# DLL Programmer's Guide for TNS/E Systems

**Abstract**

This guide describes how application programmers can use the DLL facilities provided on TNS/E systems and recommends good practices in using them.

**Product Version**

T9050 at H06.01

**Supported Release Version Updates (RVUs)**

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

| Part Number | Published |
|---|---|
| 527252-006 | April 2012 |

**Document History**

| Part Number | Product Version | Published |
|---|---|---|
| 527252-002 | T9050 | February 2005 |
| 527252-003 | T9050 | May 2005 |
| 527252-004 | T9050 | July 2005 |
| 527252-005 | T9050 | August 2010 |
| 527252-006 | T9050 | April 2012 |

# Legal Notices

# DLL Programmer's Guide for TNS/E Systems

Glossary Index Figures Tables

# 2. Essential DLL Facility Controls

## Figures

## Tables

# What's New in This Manual

## Manual Information

### Abstract

This guide describes how application programmers can use the DLL facilities provided on TNS/E systems and recommends good practices in using them.

### Product Version

T9050 at H06.01

### Supported Release Version Updates (RVUs)

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

| Part Number | Published |
|---|---|
| 527252-006 | April 2012 |

### Document History

| Part Number | Product Version | Published |
|---|---|---|
| 527252-002 | T9050 | February 2005 |
| 527252-003 | T9050 | May 2005 |
| 527252-004 | T9050 | July 2005 |
| 527252-005 | T9050 | August 2010 |
| 527252-006 | T9050 | April 2012 |

## New and Changed Information

### Changes to the H06.25/J06.14 manual:

- Updated the section What is a DLL on page 1-3.
- Updated the section Where The Linker Searches for Libraries and Archives on page 2-11.
- Updated the section The Link-Time-Defined Search Path of the Loader on page 2-17.
- Updated the section Controlling the Loader's Search Path at Load Time on page 5-13.
- Updated the table Set Attributes on page A-7.

- Added the term Neutral loadfile. on page Glossary-4.

## Changes to the H06.21/J06.10 manual:

- Added new section Special initialization and termination procedures on page 5-5.
- Add a note on `systype` on page 5-8.
- Add a note on `data2protected` attribute on page A-4.
- Added c99 compiler information throughout the manual, wherever applicable.

## Changes to the 527252-004 Manual

This is a new manual for TNS/E systems (based on the TNS/R version).

# About This Manual

## Purpose of This Manual

The *DLL Programmer's Guide For TNS/E Systems* is intended as an introduction to the process of creating and using Dynamic-Link Libraries (DLLs) on TNS/E H-series systems.

## Who Should Read This Manual

Applications and System Programmers who want to create or use DLLs.

## How This Manual Is Organized

This manual consists of the following sections:

Section 1, DLLs on a TNS/E System. This section explains the TNS/E DLL facility – what DLLs are, how they work, how they can be used, and the basic workings of the tools that create them.

Section 2, Essential DLL Facility Controls. This section explains the linker's most commonly used controls.

Section 3, Dynamic Use of DLLs. This section discusses how to dynamically load and unload a DLL from your running process and how to link your loadfile with a dynamically loaded DLL.

Section 4, Finding Symbol Definitions. This section describes how the linker and loader resolve symbol references, including cases when multiple definitions are available for the same symbol name.

Section 5, Advanced DLL Facility Controls. This section tells how you can manually override and extend previously described linker and loader defaults and options to meet special needs.

Section 6, Example Code. This section contains a set of examples to introduce you to some of the tools and capabilities for building dynamic linked libraries on a TNS/E system.

## Related Reading

The following manuals (use H06.03 or later versions) form the set that you may need to create and use DLLs:

- *eld Manual.*
  This contains details on options for the `eld` linker.
- *rld Manual.*
  This contains details on dynamic loading facilities.

- *eNOFT Manual.*
  This contains information on examining PIC object files on H-series systems.
- *TACL Reference Manual.*
  This contains information on loadfiles in processes and processes using loadfiles.
- *Guardian Procedure Calls Reference Manual.*
- *Guardian Procedure Errors and Messages Manual.*

You will also be using a programming language, so choose from the following:

- *COBOL85 For Non-Stop Systems*
- *C and C++ Programmer's Guide*
- *Guardian TNS/R Native C Library Calls Reference Manual*

If you are using the OSS programming environment, you may need the following manuals:

- *OSS Library Calls Reference Manual*
- *OSS Shell and Utilities Manual*

# Notation Conventions

## 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 this 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`*

**[ ] 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, in TAL or PTAL, 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
```

Note that some interfaces count the pair as a single parameter for error-reporting purposes, even though they constitute two separate parameters, and must be so expressed in C or C++.

**!o:i.** In procedure calls, in TAL or PTAL, 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
```

Note that some interfaces count the pair as a single parameter for error-reporting purposes, even though they constitute two separate parameters, and must be so expressed in C or C++.

## 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.

## HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

# 1 DLLs on a TNS/E System

This section explains the TNS/E DLL (dynamic-link library) facility – what DLLs are, how they work, how they can be used, and the basic workings of the tools that create them. This facility comprises the linker (`eld`), the loader (`rld`), and the portions of the HP *NonStop*™ operating system used in loading libraries. The facility runs in either the OSS or Guardian environments, and on auxiliary Windows systems you can construct, but not load, TNS/E applications that use DLLs.

DLLs or equivalent features exist on other platforms in the industry (for example, UNIX and Windows); on UNIX systems they are known as DSO's (dynamically shared objects).

Note that this manual discusses the use of the linker `eld` on TNS/E systems. There is a companion manual, *The DLL Programmer's Guide for TNS/R Systems* that covers the use of the other PIC linker, `ld`, which is designed for use on TNS/R systems.

## Libraries and Symbols

As used here, a library is a loadable object or code file (or loadfile), that offers functions and data for use by other loadable code files. In a library, each such function and data item has its own symbolic name, as defined in its original source. This symbolic name is known as a symbol, and the function or data offered under this name is called the symbol definition.

Code that needs a function or data item refers to that function or data by its symbolic name; such a reference is called a symbolic reference. Thus, a symbolic reference is the requisition for or the use of the symbol's definition. A symbolic reference is satisfied by assigning to it the symbol value, which is the virtual address of the symbol's definition. The process of assigning that value is called binding or resolving the symbolic reference. At load time, for every symbolic reference in every loadfile, the system finds the program or a library that offers a function or data item having the same name, and it binds that reference to a corresponding definition.

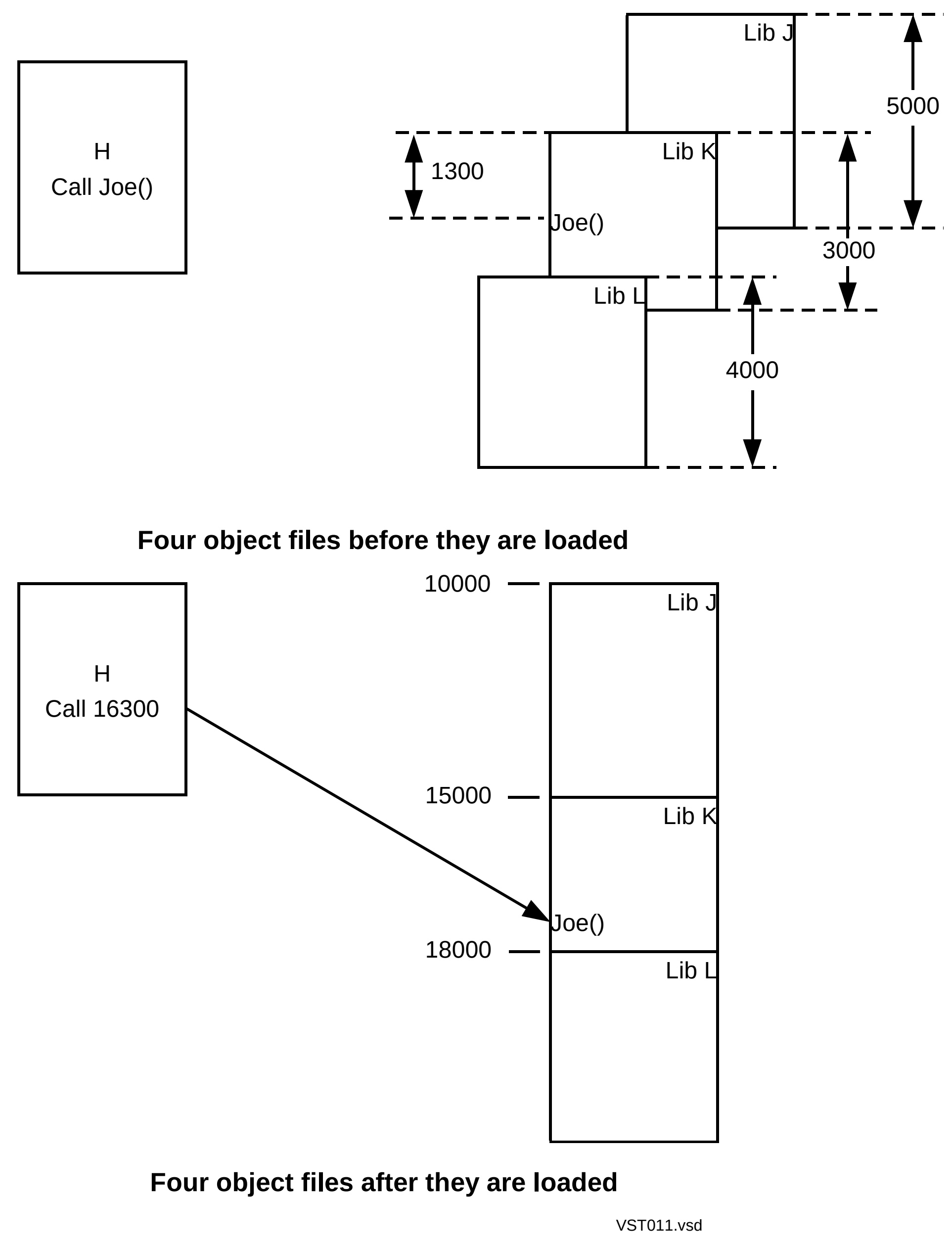
Figure 1-1 on page 1-2 is an example of how this binding works; while this example is illustrative, it is not entirely realistic, for instance, in the numbers used for virtual addresses. The upper portion of this figure shows the situation before the loadfiles are loaded. A loadfile on the left, called H, which may be a program or a library, contains a reference to a symbol, `Joe`. Three libraries are shown on the right; library J is 5000 bytes long; library K is 3000 bytes long; and library L is 4000 bytes long. Lib K contains a definition of Joe, which has an entry point offset 1300 bytes from the start of that library.

The lower portion of Figure 1-1 shows the situation after the four code files are loaded in a process. The loader placed the three libraries contiguously starting at some virtual address, say, 10000. This means that the definition of `Joe` is now located at virtual address 16300; therefore, this is the value of `Joe`. The loader has resolved, or bound, the reference in H to Joe by replacing the symbol in H with the value of Joe, namely, 16300.

A loadfile exports a symbol when it defines a symbol that can be used by another loadfile. A loadfile imports a needed symbol when that symbol reference in the program or DLL is or will be set to the value (address) of a symbol of the same name exported by another loadfile. Thus, in Figure 1-1, Library K exports `Joe` and, after H is loaded, H imports `Joe`.

In the rest of this document, the term *library* means a DLL that may be designed for either public or private use.

**Figure 1-1. Symbol Resolution by Binding at Load Time**



## Position-Independent Code (PIC) in TNS/E

All code on a TNS/E system is position-independent. Code that can be relocated in virtual memory at load time without alteration is called position-independent code (PIC). If you use DLLs in your application, your program and these DLLs must be PIC. All references in PIC files to global or external symbols are made indirectly through addresses stored in a data area so that the loader can find and bind them to reflect their virtual-memory locations at load time without modifying code. The TNS/E compilers can generate PIC files, the PIC linker (`eld`) creates either PIC programs or DLLs, and the loader and operating system load and bind the results.

PIC is more position-independent than one might imagine from the term. PIC can even be simultaneously mapped to different addresses for different processes in the same CPU.

## What is a DLL?

A DLL is a type of library that is constructed of PIC. When using DLLs on TNS/E, a complete, executable application comprises one (main) program and zero or more DLLs. The program is the root of the application, while the DLLs provide functions and data needed by the program or other DLLs. DLLs allow you to structure your applications in functional units (the DLLs). A DLL might be a library that supports a single program, it might be available to a project or a group with common computational needs, or it might be a library that is available to all users.

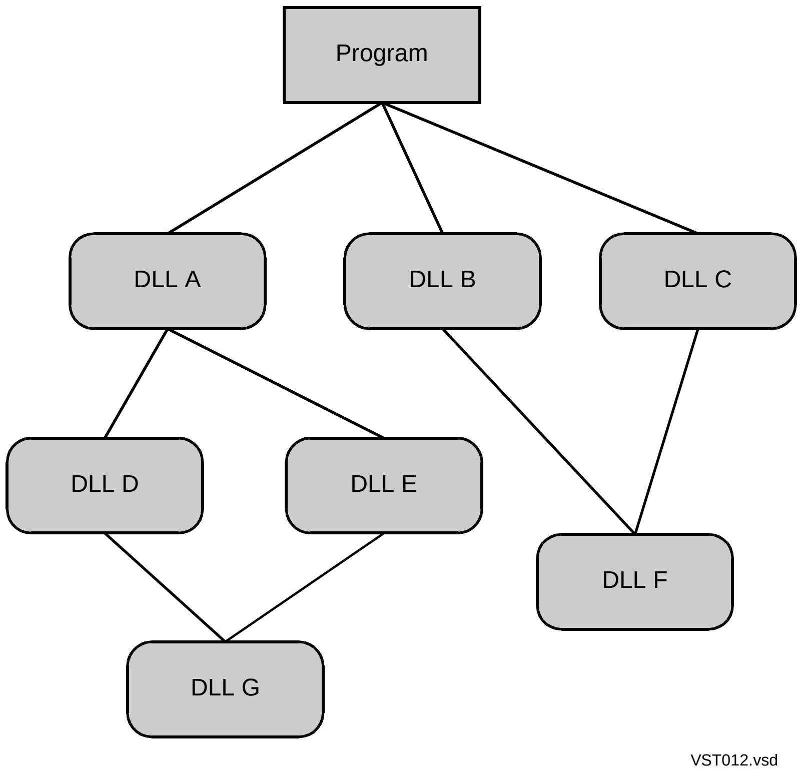
Figure 1-2 on page 1-4, shows an application comprising a program and the DLLs that it requires in order to run. Some of these (A, B, and C) offer symbol definitions that the program itself may need; others offer symbol definitions that the various libraries, but not the program, may need.

A, B, and C might be libraries that, along with Program, constitute the basic logic of the application. D and E might be libraries of supporting routines for A. F and G might be more general purpose libraries.

A DLL is written as an ordinary program with no main procedure and is designated a DLL in the course of construction.

A DLL can be a 32-bit, 64-bit, or neutral loadfile on a TNS/E OSS platform and 32-bit loadfile on all other platforms. 64-bit and neutral DLLs are supported from H06.24/J06.13 RVU onwards.

**Figure 1-2. An Application Made Up Of a Program And The DLLs It Needs**



## Why Dynamically Linked Libraries?

For statically linked programs, you must copy all shared libraries into your own programs; thus, complete copies of these libraries appear in all the programs that use them, and each copy consumes main memory and disk space. Also, whenever such a library is changed, it must be relinked into all the programs that need those changes.

Some key characteristics of DLLs are:

- A single copy of a DLL in physical memory can be shared among multiple processes.
- A DLL that is shared among processes can appear at a different virtual address in each process and each instance has its own copy of global data that is available to its process.
- Multiple processes can use different versions of the same DLL simultaneously, though each version must have a different name or be stored in a different location in the file system.
- The same program can run simultaneously in different processes with different DLLs supplying the supporting data and functions.

- A new version of a DLL can be introduced without having to alter a program or DLL that references it, even though the location of the referenced sites has changed. This gives you great freedom to change a DLL.
- A program and the DLLs loaded with it at process creation time can access each other's functions and data by simply referencing their symbolic names.
- A running application can cause a DLL to be loaded dynamically and to make its functions and data accessible.

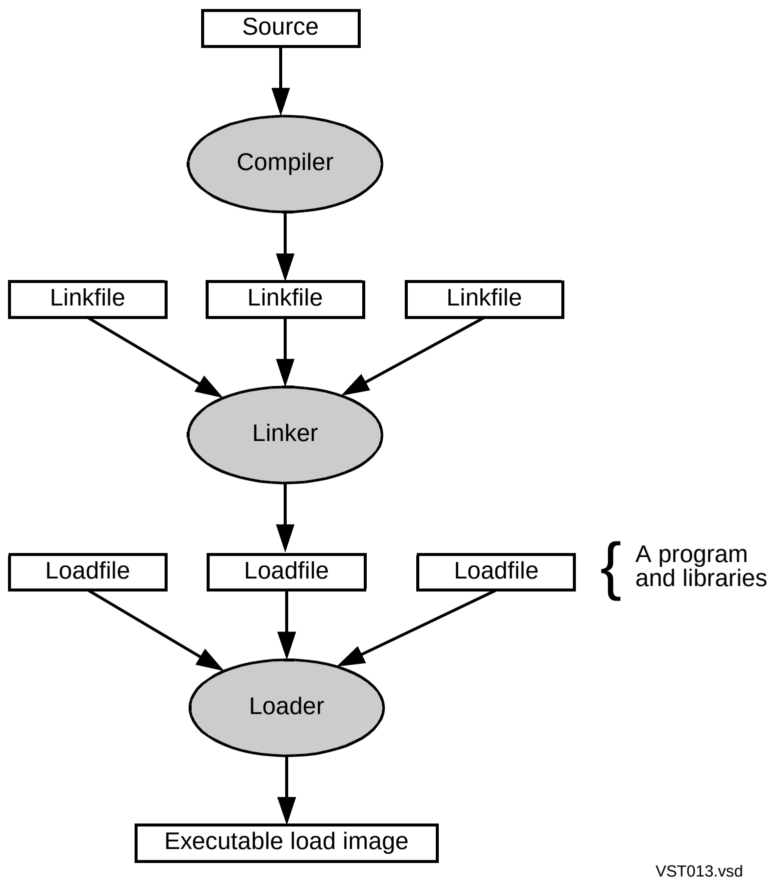
## Building and Loading Programs and Libraries

### Generating an Executable Load Image

Figure 1-3 on page 1-6 shows the steps for going from a source file to an executable PIC load image. The compiler translates a source code file into a linkable code file, called a linkfile. The linker brings together one or more linkfiles to create a loadfile, which can be designated as either a program or a DLL.

Finally, at load time, the loader determines the arrangement of the libraries in virtual memory, resolves references among loadfiles, and loads the program and libraries for execution. Your application can also cause DLLs to be loaded after the program has been loaded and put into execution.

**Figure 1-3. Simplified Code Generation and Load Processes**



## Compilers

The TNS/E compilers translate source input into a linkfile. In addition to object code and data, TNS/E compilers generate the following auxiliary information as part of their linkfile output that is used by the linker:

- A symbol table, which identifies all the symbols that this linkfile either makes available to other code files or needs from other code files.
- A relocation table, which points to all the places in the compiled code and data that must be relocated by the linker. Each entry that refers to an external symbol contains a pointer to the corresponding entry in the external symbol table.

## Linker

The TNS/E PIC linker is named `eld`; it combines one or more PIC linkfiles to create a PIC loadfile. In doing this, the linker manipulates both code and data, then places all of the loadfile's adjustable references in tables outside the code to make them available to the loader. This process, called linking or executing a link, must be applied to linkfiles after they have been compiled and before they can be loaded for execution.

You invoke the linker by a single command, `eld`, and control it by items you enter in its command stream, which comprises the options, file names, and parameters that modify the `eld` command.

Later, the loader brings together programs with their required libraries in structures like that shown in Figure 1-2 on page 1-4. To enable the linker to find the required libraries when it links a loadfile, the loadfile's programmer must enter in the command stream the names of libraries that can provide symbols that this loadfile needs. From these names, the linker creates in the loadfile a libList, which lists the names and certain attributes of each of these libraries. A library that is listed in a loadfile's libList is said to be directly referenced by that loadfile.

In simple compile-and-link operations, the compiler runs the linker automatically. When it does this, the compiler tells the linker the names of certain standard libraries. If those are the only libraries required, you need do nothing more. But if you require other libraries, you can command the compiler to pass them to the linker, or, you can run the linker yourself. In the latter case you must provide to the linker with the required library names, including the ones the compiler would have automatically done for you.

The linker can run on a TNS/E machine in either the Guardian or OSS environment. It can also run on Win32 support machines, usually in an ETK environment. In this document, these support machines are called auxiliary systems. The object file produced can only be run on TNS/E systems.

## Loading

Once a program and its DLLs have been processed by the PIC linker, they can be loaded for execution by a special library program (the run-time linker `rld)` that works with the operating system. This combined facility of `rld` and the operating system that loads programs and libraries into virtual memory for execution is called, in this document, the loader.

### FastLoad

Required libraries are not statically linked with the program. Instead, at load time, the program and its DLLs are brought into virtual memory, and the loader resolves references among them. The loader does not alter the stored file image of the loadfile; rather, it changes only the loaded memory image of the loadfile's tables and other initialized data. So this load-time adjustment might be repeated every time a program and its DLLs are loaded, although the FastLoad facility loads a preset loadfile without having to rebind it.

### Automatic Update

If `rld` has to rebind a loadfile and the loadfile import control is localized, it will update the preset bindings in the loadfile with the cooperation of the NonStop operating system. This is called automatic update. `rld` and the operating system only automatically update loadfiles at process creation time. If a loadfile is loaded via a call to `dlopen()`, the loadfile is not automatically updated.

See Section 3, Dynamic Use of DLLs for further details.

After the program and the initially loaded libraries are running, the program or a loaded DLL can also call on the loader to dynamically load yet other DLLs. References by the newly loaded DLLs are resolved among already loaded loadfiles, and subsequent function calls can retrieve symbols offered by the newly loaded DLLs.

The system automatically arranges loadfiles in virtual memory so there is no address overlap and interference.

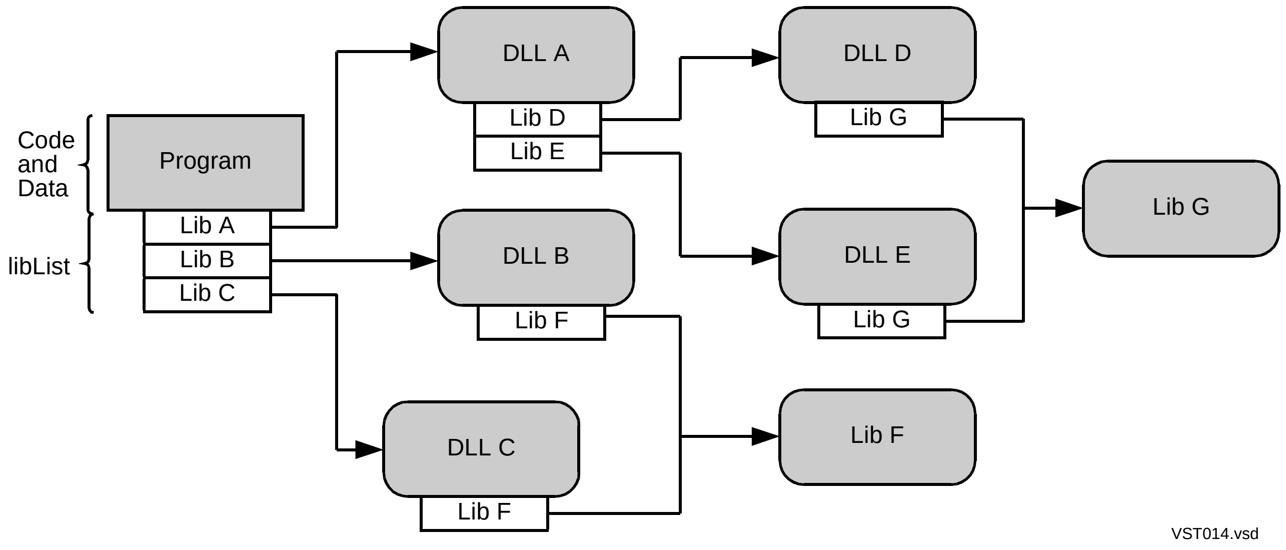
## Finding the Needed Libraries

A loadfile must be loaded with all the libraries that it directly references, that is, the libraries in its libList. However, loading only the directly referenced libraries might not be enough, because some of these libraries might directly reference still other libraries. Therefore these other libraries must also be loaded, and they might further require still other libraries. Proceeding in this way can lead to an arbitrarily long succession of required libraries, all of which must be loaded to run the given loadfile.

The loader generates a list of all the libraries that must be loaded in order to run a given loadfile by starting from the libList of that loadfile and proceeding through the libLists of all the required libraries. This generated list is that loadfile's loadList**,** and the loader orders it in the sequence that these libraries are to be loaded, as discussed in The SearchList for a Globalized Loadfile on page 4-8. Subsequently, the loader uses the loadList to control the loading operation.

Among the libraries on its loadList, the loadfile directly references those libraries on its own libList, while the rest are indirectly referenced libraries. Figure 1-4, below, shows a more detailed view of what Figure 1-2 on page 1-4 suggests. Figure 1-4 illustrates how the program directly references libraries A, B, and C, which were specified to the linker by the programmer and consequently appear in Program's libList. Also, A directly references D and E, B directly references F, and C directly references F. The programmers of A, B, and C stated these requirements for these libraries when each was linked. Likewise, D and E both directly reference G. Finally, neither F nor G reference any other libraries.

**Figure 1-4. Loadfiles and Their libLists (To Create the Structure in Figure 1-2)**



For the loadfiles in Figure 1-4:

- The program indirectly references D, E, F, and G, so these libraries do not appear in the program's libList.
- The program's loadList comprises all the libraries shown in the figure.
- A indirectly references G, and A's loadList comprises D, E, and G.

Neither a loadfile nor its programmer needs to know about its indirect needs. There is no limit to the number or depth of directly and indirectly referenced libraries in a loadList. Also, many libraries in the loadList can reference the same library, as illustrated by the multiple clients of F and G. Each library appears once in the loadLIst and in memory.

## The TNS/E Library Facility

This subsection gives an overview of the library facility and some of the controls it offers. Discussion of how to control the library-facility begins in Essential DLL Facility Controls on page 2-1.

### Public Libraries and DLLs (Implicit and Explicit).

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 Figure 1-2 on page 1-4, DLLs F and G might instead be public DLLs, because they refer to no PIC loadfiles. This is reflected in Figure 1-4 on page 1-9, where these libraries are labeled simply Lib.

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 implicit 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 DSM/SCM uses on your behalf), 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 privileged 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.

## Linkfiles and Archives

When the linker is building a new loadfile, its command stream must contain the names of one or more linkfiles to be transformed into the loadfile. A programmer brings together linkfiles to make a loadfile because these linkfiles are designed to work together and they often cross-reference each other.

A linkfile can also be stored in an archive, which is a file that holds one or more linkfiles. Normally an archive stores what might be called auxiliary linkfiles, which serve general purposes and can be included in different links. For example, you might store a number of utility routines as linkfiles in an archive and incorporate them as needed.

## Import Controls

Import controls allow you to determine from which other loadfiles your loadfile can import symbols. These controls take the form of attributes that you assign to your loadfile. This topic is discussed in detail in Import Controls and SearchLists on page 4-5. The three variants of import controls are:

- **Localized** — A localized loadfile can import symbols from certain libraries in this loadfile’s loadList. The choice of libraries is discussed in detail in The SearchList for a Localized Loadfile on page 4-5.
- **Globalized** — A globalized loadfile can import symbols from the program it is loaded with and any loadfile in the program’s loadList.

- **Semi-globalized** — A semi-globalized loadfile uses its own definition for any of its symbol references when it offers such a definition, but imports other definitions from the program it is loaded with and any loadfile in the program's loadList.

In any case, if the symbol to be imported is defined only once in the collection of loadfile candidates to supply imported symbols, that symbol is used to resolve your loadfile's need. Good practice normally avoids multiple definitions of the same symbol in the loadfiles in a loadList, because of the danger that an imported symbol could be resolved in an unexpected way. However, the library facility allows multiple definitions, and symbol resolution in such cases is discussed in Import Controls and SearchLists on page 4-5.

Localized symbol resolution is consistent with long standing conventions on NonStop systems including TNS programs and user libraries, as well as native non-PIC programs and SRLs in TNS/R. The linker's default is to localize loadfiles. On the other hand, globalized symbol resolution is an industry (UNIX) standard. One useful consequence of globalized symbol resolution is that DLLs can import symbols from their clients, including the program. Globalized symbol resolution can be especially useful when multiple loadfiles define and use the same symbol and you want to ensure that they all use the same one. If you declare all these loadfiles to be globalized, then the loader will resolve all these imports to the same exported symbol. However, object files using globalized or semi-globalized import are likely to take somewhat longer to load, because the linker is unable to preset bindings for files not seen at link time.

## Other Loader Operations

The loader and operating system assign the program being loaded and the libraries in its loadList to their positions in virtual memory and bind their symbols to their appropriate definitions in these loadfiles.

### Adjusting Symbol Values and Relocating in Virtual Memory

You do not have to relink a loadfile when a DLL it uses is changed. To accommodate this, each instance of a DLL must be relocatable in virtual memory when the DLL is loaded for execution. This requirement is the reason that programs and DLLs are written in PIC (although all code on a TNS/E system is PIC). Virtual instances of a DLL do not depend on being loaded in any particular location in virtual memory, and they do not depend on any symbols they reference being loaded in any particular location. For example, you can replace a DLL with a new one of the same name that provides the same symbols without relinking the DLL's clients, even if the locations of these symbols are different in the two DLLs.

### Dynamically Loaded DLLs

A running program or DLL can load and open a previously not-loaded DLL and gain access to the symbols it offers. DLLs invoked this way are called dynamically loaded DLLs; they are further described in Section 3, Dynamic Use of DLLs.

An important advantage of dynamically loaded DLLs is that their names and their symbols need not be known when the program is constructed. Using this facility, you can add to an existing application a new DLL that provides new functionality without even restarting the application.

You do not need to load infrequently used DLLs when the application is loaded. Instead, you can load and use these DLLs when required and unload them when they are no longer needed. They can be reloaded whenever necessary.

## User Library

A program, but not a DLL, can be linked to one user library, which is a DLL having a special relationship to the program. Instead of adding the user library's name to the program's libList, the user-library name is recorded in the program loadfile as an attribute called libname. However the linker and loader treat this DLL as if it were first in the program's libList, and a program's loadList always begins with the user library if there is one.

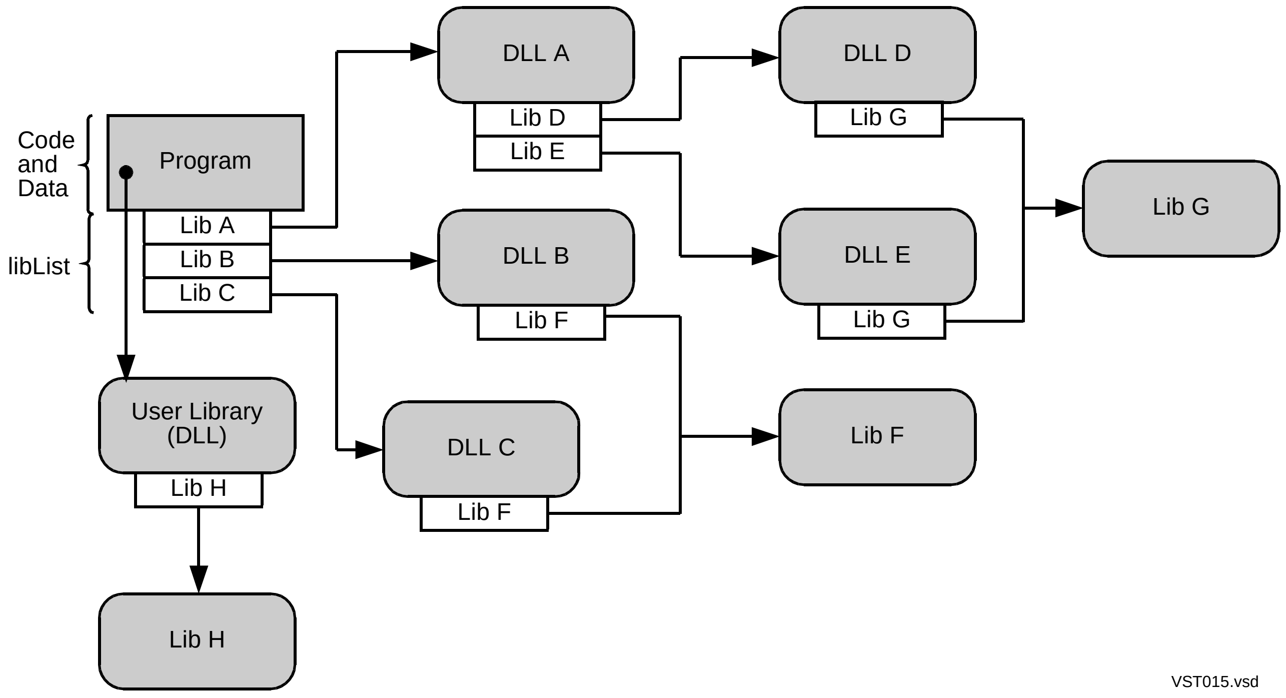
It should be noted that you can run two instances of the same program simultaneously where each instance of the program uses a different user library.

The starting virtual addresses of the program text (code) and data segments are system constants, set by the linker and enforced by the operating system. The linker also sets "preferred" virtual addresses for the text and data of DLLs, either by default or from command-string input. (DLL data immediately follows the text.) If the preferred address ranges are available at load time (do not conflict with already loaded objects or reserved areas), they are used; otherwise the operating system finds an available address range for the DLL.

Like any other DLL, a user library can require other libraries. Figure 1-5 below shows a User Library assigned to Program, and this User Library itself requires the library Lib H. Program directly references the user library and hence indirectly references H.

You can assign a DLL as a user library to a program when you link or load that program, or by a special linker command that allows you to change this assignment in an existing program. More detail is provided in How to Set Run-Time Attributes of Your Loadfile on page 5-6.

**Figure 1-5. Loadfiles of Figure 1-4 with a User Lib and Its Library Added**



A user library can also be specified at run time:

- In Guardian, the run-option lib can specify the fully qualified name of the user library:

  ```
  /lib $vol.subvol.name/
  ```

Alternatively, the option with no name, /lib/, causes any libname attribute in the program file to be ignored.

- In OSS, the shell command run has a -lib option that accepts a fully qualified Guardian file name in OSS notation:

  ```
  -lib /G/vol/subvol/name
  ```

Alternatively, use -lib=unset to disregard any libname attribute in the program file.

A /lib .../ or -lib=... specification at run time does not change the libname attribute of the program file.

Ordinary DLLs are generally more convenient, and there can be more than one, so the use of a user library with PIC programs is not encouraged. The feature is provided primarily for compatibility with legacy practice on NonStop systems. A user library can also provide an "intercept" facility: because it is loaded first after the program, a user library could export symbol definitions that take precedence over those in DLLs on the linker-provided libList.

# 2 Essential DLL Facility Controls

You control the TNS/E DLL facility by using the linker and the loader options, many of which normally run using automatic defaults. This section explains the linker's most commonly used controls, but Advanced DLL Facility Controls on page 5-1 tells you how to get more precise control over the process and its results.

The execution target for code produced by the linker is either OSS or Guardian on TNS/E (or possibly both, for a DLL); but you will often link on an auxiliary system, which usually means a Win32 workstation that supports development or administration. Certain linker options facilitate this capability. Wherever the linker performs its link is called the linker host platform, whether or not it is an auxiliary system.

Implementations of the linker utility run on each host: Guardian, OSS, or Win32.

More detailed reference information for the Guardian version may be found in the *eld Manual*.

## The Linker's Command Stream

You operate the linker by:

- Starting it as a new process
- Providing the appropriate inputs that tell the linker what to do
- Examining the textual output it produces

Most of this section describes direct use of the `eld` utility which can also be used indirectly in several ways:

- Compiler drivers, such as CCOMP and CPPCOMP (Guardian), or c89 and c99 (OSS) can run the linker during the compilation. The compiler driver provides the linker's command stream, based on defaults and some compiler options. c89 and c99 compilers also have syntax

  ```
  -Weld=...
  ```

  to pass options through to the linker.
- The Enterprise ToolKit has facilities to invoke the linker automatically.

### Direct Use of the Linker

You invoke the linker by a single command, `eld`, and control its subsequent link operation by tokens inserted following the command on the command line. These tokens come in three varieties:

- Option – A directive to the linker, which might be modified by arguments that immediately follow it. These arguments are either file names or parameters. Options without parameters are sometimes called flags.

- File name – A name of a file, which may or may not be an argument of an option. In this document, a file name in the command stream that is not a part of an option is said to be directly inserted in the command stream.
- Parameter – A non-file-name argument of an option

The command stream comprises all the tokens that modify an `eld` command; these are processed in the order they appear in the command stream. These tokens can come from the command line or from command files that are referenced in the command line. In this document, to insert an item means to make it a token in the linker's command stream either directly or as an option argument.

Herein, options are defined with the file names and parameters they require. Thus, in the next paragraph, the `-obey` option is defined as `-obey<filename>` meaning that `<filename >` is required when the option is declared.

`-obey <filename>` is a linker option that designates a command file containing tokens to be incorporated into the linker's command stream. The linker processes these tokens in sequential order before it moves on to the next token on the command line. Such a file is called an obey file. Obey files can be called from within an obey file. The linker accepts `-FL` as a synonym for `-obey`. To use the standard input file as an obey file, insert the `-stdin` option with no file names or parameters.

## Option Types

Options fall into three categories:

- Repeatable option - Each occurrence adds another element of information or causes the linker to repeat an operation.
- Toggle option - A set of options, usually a pair. One of the set turns on a designated linker behavior, and that behavior remains in effect until the linker encounters another member of the set in the command stream, which invokes a different linker behavior. Command-stream processing begins with one behavior as default, and the option members can be repeated in the command stream as many times as needed.
- One-time option - All other options, which can only appear once in a command stream.

The order in which an option appears in the command stream makes no difference except where specifically mentioned. However, when adding an option to the command stream, be sure to avoid inserting it where it separates some other option from its arguments.

## Specifying the Linker's Output

By default, the linker merges linkfiles to produce loadfiles. Following the `eld` command, you should list the names of the linkfiles to be combined, as illustrated in this command-stream fragment:

```
eld linkfile1 linkfile2
```

where `linkfile1` and `linkfile2` are names of linkfiles to be combined in the link.

### Choosing the Output File

The output file is where the linker stores the loadfile that results from a link. To specify an output file name, insert the `-o <filename>` option, where `<filename >` is the desired name of a file. For example:

```
eld linkfile1 linkfile2 -o mainout
```

This specifies that the linkfiles from the previous example are to be linked and the output loadfile is to be stored in `mainout`. `-o` is a one-time option.

### Choosing to Create a Program or a DLL

By default, the linker combines specified linkfiles to produce a loadfile that is a program, not a DLL. Thus, the previous example will cause the linker to produce a program that it stores in `mainout`. To explicitly cause the linker to produce that program, insert the `-call_shared` option.

To create a DLL in `mainout`, insert the option `-dll`, or its synonym, `-shared`, as in the following example.

```
eld linkfile1 linkfile2 -dll -o mainout
```

`-call_shared` and `-dll` are mutually exclusive, one-time options.

### Naming DLLs

Unlike programs, every DLL has an internal name that is also the name of the file where that DLL is (to be) stored on the execution target; see Choosing a DLL Name on page 2-6. This name is specified when the DLL is linked and is recorded in the DLL itself. The `-dllname` option gives the DLL its internal name, independently of the name of the file in which you store it. The `-dllname` option can be used to create an arbitrary DLL name, for example, a fully qualified file name.

You may use `-soname` as a synonym for `-dllname.` (`so` stands for shared object.)

The following example might be used to link on a Win32 system, to give the resulting DLL the name `maindll`, and to store it in a file named `mainoutput.dll` in a directory named `C:\myfiles\dlls`.

```
eld linkfile1 linkfile2 -dll -dllname maindll &
-o C:\myfiles\dlls\mainoutput.dll
```

You also have three ways to give the DLL and its output file the same name:

- The `-o <filename>` option, when used by itself, assigns `<filename>` to the internal DLL name as well as the file. When using the -o option and `<filename>` is a qualified file name, the DLL name is obtained by truncating `<filename>` to remove the path or subvolume definition and yield the corresponding unqualified name. See File-Name Qualification below.
- The `-dllname <filename>` option, when used by itself, also assigns `<filename>` to the file.
- You can use both the `-dllname` and `-o` options with the same name in each.

The following example of the first option gives the resulting DLL the name `mainout` and also stores it in a file named `mainout`.

```
eld linkfile1 linkfile2 -dll -o mainout
```

If you do not insert either a `-dllname` or a `-o` option, the linker will assign the same default name to both the DLL and the output file. If the linker is running on a Guardian host, that name is `aout`; otherwise, it is `a.out`.

## File-Name Qualification

The linker distinguishes between:

- An unqualified, or “simple” file name (also known as the file identifier), which identifies a file within a directory or subvolume but which must be appended to the directory or subvolume definition and expanded with file-name augmentation according to the file system, as described in Augmenting Library Names Automatically in Searches on page 5-2
- A partially qualified file name, which identifies the file uniquely in the file system where the name is used.
- A fully qualified file name: for Guardian, has $vol.subvol.name, perhaps prefaced by \system; for OSS, begins with /.

The linker identifies a qualified file name as one that contains:

- On Guardian – a period, backward slash, or dollar sign
- On OSS – a forward slash
- On Win32 – a forward slash, a backward slash, or a colon

All other file names are regarded as unqualified.

For any stand-alone file name (specifying a linker input), the handling is much the same:

- Guardian - Apply the =_DEFAULTS DEFINE to fill in any system, volume, or subvol that is missing. (This is a no-op for a fully qualified name). See also the MAP DEFINES note below.

- OSS - If it is "absolute" (begins with a /), take it as is; otherwise append it to the current working directory.
- Win32 - As above, except for drive: and \ issues.

The distinctions become more important for file names specified by the -lib (-l) option. In this case, only unqualified names are subject to searching through the list of paths. Partially or fully qualified names are treated as above.

## Notes About MAP DEFINES

### In General

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 cited above 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.

### eld Specifics

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 will be 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 uppercase, 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 specified 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.

## Choosing a DLL Name

On the execution target, you must store a DLL in a file having the same name as that DLL, or else the loader will be unable to find it. This is because when a loadfile being linked requires that DLL, the linker uses that DLL's internal name to enter in this loadfile's libList. Then, when the loader (on the execution target) searches for DLLs this loadfile requires, it searches the file system using names from this loadfile's libList.

The ability to name a DLL differently from its linker-output file is useful when linking on an auxiliary system, so you can name that DLL for the file you want to store it in on the execution target.

On the other hand, names in a libList may be used on both the linker platform and the execution target systems, so since libList names come from internal DLL names, portable names are recommended. A portable name is a proper Guardian name, which can be up to eight-characters long, expressed in lower-case.

There is one more consideration for making simple names portable: If they are lowercase, begin with a letter, contain only letters and digits, and are at most eight characters long, they work as either Guardian or OSS names. This can be important for a DLL that serves both environments.

To change a DLL's internal name, you must relink it.

## At a Glance: Controlling Linker Output When Producing a Loadfile

The one command, `eld`, invokes the linker for all operations; options control all subsequent linker steps.

The following table summarizes the options for creating loadfiles and for naming the files they are stored in.

| **To:** | **Action:** |
|---|---|
| Output a program | This happens by default |
| Output a DLL | Insert the `-dll` option |

| **To:** | **Action:** |
|---|---|
| Specify the name of the file in which to store the result | Assign the file name with a `-o` option |
| Name the resulting DLL differently from the output file | Assign the DLL name with the `-dllname` option and the file name with a `-o` option. |
| Simultaneously give the same name to the output DLL and the file it is stored in | Assign the name with a `-o` option and do not use a `-dllname` option, or vice versa. |

## Specifying Which Inputs Go into a Link

Primary inputs to a link are the individual linkfiles inserted by name in the command stream. Other linkfiles can come from archives that are inserted in the command stream; an archive is a file that contains one or more linkfiles. Input linkfiles are merged in the link process to form the output loadfile. Other inputs to a link are libraries that the output loadfile references directly or indirectly. The following subsections describe each of these inputs in detail.

### Linkfile Inputs

The linker's main purpose is to combine linkfiles to produce a loadfile. You must insert directly in the command stream the name of the linkfiles that are the primary inputs to the link. The names of these linkfile inputs can be unqualified if the linkfile is in the same subvolume or directory from which you invoke the linker or else they must be fully qualified file names; they cannot be part of any option. These names can be inserted anywhere as long as they do not separate another option from its parameters.

The linker always resolves symbol references in linkfiles being linked with symbol definitions those linkfiles themselves provide. In particular, if a loadfile refers to a symbol that it also exports, the linker binds that reference to that loadfile's definition. If the loadfile is localized, which the linker assigns by default, then at load time, the loader will accept this resolution. If the loadfile is not localized, the loader may revise the linker's resolution, as discussed in Ambiguity Example 2 on page 4-9.

#### Selecting Linkfiles from Archives

Like named linkfiles, by inserting their qualified names, you can specify archives for the linker to access. To learn how the linker searches for an archive, see Specifying Where the Linker Can Find Its Inputs on page 2-9. Two options allow you to select which linkfiles are brought into the link from an open archive:

| **Linkfiles to Bring into the Link** | **Option** |
|---|---|
| Bring in all linkfiles in this archive. | `-all` or the synonym `-include_whole` |
| Bring in only those linkfiles that resolve currently unresolved symbols. | `-none` or the synonym `-no_include_whole` |

`-all` and `-none` are toggle options you can insert multiple times in the command stream to set the mode for archives that are subsequently specified in the command stream. You can also insert the same archive more than once in the command stream. At the beginning of the command stream, the default mode is `-none`.

The following example brings into the link only those linkfiles in an archive called `archfile1` that resolve symbols at the time the linker has `archfile1` open, since `-none` is the undeclared mode at the beginning of the command stream.

```
eld linkfile1 linkfile2 archfile1 -dll -o mainout
```

If, instead, you want the linker to bring into the link all the linkfiles in `archfile1`, insert the following:

```
eld linkfile1 linkfile2 -all archfile1 -dll -o mainout
```

In this case, `-all` remains in effect for the rest of the command stream that follows it.

## Availability of Linkfiles from Archives

Regardless of how the linker finds an archive, that archive is opened and remains open, with its linkfiles available for inclusion in your link, only while the linker processes the token naming that archive. When the linker has finished processing that token and has extracted the appropriate linkfiles for inclusion in your link, it closes that archive. Thereafter, that archive’s other linkfiles and their symbol definitions, are no longer available to your link unless you reopen that archive. If subsequently processed files require symbols from this archive, they can be satisfied in any of the following ways:

- Open the archive only after processing all the files that may need these symbols.
- Reopen the archive again later, after processing the other files that may need these symbols.
- Open the archive with `-all` in effect, so that all its linkfiles are incorporated in the link and all their symbols are subsequently available.
- If you know ahead of time that a given symbol must be resolved later, insert the `-u` <`symbol name` > option, where `<symbol name>` names that symbol. Having seen this declaration, the linker incorporates the first linkfile it finds in the archive that resolves <`symbol name`> into the link. `-u` is repeatable, and only one symbol can be listed with each `-u` option.

Here is another way of solving the same issues:

- If a first linkfile is supposed to get a symbol definition from an archive and if you are unsure whether a second linkfile in the link also exports that symbol, then insert the archive after the first linkfile and before the second linkfile.
- Otherwise, insert the names of your archives at the end of your command stream, so that these archives can address all the outstanding symbol references generated by processing the command stream.

While the archive is open (when `-none` is in effect), the linker searches for symbols that are unresolved in the loadfiles seen so far, or specified by the `-u` option, or unresolved in linkfiles selected from the archive. The archive can have indirectly needed linkfiles. The linker finds them regardless of their order. (That is, the linker makes multiple passes over the archive while it has it open, if necessary to resolve symbols introduced by linkfiles in the archive.)

## Library Inputs

In addition to linkfiles and archives, a loadfile being linked can also obtain symbol definitions from existing loadable libraries. You must know which symbols the loadfile you are creating will import from existing libraries, and tell the linker which libraries can resolve those symbols by inserting their file names in the command stream. Using these names, the linker opens the corresponding files and reads their internal names to build the libList in the resulting loadfile. There it lists the libraries in the order that you inserted them; these libraries are called your loadfile's libListed libraries. Later, the loader uses this libList on the execution target to find the libraries that will resolve your loadfile's symbol references and to build your loadfile's loadList.

Thus, when linking your loadfile, the order in which you insert library names into the linker's command stream determines the order that the linker processes them and lists them in your loadfile's libList. The order of library names is unaffected by the mingling in the command stream of other inserted tokens among these names.

Figure 1-4 on page 1-9 shows that when linking the program, the programmer inserted the file names of DLLs A, B, and C, in that sequence. It also shows that:

- A requires D and E.
- Both D and E require G.
- Both B and C require F.

To get these results, when A was linked, its programmer inserted first the names D then E. When D and E were linked, their programmers inserted G. Likewise, when B and C were linked, their programmers inserted F.

Library names can safely be inserted anywhere in the command stream, because their symbols are made available as needed in the link regardless of their inserted position. Also remember, when a TNS/E compiler invokes the linker for you, the compiler automatically ensures that the object files are linked to any required standard run-time libraries.

# Specifying Where the Linker Can Find Its Inputs

The previous section discussed how to tell the linker what items go into a link. This section focuses on how to tell the linker where to look for these items in the file system of the linker platform. The linker can find files in several different ways:

- If the linker is given a qualified file name inserted directly in the command stream, it opens the file in the normal way for the linker host platform.
- If the linker is given an unqualified file name inserted directly in the command stream, it opens a file of that name in the current directory or subvolume.
- If an unqualified file name is used in a -lib option, the linker searches for that file following prescribed search paths, as discussed in Where The Linker Searches for Libraries and Archives on page 2-11.
- If a partially or fully qualified file name is used in a -lib option, the linker does not search; it applies the host-system defaults to a partially qualified file name, and attempts to open the file.
- The linker can look for the file among the public libraries.

## Files the Linker Opens Normally

The linker accesses files inserted directly in the command stream by making a single attempt to open them, as described above. The linker recognizes and distinguishes among a linkfile, an archive, and a DLL that it opens, and it handles each appropriately.

The following causes the linker to attempt to open `linkfile1`, `linkile2`, and `archfile1` normally in order to access their contents.

```
eld linkfile1 linkfile2 archfile1 -dll -o mainout
```

## Libraries the Linker Searches For and Opens

To cause the linker to search for a file to bring into a link, insert the `-lib <filename>` option. The file can be either an archive, which supplies linkfiles to incorporate into the link, or a loadable library, which will be listed in the output loadfile's libList. Only one filename can follow a `-lib` option, so to declare multiple file names, use multiple `-lib` option declarations. `-lib` is a repeatable option. `-l <filename>` is a synonym for `-lib <filename>`, where the `-l` must be lowercase and may, but need not, be separated from its filename parameter by white space.

The linker recognizes file names in a `-lib` (or `-l`) option as either qualified or unqualified, as discussed in File-Name Qualification on page 2-4. It accepts a qualified name in a `-lib` option and does not search for it, but instead, attempts to open the file as a library or archive; thus `-lib qf` is equivalent to a `qf` input as a separate file name, when `qf` is qualified. If the linker cannot open a file with a qualified name it declares an error and terminates the link. If the linker cannot open a file specified in a `-lib` option with an unqualified name it declares an error, unless you have instructed it to allow missing libraries, as described in Allowing Missing Libraries on page 2-12. It is an error if a file opened as a result of `-lib` is not a library or an archive.

On the other hand, when the linker recognizes an unqualified name in a `-lib` option, it searches the file system for a file with that name and opens it. Also, if the name in a `-lib` option is unqualified, the linker might augment the given name with the defaults appropriate to the linker platform and then attempt to open the resulting file. This is described in Augmenting Library Names Automatically in Searches on page 5-2.

A `-lib` option cannot specify any of the primary linkfiles in a link; these must be inserted separately in the command stream as qualified file names or unqualified names if the files are in the current directory.

## Where The Linker Searches for Libraries and Archives

The following example adds a library, whose unqualified name is `libfile1`, to the link.

```
ld linkfile1 linkfile2 archfile1 -lib libfile1 -dll -o mainout
```

If `libfile1` were a qualified name, the linker would open it normally, just as it did for the two linkfiles and the archive. However, since `libfile1` is an unqualified name, the linker searches for it, and the linker must know where in the file system to look for it. This is defined by the linker search path, which specifies the sequence of directories or subvolumes in which the linker searches for directly referenced libraries. This path is specified by the following sequence in the order shown:

1. A directory or subvolume you specify in a `-first_L <pathname >` option in the linker's command stream, where `<pathname >` is a path to a specified directory or subvolume. Only one path or directory can be inserted with each `-first_L` option, so one such option must be inserted for each path or directory needed. The linker searches the specified directories or subvolumes in the sequence that the `-first_L` options are inserted.

2. The public libraries. Because the public libraries cannot store archives, the linker will bypass this step if `-b static` is in effect, as discussed in Making the Linker Accept Only DLLs or Only Archives on page 5-1.

3. The directory or subvolume you specify in a `-libvol <pathname >` option in the linker's command stream, where `<pathname >` is a path to a specified directory or subvolume. `-L`, in upper case, is a synonym for `-libvol`. Only one path or directory can be inserted with each `-libvol` option, so one such option must be inserted for each path or directory needed. The linker searches the specified directories or subvolumes in the sequence that the `-libvol` options are inserted.

4. For OSS, where the value of the environmental value COMP_ROOT prefixes each of the following names:

   - when building a 32-bit or neutral object:
     `/lib:/usr/lib:/usr/local/lib:/G/SYSTEM/ZDLL`

   - when building a 64-bit object:
     `/lib64:/usr/lib64:/usr/local/lib64:/lib:/usr/lib:/usr/local/lib:/G/SYSTEM/ZDLL`

   Finally, unless the `-b static` option is in effect, (see Making the Linker Accept Only DLLs or Only Archives on page 5-1), the linker searches in `/G/SYSTEM/ZDLL.`

For Guardian, unless the `-b static` option is in effect, the linker searches in $SYSTEM.ZDLL.

For Win 32, the linker does not search in any standard places.

You can have the linker skip steps 2 and 4 above by inserting the `-nostdlib` option or its synonym `-no_stdlib.`In this event, the linker will search for libraries in places specified by `-first_L` or `-libvol` options. `-nostdlib` is a one-time option that applies to the linker only; it has no effect on subsequent loader operation.

The following example causes the linker to search for a library listed in a `-lib` option first in a private subvolume, `pvtsvol`, before searching for it in the public libraries.

```
 eld linkfile1 linkfile2 -lib dllfile1 &
-first_L pvtsvol -dll -o mainout
```

You may find it convenient to use a common search path that can work on different linker platforms. The linker design supports this by allowing directory and subvolume names for different platforms to be mixed in one search path definition. In this case, the linker on one platform will find in its search path some directory or subvolume names that are syntactically incorrect, because they are for the other platform. The linker ignores these and searches only the ones it deems correct.

### Allowing Missing Libraries

The linker terminates in error when it cannot find a file it searches for, unless it has been instructed to allow missing libraries, by use of the `-allow_missing_libs` option.

If the linker is directed to search for archives as well as libraries, then operating under the `-allow_missing_libs` option, the linker treats any missing file as a library; thus it can miss an archive as well. `-allow_missing_libs` is a one-time option.

If `-allow_missing_libs` is specified, and a specified library is not found, the name from the `-lib` option is placed into the libList of the output loadfile.

## Specifying a User Library for a Program

When linking a PIC Program, you can make an existing DLL the program’s user library by inserting the `-set libname <filename` > option, where `<filename` > is the name of that DLL. `-libname` is a synonym for `-set libname.<filename` > must be a fully qualified Guardian file name on the execution target. `-set libname` is only valid when linking a program.

## At a Glance: Files the Linker Brings into a Link

| **To incorporate in a link:** | **Action:** |
|---|---|
| A linkfile, archive, or library to open normally | Insert its qualified file name directly in the command stream |
| An archive or library to search for and open | Insert its unqualified file name in a `-lib` option |

## Compile-Time Control of Export and Import

The TNS/E C and C++ compilers provide facilities to specify export and import controls at compile time. The syntax involves a modifier, export$ or import$, which can be placed on the declaration of an identifier. As their names imply, they cause the compiler to mark the associated definition as exported or imported, respectively. When applied to the declaration or definition of an individual function or variable, it affects that one item. When applied to the declaration of a class, it affects all the symbol definitions within that class, including auxiliary compiler-generated definitions such as type identification variables. This facility has several important advantages:

- The identifiers can be marked in the source, avoiding the need to place details into the linker's command stream.
- The programmer need not know or enter the "mangled" form of C++ function names to export them.
- When marking a whole class, the programmer need not know its compiler-generated auxiliary identifiers.
- Not only can identifiers be marked exported in some linkfiles and loadfiles, they can be marked imported in others. Judicious use of this ability can avoid unwanted multiple definitions, reducing wasted address space and potential ambiguity. (Linker commands can offer symbols for export, but cannot force them to be imported.)

The export$ and import$ modifiers are not intended to be used explicitly, but instead to occur within the expansions of defines. For example, consider the following header file fragment:

```
#ifndef export_foo
#  define export_foo import$
#endif


export_foo class foo {...}...
```

When this file is included in routine compilations, the various symbols associated with this class definition will be marked as undefined, so they must be defined in another linkfile or loadfile. However, the compilation of the module that implements class foo can contain

```
#define export_foo export$
```

ahead of the #include directive for this header file. As a result, this compilation will define all the symbols associated with class foo, and mark them offered for export.

## Your Loadfile's Exported Symbols

By default, the linker causes your loadfile to offer for export those symbols for which your compiler sets the `xport` bit in the external symbol table. To know which these are, you must know which symbols your compiler designates this way. To ensure that your loadfile offers for export all needed symbols, insert the `-export_all` option. This causes the loadfile to offer all symbols except:

- Those used internally by the compiler and linker
- Those used only in starting up or shutting down the loadfile

Controlling Which Symbols Your Loadfile Exports on page 5-5 discusses how to override the automatic exporting of symbols your compiler designates as exportable.

## Re-Exported libraries

A DLL can also make available symbols exported by any library in its libList; that is, the given DLL can re-export the other library. Thus, when a given DLL, call it A, re-exports a library, B, any loadfile that has A in its libList can also use all the symbols offered for export by B. When linking a given DLL, the programmer must designate which libraries, if any, the given one is to re-export.

The fact that DLL A re-exports library B is only meaningful when a loadfile that has A in its libList is localized, because if that loadfile is not localized, it has access to B's symbols anyway.

As an example, suppose that in Figure 1-5 on page 1-14, Program is localized. Then it can only import symbols from User Library, A, B, and C, unless one of these four re-exports libraries in its libList. Either B or C could re-export F, in which case Program could use F's symbols. And if A re-exported either D or E, Program could use the re-exported library's symbols. In the latter case, if the re-exported library (either D or E) also re-exported G, then Program could use G's symbols, as well. This is because re-exportation is transitive, in that if Library X re-exports DLL Y and Y re-exports library Z, then X re-exports Z. User Library could also re-export H for A's use.

On the other hand, in Figure 1-5, if Program is globalized, then it can import symbols from any libraries shown in that figure, regardless of which libraries are re-exported.

## How to Make Your Loadfile Re-Export Symbols of Other DLLs

When you link your DLL, you can make it re-export the exported symbols of any library in its libList. To do this you insert the `-reexport` option, after which you insert directly or in a `-lib` option the libraries which are to be re-exported. `-reexport` and `-no_reexport` are a toggle-option pair telling the linker that all libraries inserted after the `-reexport` option are re-exported until the linker encounters a `-no_reexport` option. `-no_reexport` is the undeclared mode at the start of the command stream.

In the following example, `-reexport` makes available the symbols in `dllfile1` to any localized loadfile that has `mainout` in its libList.

```
eld linkfile1 linkfile2 -reexport -lib dllfile1 &
-first_L pvtsvol... -dll -o mainout
```

## Some Examples Using Re-Exportation

### Splitting a DLL into Two DLLs

One use of re-exportation is to allow you to split a DLL into multiple DLLs without having to relink the clients of the original DLL. You might make such a split because it is expedient to assign responsibility for parts of the original DLL to different individuals or groups. To split a DLL in two, you give one of the new DLLs the name of the original and have it re-export the other.

**Figure 2-1. DLL A Imports DLL D’s Symbols.**

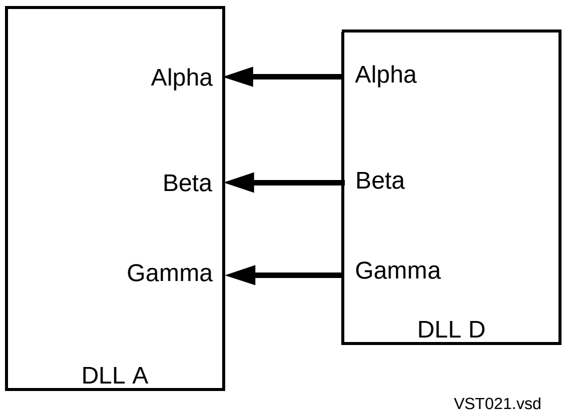


Figure 2-1 on page 2-15 illustrates DLL D exporting symbols Alpha, Beta, and Gamma, while its client, DLL A, imports those symbols. Suppose that you want to replace D with two loadfiles, one which provides the Gamma definition and the other which provides the Alpha and Beta definitions, and you do not want to affect D’s client loadfiles. You can do this as follows.

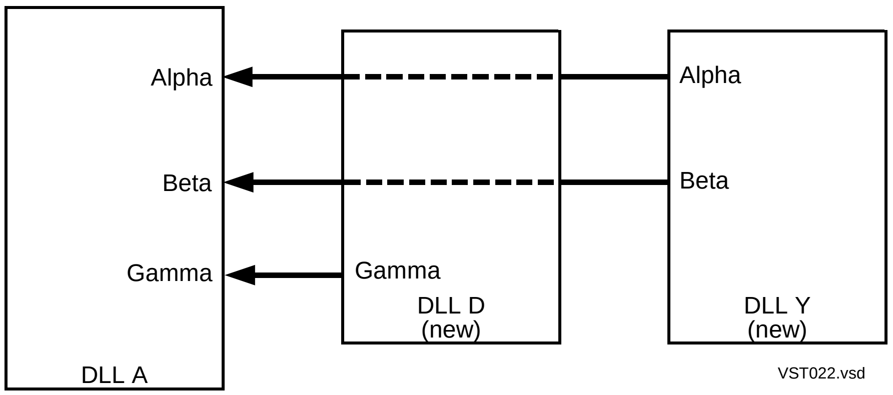
1. Split the source for D to create the two new loadfiles: one that exports Alpha and Beta, the other that exports Gamma.

2. Recompile and relink the new sources, and give one resulting loadfile (say the one that exports Alpha and Beta) a new name, Y. Give the other loadfile the old name, D. See Naming DLLs on page 2-3.

3. Set up D to re-export Y.

The result is shown in Figure 2-2 on page 2-16, where the dashed lines through D indicate re-exportation. All the clients of D, like A, still get symbols Alpha, Beta, and Gamma through D, so they need not be changed.

**Figure 2-2. Splitting a DLL Into Two Parts**



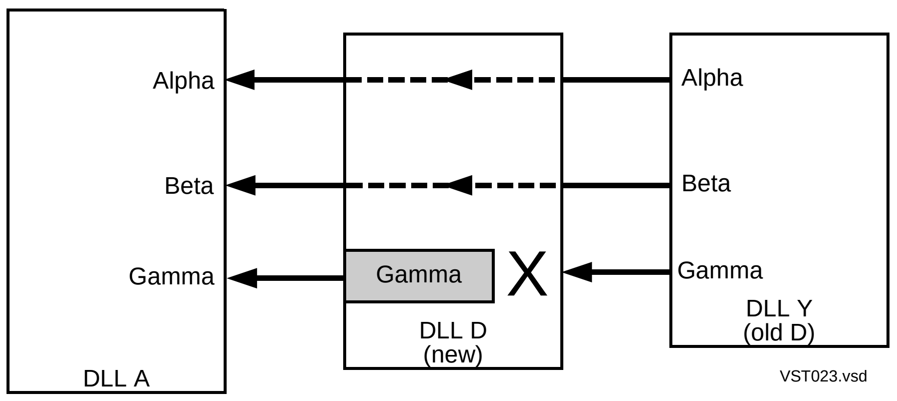
## Replacing an Existing Symbol Definition

In a working DLL you can make a new version of an existing symbol having new function without recompiling that DLL, so that all client loadfiles that use this symbol will invoke the new function without being relinked and reloaded. Starting from the situation in Figure 2-1 on page 2-15, Figure 2-3 on page 2-17 illustrates how a new DLL can combine with an existing DLL to replace the old function. The steps to accomplish this are:

1. Relink DLL D and rename it DLL Y.

2. In a new DLL D, construct the new procedure to provide the new function and name its entry point Gamma, the same name as the replaced function in the old DLL D. Put Y in D’s libList and designate Y re-exported, so users can still access the old symbols Alpha and Beta from D.

**Figure 2-3. the New Symbol, Gamma, With New Function (Shaded), Replaces the Old Gamma.**



Because the search order for the linker and loader comes to D before Y (See Finding Symbol Definitions on page 4-1), D’s Gamma masks Y’s. So without any change to the users, all of those that formerly used old D’s Gamma will now get new D’s Gamma.

# Things to Consider about the Loader

Many of the loader’s operations are automatic and driven by adequate defaults. This subsection discusses a few of which you must be aware. For more precise control of the loader, see Load-Time Operation on page 5-13.

## The Link-Time-Defined Search Path of the Loader

The linker can run on an auxiliary system while the loader must run on the execution target, so their search paths might necessarily be different. For example, the directory and subvolume names are unlikely to be the same on the auxiliary and execution-target systems. This subsection discusses how, when executing the link, you can direct the load-time search to appropriate paths on the target system.

While the loader’s search path is similar to the linker’s, the search-path information the linker gets from `-first_L`, `-libvol`, or `-L` options is not passed on to the loader. Instead, the loader’s search path is controlled at link time by `-RLD_first_L` or `-RLD_L` options. These are repeatable options.

The loader’s search path can also be modified at load time, as discussed in Controlling the Loader’s Search Path at Load Time on page 5-13; however, when not augmented at load time, the loader’s search path is:

1. The directories or subvolumes specified in a `-RLD_first_L <parameter>` option in the linker’s command stream, where `<parameter>` is a path or paths to a specified directory or subvolume. The `-RLD_first_L` option is not required, but when used, is repeatable with different path strings.

2. The public libraries.
3. The directory or subvolume specified in a `-RLD_L <parameter>` option in the linker's command stream, where `<parameter>` is a path or paths to a specified directory or subvolume. The `-RLD_L` option is not required but when used, is repeatable with different path strings.
4. Following default locations:
   - 32-bit process:

     For OSS: `/lib, /usr/lib, /usr/local/lib,` and `/G/SYSTEM/ZDLL` in the order of the paths specified here.

     For Guardian: `$SYSTEM.ZDLL`
   - 64-bit process:

     For OSS: `/lib64, /usr/lib64, /usr/local/lib64, /lib, /usr/lib, /usr/local/lib, /G/SYSTEM/YDLL,` and `/G/SYSTEM/ZDLL` in the order of the paths specified here.

Unlike the previously discussed options that guide the linker's search, `-RLD_first_L` and `-RLD_L,` which guide the loader's search, allow you to specify multiple colon-separated paths as a single argument for each option. You can also insert multiple such options, and the linker concatenates their arguments in the order you insert them to present the loader with a single list of colon-separated paths. The loader follows this list in the order listed. The path names themselves cannot have colons embedded within them.

The loader ignores invalid path names, so you can mix OSS paths and Guardian subvolumes to create a loader search path that will work on either.

As mentioned above, the loader offers the ability to specify paths at load time, but for security reasons you may wish to ensure that the loader accepts no load-time directives and uses only the paths in 1 through 4 above. To disable such run-time directives, insert the linker option `-limit_runtime_paths`, which is a one-time option.

The following command-stream fragment might appear when linking on Win32. It directs the linker to look for `dllfile1` in a directory called `mydir` before it searches the public libraries.

```
eld... -L mydir -lib dllfile1 ... -o mainout
```

However, this example passes no search directions to the loader running on the execution target; so the loader will search for `dllfile1` in only the target's default places. If `dllfile1` (or its equivalent) is stored on the execution target in a private subvolume named `pvtsvol,` you can direct the loader to search there before looking anywhere but in the public libraries by altering the previous example:

```
eld...  -L mydir -lib dllfile1 ... -o mainout -RLD_L pvtsvol
```

The `-RLD_first_L` option is rarely necessary, because the public libraries have unique names that should not overlap those supplied by the user or other agencies. It is more efficient to have the public libraries first on the path search list at load time, because the set of public libraries can be searched very quickly using a table in memory.

## Unresolved Symbols at Load Time

The loader always searches for every symbol definition that your loadfile must import. Until it finds a library that offers a symbol definition that your loadfile needs, it considers that symbol unresolved.

The loader's search path can be as described under The Link-Time-Defined Search Path of the Loader on page 2-17, but it might be more extensive, as described in Finding Symbol Definitions on page 4-1. After looking in all the files specified for this search and in the implicit system and millicode libraries, if the loader cannot resolve a symbol reference, it will likely deem this an error; see the definition of `set RLD_unresolved` under How to Set Run-Time Attributes of Your Loadfile on page 5-6.

Also, you may want the linker to help you to find unresolved symbols prior to load time. For this, see Making the Linker Look for Unresolved Symbols on page 5-3.

## Simultaneously Using Different Versions of a DLL

By controlling the loader's search path, you can allow two different versions of the same DLL with the same name to be loaded and run in two different processes simultaneously. This can be useful when testing a new version of a DLL with existing application code that you do not want to modify. One way to substitute a test DLL for a production DLL is to link the program and DLLs to allow load-time search-path specifications, as described in Controlling the Loader's Search Path at Load Time on page 5-13.

# Default Setting and Checking of File Attributes

The linker sets certain attributes of the loadfile being linked and performs certain consistency checks. This subsection discusses those that you must know about. How to set these and others to non-default values is covered in How to Set Run-Time Attributes of Your Loadfile on page 5-6.

## Floating-Point Type

### Setting the Floating-Point Type of a Loadfile Being Linked

The floating-point type attribute of the loadfile being linked is determined in one of two ways.

1. If the `-set floattype <value>` option is inserted, the linker sets the floating-point type of the output loadfile to `<value>`, where `<value>` can be `ieee`, `tandem`, or `neutral`.
2. If the `-set floattype <value>` option is not inserted, then if the input linkfiles do not contain a mixture of both `ieee` and `tandem` floating-point types, the linker sets the floating-point type of the output loadfile to the one that is represented. Input linkfiles of `neutral` floating-point type are ignored in this. However, if all input linkfiles are `neutral`, then the linker sets the output loadfile to `neutral`.

### Checking the Floating-Point Types of Linkfiles Being Linked

The linker checks for consistency of floating-point types among all input linkfiles, as follows.

1. If the floating point type of a loadfile being linked is set by inserting the `-set floattype <value>` option, as in (1) of Setting the Floating-Point Type of a Loadfile Being Linked on page 2-19, then if any of the linkfiles being linked has a floating-point type that differs from `<value>`, the linker issues a warning message.
2. If the `-set floattype <value>` option is not inserted, as in (2) of Setting the Floating-Point Type of a Loadfile Being Linked on page 2-19, then if the input linkfiles contain a mixture of both `ieee` and `tandem` floating-point types, the linker terminates in error.

### Checking the Floating-Point Types of Liblisted Libraries

When linking a PIC program, by default, the linker checks for floating-point type consistency among the libraries from which that program imports symbols. If either the program’s floating-point type is `neutral` or the floating-point type of any such library differs from floating-point type set for the program, the linker issues a warning message.

When linking a DLL, the linker does not check for floating-point type consistency among loadfiles from which it imports symbol definitions.

## C++ Dialect

Each compiler identifies its language in the generated linkfiles. Three different versions of the C++ language exist on HP NonStop systems; these dialects are called version1, version2 and version3. They differ in language constructs and in their run-time support libraries; they have slightly different "mangling" algorithms for function names. Version1, the oldest, is obsolete and is not supported for PIC files (on TNS/R or TNS/E). Version3 was new with G06.20; it complies with the ANSI standard.

For any language other than C++, the C++ dialect is defined to be neutral. The linker and loader ensure that only one non-neutral C++ dialect appears in a loadfile or a process, respectively.

## Checking the C++ Dialect of Linkfiles That Go into a Link

The linker checks for consistency of C++ dialect among all input linkfiles. If any linkfiles from C++ compilations differ in C++ dialect, the linker terminates in error.

## Setting the C++ Dialect of a Loadfile Being Linked

The C++ dialect of the loadfile being linked is determined in one of two ways.

1. If the `-set cppdialect cppneutral` option is specified, the linker sets the output loadfile C++ dialect to neutral. See Neutralizing the C++ Dialect of a Loadfile on page 2-21.
2. Otherwise, if all C++ compilation units among the input linkfiles have the same dialect, the linker sets the output loadfile's cppdialect attribute to that value.

## Checking the C++ Dialect of Loadfiles in a Process

The loader checks the consistency of C++ dialect in all the native loadfiles in the process. For PIC processes, there are two assertions:

- No loadfile has cppdialect = version1. Violation results in process creation error 78 (operation not supported) with error-detail 6 (C++ version1 is not supported).
- All loadfiles that have non-neutral cppdialect have the same value, version2 or version3. Violation results in process creation error 77 (unable to load object file) with error-detail 8 (mixed C++ dialect versions are not allowed).

The checking applies to all loadfiles: the program, DLLs, and public DLLs. It applies to libraries dynamically loaded by `dlopen()`as well as those loaded at process creation time. Violation of either assertion generates a unique process-creation error, detail (78,6 and 77,8, respectively; see the *Guardian Procedure Errors and Messages Manual*).

## Neutralizing the C++ Dialect of a Loadfile

The linker can overwrite the cppdialect attribute of an existing PIC loadfile; the command is:

```
eld -change cppdialect cppneutral filename
```

This action must be taken with care. Like the corresponding -set option at link time, it causes the presence of any C++ code in this loadfile to be suppressed. Neutralization can be useful for a library that happens to contain code compiled by C++ but does not have any dependencies on C++ run-time libraries, and does not import or export any C++ objects or compiler-generated identifiers. The neutralized library can load into the same process as other loadfiles with a different version of C++. But if applied improperly, this assertion of neutrality can lead to errors ranging from fairly obvious (such as unresolved symbols) to rather subtle (such as differing semantics in different versions of same-named support-library functions).

### SQL/MX Restriction

Public DLLs that support SQL/MX have cppdialect = version2. Therefore, SQL/MX clients cannot yet use C++ version3.

## Execution-Target System Type

The linker sets a system-type attribute in the output loadfile to the following default values.

- If the linker is hosted on a PC, the default is `OSS`.
- If the linker is hosted on `OSS`, the default is `Guardian` if the loadfile is being created in a Guardian subvolume and is `OSS` if not.
- If the linker is hosted on Guardian, the default is `Guardian.`

How to Set Run-Time Attributes of Your Loadfile on page 5-6 tells how set the system type to a non-default value.

# At a Glance: Linker Mandatory Inputs, and Defaults

## Normal Linker Inputs

To avoid a linker error when linking your loadfile, you must specify the following items in the linker command stream:

1. The linkfiles to be merged to form your loadfile
2. The archives that contain linkfiles that resolve symbols your loadfile needs.
3. The libraries that resolve symbols your loadfile needs.

Also, you may run a link in a situation where you know certain needed libraries are missing, and you may not want the linker to issue the consequent diagnostics. In that case, insert the `-allow_missing_libs` option.

## Linker Default Operation

When executing the options specified in this section, the linker automatically follows its designed default modes. These are listed below, along with references, in square brackets, to subsections that tell how to manually affect these behaviors.

1. By default, the linker builds a loadfile.
2. The linker builds a loadfile that is, by default, a program. [Choosing to Create a Program or a DLL on page 2-3]
3. The linker builds a loadfile that is, by default, localized. [Import Controls and SearchLists on page 4-5]

4. By default, the linker uses only the linkfiles from an archive that resolve currently unresolved symbols. [Selecting Linkfiles from Archives on page 2-7]
5. By default, the linker exports those symbols for which your compiler sets the `xport` bit in the external symbol table. [Controlling Which Symbols Your Loadfile Exports on page 5-5]
6. The linker builds a loadfile that, by default, re-exports no other DLLs. [How to Make Your Loadfile Re-Export Symbols of Other DLLs on page 2-15]
7. If you specify an output file for the DLL you are linking but no DLL name, the linker gives the resulting DLL the same name as the file. [Naming DLLs on page 2-3]
8. If you specify a DLL name for the DLL you are linking but no output file, the linker gives the output file the same name as the DLL. [Naming DLLs on page 2-3]
9. If you specify neither a DLL name nor an output file for the DLL you are linking, the linker gives them both the same name, as defined in Naming DLLs on page 2-3.
10. By default, the linker sets the system type of your loadfile as defined in Execution-Target System Type on page 2-22. [How to Set Run-Time Attributes of Your Loadfile on page 5-6.]

## Linker and Loader Errors

Linker errors described in this document cause the link to terminate with all the files involved in the link left as they were before the link started. Fatal loader errors cause the load operation to abort, but have no effect on other system operation.

For a complete list of `eld` error messages, with cause/effect/recovery information see the *eld Manual.*

# 3 Dynamic Use of DLLs

An important attribute of the DLL facility is that a running PIC program or DLL can load and open a previously not-loaded DLL and gain access to the symbols it offers. DLLs invoked this way are called dynamically loaded DLLs.

One advantage of dynamically loading a DLL is that its name need not be known when the program is constructed; instead, you can add this new DLL to an existing application without even restarting the application. Also, you do not need to load infrequently used DLLs when the application is loaded. Instead, you can load and use these DLLs when needed and unload them when they are no longer required. They can be reloaded whenever necessary.

Dynamic libraries make it possible to update facilities in a running application. If specific business functions are implemented in a dynamically loaded DLL, a program can unload that DLL and load an updated version that supports the same interfaces with revised algorithms, tables, etc.

This section discusses how to dynamically load and unload a DLL from your running process and how to link your loadfile with a dynamically loaded DLL.

`rld` is the facility that loads a program and its requisite libraries. It will load a preset loadfile without rebinding ordinary symbols if the loadfile bindings are correct. This is called FastLoad. A FastLoadable loadfile is one whose bindings have been preset by `eld`, or automatically updated by `rld` and the NonStop operating system.

If `rld` has to rebind a loadfile and the loadfile import control is localized, it will update the preset bindings in the loadfile with the cooperation of the NonStop operating system. This is called automatic update. `rld` and the operating system only automatically update loadfiles at process creation time. If a loadfile is loaded via a call to `dlopen()`, the loadfile is not automatically updated.

Reference information about dynamic linking may be found in the *rld Manual*.

For the C/C++ languages, declarations for the runtime dynamic linking functions and the `dlopen()` mode options are provided in a header file called `dlfcnh` on the Guardian platform and `dlfcn.h` on the OSS/Unix/PC platforms. For the pTAL language, a header file called `hdlfcn` is provided on the Guardian platform containing the same declarations.

The rest of this section details the loading facilities based on C/C++ language functions.

## Dynamic Loading Functions

Dynamic loading and linking use five associated C-language functions, `dlopen()`, `dlsym()`, `dlclose()`, `dlerror()` and `dlresultcode()`. These functions compose the dynamic library function calls described in this section, and they are declared in a header file called `dlfcn.h` and defined in the loader's library. Therefore, any C or C++ source file that uses these functions must contain the following:

```
#include <dlfcn.h>
```

The same functions have pTAL external declarations in a file named `hldfcn`. Both header files also define parameter types and constants for using with these functions.

These functions are implemented in the public library ZRLDDLL. This DLL does not use any explicit public libraries. This library is not named in LIBCOBEY or `libc.obey` (if one exists), and is not supplied automatically by the compiler driver when it runs the linker for you. Therefore, you must name this library explicitly when building a loadfile that calls any of these functions. If you run `eld` manually, you can add a -lib zrlddll option in the command stream. If you let the compiler run the linker, you can provide the option through the compiler command line; see Running the Linker Through the Compiler on page 5-10.

The following summarizes the dynamic library function calls.

**`dlopen()`** loads and opens a specified DLL and the libraries in its loadList if they are not loaded, and this function returns a handle for the named DLL.

**`dlsym()`** returns an address of a named symbol exported by a loadfile associated with a dlopen handle. The calling process can assign that address to an appropriate data or function pointer, which becomes a reference to that symbol.

**`dlclose()`** invalidates a dlopen handle and unloads any dynamically loaded libraries not required by some other handle.

**`dlerror()`** provides a textual error message that describes any error arising from an immediately preceding call to `dlopen(), dlsym(), or dlclose()`.

**`dlresultcode()`** provides an enumerated result code for the last call to `dlopen(), dlsym(), or dlclose()`.

## Opening a DLL from a Running Loadfile (dlopen)

A call to dynamically load a DLL must come from a loadfile that is already loaded. The operation also loads any libraries in that DLL's loadList that are not already loaded.

To open and load a DLL, a running process invokes the following function call.

```
void *dlhandle dlopen (const *object_pathname, int mode)
```

`dlopen()` invokes `rld` to load the DLL and the libraries in that DLL's loadList and to make them available to the calling process. The `dlopen()` function also returns to the calling process a handle for the named DLL, even when that DLL is already loaded. This handle is used by subsequent dynamic library calls.

The parameters of `dlopen()` are:

**`*object_pathname`** is either zero (NULL) or a pointer to the null-terminated file name (string) of the DLL to open and load, which is called herein the dlopen target.

**`mode`** is an enumeration of options that control symbol resolution and loading, as described in dlopen's Mode Parameter Values below.

When the object_pathname parameter is zero, the loader does not load anything, but returns a handle that enables `dlsym()` to search for symbols in the program and libraries loaded with it. Furthermore, any running loadfile object_pathname can name any currently loaded DLL as a dlopen target and get a handle to access the exported symbols of that DLL and the libraries in its loadList.

In the following discussion, the loadfiles loaded prior to a `dlopen()` call are called the prior operating load set. The prior operating load set includes the main program and its initially loaded DLLs as well as any DLLs that were previously loaded dynamically. The DLL named in a `dlopen()` is called the targeted DLL. That DLL and the libraries loaded with it compose the added load set. The prior operating load set and the added load set compose the new operating load set.

### Resolving the Added Load Set's Imported Symbols

The loader resolves symbols imported by the DLLs in the added load set following the normal import-control rules for each such DLL and treating each as part of the new operating load set. For example, the loader binds symbols imported by an added localized DLL to those offered from its own load set, as described in The SearchList for a Localized Loadfile on page 4-5; whereas, the loader binds symbols imported by an added globalized DLL to those offered by the entire new operating load set. To resolve globalized symbols, the added load set's search list is appended to the end of the old operating load set's globalized search list, which is described in The SearchList for a Globalized Loadfile on page 4-8. `dlopen()` can issue either warnings or errors as a result of unresolved symbols, as described in Unresolved Symbols at Load Time on page 2-19.

The old operating load set cannot have symbol references that must be bound to symbol definitions in the added load set, because these would have caused errors when the old operating load set was loaded. Instead, the old operating load set can have pointers to procedures and data that are initially unused. Then after `dlopen()` loads the added load set, the old operating load set can call dlsym to fill in the pointers, as described in Accessing Symbols (dlsym) on page 3-5.

## dlopen's Mode Parameter Values

`dlopen()` accepts any valid combination of the mode options described below. These values are defined in the header file dlfcn.h. A mode value of zero is invalid. The options are as follows:

`RTLD_NOW`

This option performs all linking on the newly loaded library immediately.

`RTLD_LAZY`

This option is accepted to provide compatibility for UNIX, but is treated the same as RTLD_NOW.

`RTLD_GLOBAL`

This option specifies that the newly loaded DLLs should be added to the cumulative loadList, i.e. new libraries are added to the global set, which consists of the program and libraries loaded initially by `rld`. This is the default.

`RTLD_LOCAL`

This option is accepted to provide compatibility for UNIX, but is treated the same as RTLD_GLOBAL.

`RTLD_NOLOAD`

This option specifies that `dlopen()` should not load the target library, but should provide a unique handle to it, if it is already loaded. This option causes `dlopen()` to return a 0 if the library is not already loaded. Note that any handle returned by `dlopen()` counts as a usage of the target library and until it is closed, the library is not unloaded.

`RTLD_VERBOSE(verbosity_level)`

This option specifies the diagnostic output detail from calls to dynamic linking functions regarding loading of library files, resolution of symbol names and search details for files and symbols. The `verbosity_level` values are the same as `rld`'s and are as follows:

0 - default (as though this specification is absent)
1 - none (no output to hometerm/stderr)
2 - warnings and errors
3 - also show files loaded
4 - also show symbol resolution
5 - same as 3, plus file search details
6 - same as 4, plus file search details
7 - same as 4, plus symbol search details
8 - all the above

`dlopen()` treats the mode parameter as the union of the above values and detects an error if any other value is used. Valid combinations for the mode parameter include both of the following:

- Exactly one from the set of (RTLD_NOW **|** RTLD_LAZY **|** RTLD_NOLOAD)
- No more than one from the set of (RTLD_GLOBAL **|** RTLD_LOCAL)

This means that if RTLD_NOLOAD is specified the user cannot set RTLD_NOW or RTLD_LAZY, however, the user can, optionally, set RTLD_GLOBAL or RTLD_LOCAL.

Each valid combination can be combined with the verbosity level, via RTLD_VERBOSE (verbosity_level) to create the mode parameter.

## Returned Value of dlopen

`dlopen()` returns a handle (value) for the calling process to use in subsequent calls to the dlopen target. This handle is not usable in any other way; in particular, it is not an address.

dlopen (with dlsym) is the only way for DLLs in the prior operating load set to access symbols exported by DLLs in the (dynamically loaded) added load set. Furthermore, any running loadfile can name any currently loaded DLL as a dlopen target and get a handle to access the exported symbols of that DLL and the libraries in its loadList. When the dlopen target is 0, `dlopen()` returns a handle for the main program.

### Error-Returned Value of dlopen

A returned value of 0 indicates an error, which can occur for any of the following reasons:

- The `object_pathname` parameter pointed out of bounds or to a malformed name.
- `dlopen()` could not find or could not open the specified file.
- The specified file was neither a DLL nor a public DLL, or was not valid.
- An error occurred while the loader was loading the specified file or adjusting its symbolic references.
- A target DLL specified with RTLD_NOLOAD mode was not already loaded.
- The mode parameter value is invalid.

When `dlopen()` returns 0, call `dlerror()` for a textual description of the error, or `dlresultcode()` for an encoded error code and error-detail.

# Accessing Symbols (dlsym)

Using the handle returned to the calling process by `dlopen()`, that process can access a named symbol if it is exported by the targeted DLL or any library in its loadList. `dlsym()` is the only way to access symbols in a dynamically loaded library; but it also allows a process to obtain the address of a symbol exported by any currently loaded loadfile for which the process has a dlopen handle or by the libraries in that loadfile's loadList. The following function call invokes `dlsym()`:

```
void *dlsym(dlhandle dlopen_handle, const char *symbol_name)
```

The parameters of dlsym are:

`dlopen_handle`

A handle returned by a previous dlopen call and not invalidated by `dlclose()`.

`*symbol_name`

A pointer to the null-terminated name (string) of a symbol to be accessed in the specified loadfile.

## Returned Values of dlsym

Each `dlsym()` call returns the address of a specified symbol. You can use the returned address by storing it as a pointer of a type appropriate for the function or datum designated by the symbol. `dlsym()` finds this address by searching for the first exported occurrence of the named symbol starting with the loadfile designated by the dlopen handle and followed by the libraries in that loadfile's loadList. The import and re-export controls of the searched loadfiles are ignored. If the handle is for the main program, `dlsym()` starts with the main program and searches the entire operating load set.

Each `dlsym()` call returns the address of only one symbol, so the calling process must issue a separate `dlsym()`call for the address of each symbol it needs.

### Error-Returned Value of dlsym

If `dlsym()` encounters any of the following conditions, it returns the value 0 :

- Invalid `dlopen_handle` including a handle that has been invalidated by `dlclose()`, as discussed in Closing a Running Loadfile's Handle to a DLL (dlclose) on page 3-6.
- `symbol_name` cannot be found
- The value of the symbol is 0 (rare, non-error case). Only an "absolute" symbol exported from an assembler module can have the value 0; symbols exported by higher-level languages are virtual addresses, and zero is never a valid address in native processes.

When `dlsym()` returns 0, call `dlerror()` for a textual description of the error, or `dlresultcode()` for an encoded error code and error-detail. In the rare event that `dlsym()` found the symbol and its value was 0, `dlerror()` and `dlresultcode()` return 0.

## Closing a Running Loadfile's Handle to a DLL (dlclose)

To invalidate a handle obtained by a dlopen call, a running process issues the following call.

```
int dlclose(dlhandle dlopen_handle)
```

dlclose invalidates the handle and makes it unavailable for any other call that uses that handle. If the specified handle is the last outstanding one for the referenced DLL and if that DLL was dynamically loaded (by dlopen), the DLL is also unloaded. The following is the parameter of a dlclose call:

`dlopen_handle` is a handle previously returned by a dlopen call.

Closing the last handle for the main program or for any DLLs that were loaded with the main program does not unload any of these loadfiles. Also, if dlclose causes a loadfile to be unloaded, then any DLL in its load set is also unloaded, unless that DLL also belongs to the load set of the main program or to a DLL that still has an outstanding handle for it. When dlclose unloads DLLs, the new operating load set becomes the old operating load set less the unloaded DLLs.

Referencing code or data in a DLL using a handle that has been invalidated by dlclose produces undefined results.

Issuing a dlopen that specifies a DLL previously unloaded by dlclose reloads that DLL plus the libraries in its libList that are not already loaded, and establishes a new handle to it.

## Returned Values of dlclose

dlclose returns 0 if it successfully invalidated the specified handle.

### Error-Return Values of dlclose

If dlclose cannot invalidate the specified handle, it returns a non-zero number. In that event, call `dlerror()` for a textual description of that error, or `dlresultcode()` for an encoded error code and error-detail.

# Error Reporting For Dynamic Library Calls (dlerror and dlresultcode)

## Error Text : dlerror

To obtain information in textual form about any error that occurred in a dynamic library call (dlopen, dlsym, or dlclose), a  process can immediately invoke the following function.

```
char *dlerror(void)
```

dlerror has no parameters.

The dlerror function returns 0 (a NULL pointer) if:

- The immediately prior call to dlopen, dlclose, or dlsym had no error
- dlerror has already been called since the last call to dlopen, dlclose, or dlsym

- There has never been a call to dlopen, dlclose, or dlsym in this process.

Otherwise, dlerror returns a pointer to a buffer that contains a null-terminated character string containing only displayable characters and no trailing newline character. The string is a read-only value that is overwritten by the occurrence of any subsequent error in a dynamic library call, or by any call to dlopen, dlsym, or dlclose, so to preserve or modify the string, a process should make its own copy of it.

The loader cannot recognize threads that may be used in a multi-threaded application. Therefore, if you create such an application, you must ensure that no thread switch occurs between a dynamic library call and the invocation of dlerror that retrieves information about that call. Since the NonStop operating system supports only user-level threads, this means that during this interval, you must avoid invoking functions that can cause explicit thread switching.

## Error Encoding: dlresultcode

To obtain information in encoded form about any error that occurred in a dynamic library call (dlopen, dlsym, or dlclose), a process can invoke the following function:

```
char dlresultcode(void)
```

dlresultcode has no parameters.

If the previous call of dlopen, dlsym, or dlclose encountered no error, or if none of those functions has been called in this process, dlresultcode returns 0. Otherwise, it returns a nonzero value encoding the information about the previous error. The 32-bit result is subdivided: the upper 16 bits are an error code; the lower 16 bits are an error-detail code.

Most of the errors encountered by `dlopen()` are similar to those encountered when loading a program and its requisite libraries, so they are encoded the same way, as process creation errors. These error and error-detail codes are described in the *Guardian Procedure Errors and Messages Manual* and in the PROCESS_LAUNCH_ description in the *Guardian Procedure Calls Reference Manual*.

Certain errors are unique to the dynamic loading functions. These are reported with error code 100; the error-detail is one of the following:

1 - An invalid handle argument was passed to `dlsym()` or `dlclose()`.

2 - The symbol sought by `dlsym()` was not found.

3 - An unrecognized option was passed to `dlopen()`.

4 - A bounds error occurred on the pathname parameter to `dlopen()`.

5 - 7 An inconsistent state was detected in `dlopen()` or `dlclose()` processing. Perhaps a process flag at the high-address end of the stack is corrupted.

8 - RTLD_NOLOAD was specified as an option to `dlopen()`, but the specified library was not already loaded.

Calling `dlresultcode()` does not reset the error code. `dlerror()` and `dlresultcode()` can be called in either order; neither affects the value returned by the other.

`dlresultcode()` is an HP NonStop operating system addition to the conventional set of dynamic loading functions.

### Thread Considerations

The loader is not aware of threads that may be used in a multi-threaded application. Therefore, if you create such an application, you must ensure that no thread switch occurs between a dynamic library call and the invocation of dlerror or dlresultcode that retrieves information about that call. Because the NonStop operating system supports only user-level threads, this means that during this interval, you must avoid invoking functions that can cause explicit thread switching.

## Using Dynamically Loaded DLLs to Extend an Application

You can take advantage of dynamically loaded DLLs to build your application to incorporate anticipated but as yet undefined functions. There are several ways to accomplish this. For example:

- If the application is interactive or has an interactive control mechanism, the operator can supply the name of a DLL to load.
- In a transaction processing system, individual transaction requests could name a DLL to process the transaction, or include a transaction type code that maps into a table of DLL file names; the program might update that table dynamically at operator command.

In either event, the program can load the required DLL if it is not already present. Or it could unload an existing DLL (of that name or for that transaction type) and load another, thus dynamically updating part of the application code.

Of course, loading the DLL is only the first step; the application must also find and access the necessary symbols. Again, multiple approaches are possible:

- In the simplest situation, all the DLLs that support this application export a small set of functions (and perhaps data) with canonical symbol names. The application uses `dlsym()` to find the addresses of these symbols. Each canonical function has the same prototype, so it can receive and process the same set of parameters. The DLLs differ in what their functions do. A classical example from engineering or scientific applications is a numerical integration program: a DLL supplies a function (subroutine) to compute the mathematical function to be integrated. The program calls that function repeatedly to compute an approximate definite integral over some range.

- More elaborate conventions can have the DLL export a canonical "master" function or data structure that describes the functions or data available in this library. The application uses `dlsym()` to locate the "master" symbol. For example, an exported data structure could contain a count and an array of substructures that contain information about each available function, including a function pointer and an encoding of its purpose, result and parameter types. Of course, these descriptions are limited by the conventions established between application and DLL programmers, but can be as flexible as the programmers make them. Because these structures include the function pointer, the individual functions need not be exported from the DLL; the application need not know their actual names or use dlsym to find them.

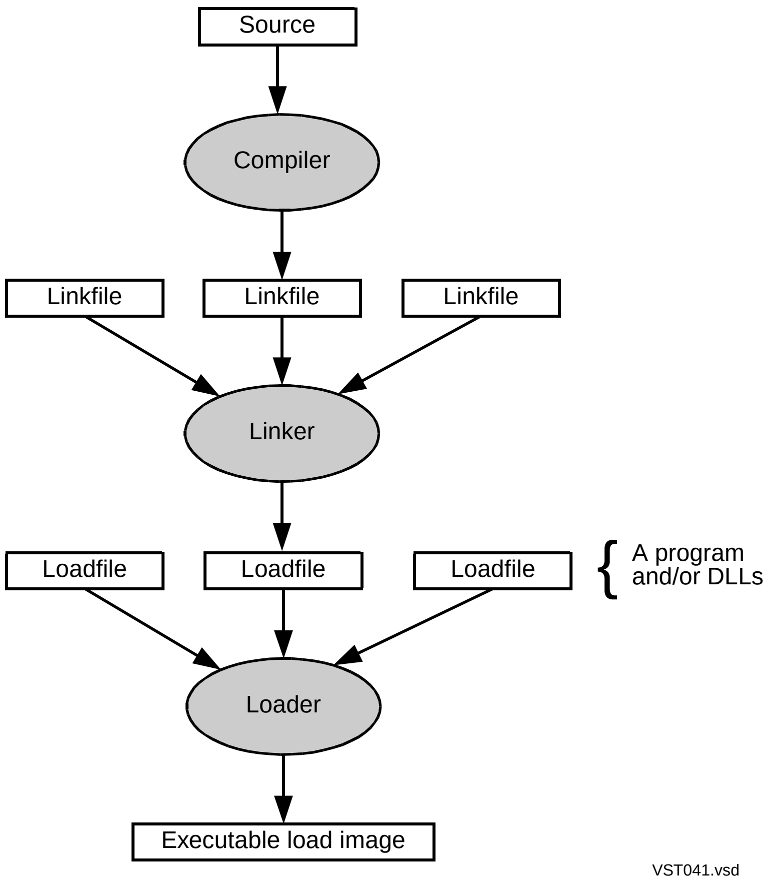
All of this can be done without changing or even stopping and restarting the original application.

# 4 Finding Symbol Definitions

This section describes how the linker and loader resolve symbol references, including cases when multiple definitions are available for the same symbol name.

Figure 4-1 on page 4-1 shows the evolution of code from source to linkfile to loadfile to executable PIC load image. At the linkfile and loadfile stages, these code objects can be brought together so they can exchange services and data. This exchange is defined by symbol references that express a need for functions or data and symbol definitions that provide functions and data.

**Figure 4-1. PIC Code Generation**



## The loadList

The loadList of a file consists of that file, the library files specified on its libList, the files specified on their libLists, and so on. This is called the *breadth-first transitive closure* of libLists. The linker develops the loadList of the loadfile being created. The loader, at process creation time, develops the loadList of the program. For dynamic loading, using `dlopen()`, the relevant loadList is that of the designated library file; that loadList extends the operating load set of the program.

The linker or loader creates the loadList for a loadfile using the following algorithm:

1. [Initialize] To an empty loadList add the name of the designated file, and set a pointer referring to this entry, making it the currently referenced loadfile.

2. If the designated file is the program and it has a user library, add the library name to the loadList.

3. If the designated file has a non-empty libList, add the libList entries in the same order they appear in the libList.

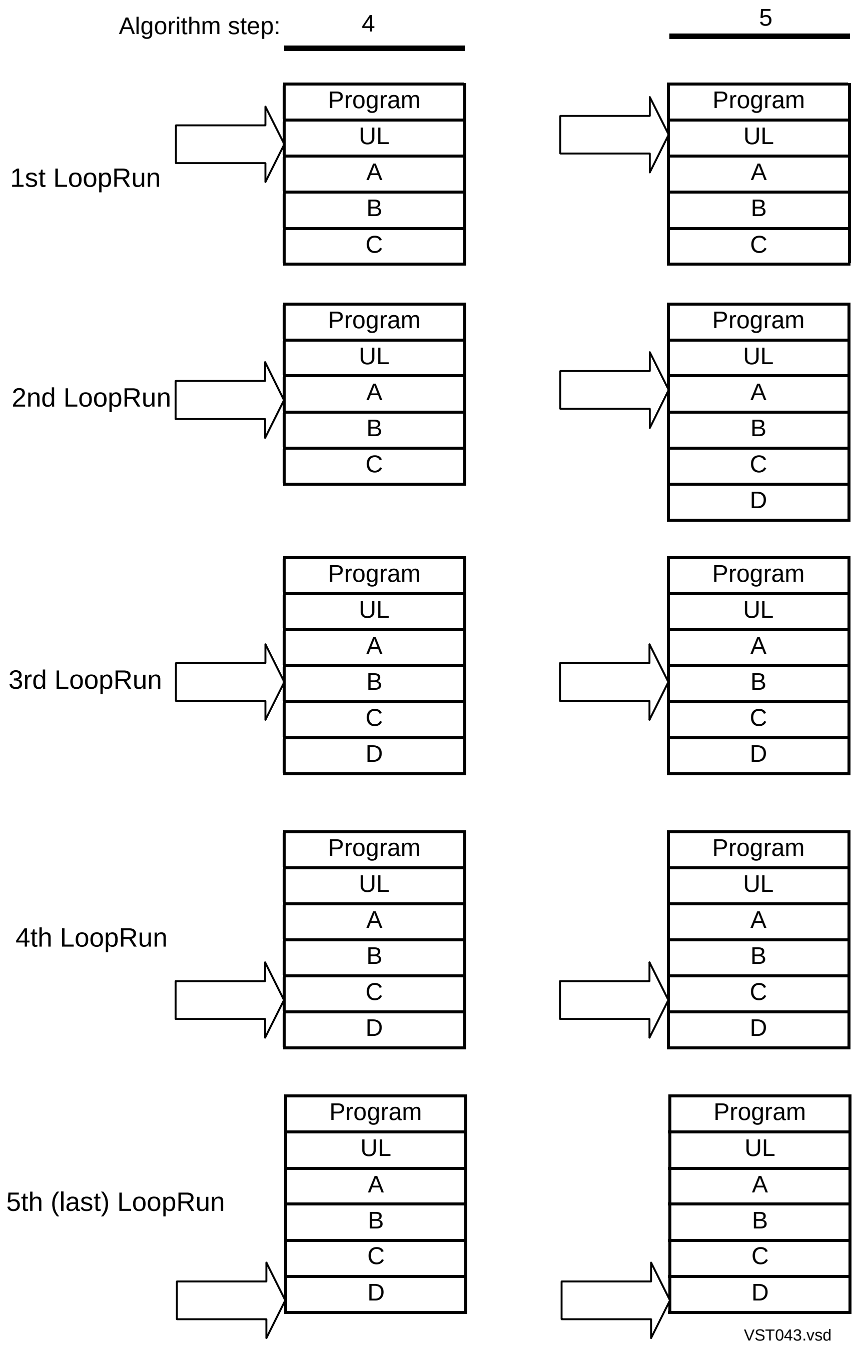
4. [Begin loop] If there is another loadList entry immediately following the referenced loadfile, set the reference pointer to it. If there are no more loadList entries, the loadList is finished.

5. Append to the loadList, in the same order, the libList entries of the referenced loadfile that are not already on the loadList.

6. Return to step 4. [End loop]

For the program and loadfiles of Figure 1-5 on page 1-14, Figure 4-2 on page 4-3 shows the evolution of the globalized loadList and the pointer status for each loop run in the foregoing algorithm at the end of steps 4 and 5. That figure shows only the first few runs of the loop and the last. The resulting loadList, is shown at the bottom of the figure.

**Figure 4-2. Development of the Globalized SearchList for the Program and Libraries based on Figure 1-5 on page 1-14**



# Global Scope, Import and Export

The concepts of *global* and *local* apply intuitively to different scopes in a programming environment. These concepts are subtle, and apply differently to individual compiler source streams, to linkfiles representing compilation units, and to loadfiles (programs and libraries). Furthermore, the application of these concepts differs from one source language to another.

In the C language, items declared within a function are local to that function and are undefined (invisible) elsewhere. Items declared outside any function have global scope and are visible in any function that does not hide that declaration with a local one of the same name. (In languages with nested functions, such as Pascal, there are multiple levels of nested scopes. In pTAL, a SUBPROC has a scope nested within that of the PROC.)

The C language defines *storage class* (sometimes also called *linkage*); the two storage classes that apply to global functions and data are `extern` and `static`. Global data and functions with `static` linkage are visible to the functions within this compilation unit, but are local to the compilation, and therefore to the linkfile that results from the compilation. Global variables with no specified storage class are implicitly `extern`. The explicit specification of `extern` linkage on an uninitialized variable declares the variable but leaves it undefined; if there are references in this compilation, they must be linked to a definition in another.

(The pTAL language has no explicit storage class. Everything is effectively extern except SUBPROCs, which are static.)

When the linker combines multiple linkfiles to build a loadfile, global extern definitions in one linkfile are available to satisfy references to an undefined symbol in another; these items are considered global in the resulting loadfile. However, static global items within each linkfile become local in the loadfile.

A key part of the linker's job is binding global references in one linkfile to definitions in another. For C compilation units, the linker enforces the language requirement that a given symbol have at most one definition in the loadfile. For some constructs common in header files, the C++ language generates definitions in every compilation that includes the header; the compiler marks these symbols to inform the linker that multiple definitions are acceptable, and the linker is expected to bind all references to one of them. Slightly different rules apply to some pTAL definitions, such as data blocks: the linker will accept multiple definitions, as long as any initialized data is the same in each.

If none of the linkfiles contributing to a loadfile defines a referenced symbol, it remains undefined in the loadfile. In this case, it will need to be bound to a definition in some other loadfile. In such a case, the referencing file is said to import the symbol, and the defining loadfile is said to export it. In a DLL with globalized import control, it is also possible for a symbol to be imported even though that symbol is offered for export; in this case the definition in this DLL is said to be preempted by a definition in a loadfile appearing earlier in the program's loadList.

Only some of the global symbols in the loadfiles can be shared with other loadfiles: those that are offered for export, and those that are undefined (require import). There are several ways that symbols become available for export. By default, the linker offers for export only symbols that the compiler designates as exportable. With the C/C++ compilers, you can specify export or import of individual symbols, or of whole classes.

With the linker, you can export individual symbols, or you can cause all global symbols in your loadfile to be exported, and then you can choose which symbols not to export. See Controlling Which Symbols Your Loadfile Exports on page 5-5.

## Import Controls and SearchLists

Just as you have control over the symbol definitions your loadfile offers for export, you can control the sources from which your loadfile can import symbol definitions. You do this by setting your loadfile's import control, which directly affects the range and sequence of the search that the linker and loader follow in locating needed symbol definitions. The search range defines which loadfiles are searched for symbol definitions; the search sequence determines the order in which these loadfiles are searched.

The search sequence is important, because each symbol reference will be bound to the first definition found whose name matches the reference. This is how the linker and loader avoid the dilemma posed by duplicate symbol definitions among loadfiles in the search range, which would otherwise be ambiguous.

The search range and sequence are encapsulated in your loadfile's searchList, which is created and used by the linker, when your loadfile is linked, then later recreated and used by the loader each time your loadfile is loaded. The searchList names the loadfiles to be searched for symbol definitions in the order they are to be searched.

The setting of your loadfile's import control tells the linker and loader how to build your loadfile's searchList. The allowed settings are localized, globalized, and semi-globalized. You choose the setting for the loadfile you are linking by inserting one of the following options:

`-b localized`

`-b globalized`

`-b semi_globalized` or its synonym, `-b symbolic`

These are one-time options. If you do not insert one of these options, then by default, the linker produces a localized loadfile.

## The SearchList for a Localized Loadfile

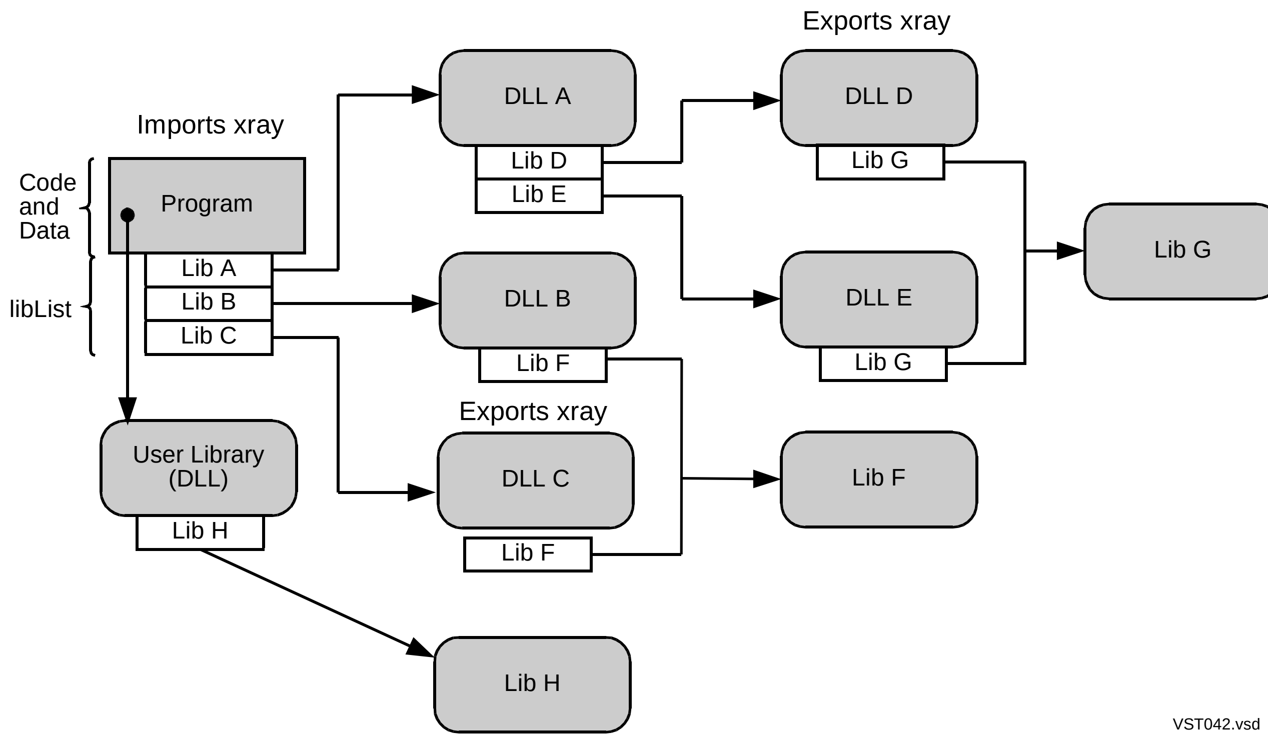
In the localized case with no re-exportation, the only loadfiles available to resolve symbols are those listed in the importing loadfile's libList. Hence, the searchList begins with the loadfile itself followed by names taken from the libList in the order listed.

However, a DLL can also make available symbols exported by any library in its libList by re-exporting that library, so a localized loadfile could import symbols from libraries not in its libList. A localized loadfile's searchList is developed starting with this loadfile itself, adding its libList (as above), appending to this the names of re-exported libraries, then appending the names of the libraries those re-exported libraries re-export, and so forth. This process is defined in the following algorithm, which the linker uses when linking a localized loadfile and the loader uses when loading it. The algorithm is illustrated in the example that comes after it.

1. [Initialize] To an empty searchList add the name of the localized loadfile being linked and set a reference pointer pointing to this entry, making it the currently referenced loadfile.
2. If the referenced loadfile is a program that has a user library, add the user library's name to the searchList.
3. If the referenced loadfile has a non-empty libList, add the libList entries from the loadfile in the same order they appear in the libList.
4. [Begin loop] If there is another searchList entry immediately below the referenced loadfile, set the reference pointer to it and make it the reference loadfile. If there are no more searchList entries, the searchList is finished.
5. Append to the searchList, in the same order, the libList entries of the referenced loadfile that are designated in the libList as re-exported and that are not already on the searchList.
6. Return to step 4.

As an example of how the foregoing algorithm works, consider Figure 4-3 on page 4-7, which shows Program and the libraries in its loadList. Program references the symbol `xray` and DLLs C and D both export it. Assume that Program is a localized loadfile. Program's imported symbols are resolved by searching UL, A, B, and C in that order, where UL stands for User Library. Suppose also, that UL does not re-export H, that A re-exports D but not E, and that neither B nor C re-exports F. Figure 4-4 on page 4-9 shows the development of the localized searchList for Program following the algorithm listed above. For each run through the loop, the rows in that figure show the situations after the execution of algorithm steps 4 and 5.

**Figure 4-3. The Loadfiles of Figure 1-5, Now Showing the Use and Availability of the Global Symbol `xray`**



The final searchList is shown at the bottom of Figure 4-4. This shows how to develop a localized searchList for Program when A re-exports D and no other DLLs re-export. The arrows point to the referenced libraries in the algorithm after the indicated loop run.

Program will use C’s definition of xray, since C precedes D in the searchList. But, assuming D is localized, D will use its own definition of xray.

If we change the conditions so that B re-exports F, then Program’s searchList becomes Program, UL, A, B, C, D, F. If now D re-exports G, then Program’s searchList becomes Program, UL, A, B, C, D, F, G. Then if User Library re-exports H, the searchList becomes Program, UL, A, B, C, H, D, F, G. If E re-exports G, this has no effect on the searchList, because E itself is not re-exported; hence, E does not appear on the searchList.

Localized import gives the programmer tight control over how imported symbols are resolved, which is consistent with previous conventions used on HP NonStop systems. Localized import facilitates load-time optimizations, because the linker's search list is the same as the loader's. Finally, localized import is necessary for security where PIC programs and DLLs support license and privilege.

## The SearchList for a Globalized Loadfile

For a loadfile with globalized import, the linker's searchList is just its loadList; the loader's searchList is the loadList of the program. (For dynamic loads, that is the operating load set, also called the cumulative loadList of the program.)

The resulting loadList, shown at the bottom of Figure 4-2, is thus the searchList; it gives the sequence followed by the loader to look for exported symbol definitions that satisfy the import needs of every globalized loadfile loaded with the program.

Because the linker sees only the file it is building and its loadList, the linker searchList of a DLL differs from the loader searchList, which includes the program and other libraries it requires. Therefore, any link-time bindings may require revision at load time.

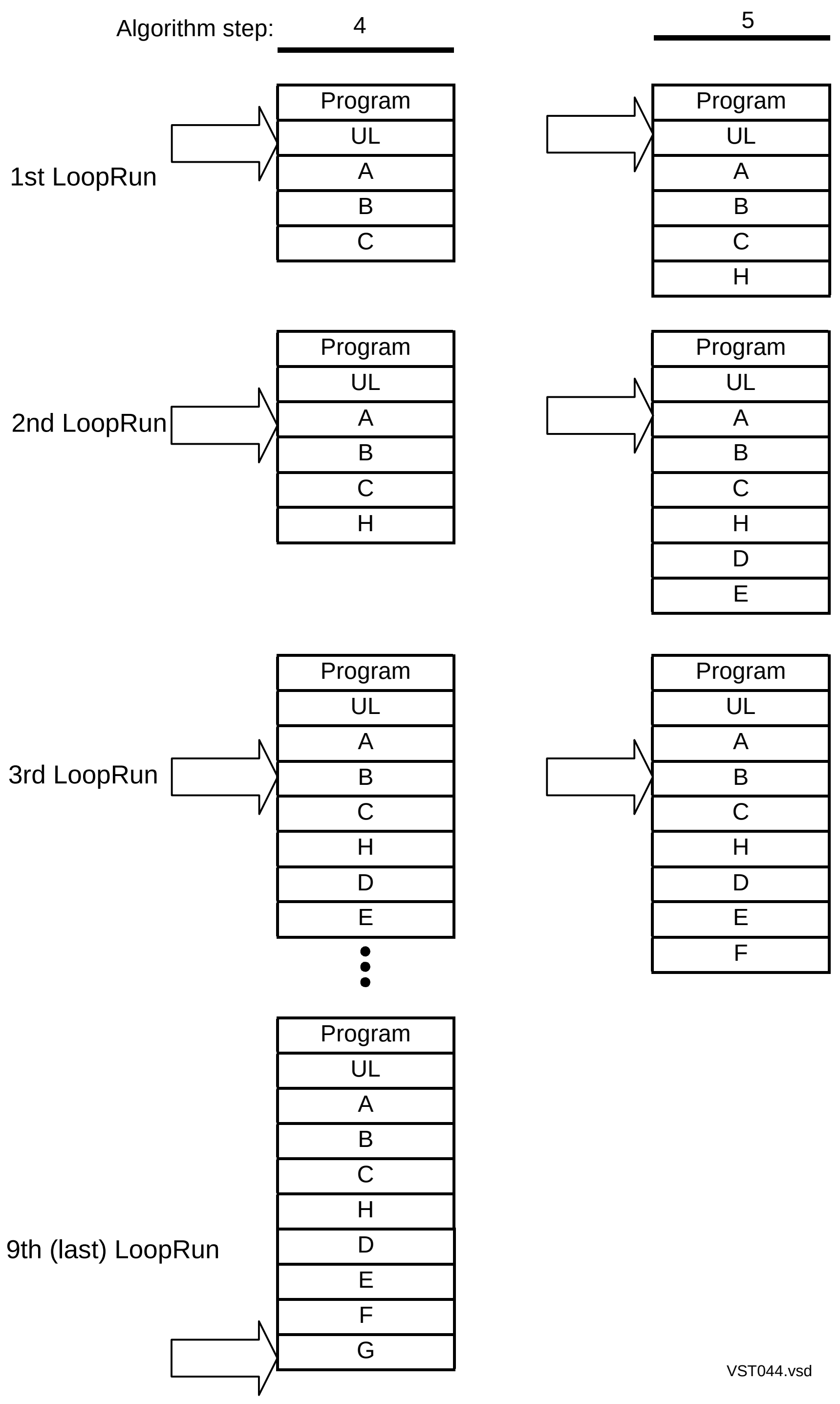
An advantage of globalized import is that the resulting bindings of symbols across loadfiles more closely resemble those that would occur if all the linkfiles that constitute the separate loadfiles had been linked into one loadfile. (This observation explains why "globalized" import is the UNIX default.)

### Ambiguity Example 1

Figure 4-3 on page 4-7 shows a case of symbol ambiguity when Program is globalized and imports a symbol named `xray`, since both C and D export a symbol named `xray`. In linking Program, the linker sees `xray` in both C and D. However, following the globalized searchList (Program, UL, A, B, C, H, D, E, F, G), the linker encounters C’s `xray` before it encounters D’s `xray`, so it resolves Program’s need with C’s definition.

Furthermore, if D is globalized and if it also references `xray`, then the loader assigns C’s definition to D’s reference as well, preempting the linker’s original assignment of D’s own definition to D’s reference. Thus, `xray` is defined the same for Program and D. If C references `xray`, it will get its own definition, regardless of whether it is localized or globalized, because it appears first in both searchLists.

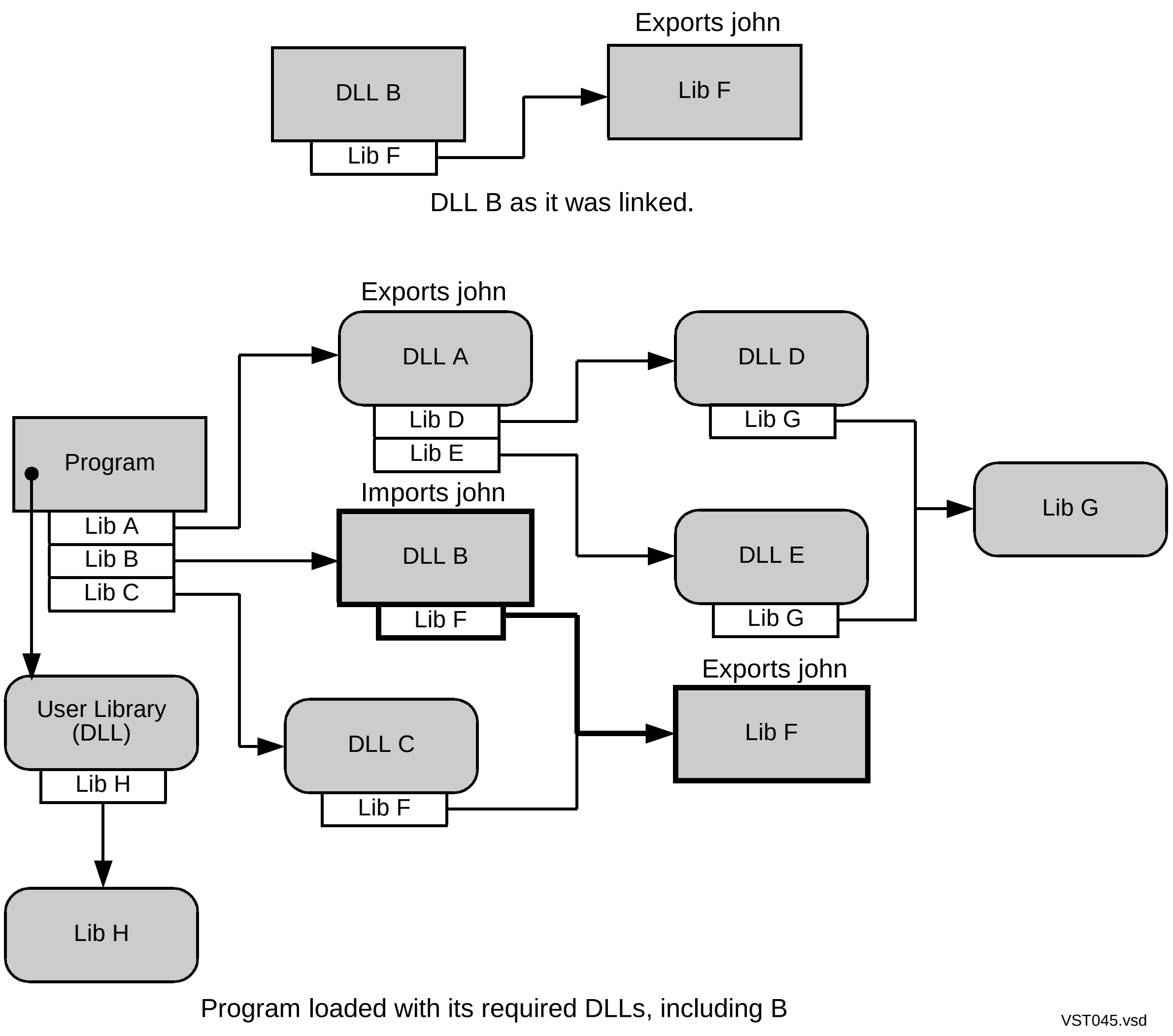
**Figure 4-4. Development of the Globalized SearchList for the Program and Libraries in Figure 4-3**



## Ambiguity Example 2

When linking a DLL, the programmer might not know about the program or the other DLLs that could ultimately be bound together to create an executable set. In Figure 4-5 on page 4-10, the programmer of globalized DLL B links it to another globalized library, called F, that exports the symbol definition of `john` that B should use. Thus, there appears to be no ambiguity, and the resolution of `john` should be simple. But, because B is globalized, the loader sees it differently.

**Figure 4-5. Localized and Globalized Symbol Resolution**



When loading Program, the loader's picture is shown in the lower part of Figure 4-5, which includes DLL B and Lib F as they were linked in the upper part of the figure. Program requires A, B, and C, and it turns out that `john` is exported by both A and F. Recall that imported symbols for every globalized loadfile are resolved using Program's globalized searchList, which is Program, UL, A, B, C, H, D, E, F, and G. Because the loader encounters A's definition of `john` before F's and because B is globalized, the loader satisfies B's need with A's definition, not F's. If B's programmer expected otherwise, there is a surprise in store.

Perhaps more surprising, if a DLL references its own global symbol definition, one might think that this definition would always be used to satisfy this DLL. However, as we saw earlier in Ambiguity Example 1 on page 4-8, the loader can preempt the linker's setting and decide to import that symbol. So, if a DLL is globalized, any of its references might be satisfied by a symbol of the same name in some other loadfile. Therefore, in the above example, if F references `john` in addition to offering it for export, then because F is globalized, the loader binds F's reference to A's definition instead of F's.

B's and F's programmers can avoid this substitution by declaring both B and F to be localized. This assures that in resolving B's imported symbols, the linker and the loader use B's localized searchList (in this case: F); it also assures that F uses its own definition of `john`. Note, however, that if any other globalized loadfile in the list needs `john`, it still gets the definition from A. So, for example, if B is localized and F is globalized, B uses F's definition of `john`, but F uses A's.

### Ambiguity Caution

The UNIX environment standardizes on globalized import controls, and hence, accepts symbol ambiguity; but it can be dangerous and is often undesirable, because a symbol might be resolved in a way that the programmer did not anticipate. Like UNIX, TNS/E permits ambiguity in the globalized case, and the loader issues no warning about it. On the other hand, this mechanism ensures that all globalized loadfiles in a process will consistently use the same definition for a given symbol.

## The SearchList for a Semi-Globalized Loadfile

Localizing a DLL prevents a locally referenced and defined symbol from being resolved by an external definition. But should the DLL need to be globalized to resolve other symbols, declaring it semi-globalized might solve the problem.

A semi-globalized searchList for a given loadfile is identical to the globalized searchList except that the given loadfile's name is ordered at the beginning of the searchList. For example, in Figure 4-5 on page 4-10, if F is declared semi-globalized, then to resolve F's imported symbols the loader will use the searchList: F, Program, UL, A, B, C, H, D, E, and G. As a result, F uses its own exported symbols; and hence, the loader will bind F's reference to `john` to its own definition of `john`.

Semi-globalized import exists primarily for compatibility with the conventional UNIX option -B symbolic. Like localized import, it prevents preemption of definitions. Localized import provides finer control.

## Import Control Summary

This table summarizes the effects of the three import options that you can apply when linking your loadfile. Both `-b globalized` and `-b semi_globalized` (or its synonym `-b symbolic`) invoke a sort range that includes every loadfile in the loadList of the program. On the other hand, `-b localized` restricts the range to within the loadlist of the loadfile itself. That range comprises the loadfile's LibList, the loadfiles that those loadfiles re-export, and so on.

**Table 4-1. Import Control Summary**

| Option | Range | Sequence |
|---|---|---|
| `-b globalized` | loadList of the program | load sequence |
| `-b semi_globalized` | loadList of the program | current loadfile followed by load sequence |
| `-b localized` | within the loadfile's loadList | load sequence |

## C++ Considerations: Globalized (Gblzd) Symbols

Constructs in the C++ language can generate symbols that might have multiple definitions (for example, one in every compilation unit that includes a particular header file). The compiler marks these symbols so that the linker can recognize them and pick one definition for use throughout the resulting loadfile. The linker also marks them so that the loader can recognize them and pick one definition for use throughout the process.

Here are a few of the situations that give rise to such symbols:

- Functions declared inline for which the compiler defines an out-of-line procedure
- Elaborations of template functions or classes
- Compiler-generated variable names used to identify types for run-time type checking, including throw and catch of exceptions

The compiler generates these symbols as needed; programmer's cannot explicitly define them in source.

For correct function of the C++ program and libraries, it is important that a single definition be used throughout the process. For example, an exception can be thrown and caught in different loadfiles only if they agree on the exception class type.

These special symbols are called globalized symbols, often abbreviated gblzd. The overloading of the term *globalized* arises because these symbols unconditionally have globalized semantics: the loader picks one definition for this symbol and binds all references to it. It does so regardless of the import control for the referencing loadfile. One must carefully distinguish the term *globalized symbols* (these symbols that automatically get special treatment) from *globalized import* (selected by the user at link time).

These gblzd symbols reside in a separate symbol table in PIC loadfiles.

## System Library and Millicode

The TNS/E system and millicode libraries are implicitly attached to the end of every searchList and cannot be mentioned in the command stream. So, if the loader does not find a needed symbol definition in the loadfiles in the searchList, it will automatically look for one in the system and millicode libraries.

## Symbol Resolution at a Glance

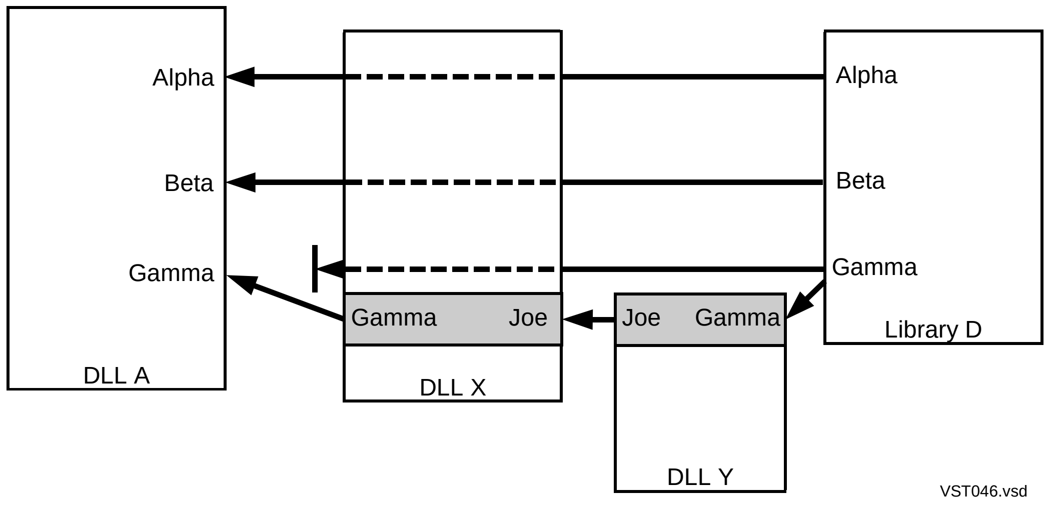
Globalized symbols defined in C++ receive special handling, described above. The following statements apply to all other symbols:

- Whether a given loadfile is declared localized, semi-globalized, or globalized determines how its imported symbols are resolved. That declaration has no effect on how imported symbols of other loadfiles are resolved.
- A loadfile's searchList names the loadfiles to search in the sequence they are to be searched in order to find symbols referenced by the loadfile.
- For a globalized loadfile, at load time, symbols are resolved using the globalized searchList for the program with which the loadfile is loaded.
- For a semi-globalized loadfile, at load time, symbols are resolved using that loadfile itself followed by the globalized searchList for the program with which it is loaded.
- For a localized loadfile, at load time, symbols are resolved using the localized searchList for that loadfile, which begins with the loadfile itself. If that loadfile's libList includes DLLs that re-export other libraries, then that searchList can include libraries that are not listed in the loadfile's libList.
- The loader appends the load list of any dynamically loaded DLLs to the end of the globalized searchList of the prior operating load set.

## Example: Intercepting an Exported Symbol

This allows you to introduce a new function that modifies an existing one in an existing library. This modified function may be used by some loadfiles in the installation, but need not be used by all; they can still access the original function without modification. This case is similar to previous examples in Section 2, Essential DLL Facility Controls but it takes advantage of searchList order to override a symbol definition and leaves the existing library unmodified, as illustrated in Figure 4-6 on page 4-14  The following description assumes the starting condition shown in Figure 2-1 on page 2-15.

**Figure 4-6. Intercepting a Call to Library D**



The symbol `Gamma`, provided by the existing Library D, is intercepted for selected calling loadfiles (including DLL A). These selected loadfiles must be relinked to put DLL X in their libLists and remove Library D. Hence, their `Gamma` references are bound to the definition in DLL X. The new function that modifies the original `Gamma` goes into X and Y, and Y accesses Library D to get the original `Gamma` function, so you do not have to rewrite it. Calls to `Gamma` from unmodified calling loadfiles continue to go directly to Library D.

To create this library intercept you:

1. Relink the calling modules (such as A) that are to use the intercepted `Gamma` in order to replace D in their liblists with X. These calling loadfiles must be localized.
2. Write a new localized DLL X that contains the new (intercept) function and offers the intercepted symbol, `Gamma`. Make X re-export D, so the other symbols D offers are available to the callers.
3. So that you do not have to rewrite the original `Gamma` definition, the new function in DLL X needs to pass control to `Gamma` in DLL D. But DLL X cannot call `Gamma` in DLL D directly, because X's reference to `Gamma` would be resolved to X's own definition of `Gamma`. Hence, you introduce an intermediary DLL, Y, which references `Gamma`. Y must be localized and have D in its liblist. If for some reason X must be in Y's liblist, then be sure that D precedes it in that list.
4. Instead of calling `Gamma`, X calls Joe to pass the information to Y. Y's definition of Joe converts this to a call for `Gamma` in D.

# 5 Advanced DLL Facility Controls

This section tells how you can manually override and extend previously described linker and loader defaults and options to meet special needs. These advanced options fall loosely into four categories that compose the major headings in this section:

- Linker Input Controls on page 5-1
- Linker Output Controls on page 5-4
- Link-Time Operation on page 5-10
- Load-Time Operation on page 5-13

## Linker Input Controls

Specifying Which Inputs Go into a Link on page 2-7 describes the essential controls over linker inputs. This section covers less usual situations where more precise control over inputs is needed.

### Making the Linker Accept Only DLLs or Only Archives

The following options act as a three-way switch that sets the mode for processing subsequent items in the command stream. They allow you to constrain the class of files the linker accepts in a link operation:

| Option | Constraint on Linker |
|---|---|
| `-b static` | Accept only archives, not DLLs |
| `-b dllsonly` | Accept only DLLs, not archives |
| `-b dynamic` | Accept both archives and DLLs |

These options can be inserted multiple times, and each time, the option sets the mode for processing subsequent input-file names in the command stream, until another of these options appears and changes the mode again. At the beginning of the command stream, the undeclared mode is `-b dynamic`. The following examples illustrate error conditions that can occur when the linker opens a file named either directly on the command line or in a `-lib` option.

- If `-b static` is in effect and the file is a DLL, the linker terminates in error.
- If `-b dllsonly` is in effect and the file is an archive, the linker terminates in error.

Also, recall that if the linker is searching for both archives and DLLs, `-allow_missing_libs` can cause the linker to overlook missing archives as well; see Allowing Missing Libraries on page 2-12. However, if `-b static` is in effect, then the linker can only be looking for archives, and even if `-allow_missing_libs` has been inserted, the linker will terminate in error if it doesn't find the specified archive file.

## Augmenting Library Names Automatically in Searches

By convention in OSS and Windows, names for DLLs and archives have a common prefix, `lib`, and have distinguishing suffixes (filename extensions), `.so` for DLLs and `.a` for archives. However, especially if the linker host is Windows but the target is Guardian, it might be convenient to use 'simple' filenames directly.

Therefore, if you tell the linker to search using `-l abc`:

1. `eld` first attempts to open `abc`.
2. If `eld` fails to find that file in the Guardian file system, and -b static is not in effect, it looks for `libabc.so`.
3. If `eld` still has not found a file and -b dllsonly is not in effect, it looks for `libabc.a`.
4. Again if -b static is not in effect, `eld` will try to open a file named `zabcdll.`

The prefix and the suffixes are appended automatically at each location in the search path list, including the public library set.

The augmentation of library names occurs only in the linker, in all environments. (The Guardian loader searches only for the name found in the libList, which the linker takes from the DLL name of the target file.)

## Handling Duplicate Symbols among Linkfiles in a Link

The same symbol can be defined in more than one linkfile in a link, and the linker may or may not treat this as an error. The following subsections tell when duplication is treated as an error and when it is accepted.

### Deciding When to Accept Duplicate Symbol Definitions in Linkfiles

#### Either Data or Procedure Definitions

The linker treats multiple definitions of the same symbol in linkfiles as an error when either all the definitions of a symbol are not data items or all are not procedures.

#### Data Definitions

The linker accepts multiple data-item definitions of the same symbol in input linkfiles when both of the following are true:

- An item is defined in more than one file, and the compiler has marked every instance of this symbol to allow multiple definition (so this will become a gblzd symbol; see C++ Considerations: Globalized (Gblzd) Symbols on page 4-12 .

- The linker can determine that they are the same size and that their initial values (if any) are the same.

Otherwise, the linker treats multiple definitions of a data item in input linkfiles as an error.

#### Procedure Definitions

Multiple procedure definitions of the same symbol are permitted when the compiler generates and appropriately labels the duplicates. In this case, the linker checks that the attributes of the duplicated procedures are identical except for `EDITLINE`. If the others are not identical, the linker declares an error.

### Choosing Among Accepted Duplicate Symbol Definitions in Linkfiles

In reading the following, be aware that the linker processes linkfiles in the order that they, or the archives they come from, appear in the command stream.

#### Data Definitions

When the linker accepts duplicate data-item definitions of a symbol in linkfiles, it chooses the instance to use according to the following priority:

1. An instance that is initialized over one that is not.
2. An instance which retains its symbol-table information over one that has been stripped of this information.
3. An instance that come from the following sources in preferred order: C, C++, pTAL, COBOL.
4. The first instance the linker finds in processing the linkfiles.

#### Procedure Definitions

When the linker accepts a duplicate procedure definition of a symbol in linkfiles, it selects the instance to use from the first version of the procedure it finds in processing the linkfiles.

## Making the Linker Look for Unresolved Symbols

As part of a link, you can ask the linker to check that the symbol definitions imported by the output loadfile are available through the libraries in its libList. You do this by inserting `-unres_symbols <parameter>`, where `<parameter>` can have the values `error`, `warn`, or `ignore`. When `-unres_symbols` is in effect, if the loadfile being linked is localized, the linker searches for libraries and their symbols as defined in The SearchList for a Localized Loadfile on page 4-5. On the other hand, if the loadfile being linked is globalized or semi-globalized, the linker searches for libraries and their symbols following a searchList that is identical to this loadfile's loadList.

Until the linker finds a library that offers a symbol that the output loadfile needs, it considers that symbol unresolved. After searching the libraries described above and looking in the millicode libraries, if the linker cannot find an exported symbol to match an imported one in the output loadfile, it will respond as specified by `<parameter>`. If `error` is specified, the linker terminates and reports an error; if `warn` is specified, the linker issues a warning message and continues; if `ignore` is specified, the linker does not attempt to find matching symbols. If you do not specify `-unres_symbols`, the default on TNS/E is to ignore unresolved symbols.

Also, when linking on a system other than the execution target, such as an auxiliary system, be aware that certain libraries may be unavailable. This may cause an inevitable error or warning if either is specified.

Also note that if you insert the option, `-unres_symbols` and also allow missing libraries, then `-unres_symbols` automatically defaults to `ignore`.

# Linker Output Controls

## Designating the Main Entry Point of Your Program

When a program loadfile is linked, it must have a main entry point designated in one of the following ways:

- The input to the link must include a linkfile that contains a procedure that has the `main` attribute. If you do not insert the `-e <entrypoint>` option, as described in the next bullet, the linker will designate the procedure with the main attribute as the entry point.
- You explicitly designate the main entry point using the `-e <entrypoint>` option, where `<entrypoint>` is the name of a global symbol definition. (It need not be offered for export.)

PIC programs written in C or C++ are usually linked including a linkfile named CCPLMAIN (Guardian) or ccplmain.o (OSS, Windows), which provides a function named _MAIN() that has the main attribute. COBOL programs include a compiler-supplied main program. Therefore, the `-e` command is rarely needed.

In a program, the procedure having the `main` attribute is the default entry point for starting the program, and it is usually a mistake for you to force a program to start at some other point. The linker will not accept a program in which there are more than one procedure having a `main` attribute, unless you insert the `-allow_multiple_mains` option. In this case, if there are two main procedures, you must resolve the choice by inserting the `-e <entrypoint>` option.

The linker will not accept a DLL with a procedure having the `main` attribute; you can force a DLL to have an entry point by inserting the `-e <entrypoint>` option when linking that DLL, but the loader ignores it.

## Controlling Which Symbols Your Loadfile Exports

Symbols offered for export are those made available to other loadfiles. By default, the linker offers for export those symbols designated by the compiler to be exported. If you insert the `-export_all` option, the linker offers all defined global symbols except for the following:

- Special initialization, construction, destruction, and termination procedures with reserved name prefixes, namely __INIT__, __sti__, __std__, and __TERM__.
- Symbols that the linker creates, only for use within the current loadfile.

The linker interprets the option `-ul` as synonymous with the combination of the `-dll` (or `-shared`) option and the `-export_all` option. As the option name `-ul` suggests, it is often convenient, but not necessary, to use this option when linking a user library. This option can be used when linking any DLL.

To assure that a particular symbol is unconditionally offered for export, insert the `-export <symbolname>` option. To assure that a symbol is unconditionally not offered, insert the `-export_not <symbolname>` option or its synonym `-hidden_symbol`.

### Special initialization and termination procedures

Special initialization, construction, destruction, and termination procedures are procedures whose names begin with certain prefixes. These prefixes and their order of execution are as follows:

`__INIT__`

Alphabetical order of procedure names.

`__sti__`

The order in which `eld` sees the procedures.

`__std__`

The reverse of the order in which `eld` sees the procedures.

`__TERM__`

Reverse alphabetical order of procedure names.

As the names suggest, these procedures are called during the initialization (__INIT__ and __sti__) and termination of a process (__std__ and __TERM__). These procedures are called without the program explicitly calling them.

`__sti__ and __std__`

The C++ compiler generates local functions to invoke static constructors and destructors for global variables that are instances of classes. The linker builds two `iniTerm` lists corresponding to these procedures, and places pointers (`ctors` and `dtors` addresses) to them in the `tandeminfo` segment of the object file.

`___INIT__ and __TERM__`

Apart from the procedures from runtime-library, the user program itself might contain procedures with these names (for example, __INIT__< *name*>). These functions need not be called by the program explicitly; they are executed in alphabetical order of the procedure names in case of __INIT__ procedures and reverse alphabetic order in case of __TERM__ functions. The linker builds two `iniTerm` lists corresponding to these procedures and places pointers (`initz` and `termz` addresses) to them in the `tandeminfo` segment of the object file.

## C++ Mangled Symbol Names

In order to identify to the linker a C++ function (other than one declared with `extern` “c”), you must use the symbol’s mangled name. The linker does not make the correspondence between unmangled and mangled names. For example, if `<symbolname>` or `<entrypoint>` in the two previous subsections are in a C++ program, these values must be the mangled version of their source-code names.

## How to Set Run-Time Attributes of Your Loadfile

At link time, the linker sets certain run-time attributes of the loadfile it is building to default values, unless you override a default by inserting the `-set attribute <value>` option.

See also Default Setting and Checking of File Attributes on page 2-19.

Each `-set attribute` option can affect only one attribute, so this option must be re-inserted for each attribute that is changed. Permitted attributes are shown in the following list, which summarizes the values that can be set. The linker places these attribute values in the loadfile being linked in order to make them available to the system at load time.

`cppdialect`

sets the C++ dialect of the output loadfile. The values are `cppneutral` or `v2` or `v3`.

See also C++ Dialect on page 2-20.

`float_lib_overrule`

inhibits float-type checking of the program and the libraries from which the program imports symbols. Possible values are on or `off`, and the default is `off`. See Checking the Floating-Point Types of Liblisted Libraries on page 2-20.

`floattype`

sets the floating-point type of the output loadfile. Possible values are `ieee`, `tandem`, and `neutral`. See Setting the Floating-Point Type of a Loadfile Being Linked on page 2-19.

`heap_max`

has a numeric (hexadecimal) value with a default of zero (0).

`highpin`

has possible values of on and `off`. The default is on.

`highrequestors`

has possible values of on and `off`. The default is on.

`incomplete`

has only one possible value, on. If this is not specified, and an import library is being created, it is a complete import library.

`inspect`

has possible values of on and `off`. The default is `on`.

`libname`

is the name of a DLL to be the user library of the PIC program being linked. See Specifying a User Library for a Program on page 2-12.

`mainstack_max`

has a numeric (hexadecimal) value with a default of zero (0).

`oktosettype`

has possible values of on and `off`. The default is `off`.

`pfsize`

is a number. It is accepted for compatibility with TNS/R systems, but is ignored.

`process_subtype`

has a numeric value with a default of zero (0).

`rld_unresolved`

has possible values of `error`, `warn`, and `ignore`, and the default is error. If `error` is set (or by default), then an unresolved symbol encountered by the loader (`rld`) will result in its error termination. If the loader cannot resolve a symbol that references data, an error occurs regardless of the setting of `rld_unresolved`.

However, if the loader cannot resolve a symbol that references code, and if `warn` or `ignore` is set, it searches for a symbol definition named `UNRESOLVED_PROCEDURE_CALLED_`. If this definition is found, the symbol is bound to this definition and either a message is put out if `warn` is set or no message is put out if `ignore` is set.

By default, the loader finds this function defined in the system library. If invoked, it generates a non-deferrable SIGILL signal. You can provide your own function if you wish; the loader searches normally for this symbol, so it will use any definition found in the searchList of the loadfile containing the unresolved reference.

`runnamed`

has possible values of on and `off`. The default is `off.`

`saveabend`

has possible values of on and `off`. The default is `off`.

`space_guarantee`

has a (hexadecimal) numeric value with a default of zero (0).

`systype`

sets the system type of the loadfile being linked. Possible values are `OSS` and `Guardian`. See Execution-Target System Type on page 2-22.

**Note.** The `systype` attribute has no meaning for a DLL. A DLL can be used by a Guardian process, or an OSS process, or both, depending on how the various parts of it were written and compiled. You must have this information about the DLL to use it appropriately.

## Controlling the Load Image of DLLs

### Segmenting Loadfiles

The linker produces the output loadfile in two segments, text and data.

The linker can be directed to produce two data segments - one for constant data (the constant data segment) and one for variable data (the data variable segment). Also, if the input linkfile contains PRIV code, the linker will also produce a gateway segment.

### Specifying the Preferred Location of a Loadfile in Virtual Memory

The linker assigns addresses for these segments by default; you can specify non-default values with the following options:

`-t` *`address`*

this is a hexadecimal number that sets the starting address of the text segment (headers and code).

`-d` *`address`*

this is a hexadecimal number that sets the starting address of the data segment.

In both cases, the linker rounds up *`address`* to a virtual memory page boundary. If `-t,` but not `-d,` is inserted, then `-t` specifies the starting address of the entire contiguous DLL. By inserting `-d`, you can cause the DLL to load in two separate regions.

By default, the loader loads the text and data of a DLL into a contiguous virtual-memory area, starting with the text (including code) section. This area is chosen to avoid memory interference with other loaded files. These linker-assigned addresses are called the preferred addresses; if this address range for a DLL is available, the loader will use it.

If `-t`, but not `-d`, is inserted, then `-t` specifies the preferred starting address of the entire contiguous DLL. By inserting `-d`, you can cause the linker to assign the two DLL segments to disjoint address ranges. If you do that, the loader will reject the DLL.

In general, you should not use the `-d` option because it may result in the creation of an object file that the operating system will refuse to load.

In contrast to a DLL, a program is always divided into two separate regions. For a program, the default for `-t` is 0x70000000 and for `-d` is 0x08000000. If a non-default value is specified, the loader will reject the program.

Summarizing these two options: `-t` is useful to specify a preferred address for a DLL, so that it will not overlap the preferred address ranges of other DLLs that will appear in the same process; with non-overlapping address ranges the DLLs will load slightly faster. Otherwise, `-t` and `-d` are useful only in unusual circumstances that are beyond the scope of this manual.

## Using the Linker to Change an Existing Loadfile

The options described in this subsection act on existing loadfiles, rather than a file being linked. In the following options, `<loadfileName>` is the name of the target loadfile.

### Changing the Attributes of an Existing Loadfile

To alter any attribute of an existing loadfile, invoke the linker and insert the `-change <attribute> <value> <filename>` option.

The `<attribute>` and `<value>` choices are the same as for `-set`, as described in How to Set Run-Time Attributes of Your Loadfile on page 5-6. The `-change` option overwrites the attribute in the existing loadfile. You must be able to open the existing loadfile for update. If an error occurs in `-change` processing, the specified change may or may not have occurred and the file will be otherwise unmodified.

You can insert multiple `-change` options, where each must be followed by the name of the affected loadfile, even if the loadfile is the same in each case. These options are executed one at a time in the order inserted.

Your user ID must have write access to the file to execute the `-change` option on it.

# Link-Time Operation

## Running the Linker Through the Compiler

The C and C++ compilers can run the linker for you, saving you a separate step in straightforward situations. In each case, the compiler automatically names many of the appropriate input files and library files in the linker command stream. These are:

- The usual C libraries
- The usual C++ runtime libraries and cppinit module for the selected dialect version, for C++
- CCPLMAIN or ccplmain.o

The compiler does not automatically include more specialized libraries, such as those for tools.h++, or the loader library (ZRLDDLL).

### Guardian

CCOMP and CPPCOMP will run the linker when building a program, if you add the RUNNABLE pragma to the command line. For example:

```
ccomp /in myprogc/ myprog; suppress,runnable,call_shared,symbols
```

These compiler drivers will also run the linker when building a DLL, if you use the SHARED rather than the CALL_SHARED option. For example:

```
ccomp /in mylibc/ mylib;suppress,shared,symbols
```

This example compiles a program that uses that DLL:

```
ccomp /in myprog2c/ myprog2;suppress,runnable, &
call_shared,symbols,search "mylib"
```

The search pragma causes the “mylib” to be included in the linker command stream.

A more flexible way to add linker commands through the compiler command line is the pragma LINKFILE. For example, suppose that you create an edit file named obeyrld that contains one line:

```
-lib zrlddll
```

Then the following example will link a program that requires the dynamic loader library (in addition to the usual libraries that are provided automatically):

```
ccomp /in mydynpc/ mydynp;suppress,runnable, &
call_shared,symbols,linkfile "obeyrld"
```

### OSS

The c89 or c99 compiler will run the linker for each module it compiles, unless the `-c` option is present. It is somewhat more flexible in allowing you to specify linker commands to the compiler:

- The `-L` and `-l` options are passed straight through, with their arguments.
- Several other linker options are recognized in `-W` form; run c89 or c99 with no command input to see a list.
- The `-Weld=args` option passes args into the `eld` command stream. For example:

    - pass a parameterless option: `-Weld=-verbose`
    - pass an input filename: `-Weld=foo.so`
    - pass a sequence of tokens including blanks: `-Weld="-set floattype neutral_float"`

The `-Weld_obey=file` option passes the named file as an obey file for the linker. Here's an example that compiles and links a DLL:

```
c89 dllx.c -o dllx.so -Wshared
```

(Note that `.so` is the conventional extension for a DLL on OSS.)

The following example compiles and links a program that uses that DLL and also uses dynamic loading:

```
c89 prg.c -o prg -Wcall_shared dllx.so -l zrlddll
```

## Naming Intermediate Linker Output Files

The linker always creates the output in an intermediate file in a link operation, and the linker gives this intermediate file an internal name. When the operation is complete, the intermediate file contents are renamed to the file name you specified as the output file. See Choosing the Output File on page 2-3. If you want to allow for the linker being unable to open the output file and store the finished loadfile under the desired name, you can give the intermediate file a name that you can locate easily; otherwise, you might have to relink. To do this, you insert the `-temp_o <filename>` option, where `<filename>` is the name you choose.

## Controlling What Checks the Linker Makes and Reports

### Automatic Messages

The linker reports on its operation using messages that appear in an output listing. These are reported with a message identifier that indicates the message's severity as follows:

| Identifier | Severity | Meaning |
|---|---|---|
| 500-19999 | Fatal Error | The linker cannot do what was requested and it stops immediately. |
| 20000-29999 | Error | The linker cannot do what was requested and will eventually stop. |
| 30000-39999 | Warning | The linker can continue but the results seem questionable. |
| 40000-49999 | Informational | This does not indicate a problem. |

Messages are the same on all linker host platforms.

## Message-Control Options

The appearance of these messages is controlled by the following options:

`-verbose` tells the linker to show all messages.

`-warn` tells the linker to show all fatal-error, error, and warning messages, plus messages requested by command-stream options, as described in the next subsection.

`-no_verbose` tells the linker to show all fatal error and error messages, plus messages requested by command-stream options, as described in the next subsection. `-noverbose` is a synonym for `-no_verbose`.

The default is `-no_verbose`, except on Guardian, where the default is `-verbose`.

If `-verbose` or `-warn` is in effect, then when a linking operation finishes, the linker puts out a completion message that covers the entire process and indicates how many of each type of message (error, warn, or information) were generated during the linking process.

## Command Stream Requests for Linker Messages

In addition to the progress-reporting linker messages described above, you can request informational messages by inserting one of the following options. These are effective regardless of the message-control option in effect.

`-y <symbolname>` tells the linker to show the names of linkfiles that mention `<symbolname>` in their external symbol tables and the information in those tables about the symbol. *`-y`* reports on linkfiles that either reference the symbol or offer it for use beyond the same compilation.

`-show_multiple_defs` tells the linker to show information about all multiply defined symbols. Unlike `-y`, the linker only provides information about linkfiles that offer the symbol for use beyond the same compilation, not about those that reference it.

`-map` tells the linker to report the virtual addresses and sizes of the segments and the sections, within each segment, of a loadfile being built. Further, the linker shows the library and file names of all liblisted libraries. `-m` is a synonym for `-map`.

# Load-Time Operation

To load a program and its DLLs, the system is commanded simply to load the program. The loader automatically ensures that the program's entire loadList is loaded and that all symbols are resolved with proper bindings.

## Controlling the Loader's Search Path at Load Time

Like the linker (`eld`), the loader (`rld`) searches for DLLs, but the loader does it according to `rld`'s search path, which was initially specified when the loadfile was linked; see The Link-Time-Defined Search Path of the Loader on page 2-17. To this path, other directories or subvolumes may be added at load time as follows.

On OSS, you can specify the following environmental variables:

```
EXPORT _RLD_FIRST_LIB_PATH=path1[:path2]...
EXPORT _RLD_LIB_PATH=path1[:path2]...
```

where *path1*, *path2*, etc. are the paths the loader is to search.

On Guardian, you can use the following Search Class Defines, specified according to the *Guardian Programmers Guide*.

```
ADD DEFINE =_RLD_FIRST_LIB_PATH
ADD DEFINE =_RLD_LIB_PATH
```

The list of paths is the set of subvolumes specified by the define's attributes in the order SUBVOL0, RELSUBVOL0, SUBVOL1, ..., RELSUBVOL20. Each attribute can specify a single subvolume or a parenthesized, comma-separated list of subvolumes. Any attributes unspecified are ignored.

For example the following TACL statements specify search list $A.B:$B.C:$C.D:$D.E:

```
SET CLASS SEARCH, SUBVOL0 $A.B, RELSUBVOL0 $B.C, &
SUBVOL1 ($C.D,$D.E)
ADD DEFINE = _RLD_FIRST_LIB_PATH
```

The same list can be specified more simply with a single statement and single attribute:

```
ADD DEFINE =_RLD_LIB_PATH, CLASS SEARCH, &
SUBVOL0 ($A.B,$B.C,$C.D,$D.E)
```

No blank spaces are permitted within the parenthesized list.

In either case, the loader's search path that was defined at link time is augmented as follows, where the paths identified at link-time were steps 2, 3, 6, and 7, as described in The Link-Time-Defined Search Path of the Loader on page 2-17.

1. The directories or subvolumes specified at load time by `_RLD_FIRST_LIB_PATH`.
2. The directories or subvolumes specified in `-RLD_first_L` options at link time
3. The public libraries (DLLs)
4. The directory or subvolume that stores the program being loaded
5. The directories or subvolumes specified by `_RLD_LIB_PATH` at load time
6. The directories or subvolumes specified in `-RLD_L` options at link time
7. Following default locations:
    - 32-bit process:

      For OSS: `/lib, /usr/lib, /usr/local/lib,` and `/G/SYSTEM/ZDLL` in the order of the paths specified here.

      For Guardian: `$SYSTEM.ZDLL`
    - 64-bit process:

      For OSS: `/lib64, /usr/lib64, /usr/local/lib64, /lib, /usr/lib, /usr/local/lib, /G/SYSTEM/YDLL,` and `/G/SYSTEM/ZDLL` in the order of the paths specified here.

- Steps 3, 4, and 7, the loader follows automatically.
- Steps 2 and 6 were set up at link time.
- Steps 1 and 5 are load-time additions.

Steps 1 and 2 are provided for completeness, but are rarely required and are not recommended for routine use. Because the public libraries can be searched in a memory table, but all other searches require messages to name servers or disk processes, it is more efficient to start at step 3. The public libraries have unique names (Z*DLL) that should not interfere with finding ordinary DLLs later.

Note that the path search order for finding libraries is irrelevant to symbol search order, which depends upon the searchList order, developed from the libLists.

`_RLD_FIRST_LIB_PATH` and `_RLD_LIB_PATH` can be stated only once for loading each program and its libraries, so you must specify all the desired search paths for each of these two environmental variables or defines in a list of colon-separated path names.

You can prevent the loader's search path for your loadfile from deviating from what you set up at link time, that is, restrict the loader's search path to steps 2, 3, 6, and 7 above. You do this by having inserted the `-limit_runtime_paths` option in the linker's command stream; see The Link-Time-Defined Search Path of the Loader on page 2-17.

## Changing Run-Time Options for C and C++ Programs

These options are available to you only when linking programs, not DLLs.

When linking a program written in C, if the target platform is OSS, then by default, the linker sets the program to use the code 180 C text files. But if the target is Guardian, then by default, it sets the program to use code 101 C text files. If the target is Guardian and you want the program to use code 180 C text files instead, then insert `-ansistreams` option when linking the program.

When linking a program written in C or C++, by default, the linker sets the program to use the standard C/C++ I/O files. To avoid this setting, insert the `-nostdfiles` option when linking the program. `-no_stdfiles` is a synonym for `-nostdfiles`. This means that when the program is run, the program's stdin, stdout and stderr files will not be opened automatically and you may thus need to add logic to the program to open these files.

# 6 Example Code

This section contains a set of examples to introduce you to some of the tools and capabilities for building dynamic linked libraries on a TNS/E system.

Example One. This example creates a PIC program running with a DLL, using the TNS/E native compiler and linker.

Example Two. This session demonstrates dynamic loading. We create and compile a main and a DLL, but don’t load the DLL. We invoke rld, the run-time loader, from inside the process by using a dlopen() call.

## Example One

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.* In Example Two we show the main built without reference to a DLL. We will use `rld` to load the DLL dynamically.

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
```

## Example Two

This example was created using TNS/R tools, thus uses `ld` instead of `eld` as the linker. The use of the dynamic loader `rld` is the same on either TNS/R or TNS/E.

This session demonstrates dynamic loading. We create and compile a program and a DLL, but don't load the DLL with the program. We invoke the loader from inside the process with a dlopen() call.

If you set breakpoints in Visual Inspect just before and after executing such calls, and use the new TACL command LOADEDFILES, you can see the process in action.

### Display the Source Code

Again there is a program and a library in the session, *mainc* and *helloc*. In *mainc* we use dlopen() to load *hellodll,* which will be created from *helloc*. We use dlsym() to find function hello(), which we invoke. We then use dlclose() to dynamically drop that loaded file. Just to demonstrate an error routine, we then attempt to load a non-existent DLL called *NotThere*.

Note that `dlopen("hellodll",RTLD_NOW)` takes a pathname that typically is a relative pathname. It is qualified by the default directory searchlist which can be overridden at link or load time. At load time, the searchlist is modified by the `=RLD*` defines (for Guardian) and environmental variables (for OSS processes).

The source code for *mainc* is as follows:

```
#include <dlfcn.h>
#include "hello.h" nolist
#include <stdio.h> nolist

/************************************************************
|    main:
|
\************************************************************/
int main(int argc, char *argv[]) {

  int result = 0;
  union {
    int resCode;
    short err[2];
    } res;

  typedef void (*HelloFPtr)(void);
  HelloFPtr pHello;
  dlHandle dlopenHandle;
  dlopenHandle = dlopen("hellodll",RTLD_NOW);
   if (dlopenHandle == 0) {
        printf( "%s\n", dlerror( ) );
        return (1);
       }

   pHello = (HelloFPtr)dlsym(dlopenHandle, "hello");
   if (pHello == 0) {
       printf( "%s\n", dlerror( ) );
       return (2);
     }
   pHello( );
   result = dlclose(dlopenHandle);
   dlopenHandle = 0;

/* demonstrate the error routines */

  dlopenHandle = dlopen("NotThere",RTLD_NOW);
   if (dlopenHandle == 0) {
        res.resCode = dlresultcode( );
        printf("dlopen of a non-existent dll returns result %i\n"
               ,res.resCode);
        printf("\terror( %i,%i)\n"
               ,res.err[0], res.err[1]);
        printf( "%s\n", dlerror( ) );
        return (1);
       }

   return 0;

} /* of proc main */
```

The source code for *helloc* is as follows:

```
#include <stdio.h>    nolist
void hello( ) {
   printf("Hello, I am a DLL!\n");
}
```

## Compile the Program and Library

Both programs are compiled using the CALL_SHARED option to create PIC.

Note that we could have compiled *helloc* using the -shared compile-time option to cause the compiler to invoke ld to produce a DLL. Instead, to illustrate an alternative, we invoke ld manually to produce the DLL after compiling *helloc* as a PIC linkfile.

This is the compile for the main program:

```
NMC /IN mainc/maino;suppress,OPTIMIZE 0,SYMBOLS,ERRORS
5,EXTENSIONS,call_shared
TNS/R Native Mode Risc C - T9577D46 - 30APR2003 (Mar 23 2003 19:42:26)
(C)2000 Compaq (C)2003 Hewlett Packard Development Company, L.P.


0 remarks were issued during compilation.
0 warning was issued during compilation.
0 errors were detected during compilation.
Object file: maino
Compiler statistics
  phase     CPU seconds   elapsed time   file name
  NMC                                    \DLLQA.$SYSTEM.SYSTEM.NMC
  CFE               0.2       00:00:01   \DLLQA.$SYSTEM.SYSTEM.CFE
  UGEN              0.1       00:00:00   \DLLQA.$SYSTEM.SYSTEM.UGEN
  AS1               0.0       00:00:00   \DLLQA.$SYSTEM.SYSTEM.AS1
  total             0.3       00:00:02
All processes executed in CPU 03 (NSR-G)
Swap volume: \DLLQA.$SYSTEM
```

The compile for *helloc* is as follows:

```
NMC /IN helloc/helloo;suppress, OPTIMIZE 0,SYMBOLS,ERRORS
5,EXTENSIONS,call_shared
TNS/R Native Mode Risc C - T9577D46 - 30APR2003 (Mar 23 2003 19:42:26)
(C)2000 Compaq (C)2003 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: helloo
Compiler statistics
  phase     CPU seconds   elapsed time   file name
  NMC                                    \DLLQA.$SYSTEM.SYSTEM.NMC
  CFE              0.2       00:00:01   \DLLQA.$SYSTEM.SYSTEM.CFE
  UGEN             0.0       00:00:00   \DLLQA.$SYSTEM.SYSTEM.UGEN
  AS1              0.0       00:00:00   \DLLQA.$SYSTEM.SYSTEM.AS1
  total            0.2       00:00:01
All processes executed in CPU 03 (NSR-G)
Swap volume: \DLLQA.$SYSTEM
```

## Build the DLL

The DLL is created with the following command line to ld.

```
ld helloo -o hellodll -obey $SYSTEM.SYSTEM.libcobey -L $OSS.NSKAPAT4 -shared
 -export_all
LD (T0429G09 - 30APR2003)
(C)2003 Hewlett Packard Development Company, L.P.

T0429's command line: helloo -o hellodll -obey $SYSTEM.SYSTEM.libcobey -L
 $OSS.NSKAPAT4 -shared -export_all

LD INFORMATIONAL MESSAGE **** [40056]:
     Entering OBEY file '$SYSTEM.SYSTEM.libcobey'.

LD INFORMATIONAL MESSAGE **** [40057]:
     Exiting OBEY file '$SYSTEM.SYSTEM.libcobey'.

LD INFORMATIONAL MESSAGE **** [40052]:
     '-export_all' specified

LD INFORMATIONAL MESSAGE **** [40063]:
     Creating a DLL and -dllname was not specified; using 'hellodll' as
     -dllname value.

Comment: lots of other informational messages edited out . . .

LD INFORMATIONAL MESSAGE **** [40065]:
     Creating a DLL without -t option specified. LD will use 0x60000000 as
     the default value.

LD reported 0 errors.
LD reported 0 warnings.
LD reported 16 informational messages.

LD created the following named loadfile as type:
     hellodll (ELF, dynamic link library).
LD Timestamp:  31MAR2003 13:36:57
Elapsed Time:  00:00:01
```

Next, the main executable file is built. Note that the main program does not use -l to refer to *hellodll* at link time.

```
ld maino  -o mainexe -obey  $SYSTEM.SYSTEM.libcobey -L $OSS.NSKAPAT4
$SYSTEM.SYSTEM.CCPPMAIN -l ZRLDSRL
LD (T0429G09 - 30APR2003)
(C)2003 Hewlett Packard Development Company, L.P.

T0429's command line: maino -o mainexe -obey $SYSTEM.SYSTEM.libcobey -L
 $OSS.NSKAPAT4 $SYSTEM.SYSTEM.CCPPMAIN -l ZRLDSRL

LD INFORMATIONAL MESSAGE **** [40056]:
     Entering OBEY file '$SYSTEM.SYSTEM.libcobey'.

Comment: lots of other informational messages edited out . . .

LD INFORMATIONAL MESSAGE **** [40049]:
     The library specified as 'ZRLDSRL' in a -l or -lib option was resolved
     to the library named '\DLLQA.$SYSTEM.SYS00.ZRLDSRL'.

LD reported 0 errors.
LD reported 0 warnings.
LD reported 14 informational messages.

LD created the following named loadfile as type:
     mainexe (ELF, executable, main entry point is 0x70000e98).
LD Timestamp:  31MAR2003 13:37:05
Elapsed Time:  00:00:01
```

## Run The Program

This is a simple run of the program. The first DLL (*Hellodll*) loads dynamically, and works just fine. Then rld attempts to load the second DLL (*NotThere*), but it does not appear to be there!

```
RUN mainexe

Hello, I am a DLL!
dlopen of a non-existent dll returns result 75,11
dlopen (70000F3C->NotThere, 1): FileSystem Error 11: File
$SYSTEM.ZDLL.NotThere
 NOT Found
ABENDED: 3,305
CPU time: 0:00:00.010
1: Process terminated with warning diagnostics
```

We also ran the program under Visual Inspect and set breakpoints before and after the call to *Hellodll*. In both those places we used a new TACL command LOADEDFILES to see which files had been loaded at that point. The output from that VI and TACL session is simple, but nevertheless demonstrates a useful capability in more complex situations.

LOADEDFILES (and new option for STATUS that shows which process uses a specified file) is fully documented in the *TACL Reference Manual* (for G06.20 and later).

You can use the process name with LOADEDFILES, but can also refer to it by *cpu, number* which we do in this TACL example.

```
status *,user,pri

Process          Pri PFR %WT Userid  Program file            Hometerm
        3,301   168     000 180,1   $OSS.NSKAPAT4.MAINEXE   $ZTN10.#PTBSQW8
```

**Comment: This is before the first dlopen().**

```
$OSS NSKAPAT 8> loadedfiles 3,301

PROCESS NAME =     UNNAMED PROCESS
TYPE      FILENAME
PPROG     \DLLQA.$OSS.NSKAPAT4.MAINEXE
SRL       \DLLQA.$SYSTEM.SYS00.ZCRESRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSECSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZRLDSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZINETSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZICNVSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSTFNSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSHSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZI18NSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZCRTLSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSKSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSFSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSESRL
Total No. of Loadedfiles = 13
```

**Comment: This is after the first dlopen() ;note the appearance of the DLL.**

```
$OSS NSKAPAT 9> loadedfiles 3,301

PROCESS NAME =     UNNAMED PROCESS
TYPE      FILENAME
DLL       \DLLQA.$OSS.NSKAPAT4.HELLODLL
PPROG     \DLLQA.$OSS.NSKAPAT4.MAINEXE
SRL       \DLLQA.$SYSTEM.SYS00.ZCRESRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSECSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZRLDSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZINETSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZICNVSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSTFNSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSHSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZI18NSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZCRTLSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSKSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSFSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSESRL
Total No. of Loadedfiles = 14
```

**Comment: This is after the first dlclose();note the disappearance of the DLL.**

```
$OSS NSKAPAT 10> loadedfiles 3,301

PROCESS NAME =     UNNAMED PROCESS
TYPE      FILENAME
PPROG     \DLLQA.$OSS.NSKAPAT4.MAINEXE
SRL       \DLLQA.$SYSTEM.SYS00.ZCRESRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSECSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZRLDSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZINETSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZICNVSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZSTFNSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSHSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZI18NSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZCRTLSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSKSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSFSRL
SRL       \DLLQA.$SYSTEM.SYS00.ZOSSESRL
Total No. of Loadedfiles = 13
```

# A Linker Options List

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.

If no hyperlink exists the option is either a synonym so look for the other option name, or the option is not covered in this manual and so you should consult the *eld Manual* for further details. There is a similar options list in the *eld Manual* that will give you the exact page reference in that manual.

`-alf <filename>`

Rebase or rebind an existing loadfile (or both), recreating the file.

`-all`

Use all members from archives.

`-allow_duplicate_procs`

Do not consider it an error if there are multiple definitions of procedures with the same name.

`-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 Allowing Missing Libraries on page 2-12 .

`-allow_multiple_mains`

Do not consider it an error if more than one procedure has the MAIN attribute. See Designating the Main Entry Point of Your Program on page 5-4.

`-ansistreams`

At runtime, the program will use the ANSI version of C I/O. See Changing Run-Time Options for C and C++ Programs on page 5-15.

`-b { dllsonly | dynamic | static }`

These options specify whether the linker accepts DLLs or archives (or both). See Making the Linker Accept Only DLLs or Only Archives on page 5-1.

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

These options affect how references are resolved across loadfiles. See Global Scope, Import and Export on page 4-3.

`-call_shared`

Create a program. See Choosing to Create a Program or a DLL on page 2-3.

`-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 Changing the Attributes of an Existing Loadfile on page 5-9 .

`-check_registry <filename>`

Use the specified DLL registry to tell where the DLL being built must be placed in memory.

`-d <hexadecimal number>`

Use the specified value as the starting address of the data (constant) segment. See Specifying the Preferred Location of a Loadfile in Virtual Memory on page 5-8.

`-data_resident`

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

`-dll`

This is a synonym for `-shared`.

`-dllname`

This is a synonym for `-soname`.

`-e <symbol name>`

Use the address of the specified procedure as the main entry point. See Designating the Main Entry Point of Your Program on page 5-4.

`-error_unresolved`

This is a synonym for `-unres_symbols error`.

`-export`

This is a synonym for `-exported_symbol`.

`-export_all`

Export all symbols that you might normally want to have exported without naming them explicitly. See Controlling Which Symbols Your Loadfile Exports on page 5-5.

`-exported_symbol <symbol name>`

Export the specified symbol from the loadfile being created. See Controlling Which Symbols Your Loadfile Exports on page 5-5.

`-export_not`

This is a synonym for `-hidden_symbol`.

`-first_L <filename>`

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 Where The Linker Searches for Libraries and Archives on page 2-11.

`-FL`

This is a synonym for `-obey`.

`-grow_data_amount <hexadecimal number>`

Leave the specified amount of slack space in virtual memory for the data of this DLL.

`-grow_limit <hexadecimal number>`

Use the specified value as the total amount of memory reserved for this DLL.

`-grow_percent <number>`

Leave the specified percentage of slack space in virtual memory for each of the text and data segments of this DLL.

`-grow_text_amount <hexadecimal number>`

Leave the specified amount of slack space in virtual memory for the text of this DLL.

`-hidden_symbol <symbol name>`

Do not export the specified symbol. See Controlling Which Symbols Your Loadfile Exports on page 5-5.

`-import_lib <filename>`

Build a complete or incomplete import library with the specified filename in addition to creating a new DLL.

`-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.

`-include_whole`

This is a 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.

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

`-L <filename>`

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 uppercase. See Where The Linker Searches for Libraries and Archives on page 2-11.

`-l <filename>`

Use the specified filename to locate a DLL or archive. The “`-l`” must be specified in lowercase. See Libraries the Linker Searches For and Opens on page 2-10.

`-lib`

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

`-libname`

This is a synonym for `-set libname`.

`-libvol`

This is a 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 The Link-Time-Defined Search Path of the Loader on page 2-17.

`-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.

`-m`

This is a synonym for `-map`.

`-make_implicit_lib`

Mark the DLL being created as an implicit library.

`-make_import_lib <filename>`

Create a complete or incomplete import library with the specified filename, to represent the other DLL or DLLs whose filenames are found in the command stream.

`-map`

Produce a map showing how memory has been laid out. See Command Stream Requests for Linker Messages on page 5-12.

`-must_preset`

Consider it an error if presetting fails.

`-must_use_iname`

`eld` reports an error if the linker is not able to delete an existing file of the same name when creating an import library.

`-must_use_oname`

`eld` reports an error if the linker is not able to delete an existing file of the same name when creating its main output object file.

`-must_use_rname`

`eld` reports an error if the linker is not able to delete an existing file of the same name when it is recreating a private DLL registry.

`-no_include_whole`

This is a synonym for `-none`.

`-none`

Only include archive members in the link if they satisfy needed references.

`-no_optional_lib`

Do not consider later DLLs in the command stream to be optional.

`-no_preset`

Do not preset the loadfile being created.

`-no_reexport`

Do not re-export DLLs found after this point in the command stream.

`-nostdfiles`

At runtime, do not automatically open the standard C I/O files.
See Changing Run-Time Options for C and C++ Programs on page 5-15.

`-no_stdfiles`

This is a synonym for `-nostdfiles`.

`-nostdlib`

Do not look in the standard places for DLLs and archives. See Where The Linker Searches for Libraries and Archives on page 2-11.

`-no_stdlib`

This is a synonym for `-nostdlib`.

`-noverbose`

This is a synonym for `-no_verbose`.

`-no_verbose`

Do not show warnings or informational messages unless they are requested by a linker option. See Message-Control Options on page 5-12.

`-o <filename>`

Use this as the name of the output object file. See Choosing the Output File on page 2-3.

`-obey <filename>`

Use the specified file as an obey file. See Direct Use of the Linker on page 2-1.

`-optional_lib`

Consider later DLLs in the command stream to be optional.

`-public_registry <filename>`

Use the specified file as the public DLL registry file.

`-r`

Create a linkfile rather than a loadfile.

`-reexport`

Re-export DLLs found after this point in the command stream. See How to Make Your Loadfile Re-Export Symbols of Other DLLs on page 2-15.

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

Change the name of a symbol while creating a new file.

`-rld_first_L <path>`

The string specified by <path> should be a list of directories or subvolumes separated by colons. At runtime, the specified directories or subvolumes are places where `rld` will look for DLLs before it looks for public DLLs. SeeThe Link-Time-Defined Search Path of the Loader on page 2-17.

`-rld_L <path>`

The string specified by <path> should be a list of directories or subvolumes separated by colons. At runtime, the specified directories or subvolumes are places where `rld` will look for DLLs after it looks for public DLLs. See The Link-Time-Defined Search Path of the Loader on page 2-17.

`-rpath`

This is a synonym for `-rld_L`.

`-s`

Omit the DWARF symbol table when creating the output file.

`-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 and their defaults.

**Table A-1. Set Attributes**

| Attribute Name | Allowable Values | Default |
|---|---|---|
| CPPDialect \| CPlusPlusDialect | neutral \| v2 \| v3 | The value comes from the input linkfiles |
| data_model | ILP32, LP64 and neutral | ILP32 |
| floattype | ieee \| neutral \| tandem | The value comes from the input linkfiles |
| float_lib_overrule | on \| off | off |
| heap_max | <hexadecimal number> | 0 |
| highpin | on \| off | on |
| highrequester \| highrequesters \| highrequestor \| highrequestors | on \| off | on |

**Table A-1. Set Attributes** (continued)

| Attribute Name | Allowable Values | Default |
|---|---|---|
| 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. |
| inspect | on \| off | on |
| 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. |
| mainstack_max | <hexadecimal number> | 0 |
| oktosettype | on \| off | off |
| pfs \| pfssize | <number> | Option is a no-op. |
| process_subtype \| subtype | <number> | 0 |
| rld_unresolved | error \| warn \| ignore | error |
| runnamed | on \| off | off |
| saveabend | on \| off | off |
| space_guarantee | <hexadecimal number> | 0 |
| systype | guardian \| oss | (depends on the platform) |

`-shared`

Create a DLL. See Choosing to Create a Program or a DLL on page 2-3 .

`-show_multiple_defs`

Put information into the listing about symbols that are defined in more than one of the input linkfiles. See Command Stream Requests for Linker Messages on page 5-12.

`-soname <filename>`

Specify the DLL name for the DLL being created. See Naming DLLs on page 2-3 .

`-stdin`

Use the standard input file as an obey file. See Direct Use of the Linker on page 2-1.

`-strip <filename>`

Remove the DWARF symbol table from an existing loadfile or import library.

`-t <hexadecimal number>`

Use the specified value as the starting address of the text segment of the loadfile being built. See Specifying the Preferred Location of a Loadfile in Virtual Memory on page 5-8.

`-temp_i <filename>`

Use the specified filename as the name of the intermediate file during the creation of an import library.

`-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 Naming Intermediate Linker Output Files on page 5-11.

`-temp_r <filename>`

Use the specified filename as the name of the intermediate file during the recreation of a DLL registry.

`-u <symbol name>`

Consider the specified symbol to be needed when deciding which files to take from archives. See  Availability of Linkfiles from Archives on page 2-8.

`-ul`

Create a user library. In effect, this option is a synonym for `-shared` plus `-export_all`. See Controlling Which Symbols Your Loadfile Exports on page 5-5.

`-unres_symbols { error | ignore | warn }`

Handle unresolved references in the way specified. See Making the Linker Look for Unresolved Symbols on page 5-3.

`-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.

`-verbose`

Show all messages. See Message-Control Options on page 5-12.

`-warn`

Show all error and warning messages. See Message-Control Options on page 5-12.

`-warning_unresolved`

This is a synonym for `-unres_symbols warn`.

`-x`

Omit the DWARF symbol table when creating the output file.

`-y <symbol name>`

Provide information about how this symbol is mentioned in the ELF symbol tables of the linker’s input files. See Command Stream Requests for Linker Messages on page 5-12.

# 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 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.

**iniTerm Lists.** Lists of initialization and termination functions used in the support of runtime dynamically-loaded libraries on the HP NonStop operating system.

**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 processes 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.** This 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.

**Neutral loadfile.** This can be loaded with either a 32-bit or 64-bit program.

**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. PIC is 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(ped) 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)<br>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 identifier used to identify which version of the product you are using.

**Zimpimp file.** The name of the imp-imp file on a system is $SYSTEM.<SYSnn>.ZIMPIMP. Also called the "import library that represents the implicit DLLs", it is the file that tells which symbols are available in the set of implicit DLLs, which collectively correspond to what was previously called the system library. See also Implicit library import library (imp-imp).

**Zreg file.** This is the name of the public DLL registry file, which lists the names of all the public DLLs.

# Index