for the second member
contents of the second member

...

Tools That Work With Object Files

Here is a list of some of the tools (i.e., customer products) that read and/or write object files (or archives):

- Compilers and the assembler create object files.
- The linker (`eld`) reads and writes object files, and reads archives.
- enoft reads object files to display their contents.
- VPROC can read object files to print out version procedures (i.e., a very special case of what NOFT does).
- The HP NonStop operating system operating system, including the runtime loader (`rld`), reads object files to bring them into memory.

- Debuggers read object files as well as their memory images, and can modify the memory images.
- The archive creation tool (`ar`) reads object files, and reads and writes archives.
- SQLCOMP can read and write object files in order to create or update their object RTDU's.

Glossary

Archive file. This file contains copies of other files, called the "members" of the archive. An archive may be used for various purposes, one of which is to be an input for the linker. The linker uses archives as a source of linkfiles. Archives are not used at load time.

Big endian. This term describes a method of storing data so that the most significant byte appears in a lower-numbered location in memory. As with TNS/R, TNS/E data structure is big endian. Code on the TNS/E platform is always little endian.

Bundle. This term describes a three-instruction-wide 128-bit word used by Intel to facilitate parallel processing of code instructions.

Code file. A file comprising instructions that can be executed or emulated by a computer. Native code files can be either linkable (linkfiles) or loadable (loadfiles). Object files and binaries are other names for code files.

Client (of a loadable library). A loadfile that uses functions or data from a library.

Default. The choice made when the user does not direct otherwise.

Direct reference (of a loadfile). A library listed in a loadfile's libList.

DLL file. This is a PIC library loadfile with symbols that can be referenced by another loadfile to resolve symbolic references at link time and/or runtime. It is therefore a loadfile that offers functions or data for use by other loadfiles. For TNS/E, DLLs replace SRLs commonly associated with the TNS/R architecture. The object file linker `eld` generates DLLs for TNS/E (as does `ld` for the TNS/R DLLs). In UNIX, this type of file is known as a shared object file or dynamic shared object (DSO).

Dynamic loading. Loading and opening DLLs under programmatic control after the program is loaded and execution has begun.

EDIT Line Number. The conventional source line numbering convention is where the source lines are numbered sequentially using integers starting at 1. The Guardian EDIT text file (file code "101") uses a source line number convention where the lines are assigned numbers that have three places after the decimal point, and can be sparse within all such possible numbers.

ELF. This term stands for "executable and link format" and describes an extensible file structure that can deal with various target platforms. Like TNS/R, TNS/E uses the ELF file structure with Tandem extensions. However TNS/E is ELF all-inclusive whereas TNS/R uses both ELF and COFF file structures. All TNS/E compiler/assemblers, linkers, and loaders generate object files with this file structure.

Explicit library. Any library that is named in the libList of any client loadifle or is a user library of a client program.

Export. To provide a symbol definition for use by other loadfiles. A loadfile offers for export a symbol definition for use by other loadfiles that need a data item or function having that symbolic name.

Gateway. For every callable function there is a gateway; all calls to the function jump first to the gateway, which effects the transition to privileged state if the caller is not already privileged. There are two types of gateway pages, those that promote to kernel and those that promote to executive level.

Gblzd. globalized [symbol]

Globalized import. The import-control characteristic of a loadfile that allows it to import symbols from any loadfile in the loadList of the program with which it is loaded. When those loadfiles offer multiple definitions of the same symbol, those loadfiles are searched in loadList sequence and the first definition found takes precedence. See also searchList.

Globalized symbol. An exported symbol generated by the C++ compiler that may have multiple definitions, of which the linker and loader must assure only one is used throughout the process.

Hybrid file. This term describes a 'pseudo-DLL' that contains non-PIC text to allow a PIC process to call (as inputs) when building or relinking a program or DLL file. Hybrids do not exist in TNS/E.

Implicit library. A library supplied by HP that is available in the read-only and execute-only globally mapped address space shared by all processes without being specified to the linker or loader. The public libraries on TNS/E that replace System Code, System Library, and millicode. These libraries are called implicit because every loadfile is implicitly a user of them. Contrast with public DLLs, which are explicit because a loadfile explicitly asks to use a public DLL, although it does not specify where to find the public DLL. See also System library. and Public Libraries.

Implicit library import library (imp-imp). An import library that can be used by the Linker as a proxy for a set of implicit libraries. See Import library and Zimpimp file.

Import. To refer to a symbol definition from another loadfile. A loadfile imports a symbol definition when it needs a data item or function having that symbolic name.

Import control. The characteristic of a loadfile that determines from which other loadfiles it can import symbol definitions. The programmer sets a loadfile's import control at link time. That import control can be localized, globalized, or semiglobalized. A loadfile's import control governs the way the linker and loader construct that loadfile's searchList and affects the search only for symbols required by that loadfile.

Import library. This term describes one type of a loadfile whereby only enough parts of the file are contained therein to allow the linker to resolve references, but not enough to expose its source code; i.e., exports the symbols of the DLL . It is a file that can be used by the Linker as a proxy for one or more DLLs, but that cannot actually be loaded

and run. It is useful in cross-linking. See Implicit library import library (imp-imp) and Zimpimp file.

Indirect reference (of a loadfile). A library in a loadfile's searchList that is not named in its libList.

Instance. A particular case of a class of items, objects, or events. For example, a process is defined as one instance of the execution of a program; multiple processes might be executing the same program simultaneously. Also, instance data refers to global data of a program or library; each process has its own instance of this data.

Library. Generically, a collection of functions and data offered for use by clients. Libraries can exist as source files, linkable object files, archives (aggregated of linkfiles), and loadable object files. See also Loadable Library..

LibList. The list of libraries to be loaded along with a loadfile. However, it may not be the complete list of loadfiles that must be loaded; see loadList definition below.When linking the loadfile, the linker constructs the libList from the names of libraries specified in the linker's command stream; it stores the libList within the loadfile.

Libname. An attribute of a program loadfile, which can be set by the linker, specifying the name of a user library to be loaded with this program.

Linker. A utility whose basic function is to process one or more linkfiles to create a loadfile.

Linker platform. The system on which the linker executes. Also called *host* or *host platform*.

LIC. Library Import Characterization: A data string that characterizes the information used by a linker or loader to bind the global symbols of a particular loadfile. If the same loadfile is bound on two occasions, and its LIC has not changed, the two bindings are the same. Thus it is possible to reuse a set of bindings if it has the same LIC as that determined for this loadlfile in the presence of the other loadfiles with which it is being loaded.

Linkfile. This term describes the output of the compiler and input to the linker. This object file has accompanying tables required to build it into a PIC loadfile and can be all or part of a loadfile. The code of a linkfile is not executable until linked. In the default mode, the linker process one or more linkfiles to produce a loadfile. This term is synonymous with the term "relinkable" in TNS/R .

Loader. A programming utility that transfers a program into memory so it can run. The mechanism that brings loadfiles into memory for execution, maps them into virtual address space, and resolves symbol references among them. Synonyms include run-time loader and run-time linker. The loader for TNS and for TNS/R native programs and libraries that are not position-independent code (PIC) is part of the operating system. For PIC loadfiles and all TNS/E native programs, the loader called `rld` works with the operating system to load programs and libraries.

Loadfile. hThis term describes the input to the runtime loader and default output of the linker. This object file may contain name references to symbols that exist in other loadfiles in the same process. Such references are typically resolved when the loadfiles are brought into memory by the runtime loader `rld` . This term is synonymous with the term "executable" file. An executable object code file is one that is ready for loading into memory and executing on the computer. Loadfiles are further classified as executable programs (containing a main routine at which to begin execution of that program) or executable libraries (supplying routines or variables to multiple programs or separately loaded libraries). A TNS code file might be both a loadfile and a linkfile. Native code files are never both. Contrast with Linkfile.

LoadList. A list of all the libraries that must be loaded for a given loadfile to execute. A loadfile's loadList includes all the libraries in the given loadfile's libList plus all the libraries in those loadfiles' libLists, and so on. It does not include the implicit libraries. The loadList order is the sequence in which these loadfiles are to be loaded when they are not already loaded by a previous operation. The loadList of the program includes all the loadfiles present in the process, in the order they were loaded.

Loadable Library. A loadfile that offers functions and data to other loadfiles. In this document, DLLs are such libraries. A library cannot be invoked externally, for example, by a RUN command; instead, it is invoked by calls or data references from client loadfiles. In TNS/E, functions and data can also be obtained from the system library and millicode.

Loader Library. A public library for loading PIC programs and libraries. It works in close cooperation with the operating system. It is called "`rld`" when loading a program and its libraries at process creation time. It also exports a set of functions for dynamic loading.

Localized. The import-control characteristic of a loadfile that allows it to import symbols only from the loadfile itself followed by the libraries in its libList, libraries that those libraries re-export, and from these, any successions of re-exported libraries.

MCB. The Master Control Block. This contains global information such as the product version number, valid file types, language dialects and floating point types that may be used.

Millicode library. Low-level library routines. Although separate from it, the millicode can be considered an adjunct of the system library.

Presetting. This is the process of resolving references to DLLs at linktime.

PIC. This term stands for 'position independent code' and describes a nomenclature associated with DLLs whereby PIC text contains references do not have to be resolved at link time. Executable code that need not be modified to run at different virtual addresses. External reference addresses appear only in a data area that can be modified by the loader; they do not appear in PIC code. PIC code is even more position independent than one might imagine from the term; it can be simultaneously

mapped to different addresses for different processes in the same CPU. PIC introduces several new elements into ELF files, some of which are adapted from the Intel LP64 ELF structure. TNS/E supports only PIC files. TNS/R supports PIC and non-PIC file types.

Program. This term describes one type of loadfile that is capable of being run on the system. This is the main program and there can only be one program associated with a process.

Public Libraries. A set of libraries (offering widely-used functions) that are managed as part of the system, available to all users of the system, and in large part supplied by HP, although it is possible for customers and third parties to provide DLLs to be added to the public DLLs. A loadfile must explicitly reference a public library in order to access it.

Preempt. When the linker's binding of a symbolic reference to a symbol defined in the same DLL is rebound by the loader to a definition in another loadfile.

Process. An instance of the execution of a program.

Re-exported library. A library whose symbols are made available by another DLL to any localized client of that DLL. Re-export is an attribute of the DLL's libList entry for that library. This attribute is specified by the DLL's programmer and recorded by the linker as a DLL is built. It affects only localized clients of the DLL. This feature allows a symbol to be moved from one DLL to another without relinking clients of the original DLL.

Re-exporting is transitive; i.e., if A re-exports B and B re-exports C, then A re-exports C. Thus, re-exported libraries can re-export other libraries to form a succession of re-exported libraries of arbitrary length.

Region. The Itanium® architecture divides the address space into eight regions, indexed by the high-order three bits of the 64-bit address. TNS/E initially implements just two, regions 0 and 7: region 0 is mapped per-process; region 7 is shared by all processes. Sign extension places "negative" 32-bit addresses in region 7. Note that the high bit of the 32-bit address on TNS/E determines global addressing, and privilege is an attribute of the page; the MIPS architecture on TNS/R is just the opposite.

Relocation. the process of assigning load addresses to the different parts of a program, adjusting the code and data in the program to reflect the assigned addresses.

SearchList. For each loadfile, a list that specifies which libraries to examine, and in which order, to locate symbol definitions needed by that loadfile. The linker and loader construct the loadfile's searchList in accordance with that loadfile's import control, which is set at link time. The system library and millicode are appended to every searchList. A loadfile's searchList is unaffected by the import control of any other loadfile.

Sections and Segments. The TNS/E object file is organized into contiguous items called sections. There is an array of ELF section headers that contains the type and name of

each of these section items. A section is not required to be present if it would not contain any useful information for a given object file. In loadfiles, some of the sections are further organized in segments that get loaded into virtual memory.

Strip file. These are files do not have debugging information; i.e., DWARF symbol table, in it. Stripping can be done on any object file. It is still possible for the linker to process a linkfile that has been stripped because the DWARF symbol table does not contain any essential information to it. An import library can be stripped even if the corresponding DLL is not stripped.

Symbol Resolution. When a program is built from multiple subprograms, the references from one subprogram to another are made using symbols. For example a main program might use a square root routine called `sqrt` and the math library defines `sqrt`. A linker resolves the symbol by noting the location assigned to `sqrt` in the math library and patches the caller's object code so the call instruction refers to that location.

Semi-globalized. An import control characteristic of a loadfile that allows the loadfile first to obtain symbols from its own definitions and then to obtain others as for a globalized loadfile. Thus, a semi-globalized loadfile cannot have its symbol references to itself preempted. See also SearchList..

Symbol. The symbolic name of a function or data item. Symbols are defined in loadfiles and referenced in the same or other loadfiles.

Symbol definition. a function or data item whose name is the symbol.

Symbol value. the address of a definition of that symbol.

Symbolic reference. An occurrence in code or data of a symbol that is or must be bound to a definition of that symbol. The symbolic reference is bound (resolved and made usable) by assigning to it the value of a definition of that symbol.

System library. TNS/E library routines required to access TNS/E operating system functions. (Similar for TNS/R.) The loader automatically searches the system library for definitions that satisfy a loadfile's unresolved symbols after searching all the loadfiles in the loadfile's searchList.

TNS/E. The hardware platform based on the Itanium™ architecture and the HP NonStop operating system and software that are specific to that platform. All code is PIC.

TNS/R. The hardware platform based on the MIPS™ architecture and the HP NonStop operating system and software that are specific to that platform. Code may be PIC or non-PIC.

TLB. Translation Lookaside Buffer: a cache of page table entries, where each entry designates the physical memory page corresponding to a range of virtual addresses. Information within the entry can make the translation unique to the accessing process. Unless the appropriate TLB entry is present, the page cannot be accessed; typically

the processor generates a fault to allow software to find and load the missing entry from a memory-management structure.

TNS/E object file format. This object file format is an amalgam of Intel IA-64 code architecture and the HP NonStop operating system extensions.

TNS/E object files are categorized into three types of files: linkfiles, loadfiles, and import libraries. The following are key differences between TNS/R and TNS/E platforms:

Platform	TNS/R	TNS/E
Processor	MIPS RISC	Itanium
Architecture	SGI	Intel IA-64
Programming model	32-bit (ILP32)	32-bit (ILP32) and in future: 64-bit LP64
Object type	ELF and COFF	ELF exclusive
Debugging symbols	Third-Eye	DWARF2
Compiler Backend	SGI w/ HP extensions	Intel w/ HP extensions
Linker, PIC	`ld`	`eld`

User library. A loadable library; primarily a legacy feature for NonStop systems. For PIC programs, a user library is a DLL treated as if it were the first library in the program's libList and therefore is searched first for symbols required by the program. However, a user library does not appear in the program's libList; instead, its name is recorded in the program's loadfile as the libname attribute. A program can be associated with at most one user library; the association can be specified using the linker at link time or in a later change command, or at run time using the process creation interfaces. (The /LIB.../ option to the RUN command in TACL uses these interfaces.)

VHPT. Virtual Hash Page Table: an Itanium® architecture feature that can supply missing TLB entries without generating faults.

VPROC. The version procedure number used to identify which version of the product you are using.

Zimpimp file. The internal name of the imp-imp file. Also called the "import library that represents the implicit DLL's", it is the file that tells which symbols are available in the set of implicit DLL's, which collectively correspond to what was previously called the system library. See also Implicit library import library (imp-imp).

Zreg file. This is the internal name of the public DLL registry file, which lists the names of all the public DLL's.

Index

L

M

N

O

P

R

S

T

U

V

W

Z

Special Characters