eNOFT Manual

Abstract

This document describes the external features of eNOFT, a stand-alone utility that displays object files native to TNS/E systems. This document also describes the external features and usage of the ar archive facility that may be used on either TNS/R or TNS/E systems.

Product Version

N/A

Supported Release Version Updates (RVUs)

This manual supports H06.03 and all subsequent H-series release version updates until otherwise indicated in a replacement manual.

Part Number	Published
527507-005	February 2013

Document History

Part Number	Product Version	Published
527507-002	N/A	November 2004
527507-003	N/A	May 2005
527507-004	N/A	July 2005
527507-005	N/A	February 2013

Legal Notices

eNOFT Manual

Glossary Index Tables

Tables

What's New in This Manual

Manual Information

Abstract

This document describes the external features of `eNOFT`, a stand-alone utility that displays object files native to TNS/E systems. This document also describes the external features and usage of the `ar` archive facility that may be used on either TNS/R or TNS/E systems.

Product Version

N/A

Supported Release Version Updates (RVUs)

This manual supports H06.03 and all subsequent H-series release version updates until otherwise indicated in a replacement manual.

Part Number	Published
527507-005	February 2013

Document History

Part Number	Product Version	Published
527507-002	N/A	November 2004
527507-003	N/A	May 2005
527507-004	N/A	July 2005
527507-005	N/A	February 2013

New and Changed Information

Changes to the 527507-004 Manual

- This is a new manual. Displays have been updated between 001 and 002 EAP versions.

Changes to the 527507-005 Manual

- Added some additional information now displayed by the LISTATTRIBUTE DETAIL command on page 2-41.
- Replaced the example completely with the latest example output on page 2-42.

About This Manual

This manual explains how to use the following TNS/E native object file utilities:

- `eNOFT`, which displays object files
- `ar`, which creates and maintains archives of object files

Subsections:

Audience

This manual is intended for systems programmers and application programmers who are familiar with the following:

- HP NonStop™ servers
- NonStop Kernel (NSK) operating system
- The compilers and linkers supported by the NSK operating system running on TNS/E processors.
- Object file formats supported by the NSK operating system; see Appendix A, TNS/E Native Object Files in this manual.
- Reference manuals and programmer's guides for the languages in which programs are written (C/C++, COBOL, and pTAL).

Related Reading

- eld Manual
- COBOL85 Manual for TNS/E

Notation Conventions

This section contains generic information. For notations specific to `eNOFT`, see Common Formats of Command Arguments on page 1-3.

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

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

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

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

Notation for Messages

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

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

```
ENTER RUN CODE

?123

CODE RECEIVED:      123.00
```

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

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

```
Backup Up.
```

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

p-register

process-name

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

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

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

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

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

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

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

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

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

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

```
%005400
```

```
%B101111
```

```
%H2F
```

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

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.         !o
```

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 Introduction to eNOFT and ar

The `ar` utility creates and maintains archives composed of groups of object files. You can mix PIC and non-PIC files in an archive, but you may not mix TNS, TNS/R or TNS/E object files within the same archive. After an archive has been created, new files can be added and existing files can be extracted, deleted, or replaced. For details on the use of `ar`, see Section 3, The ar Utility and Section 5, ar Diagnostic Messages.

The rest of the sections of this manual are concerned with `eNOFT`, the object file reader. `eNOFT` reads and displays linkfiles, loadfiles and import libraries created by the TNS/E compilers, assembler and linker.

`eNOFT` is comparable to the `NOFT` utility that runs on TNS/R systems. See Appendix B, Differences Between eNOFT and NOFT for a detailed comparison of the two utilities. At each command description in this manual you will also be given information about the different options or actions of the two utilities.

Output of `eNOFT` can be sent to the display terminal or to file types native to that environment. For TNS/E native versions, this requires presence of the Common Run-Time Environment (CRE) and C runtime library (Crtl) in the Guardian and the Open System Services (OSS) environments. For the Windows PC version, `eNOFT` runs at the cmd.exe command prompt; no GUI is supplied.

`eNOFT` has the same capabilities in each environment (Guardian, OSS, PC), but each environment may require slightly different syntax. Differences, if any, are often related to the different types of file names and file characteristics on each of these platforms.

`eNOFT` commands are organized into four categories:

1. SET and RESET commands that control the format of output; for example, raw or formatted forms
2. dump commands that displays specific parts of the object file
3. list commands that organizes and lists attributes from various parts of the object file
4. user interface commands to specify the input object and control the `eNOFT` environment

These four categories are used to classify the complete list of commands and command options shown in Section 2, eNOFT Options.

Starting eNOFT

The `eNOFT` utility can be used as an interactive or batch process.

Interactive (Command-Line) Mode

`eNOFT` launches and runs as an interactive process if no command is specified. In addition, the PC version of `eNOFT` can also be launched by double-clicking on its file name.

`eNOFT` can perform most tasks from the terminal prompt by specifying the commands serially on the command line. `eNOFT` terminates after the last command is processed. Unless NOEXIT is specified, the EXIT command is not required to terminate the `eNOFT` process.

In Guardian,

```
ENOFT [ <COMMAND1>; <COMMAND2>;  ... ]
```

whereby commands must be separated by the delimiter semicolon ";". An example follows:

```
>ENOFT FILE $VOL.SUBVOL.HELLOOUT; SET FORMAT INNERLIST; DUMPCODE
```

For OSS and PC Windows, if the commands contain characters that are considered special to that shell environment such as spaces and semicolons, the commands must be surrounded by double quotes or prefixed with backslashes to prevent the shell environment from trying to parse the `eNOFT` commands. See examples for OSS below:

```
>enoft "file /usr/subdir/hello.out; set format innerlist; dumpcode"
```

or

```
>enoft file /usr/subdir/hello.out\; set format innerlist\; dumpcode
```

Alternately, commands may be prefixed with a hyphen in OSS and PC Windows:

```
enoft [ command1 -<command2> ... ]
```

whereby commands must be separated by delimiter blank space and prefixed with a hyphen " -". Use of the prefix for the first command is optional.

An example on PC Windows,

```
>enoft file c:\dir\hello.exe -set format innerlist -dumpcode
```

Batch (Command File) Mode

`eNOFT` can be run as a batch process by issuing commands through a command (obey) file as follows:

In Guardian:

```
ENOFT / IN INFILE [, OUT OUTFILE ] /
```

In OSS or PC Windows:

```
enoft < infile [ > outfile ]
```

`infile` is a text file containing one or more `eNOFT` commands. Commands listed in this file must follow the same rules for the command-line processing. If infile does not exist, a syntax error will be generated and the program terminates normally.

If both an infile and commands are given, `eNOFT` ignores the infile.

`outfile` specifies a file name to which `eNOFT` writes its output. If the specified file does not exist, `eNOFT` creates a text file with that name. If the specified file exists, `eNOFT` appends to the file. `eNOFT` writes its output to the display terminal if no output file is specified.

Common Formats of Command Arguments

The following lists common rules associated with `eNOFT` commands. Exceptions, if any, are listed with individual commands.

Argument Groupings

[ ] denote grouping of optional arguments. The first argument in the options list is the default.

{ } also denote groupings of arguments except at least one argument is mandatory.

Input Format

All commands are limited to 256 characters.

For commands that have the asterisk symbol "*" as an option, `NOFT` displays syntax errors when these commands are typed without an option.

New to `eNOFT`, all available or applicable items will be shown when these commands are entered with or without an asterisk.

The (default) `NOFT` options "BRIEF" and "B" are not supported.

Output Display Format

Most outputs of `eNOFT` will be left justified instead of center-justified in `NOFT`.

All displays are limited to 79 characters per line. If an output file is specified, information in excess of this limit will wrap around the following line to maximum number of characters per line allowed by that output file type; for example, 239 characters for the EDIT file type in Guardian. In such a case, information may be truncated.

Unless otherwise specified, dumps are presented in multiples of 32-bit values; for example, 10 decimal digits for decimal and 8 hexadecimal digits for the hexadecimal format.

All virtual addresses are represented as 32-bit hexadecimal numbers (that is, 8 hexadecimal digits) whereby the top halves of the 64-bit addresses are ignored . If the top half of the 64-bit address was not the sign extension of the bottom half, `eNOFT` will give a warning message with the truncated display.

Hexadecimal numbers are prefixed with "0x".

Sizes are represented as decimal values in bytes.

All code outputs are in multiples of 16-byte bundles of instructions.

<proc-spec>

This form of argument specifies the procedure or subprocedure.

```
<proc-num> | procname
```

`<proc-num>` is the procedure number that is available from LISTPROC or LP on page 2-48.

`procname` is the procedure name and is case-sensitive in C and C++ but not in COBOL or pTAL. This option limits the scope to the specified procedure and its subprocedures. Note the demangled form of the procname cannot be used because `eNOFT` does not support blank spaces in the name.

New to `eNOFT`, a wildcard input string may be entered to search for items containing a match to the given pattern. For SET SCOPEPROC or SSP on page 2-6 and DUMPPROC or DP on page 2-13, only the first item that matches the given pattern will be shown.

For pTAL, procname.subprocname or subprocname limits the scope to the specified subprocedure. For COBOL, subprograms beyond the second level can also be specified.

Only the first subprocedure identified is displayed if there are multiple subprocedures with the same name. To display other subprocedures, their fully qualified names (`procname.subprocname`) must be specified.

The scope of procedures to display is determined by SET SCOPESOURCE or SSS on page 2-7 or SET SCOPEPROC or SSP on page 2-6 if this option is not used. If no global scope is set, all procedures are shown.

<source-spec>

Native mode object files are loosely organized in the order compilation units were compiled into a particular object file. This organization is prevalent in the symbols for the object file and a loose correlation can also be made for other parts of the object file. Sometimes this organization can be taken advantage of by using the SET SCOPESOURCE command to restrict eNOFT to looking at things related to only a certain compilation unit. A compilation unit typically represents a single relocatable object file specified to the compiler to form an executable object. This relocatable object file may be derived from several source files, including preprocessed “include files." The <source_spec> form of argument specifies the compilation unit entry in the object file.

```
   <sourcendx> | sourcename
```

`<sourcendx>`

compilation unit index available from LISTSOURCE.

`sourcename` may specify a fully qualified path (for example, node name, volume, and subvolume for Guardian) or a filename only.

New to eNOFT, wildcard input string may be entered to search for items containing a match to the given pattern. For SET SCOPESOURCE, a list of items that matches will be shown if multiple items match.

<format-specifier>

This form of argument specifies how the information is to be formatted.

```
READABLE | R | ASCII | A | DECIMAL | D | HEX | H | ICODE | IC
| INNERLIST | IN
```

See SET FORMAT or SF on page 2-3 for details on these options.

<scope-range>

This form of argument specifies the display range in numerical value.

```
   <start> [ TO <end> |
           FOR { <number> | * } [ UNITS | U | BYTES | B ] ]
```

BYTES are in multiples of 8-bit values and UNITS as either 4 or 16 bytes, depending on the part of the object file. New to `eNOFT`, this unit of output is in multiples of 16-byte bundles of code instructions when accessing any code section, otherwise in multiples of 4 bytes all other sections.

`<start>`and `<end>` values must be in the form 0XXXXXXXXX where X is a hexadecimal digit. New to `eNOFT`, values must be on a 16-byte boundary when displaying in ICODE or INNERLIST formats. For other formats, any value within a valid range for that display is acceptable. New to `eNOFT`, the `<start>` no longer needs to be on a 4-byte or 16-byte alignment; that is, `eNOFT` will round down to the beginning of the bundle when dumping within a code section.

If `<end>` is not specified, one "unit" of information will be displayed.

For the DUMPADDRESS command, `eNOFT` will display information to the end of the section if the asterisk option "*" is specified else to the end of file for the DUMPOFFSET command. New to `eNOFT`, section headings will be displayed when trespassing into another section. For the DUMPPROC command, `eNOFT` will display information either to the end of the procedure or subprocedure if `<proc_spec>` is specified.

If `<number>` is specified without any keyword, output will be in appropriate "units" of bytes, depending on the part of the object file. New to `eNOFT`, `<number>` defaults to decimal unless its value is prefixed with 0x for hexadecimal.

Notes:

- `<start>` and <end> values differs for DUMPADDRESS, DUMPOFFSET and DUMPPROC. For DUMPADDRESS, the values denote the virtual address range set in the object file. For DUMPOFFSET, these values denote the file offset values whereby 0x0 denotes the beginning of the file. Use LAYOUT to find the starting value of a section for DUMPADDRESS and DUMPOFFSET. For DUMPPROC, the values denote the relative offset from the procedure containing the address range; that is, 0x0 denotes the beginning of the specified procedure.
- The name of the procedure that the starting address points to will be given in the heading. If the address range crosses into another procedure, a line displaying that procedure name will be inserted in the output.
- Except for the DUMPOFFSET command, the range of addresses must be entirely within one section. If `<proc_spec>` is specified, the range of address must be entirely within one procedure. If an address range ends outside this section or procedure, `eNOFT` will stop at the end of that section (or procedure for DUMPPROC) and a warning message will be emitted. When displaying in ICODE or INNERLIST formats, the address range will be rounded up to a whole number of 16-byte bundles of code instructions. `eNOFT` will display information either to the end of the procedure or subprocedure if `<proc_spec>` is specified or to the end of the section if no range is specified; for example, `<start>` without `<end>`, or with asterisk option "*".

2 eNOFT Options

This section contains the following information:

SET and RESET Commands - globally set how and where eNOFT will dump specified parts of the object file.

Dump Commands - display contiguous parts of the object file.

List Commands - organize and list specific sections of the object file.

File Handling Commands - affect the user interface.

SET and RESET Commands

`SET` and `RESET` commands globally set how and where `eNOFT` will dump specified parts of the object file. These global commands affect the output display of dump and list commands and remain in force until the specified `SET` command is changed or reset.

- `SET <set-cmd> <argument>` specifies the set-cmd setting where <argument> is a mandatory parameter. Use of the `SET <set-cmd>` without an argument will generate a syntax error message.
- `RESET <set-cmd>` sets the current `SET <set-cmd>` setting back to the default setting. The `NOFT` argument `OFF` is not supported.

All `SET` and `RESET` commands will echo back the new setting, thus acting as a check for the user.

The `NOFT` argument "?" is not supported. Use `SHOW <set-cmd>` to display the current setting for `set-cmd` and `SHOW` without any option to display all `SET` settings.

RESET

```
RESET [ * | set-cmd ]
```

This command resets one or more `eNOFT` target object file parameters to their default values.

This command sets all prior `SET` command settings to their respective defaults if the asterisk option "*" is specified. New to `eNOFT`, typing `RESET` without any option also resets all the `SET` command settings.

SET

```
[ set-cmd [ argument ] ]
```

This command displays the current setting for one or all <`set`-cmd> attributes of the target object file, including `DEMANGLE` if C++ sources are opened and within scope.

New to `eNOFT`, `SET <set-cmd>` without an argument will echo the current setting of `<set-cmd>` and `SET` by itself shows all `<set-cmd>` settings.

`SET <set-cmd> <argument>` sets the set-cmd to argument.

The default display shows all current `SET` commands settings:

```
enoft> set
Sorting:          (none)
Formatting:       Readable
Current Scope:    (none)
Input Case:       Sensitive
History Size:     50
History Window:   10
Lines Per Page    0
Brief/Detail:     (none)
C++ Demangle:     Off
```

SET CASE or SC

```
SC { OFF | ON }
```

This command sets case sensitivity for source names, procedure names, and path names.

The default setting is "Not Sensitive". `eNOFT` becomes case-sensitive on opening of the object file with C/C++ code and case insensitive for all other object files, regardless of the platform environment `eNOFT` runs on. `RESET CASE` sets case sensitivity to the default setting, regardless of the current object file.

`OFF` forces this setting to "Not Sensitive" and vice versa for `ON`. Note certain source files or procedures may not be matched correctly depending on the language used in the object file.

This setting is automatically set to the default setting upon opening a new object.

For object files with C/C++ and non-C/C++ source codes, if proc or source scoping restricts to a source file that does not have C/C++ code, `CASE` automatically reverts to "Not Sensitive" mode and returns to "Sensitive" when restricted scoping is removed.

SET DEMANGLE or SDE

```
SDE [ ON | OFF ]
```

This command sets C++ symbols name display to demangled format.

With the default “OFF”, names are displayed in the mangled format. This command is only applicable to files with C++ symbols.

SET DISPLAY OR SD

```
SD [ BRIEF | B | DETAIL | D ]
```

New to `eNOFT`, this command globally sets the display format to `BRIEF` or `DETAIL` and is applicable to the following commands: `COMP, RTDU, UNWINDINFO, LISTATTRIBUTE, LISTCOMPILERS, LISTDEBUG, LISTPROC, LISTSOURCE, LISTUNRESOLVED, LISTUNREFERENCED,` and `XREFPROC`.

In the default format, “none”, queries of single records (for example, procedure specified) will be displayed in `DETAIL` format whereas queries resulting in multiple records will be displayed in `BRIEF` format. This command overrides this default behavior resulting in one consistent format regardless of the result of the query.

SET FORMAT or SF

```
SF { READABLE | R | ASCII | A | DECIMAL | D | HEX | H | ICODE |
IC | INNERLIST | IN }
```

This command sets the output display format.

`READABLE or R` is the default format. In this mode, `eNOFT` determines the applicable format based on the item type and part of the object file being displayed. For example, `eNOFT` displays code sections in disassembled program code (ICODE), ASCII format for literals, hexadecimal format for virtual addresses, and decimal for non-address values.

`ASCII or A` displays the specified part of the object file in ASCII text format. The following sample shows a continuous dump of 64 bytes per line:

```
0x70000420:  '......4...8...<<...<.p.$.2@....$$..<...<.p.$.2H....$$..<...<.p.
0x70000460:  $.2T....$$..<...<.p.$.2\....$$..<...<.p.$.2h....$$..<...<.p.$.2l
```

`DECIMAL or D` displays the specified part of the object file in decimal format. The following sample shows 16 bytes per line in chunks of ten-digit decimal values:

```
0x70000420:  0666763208 2948530228 2946760760 2946826300      '... ...4 ...8
...<
0x70000430:  1006700544 1006989312 0614806080 0243237392      <... <.p. $.2@
....
```

`HEX or H` displays the specified part of the object file in hexadecimal format. The following sample shows 16 bytes per line in chunks of eight-digit hexadecimal values:

```
0x70000420:  27bdffc8 afbf0034 afa40038 afa5003c          '... ...4 ...8 ...<
0x70000430:  3c010800 3c057000 24a53240 0e7f8210         <... <.p. $.2@ ....
```

`ICODE or IC` displays the specified part of the object file in disassembled code. A syntax error may generate if used in an area not corresponding to actual instructions. The following sample shows 2 bundles of code instructions. Note the values

"2:001:0013:0" shown next to the procedure "main" denotes the source index, file index, line number (ordinal shown here), and bundle instruction, respectively:

```
[  2:001:0013:0  main]
0x70010cc0: {0: 02c00916800 M alloc r32 = ar.pfs, 0x09, 0x00, 0x02, 0x00
             1: 00000000000 M adds sp = -48, sp
             2: 10800100880 I adds r34 = 0x00, gp ;;
             template: 0x09}

[   :   :    :   main]
0x70010cd0: {0: 00008000000 M nop.m 0x00
             1: 00000000000 I mov r33 = rp
             2: 00008000000 I nop.i 0x00 ;;
             template: 0x01}
```

INNERLIST or IN displays the specified part of the object file in disassembled program code with the source code interspersed throughout the dump. The source code will not be shown if the source file is not at the path specified when the object file was built. This option only applies to code dumps and reverts to the default format for all other dumps. New to eNOFT, object files compiled on OSS for Guardian target execution ("System Type" in LISTATTRIBUTE DETAIL) can now have the source code (remaining) on OSS be shown in the Guardian platform.

```
******** Innerlist Dump of Procedure main

[Src:Fil:Line:i  Procedure]
MemAddress   Contents
---------------------------------------------------------------------

DW_LINE File: 1 d:\temp\foo.cpp

   13 {
[  2:001:0013:0  main]
0x00000000: {0: 02c00916800 M alloc r32 = ar.pfs, 0x09, 0x00, 0x02, 0x00
             1: 00000000000 M adds sp = -48, sp
             2: 10800100880 I adds r34 = 0x00, gp ;;
             template: 0x09}
```

```
[    :    :     :    main]
0x00000010: {0: 00008000000 M nop.m 0x00
             1: 00000000000 I mov r33 = rp
             2: 00008000000 I nop.i 0x00 ;;
             template: 0x01}

[    :    :     :    main]
0x00000020: {0: 10800c60500 M adds r20 = 0x030, sp ;;
             1: 00000000000 M st8 [r20] = r34
             2: 00008000000 I nop.i 0x00
             template: 0x0a}

   14    T399myClass<long> thisMyClass;
[  2:004:0014:0  main]
0x00000030: {0: 00008000000 M nop.m 0x00
             1: 00000000000 I nop.i 0x00
             2: 0b1fe7db000 B br.call.sptk.many rp = $-49456 ;;
             template: 0x11}

[    :    :     :    main]
0x00000040: {0: 10800c60500 M adds r20 = 0x030, sp ;;
             1: 00000000000 M ld8 r34 = [r20]
             2: 00008000000 I nop.i 0x00 ;;
             template: 0x0b}
```

When specified, the local command format option `<format_spec>` will take precedence over this command.

SET HISTORYBUFFER or SHB

```
SHB <num>
```

This command sets maximum number of command lines available for retrieval by the `HISTORY` command.

`<num>` specifies the number of command lines in the history buffer. New to `eNOFT`, this argument is mandatory.

The default is 50 with a range from zero to 65535.

Notes:

- If the buffer size is reduced, the number of command lines in storage is truncated with the oldest commands in this queue deleted first (FIFO). The command lines deleted from the buffer are not retrievable.
- If the buffer size is increased, commands in queue are retained with additional buffer allocated to handle up to the specified buffer size.

SET HISTORYWINDOW or SHW

```
SHW <num>
```

This command specifies the number of command lines displayed with the `HISTORY` command.

`<num>` changes this command window to show the specified number of commands. New to `eNOFT`, this argument is mandatory.

The default value is 10 with a range from zero (0) to 50.

SET LINES or SL

```
SL <num>
```

This command specifies the number of lines to display before pausing so that an area of output does not scroll beyond the terminal display line size.

`<num>` changes the number of lines that will be shown before a prompt is given to continue. New to `eNOFT`, this argument is mandatory.

This command is not applicable in the command-line mode.

The default value is zero and causes output to continue until all results are displayed. The range is from 0 to 65535.

SET SCOPEPROC or SSP

```
SSP <proc-spec>
```

This command narrows the scope to a single procedure or subprocedure and affects the following commands: `LISTCOMPILERS, LISTDEBUG, LISTEXPORTS, LISTOPTIMIZE, LISTPROC, LISTSOURCE, LISTUNRESOLVED, LISTUNREFERENCED, DWARF INFO, SYMTAB, UNWIND, UNWINDINFO, XREFPROC,` and `DUMPALL.`

The following commands are not affected by this command: LISTDATA, RELOC.

The default setting is "none" ; all procedures (and subprocedures as applicable) are considered.

This command is automatically reset upon opening a new object file.

Use of this command or its equivalent `RESET` command overrides any existing `SET SCOPESOURCE` setting.

This command takes precedence over the local command scope option `<proc-spec>` in event of a conflict.

The alias `NOFT` command `PROC` is not supported.

SET SCOPESOURCE or SSS

```
SSS <source-spec>
```

This command narrows the scope to a single source file. This is helpful when trying to find unique items within a source file and to limit the output to a range within the designated scope.

The following commands are affected by this command: `LISTCOMPILERS, LISTDEBUG, LISTPROC, LISTSOURCE, LISTUNRESOLVED, LISTUNREFERENCED, UNWIND, UNWINDINFO, DWARF, SYMTAB, LISTEXPORTS, LISTOPTIMIZE,` dumps in `ICODE` or `INNERLIST formats, XREFPROC,` and `DUMPALL.`

The default setting is "none" ; all source files are considered.

This command is automatically reset upon opening a new object file.

Use of this command or its equivalent `RESET` command overrides any existing `SET SCOPEPROC` setting.

New to `eNOFT`, this command takes precedence over the local command scope option <proc-spec> in event of a conflict.

The alias `NOFT` command `SOURCE` is not supported.

SET SORT or ST

```
ST { NONE | N | ALPHA | A | LOC | L }
```

This command specifies the sorting order of the output.

It is applicable to the following commands: `PROCINFO, SYMTAB, UNWIND, UNWINDINFO, LISTDEBUG, LISTEXPORTS, LISTOPTIMIZE, LISTPROC, LISTSOURCE (ALPHA` only`), LISTUNREFERENCED (ALPHA` only`), LISTUNRESOLVED (ALPHA` only`),` and `XREFPROC (LOC` only`).`

`NONE or N` is the default order of sorting. This order is determined by the system and depends on the applicable table and whether the items therein are procedures, source files, or other attributes.

`ALPHA or A` sorts the output in alphabetical order. Depending on the `eNOFT` command, the names being alphabetized may be source file names, procedure names, or similar attributes.

`LOC or L` sorts the output in virtual address order and is not applicable to linkfiles. Often the addresses being sorted are addresses of procedures. When source files are being sorted, the address of the source file is the same as the address of the first procedure or subprocedure in the source file.

Dump Commands

The dump commands display contiguous parts of the object file. The following are methods to narrow the scope of the display:

- `DUMPADDRESS` sets the virtual address.
- `DUMPOFFSET` sets the offset within the object file.
- `DUMPPROC` sets the procedure or subprocedure name.
- `DUMPSECTION` sets the object file section.

Additional commands further organize sections by similar interests (for example `DUMPCODE, DWARF, RTDU`) or serve as aliases to `DUMPSECTION` for common sections (for example `DYNAMIC` and `GOT`).

Unless otherwise specified, the default format is `READABLE`.

DUMPALL or ALL

```
DUMPALL [ * | LIST ]
```

This command displays all non-zero size sections in the object file in order of their relative offsets. It expands the command `LAYOUT` to show the section contents. Unlike `DUMPSECTION * DETAIL`, this command includes displays of the file, program and section headers, and stops on any command that fails, for example with a data error.

The asterisk additionally displays all applicable list commands after displays of all sections in the object file.

`LIST` restricts the display to only applicable List Display commands, so that it complements `DUMPSECTION * DETAIL.`

DUMPADDRESS or DA

```
DA <scope-range> [ IN <format-specifier> ]
```

This command displays the object file contents from a virtual address inside a loadfile's memory space and is applicable to loadfiles and import libraries. Use `SECTHDRS` or `LAYOUT` to locate the starting address and size of any particular section.

The following sample shows two 16-byte bundles of code instructions for the .text section, automatically shown in format ICODE because this is in a code area.

```
enoft> dumpaddress 0x70000420 for 2

 ******** Readable Dump Of .text Section (File Offset 0x0390)

[demo                    2]::
0x70000420: {0: 08082000800 M ld4     r32 = [r32]
             1: 10801800dc0 I adds  r55 = 0x0, r24
             2: 10800a40b00 I adds  r44 = 0x20, ret2
             dispersal:   00}
[demo                    2]::
0x70000430: {0: 00008000000 M nop.m   0x0
             1: 00008000000 F nop.f  0x0
             2: 12000006440 I addl  r17 = 0x3, r0 ;;
             dispersal:   0d}
```

Note that if a single bundle is specified and no source file and line number information is available for that bundle, `eNOFT` will display from nearest preceding bundle with available line info.

DUMPCODE or DC

```
DC [ BRIEF | B | IN <format_spec> ]
```

New to `eNOFT`, this command displays all available code in the object file: .gateway, .plt, and every section that begins with ".text" or ".restext". Use `DUMPSECTION` to display individual code sections.

`BRIEF` or `B` is an alias to command `LAYOUT CODE`.

`<format_spec>` defaults to the ICODE format if not specified. See `SET FORMAT` for details.

The following sample shows the content of the .plt section of a loadfile in the default ICODE format. Each code section displayed will be prefixed with its name in the header.

```
enoft> dumpcode

******** Section .plt in Icode (File Offset 0x800)

(Source file not located for this symbol; no proc or
 or src line number is available for this kind of symbol.)

Src:Fil:Line:i  Procedure]
MemAddress   Contents
---------------------------------------------------------------------
[   :   :    :  #import_stubs]
0x70000800: {0: 120001803c0 M addl r15 = 0x040, gp ;;
             1: 00000000000 M ld8 r16 = [r15], 0x08
             2: 10800100380 I adds r14 = 0x00, gp ;;
             template: 0x0b}

[   :   :    :  #import_stubs]
0x70000810: {0: 080c0f00040 M ld8 gp = [r15]
             1: 00000000000 I mov b6 = r16, $+0x0
             2: 0010000c000 B br.cond.sptk.few b6 ;;
             template: 0x11}

[   :   :    :  #import_stubs]
0x70000820: {0: 120001a03c0 M addl r15 = 0x050, gp ;;
             1: 00000000000 M ld8 r16 = [r15], 0x08
             2: 10800100380 I adds r14 = 0x00, gp ;;
             template: 0x0b}
    . . .
```

This command replaces the following `NOFT` commands: `ALLTEXT, USERGATE, RESTEXT,` and `TEXT.`

DUMPDATA or DD

```
DD [ BRIEF | B | IN <format_spec> ]
```

New to `eNOFT`, this command displays all initialized user data sections in the object file: .data, .sdata, .rdata, and .rconst . Use `DUMPSECTION` to display individual data sections.

`BRIEF` or `B` is an alias to command `LAYOUT DATA`.

`<format_spec>` defaults to the HEX format if not specified. See `SET FORMAT` for details.

The following sample shows all available data sections in the default hexadecimal format. Each data section displayed will be prefixed with its name in the header.

```
enoft> dumpdata

******** Section .rconst in Hex (File Offset 0x7d0)

MemAddress   Contents
-----------------------------------------------------------------------
              32 zero bytes skipped.

******** Section .data in Hex (File Offset 0x2000)

MemAddress   Contents
-----------------------------------------------------------------------
0x08000000:  080001b0 01000000 02000000 00000000    .... .... .... ....
0x08000010:  aaaa4d43 4220aaaa 00000000 00000000    ..MC B .. .... ....
             112 zero bytes skipped.
0x08000090:  00000000 00000000 20010000 00000000    .... ....  ... ....

               256 zero bytes skipped.



******** Section .data1 is empty.

******** Section .rodata in Hex (File Offset 0x21a0)

MemAddress   Contents
-----------------------------------------------------------------------
0x080001a0:  25730000 00000000 00000000 00000000    %s.. .... .... ....
0x080001b0:  48656c6c 6f2c2057 6f726c64 0a000000    Hell o, W orld ....

******** Section .rdata in Hex (File Offset 0x21c0)

MemAddress   Contents
-----------------------------------------------------------------------
```

```
0x080001c0:  00000000 080001d0 00000000 00000000    .... .... .... ....

******** Section .pdata is empty.

******** Section .xdata is empty.

******** Section .sdata is empty.

******** Section .srodata is empty.

******** Section .sdata1 is empty.
    . . .
```

This command replaces the following `NOFT` commands: `DATA, READONLY, LARGEDATA,` and `SMALLDATA`.

DUMPOFFSET or DO

```
DO <scope-range> [ IN <format-specifier> ]
```

This command displays code and data range from a physical offset within the object file whereby 0x00000000 denotes the beginning of the object file. Use `LAYOUT` to find the starting offset value of a section.

The following sample shows 128 bytes of the .text section in a locally specified hexadecimal format instead of the default ICODE format for this code part of the object file.

```
enoft> dumpoffset 0x390 to 0x410 in hex

******** Hex Dump of Offset 0x00000390 To 0x0000410 ********

Offset       Mem Addr                                  Contents
--------------------------------------------------------------------------
0x00000390:  27bdffc8 afbf0034 3c010800 3c065800       '... ...4 <... <.X.
0x000003a0:  24c60068 24250000 3c04c000 0e76800c       $..h $%.. <... .v..
0x000003b0:  afa0002c 8fbf0034 00000000 03e00008       ..., ...4 .... ....
0x000003c0:  27bd0038 27bdffd8 afbf0014 3c015800       '..8 '... .... <.X.
0x000003d0:  24260060 27a40026 0e768008 27a50020       $&.` '..& .v.. '..
0x000003e0:  3c015800 8c260060 87a40026 8fa50020       <.X. .&.` ...& ...
0x000003f0:  0c00017e 00000000 0e7681d8 00402025       ...~ .... .v.. .@ %
```

DUMPPROC or DP

```
DP <proc-spec> [ <scope-range> ] [ IN <format-specifier> ]
```

This command displays some or all parts of a procedure. Use `LISTPROC` to find all available procedures or subprocedures.

The default format is ICODE.

The following shows procedure "main" in the default disassembled code format:

```
enoft> dumpproc main

******** Icode Dump of Procedure main

[Src:Fil:Line:i  Procedure]
Offset       Contents
---------------------------------------------------------------------
[  0:004:1812:0  main]
0x00000000:  {0: 02c0163c880 M alloc r34 = ar.pfs, 0x16, 0x0, 0x8, 0x0
              1: 119f0c80300 M adds sp = -192, sp
              2: 10800100900 I adds r36 = 0x0, gp ;;
              template: 0x09}

[   :   :    :   main]
0x00000010:  {0: 00008000000 M nop.m 0x0
              1: 00008000000 I nop.i 0x0
              2: 001880008c0 I mov r35 = rp
              template: 0x00}
    . . .
```

DUMPSECTION or DS

```
DUMPSECTION or DS
[ * [ DETAIL | D ] | sect_name | <sect_num> ]
[ IN <format_spec> ]
```

New to `eNOFT`, this command displays one specified section of the object file. The default display is a subset listing from the `LAYOUT` command whereby only the sections are listed; that is, it excludes file, program, and section headers.

`DETAIL` or `D` dumps all non-zero size sections in the object file. Unlike `DUMPALL`, this command excludes the file, program, and section headers and continues even if any dump command fails (for example, data error).

`<sect_num>` and `sect_name` specifies the section index and name, respectively, that are available from `SECTHDRS` or `LAYOUT`.

The following sample shows parts of the .rela.dyn section in (raw) hexadecimal format. See `RELOC` for the default `READABLE` format display of this section.

```
enoft> dumpsection .rela.dyn in hex
********** .rela.dyn Section (File Offset 0x1880)
Offset Mem Addr Contents
-----------------------------------------------------------------
0x00000000 0x00000880 00000000 00003260 00000000 0000006e
0x00000010 0x00000890 00000000 00000000 00000000 00003268
0x00000020 0x000008a0 00000000 0000006e 00000000 00000000
0x00000030 0x000008b0 00000000 000031c0 00000015 00000046
0x00000040 0x000008c0 00000000 00000000 00000000 00003240
0x00000050 0x000008d0 00000017 00000026 00000000 00000000
0x00000060 0x000008e0 00000000 000031b8 0000001a 00000026
```

DWARF

```
[ * | INFO | ABBREV | LINE [ ORDINAL ] | LOC ]
```

New to `eNOFT`, this command displays the DWARF Version 2 symbol table.

The default display shows all available DWARF sections, .debug_info, .debug_abbrev, and .debug_line.

`INFO` displays the .debug_info section which contains the debugging information entries ("dies").

`ABBREV` displays the .debug_abbrev abbreviation tables section which contains a concatenation of abbreviation tables from all compilation units.

`LINE` displays the line number table with information for code contributed to an object file. This section is referenced by a corresponding debugging information entry in the .debug_info section.

If the .debug_line section has modifications to support the EDIT line table (that is, there is a .debug_line_nsk section), the EDIT line table format will be displayed instead of the standard line table format. Use keyword `ORDINAL` to display the line number table in the standard line table format.

`LOC` displays the list for data items that can change location. This section is referenced by a corresponding debugging information entry in the .debug_info section.

Note the .debug_relocs section is displayed using the command `RELOC` and is applicable to DLLs only.

The following shows displays for all available dwarf debug sections of a loadfile:

```
enoft> dwarf
```

```
******** Section .debug_info (File Offset 0x3000)

0x0000b  DW_AB_CODE_compile_unit (0x1)
         DW_TAG_compile_unit (0x11)   DW_CHILDREN_yes
         DW_AT_               DW_FORM_         VALUE
       comp_dir (0x001b)        string (0x08)  "\SPEEDY.$DATA06.T8432H01"
       language (0x0013)         data1 (0x0b)  DW_LANG_C89
           name (0x0003)        string (0x08)
"\SPEEDY.$DATA06.T8432H01.CPLMAIN
C"
       producer (0x0025)        string (0x08)
"T0549H01_01OCT2004_CCOM_22Mar200
4_GRD 3.3 TOOLSY02 Release"
HP_compile_time (0x2024)        string (0x08)  [unsupported attr 0x2024]
      stmt_list (0x0010)         data8 (0x07)  0x0
 <unknown attr> (0x2025)         data8 (0x07)  [unsupported attr 0x2025]

0x000b0  DW_AB_CODE_compile_unit_nopc (0x2)
         DW_TAG_file_type (0x29)   DW_CHILDREN_no
         DW_AT_               DW_FORM_         VALUE
           name (0x0003)        string (0x08)  "EDIT"

0x000b6  DW_AB_CODE_compile_unit_prof (0x3)
         DW_TAG_base_type (0x24)   DW_CHILDREN_no
         DW_AT_               DW_FORM_         VALUE
      byte_size (0x000b)         data1 (0x0b)  0
       encoding (0x003e)         data1 (0x0b)  DW_ATE_signed
           name (0x0003)        string (0x08)  "void"
    . . .
******** Section .debug_abbrev (File Offset 0x3958)

0x00000  DW_AB_CODE_compile_unit (0x1)
         DW_TAG_compile_unit (0x11)   DW_CHILDREN_yes
         DW_AT_               DW_FORM_
       comp_dir (0x001b)        string (0x08)
       language (0x0013)         data1 (0x0b)
           name (0x0003)        string (0x08)
       producer (0x0025)        string (0x08)
HP_compile_time (0x2024)        string (0x08)
```

```
      stmt_list (0x0010)         data8 (0x07)
 <unknown attr> (0x2025)         data8 (0x07)

0x00015  DW_AB_CODE_compile_unit_nopc (0x2)
         DW_TAG_file_type (0x29)   DW_CHILDREN_no
         DW_AT_               DW_FORM_
           name (0x0003)        string (0x08)

0x0001c  DW_AB_CODE_compile_unit_prof (0x3)
         DW_TAG_base_type (0x24)   DW_CHILDREN_no
         DW_AT_               DW_FORM_
      byte_size (0x000b)         data1 (0x0b)
       encoding (0x003e)         data1 (0x0b)
           name (0x0003)        string (0x08)

0x00027  DW_AB_CODE_subprogram_void (0x4)
         DW_TAG_subprogram (0x2e)   DW_CHILDREN_no
         DW_AT_               DW_FORM_
 edit_decl_line (0x201e)         data2 (0x05)
      decl_line (0x003b)         data1 (0x0b)
    . . .
******** Section .debug_line_nsk (File Offset 0x3ddc)

Source: 0 \SPEEDY.$DATA06.T8432H01.CPLMAINC

Prologue:
 Length:             265
 DWARF Version:      2
 Prologue Length:    138
 Min Instr Length:   1
 Default isStmt:     0
 Line Base:          -1
 Line Range:         4
 Opcode Base:        10

File Directories:
 0 \SPEEDY.$DATA06.T8432H01
 1 \SPEEDY.$DATA06.T8432H01
 2 \SPEEDY.$DATA01
 3 \SPEEDY.$DATA01.TOOLSY02

FiNdx Dir File_Name   Size Time
    0   1 CPLMAINC
    1   2 #0062090
    2   3 STDIOH
    3   3 SYSTYPEH
    4   3 ERRNOH


ADDRESS                LINE  COL STMT BB  FILE(II)
0x0000000070000880:0    45.   0    T   F  \SPEEDY.$DATA06.T8432H01.CPLMAINC
0x00000000700008a0:1    49.   0    T   F  \SPEEDY.$DATA06.T8432H01.CPLMAINC
    . . .
******** Section .debug_loc (File Offset 0x9e4)

  OFFSET          BEGIN                END             EXPRESSION
0x00000000  0x0000000000000000  0x0000000000000020    DW_OP_regx  32

  OFFSET          BEGIN                END             EXPRESSION
0x00000024  0x0000000000000000  0x0000000000000020    DW_OP_regx  33
```

This command replaces the following `NOFT` commands: `AUXSYMTBL`, `EXTSYMTBL`, `FILETBL`, `INDFILETBL`, `LINBRTBL`, `LOCSYMTBL`, `PROCTBL`, `SYMHDR`, and `SYMBOLS`.

DYNAMIC

This is an alias for `DUMPSECTION.dynamic` and is applicable to loadfiles and import libraries.

The dynamic section contains information about other sections that are needed by the runtime loader, such as virtual addresses of the .liblist through .rela.dyn sections and sizes.

The following shows the dynamic section of a loadfile.

```
enoft> dynamic


******** Section .dynamic (File Offset 0x498)


Index Tag                                Value (e.g., Address/Size)
---------------------------------------------------------------------
    0 HASH                               0x70000f70
    1 STRTAB (.dynstr)                   0x70001630
    2 SYMTAB (.dynsym)                   0x700010f0
    3 STRSZ (.dynstr sz)                 179
    4 SYMENT (.dynsym entry sz)          24
    5 HASHVAL                            0x700016e4
    6 LIBLIST                            0x700005d8
    7 LIBLISTNO (nbr. entries)           4
    8 SYMTABNO (nbr. entries)            56
    9 STRTAB2 (.dynstr2)                 0x700005f8
   10 STRSZ2 (.dynstr2 sz)               36
   11 PLTGOT (GP value)                  0x080001f0
   12 RELA   (.rela.dyn)                 0x700017c8
   13 RELASZ (.rela.dyn sz)              288
   14 RELAENT (.rela.x entry sz)         24
   15 TANDEM_FPTR (.fptr)                0x080001d0
   16 TANDEM_FPTRSZ (.fptr sz)           32
   17 LIC                                0x70000328
   18 LICSZ (.lic sz)                    368
```

FILEHDR

This command displays the ELF header information for the entire file and is always found at the start of an ELF file.

A sample display of this command is shown below for a loadfile:

```
enoft> filehdr

******** ELF File Header

Ident:                     ELF64-bit Big_Endian VER_1 NSK
Type:                      PIC_Program (loadfile)
Target Machine:            IA64
Version:                   1 (current)
Entry Point Address:       0x70000ae0
Program Hdr file offset:   0x00000040
Sections Hdr file offset: 0x000040b0
Flags:                     (0x2500000)
                           Target OSS
                           32-bit DATA MODEL
                           IEEE_float
                           Non-empty Liblist
                           Localized
                           DATA1 (unprotected)
File Hdr Size:             64
Program Hdr Size:          56
Nbr Program Hdrs:          6
Sections Hdr Size:         64
Nbr Section Hdrs:          36
String Table Index:        35
```

This command can no longer be called from its "parent" command `HEADERS` because that command is not implemented in `eNOFT`.

FUNCDESC or FD

New to `eNOFT`, this command displays the contents of the function descriptor sections and is only applicable to loadfiles. The two possible sections are the local function descriptors .IA_64.pltoff and the official function descriptor .fptr.

The following shows the function descriptor section of a loadfile.

```
enoft> funcdesc

******** Section .fptr (File Offset 0x21d0)

Mem_Addr    Proc_Addr          GP_Value
---------------------------------------------------------------------
0x080001d0  0x0000000070000880 0x00000000080001f0
0x080001e0  0x0000000070000880 0x00000000080001f0

******** Section .IA_64.pltoff (File Offset 0x2230)

Mem_Addr   GP_Offset   Proc_Addr          GP_Value
---------------------------------------------------------------------
0x08000230 0x00000040  0x0000000078154160 0x00000000781bc170
0x08000240 0x00000050  0x00000000780c2900 0x00000000781bc170
0x08000250 0x00000060  0x0000000078154f00 0x00000000781bc170
0x08000260 0x00000070  0x0000000078155940 0x00000000781bc170
```

GOT

This is an alias for `DUMPSECTION .got` and is only applicable to loadfiles.

The .got global offset table section contains 64-bit addresses of data items that are referenced indirectly, as well as the addresses of official function descriptors and EnterPriv labels. The following shows the global offset table section of a loadfile.

```
enoft> got

******** Section .got (File Offset 0x21f0)

GP: 0x080001f0

      Address_of Computed   Value_of
Index GOT_Entry  GP_offset  GOT_Entry          Symbols_Name
-------------------------------------------------------------
--------
    0 0x080001f0 0x00000000 0x0000000008000008
    1 0x080001f8 0x00000008 0x0000000078167570
    2 0x08000200 0x00000010 0x0000000078167590
    3 0x08000208 0x00000018 0x0000000008000010
    4 0x08000210 0x00000020 0x0000000008000000
    5 0x08000218 0x00000028 0x0000000008000004
    6 0x08000220 0x00000030 0x00000000080001a0
```

(For this file, the GP value is the start of the GOT.)

HASH

This command displays the contents of the .hash and .hash.gblzd sections and is only applicable to loadfiles. Section .hash also applies to import libraries.

The .hash and .hash.gblzd sections are used for looking up symbols in the .dynsym and .dynsym.gblzd sections, respectively.

The `NOFT` option `GLOBALIZED` is not supported. Use `DUMPSECTION` to display the .hash.gblzd section.

The following shows the hash section of a loadfile. (No .hash.gblzd section is available for this file.)

```
enoft> hash

******** Section .hash (File Offset 0xf70)

Number of buckets: 37   Symbol Chain Length: 56

        (Computed)  Chain Dynsym  Dynsym
Bucket  Hash Value  Index  Index  Name
---------------------------------------------------------------------
     0  0x006c994f            40  errno
     1  0x0f021913            52  __INIT__1_C
     2  0x00000000
     3  0x00000000
     4  0x000737fe     48     48  main
     5  0x00000000
     6  0x00000000
     7  0x000006f7            50  hw
     8  0x00000000
     9  0x00000000
    10  0x00064172            47  _MCB
    11  0x00000000
    12  0x00000000
    13  0x035b39df     35     35  C_INT_INIT_COMPLETE_
    14  0x00000000
    15  0x077905a6            36  printf
    16  0x05079511            41  myproc1
    17  0x08384a3e            55  T8432H01_01OCT2004_CCPLMAIN
    18  0x00000000
    19  0x066050ba            42  _initz
    20  0x00000000
    21  0x00000000
    22  0x06399473           44  _DTORS
    23  0x00000061     54     54  a
    24  0x00000062           37  b
. . .

        (Computed) Next in  Dynsym  Dynsym
Chain   Hash Value   Chain   Index  Name
---------------------------------------------------------------------
     0  0x00000000
     1  0x00000000
       . . .
    34  0x00000000
    35  0x06389473              43  _CTORS
    36  0x00000000
    37  0x00000000
       . . .
```

HASHVAL

This command displays the contents of the .hashval and .hashval.gblzd sections and is only applicable to loadfiles.

The .hashval section contains the precomputed hash values for the symbols listed in the .dynsym section. When filling in relocation sites for references across loadfile boundaries, these hash values provide a convenient means to look up corresponding symbols in sections of other loadfiles without first having to calculate the value from the hash function.

The NOFT option GLOBALIZED is not supported. Use DUMPSECTION to individually display the .hashval and .hashval.gblzd sections.

The following shows the hash section of a loadfile:

```
enoft> hashval

******** Section .hashval (File Offset 0x16e4)

Index Hash_Value       Info Bucket Dynsym_Name
-------------------------------------------------------------------
    0 0x00000000 0x00000000      0 *** No symbol ***
    1 0x00000000 0x00000000      0 *** No symbol ***
. . .
   17 0x00000000 0x035b39df      0 *** No symbol ***
   18 0x077905a6 0x00000062     15 printf
   19 0x006415de 0x0efd772f     23 _MAIN
   20 0x006c994f 0x05079511      0 errno
   21 0x066050ba 0x06389473     19 _initz
   22 0x06399473 0x0c5d093e     22 _DTORS
   23 0x066ac94a 0x00064172      4 _termz
   24 0x000737fe 0x065bb693      4 main
   25 0x000006f7 0x065ab693      7 hw
   26 0x0f021913 0x0006cf04      1 __INIT__1_C
   27 0x00000061 0x08384a3e     23 a
```

LIBLIST

This is an alias for `DUMPSECTION.liblist` and is applicable to loadfiles and import libraries.

This section contains the names of the DLLs that were on the linker command line when the linker built this loadfile. It is not required if no DLLs were so specified. In an import library that represents a single DLL, it contains the same information as in that DLL. Neither the user library nor implicit libraries appear in the program's liblist.

The following shows the .liblist section of a loadfile:

```
enoft> liblist

******** Section .liblist (File Offset 0x5d8)

Count ReExport NotFound DLL_Name
---------------------------------------------------------------------
    1 no       no       zcrtldll
    2 no       no       zcredll
    3 no       no       ZI18NDLL
    4 no       no       ZICNVDLL
```

LIC

New to `eNOFT`, this is an alias for `DUMPSECTION.lic`, the library import characterization section, and is only applicable to loadfiles that have been preset. For loadfiles that have not been preset, this section serves as a placeholder.

This section contains information about the DLLs that were used to preset the loadfile. It is created by the linker upon binding all symbols in the target loadfile to the DLLs.

The following shows the .lic section for a loadfile with 3 export digests.

```
enoft> lic

 ********** .lic Section (File Offset 0x34c67)

Flag: HasUnres
Index Reloc Export Digest
----------------------------------------------------------------
    1    -1 8b 32 f3 87 21 88 4f 6f 71 e5 84 9d a9 c8 ce 93
    2    -1 21 88 4f 6f 71 e5 84 9d a9 c8 ce 93 56 1a 3c 5f
    3     0 71 e5 84 9d a9 c8 ce 93 63 58 5f 4d 89 30 6b 3c
```

PROCINFO or PI

New to eNOFT, this is an alias for DUMPSECTION.procinfo and is only applicable to linkfiles.

The names, addresses, "attributes", and nesting of procedures and subprocedures parts of the section are used by the linker to create stack unwinding information in loadfiles. The entry points that have the callable attribute are used to create gateways.

The following shows the .procinfo section for a linkfile:

```
enoft> procinfo


******** Section .procinfo (File Offset 0x6e06b0)


PiNdx Offset     Proc_Name
      Attributes Opt  Parm Parent             Section
---------------------------------------------------------------------
    0 0x00000000 T9219M01^01MAY03^ACKPT^M01
                   0    0  none               .text0


    1 0x00000000 IOPRM_CHECKPOINT_SEND_
        E     PR   0   10  none               .restext1


    2 0x00000000 IOPRM_CHECKPOINT_SEND_.CLEAR^CKPT^PKG
              R    0    0     1               .restext2


    3 0x00000000 IOPRM_ALTER_OPEN_
        E     PR   0    3  none               .restext0
. . .
Key:  A = alt entry point  C = callable  E = extensible
       G = Gateway    K = kernel callable  M = main  N = Cobol nonstop
      P = privileged  R = resident        S = shell
      pTAL subprocs and Cobol nested procs have a 'parent'.
```

PROGHDRS

This command displays the contents of the program headers and is applicable to loadfiles and import libraries.

The following shows a listing of the program headers for a loadfile.

```
enoft> proghdrs


******** ELF Program Headers (File Offset 0x40)


Type            Offset     VirtAddr   File_Size  Mem_Size   Align RWX
--------------------------------------------------------------------
LOAD            0x00000000 0x70000000 0x00002000 0x00002000  4096 R-X

LOAD            0x00002000 0x08000000 0x00001000 0x00001000  4096 RW-

DYNAMIC         0x00000498 0x70000498 0x00000140 0x00000140     0 R--

TANDEMINFO      0x00000190 0x70000190 0x00000198 0x00000198     0 R--

LIB_IMPORT_CHAR 0x00000328 0x70000328 0x00000170 0x00000170     0 R--

UNWIND          0x00000620 0x70000620 0x000000a8 0x000000a8     0 R-
```

This command can no longer be called from its "parent" command `HEADERS` because that command is not implemented in `eNOFT`.

RELOC

This command displays the contents of all relocation tables in the object file. This command in `NOFT` had included the following sections: .text, .data, and .rdata. The possible relocation table sections are the sections whose names start ".rela". There are typically several of these in linkfiles. There are at most two in loadfiles, named .rela.dyn and .rela.gblzd.

For linkfiles, the relocation tables .rela.x (for respective sections named .x) describe the relocation sites as offsets relative to the symbols listed in the .symtab symbol tables.

The relocation table describes the relocation site (the location where the contents of one place need to be filled in with the address of another) for that section.

For loadfiles, dynamic relocation tables .rela.dyn and .rela.gblzd describe the relocation sites as offsets relative to the symbols listed in .dynsym and .dynsym.gblzd dynamic symbol tables, respectively.

The following shows the dynamic relocation table for a loadfile:

```
enoft> reloc

******** Section .rela.dyn (File Offset 0x17c8)

Index Reloc_Site         Reloc_Type   Addend             Section
      Target_Symbol
---------------------------------------------------------------------
    0 0x0000000008000000 REL32MSB     0x0000000000000000


    1 0x0000000008000220 REL64MSB     0x0000000000000000


    2 0x0000000008000230 IPLTMSB      0x0000000000000000
      C_INT_INIT_COMPLETE_
    3 0x0000000008000240 IPLTMSB      0x0000000000000000
      printf
    4 0x00000000080001f0 DIR64MSB     0x0000000000000000 .data
      b
    5 0x0000000008000250 IPLTMSB      0x0000000000000000
      C_INT_INIT_START_
    6 0x00000000080001f8 DIR64MSB     0x0000000000000000
      errno
    7 0x0000000008000200 DIR64MSB     0x0000000000000000
      environ
    8 0x0000000008000208 DIR64MSB     0x0000000000000000 .data
      _MCB
    9 0x0000000008000210 DIR64MSB     0x0000000000000000 .data
      hw
   10 0x0000000008000260 IPLTMSB      0x0000000000000000
      exit
   11 0x0000000008000218 DIR64MSB     0x0000000000000000 .data
      a
```

This command incorporates the `NOFT` command DYNREL.

RTDU

```
[ { * | SOURCE | OBJECT } [ DETAIL | D ] ]
```

New to eNOFT, this command displays the header information for the RTDU sections in the object file.

Note linkfiles only have source RTDU's .source.rtdu, while programs have both source RTDU's and at most one object RTDU .object.rtdu. Use `DUMPSECTION` to display an individual section else `STRTAB RTDU` for the name section.

`DETAIL` or `D` option adds the memory content of the RTDU data section information for each procedure.

The following shows the RTDU section of a linkfile:

```
enoft> rtdu


******** Section .source.rtdu (File Offset 0x2e9c7)


Index DataOffset   Size Name
---------------------------------------------------------------------
    0 0x00000000  20553 SQLTEST4
    1 0x00005049   1622 SQLTEST4-SUB1
    2 0x0000569f    842 SQL-CALL-1
    3 0x000059e9    830 SQL-CALL-4
    4 0x00005d27    844 SQL-CALL-2
```

SECTHDRS

This command displays the contents of the section headers.

The following shows a listing of sections for a loadfile:

```
enoft> secthdrs

******** ELF Section Headers (File Offset 0x40b0)

ShNdx Name                  FileOffset VirtAddr     Size
      Type            Flags Link   Info AddrAlign Entsize
---------------------------------------------------------------------
    0                       0x00000000 0x00000000      0
      NULL                     0      0      0         0

    1 .tandem_info          0x00000190 0x70000190    408
      TANDEM_INFO       A      0      0      8         0

    2 .lic                  0x00000328 0x70000328    368
      TANDEM_LIC        A      0      0      8         0

    3 .dynamic              0x00000498 0x70000498    320
      DYNAMIC           A      5      0      8        16

    4 .liblist              0x000005d8 0x700005d8     32
      TANDEM_LIBLIST    A      5      4      8         8

    5 .dynstr2              0x000005f8 0x700005f8     36
      STRTAB            A      0      0      1         0
    . . .
 Key to Flags:
 W-write, A-alloc, X-execute, R-resident, G-gateway, S-short
```

This command can no longer be called from its "parent" command `HEADERS` because that command is not implemented in `eNOFT`.

STRTAB

```
STRTAB [ * | DYNSTR | DYNSTR2 | PROCNAMES | RTDU | SHSTRTAB |
STRTAB | UNWIND ]
```

New to `eNOFT`, this command displays the contents of all string tables in the object file. For linkfiles, the possible string tables are sections named .procnames, .shstrtab, and .strtab. For loadfiles, the possible string tables are sections named .dynstr, .dynstr2, .IA_64.unwind.strings, and .shstrtab, as well as the source RTDU names section.

The default display shows all string tables in the object file.

`DYNSTR` displays the .dynstr section, which is the string space that is pointed at from the .dynsym section.

`DYNSTR2` displays the .dynstr2 section that contains the string space pointed at from the .dynamic, .liblist, and .dynsym.gblzd sections. It is not required if the .liblist and .dynsym.gblzd sections are absent and there are no strings represented in the .dynamic section.

`PROCNAMES` displays the .procnames section that contains the standard ELF string space names pointed at from the .procinfo section.

`SHSTRTAB` displays the .shstrtab section that contains the names of the sections.

`STRTAB` displays the .strtab section that contains that contains the string space pointed at from the .symtab section.

`RTDU` displays the section that contains that contains the string space pointed at from the .rtdu section.

`UNWIND` displays the .unwind.strings section that contains the string space pointed at from the .unwind section.

The following shows the .dynstr section for a loadfile:

```
enoft> strtab dynstr

******** Section .dynstr (File Offset 0x1630)

Count Offset String
---------------------------------------------------------------------
    1 0x0000 <null string>
    2 0x0001 _ctors
    3 0x0008 _CTORS
    4 0x000f _dtors
    5 0x0016 _DTORS
    6 0x001d _initz
    7 0x0024 _termz
    8 0x002b C_INT_INIT_COMPLETE_
    9 0x0040 _MAIN
   10 0x0046 C_INT_INIT_START_
   11 0x0058 errno
   12 0x005e environ
   13 0x0066 _MCB
   14 0x006b main
   15 0x0070 __INIT__1_C
   16 0x007c exit
   17 0x0081 T8432H01_01OCT2004_CCPLMAIN
   18 0x009d myproc1
   19 0x00a5 hw
   20 0x00a8 a
   21 0x00aa b
   22 0x00ac printf
```

This command replaces `NOFT` commands DYNSTR and DYNSTR2.

SYMTAB

`[ * | { EXPORT | E } | { PROC | P } | { DATA | D } ]`

Similar to `NOFT` command `DYNSYM`, this command displays the contents of the .symtab symbol table in linkfiles and the .dynsym and .dynsym.gblzd in loadfiles.

The .dynsym dynamic symbol table provides symbolic information in loadfiles as needed by the loader, or information in loadfiles and import libraries as needed by the linker.

By definition, the .dynsym.gblzd section only contains symbols that are exported or undefined.

Option `EXPORT or E` lists all exported symbols that are global and defined from sections .dynsym and .dynsym.gblzd in loadfiles and import libraries. Symbols available only to other linkfiles and local to loadfiles (not exported) are not listed.

Option `DATA or D` lists data symbols and option `PROC or P` lists code symbols.

The `NOFT` option `GLOBALIZED` is not supported herein. Use `DUMPSECTION` to exclusively display the .dynsym.gblzd section.

The following shows the dynamic symbol table for a loadfile:

```
enoft> symtab

******** Section .dynsym (File Offset 0x10f0)

SymNdx Address/Value      Symbol_Name
       Bind Type EM Lang  FDescAddr/Size    Section
---------------------------------------------------------------------
     0 0x00000000
       Lcl  None    Asm   0                 UNDEF

     1 0x70000190
       Lcl  Sect    Asm   0                 .tandem_info

     2 0x70000328
       Lcl  Sect    Asm   0                 .lic
    . . .
    53 0x00000000         exit
       Glob Code    Ptal  0                 UNDEF

    54 0x08000004         a
       Lcl  Data    C     1                 .data

    55 0x70000cc0         T8432H01_01OCT2004_CCPLMAIN
       Lcl  Code    C     0                 .text

 ('Bind' tells if the symbol is local or global)
 ('Type' tells if it is code or data or some special kind of entry)
 ('E' = STO_EXPORT set, 'M' = STO_MULTIPLE_DEF_OK set)
```

This command replaces `NOFT` commands `ELFSYMTBL` and `DYNSYM`.

TANDEMINFO

This is an alias for `DUMPSECTION .tandem_info` and is applicable to loadfiles and import libraries.

The following shows the .tandem_info section for a loadfile:

```
enoft> tandeminfo

******** Section .tandem_info (File Offset 0x190)

Create_Timestamp: May 06 14:42:58 2004 (Julian)
Update_Timestamp: May 06 14:42:58 2004 (Julian)
Flags:            0x00f08018
                  HIGHPIN
                  HIGH_REQUESTORS
                  RUNTIME_UNRES_CHECKING: ERROR
                  CPP_DIALECT: (Neutral)
                  DEFAULT_DEBUGGER: Visual Inspect
Version:          0
Unwind Offset:    0x00000620
Unwind Size:      7
Libname:          (none)
Fingerprint:      Ver 0, Value 0 0
Process Subtype:  0
Heap Max:         0x0
Mainstack Max:    0x0
Space Guarantee:  0x0
DLL Name:         (none)
Export Digest:    41 16 b5 32 d3 c2 b1 06 64 45 5b bf ce ce 06 ce
ctors Address:    0x00000000
dtors Address:    0x00000000
initz Address:    0x080001c0
termz Address:    0x00000000
Linker Vproc:     TNS/E Linker Internal Build Date April 8, 2004.
```

This command incorporates the DLL- equivalent features of `NOFT` command `SRLDIGEST` which dumps the export digest.

UNWIND

New to `eNOFT`, this command displays the contents of the sections of stack unwinding information and is applicable to linkfiles and loadfiles.

.IA_64.unwind and .IA_64.unwind_info sections describe the stack frame information about procedures from the .text and .restext sections.

For loadfiles, the .IA_64.unwind.strings section pointed at from .IA_64.unwind function will be dumped using the `STRTAB` command.

The following shows the .IA_64.unwind section of a loadfile. Display for the complementary .IA_64.unwind_info is not available as of this writing.

```
enoft> unwind
******** Section .IA_64.unwind (File Offset 0x620)

UwNdx Proc_Addr  Symbol_Name
      InfoPtr_Addr       UnwindAddr         Parent Attribute Section
      info_ptr           begin_address      name_offset
--------------------------------------------------------------------
    0 0x70000800 #import_stubs
      0x70000624         0x70000620         none             .plt
      0x00000000         0x000001e0         0x0000012d

    1 0x70000880 __INIT__1_C
      0x700006c8         0x70000638         none             .text
      0x0000008c         0x00000260         0x00000147

    2 0x70000ae0 _MAIN
      0x700006e8         0x70000650         none       M     .text
      0x00000094         0x000004c0         0x0000013b

    3 0x70000cc0 T8432H01_01OCT2004_CCPLMAIN
      0x70000708         0x70000668         none             .text
      0x0000009c         0x000006a0         0x00000129

    4 0x70000ce0 myproc1
      0x70000720         0x70000680         none             .text
      0x0000009c         0x000006c0         0x0000012d

    5 0x70000d00 main
      0x70000738         0x70000698         none             .text
      0x0000009c         0x000006e0         0x0000011d

    6 0x70000f70 #end_of_code
      0x700006b4         0x700006b0         none             not found
      0x00000000         0x00000950         0x000000c2

Key:  A = alt entry point  C = callable  E = extensible
     G = Gateway    K = kernel callable M = main N = Cobol nonstop
     P = privileged R = resident        S = shell
     pTAL subprocs and Cobol nested procs have a 'parent'.
```

For linkfiles, this command shows the following:

```
enoft> unwind
******** Section .IA_64.unwind (File Offset 0x510)

UwNdx Start_Offset End_Offset Info_Ptr   Section_Name
---------------------------------------------------------------------
    0 0x00000000   0x00000020 0x00000000 .text
    1 0x00000020   0x00000290 0x00000018 .text
```

This command replaces `NOFT` command `RUNTIMEPROC`.

List Commands

The following commands organize and list specific sections of the object file.

Unless otherwise specified, the default format is READABLE.

DBGINFO

```
{ <proc_addr> | <proc_spec> }
```

New to eNOFT, this command lists compilation source and debug file information for a given procedure name, index, or address. When `<proc_addr>` is specified, line number and instruction bundle index will also be given. This command is applicable to loadfiles and import libraries.

`<proc_addr>` value must be in the form 0XXXXXXXXX where X is a hexadecimal digit. The value must be in a code section. In addition, the address value must be on a 16-byte alignment otherwise `eNOFT` will round down to the beginning of the address bundle. Note that if line number information is not available for that bundle, `eNOFT` will display from nearest preceding bundle with available line information.

Note that if the demangled (original) name exists, that name will be shown.

```
enoft> dbg 245

Procedure:        0x600309a0
CLS_SC_REBIND_REBASE::bfnLookupInLoadfile(CLS_SC_REBIND_REBASE *,
GR_SYMBOL_ADDR_INFO *, bool, bool *)

Source:           0 \SPEEDY.$DATA06.T0428TSK.ZRLDSRLP

File:             25 \SPEEDY.$DATA06.T0428TSK.rebndbsp


enoft> dbg 0x600309a0

Procedure:        0x600309a0
CLS_SC_REBIND_REBASE::bfnLookupInLoadfile(CLS_SC_REBIND_REBASE *,
GR_SYMBOL_ADDR_INFO *, bool, bool *)

Source:           0 \SPEEDY.$DATA06.T0428TSK.ZRLDSRLP

File:             25 \SPEEDY.$DATA06.T0428TSK.rebndbsp

Line Number:      2210


Enoft> dbg 0x60000000


******** Debug information matching address not found.
```

LAYOUT

```
[ * | CODE | DATA ]
```

This command lists the parts of the current object file in the order of their relative file offsets.

New to `eNOFT`,`CODE` limits the display to code sections only and `DATA` limits the display to data sections only.

New to `eNOFT`, virtual addresses are shown for all applicable sections. The section type is only displayed for sections of unknown name. The following shows the layout of a loadfile:

```
enoft> layout

******** Layout of ELF File Sections

ShNdx FileOffset    Size Content
      VirtAddr
---------------------------------------------------------------------
    - 0x00000000      64 File Header
      (no value)

    - 0x00000040     336 Program Headers
      0x70000000

    1 0x00000190     408 .tandem_info operating system info
      0x70000190
. . .

   33 0x00003b8e     590 .debug_line of type PROGBITS
      (no value)

   34 0x00003ddc     379 .debug_line_nsk of type PROGBITS
      (no value)

   35 0x00003f57     343 .shstrtab strings for section headers
      (no value)

    - 0x000040b0    2304 Section Headers
      (no value)
```

LISTATTRIBUTE or LA

```
[ DETAIL | D ]
```

This command lists common file and process attributes associated with the object file. These attributes are from the .tandeminfo section, unless otherwise noted. The following attributes are displayed for all object types:

Name (was "Object File" in `NOFT`)
File Format (from ELF Header)
Type (was "Type of Executable")
Debugging Symbols (was "Symbols/INSPECT Region")
Floating-Point Type (from ELF Header)

In addition, the following attributes are displayed for loadfiles and import libraries:

Float-overrule (from ELF Header; n/a to DLLs)
System Type (from ELF Header)
Creation Timestamp (was "Timestamp")
Process Subtype
Highrequestors
Runnamed
Highpin
Saveabend
Main is PRIV/CALLABLE (was "Priv or Callable")
CALLABLE Procs (was "Callable")

The following shows file and process attributes for a program file:

```
enoft> listattribute


******** List of Common File Attributes
Name:                   d:/temp/hello.out
File Format:            ELF64-bit, Big_Endian, IA64
Type:                   PIC_Program (loadfile)
Debugging Symbols:      Yes
Float-Point Type:       IEEE_FLOAT
---------------------------------------------------------------------
System Type:            OSS
Creation Timestamp:     2004 May 6, 14:42:58
Process Subtype:        0
Highrequestors:         Yes
Runnamed:               No
Highpin:                Yes
Saveabend:              No
PRIV or CALLABLE Main:  No
CALLABLE Procs:         No
Default Debugger:       Visual Inspect
```

New to `eNOFT`, the `DETAIL` or `D` option adds the following parameters to the display, as applicable to the file type:

Entry Point
Unresolved References (PIC only)
C++ Dialect
Maximum Heap Size
Main Stack Size
Space Guarantee
Fingerprint
Fingerprint Version
Ctors_vaddr
Dtors_vaddr
Initz_vaddr
Termz_vaddr
User Library Name
Interpose User Library
Globalized Symbols (PIC only)
Limit Runtime Paths (PIC only)
MCB Address (from .data section)
Nbr Procedures
Nbr Ptal AltEntrPts
Languages and Dialects

Instrumented File
Ansistreams

The following NOFT attributes do not apply to TNS/E and thus are not shown in eNOFT:

DEBUG/INSPECT
PFS Size
Directly Needed Public SRLs
Directly Needed Public SRL Bitmap
Userlibrary Timestamp
SRL Client. Attribute "Executable" in the NOFT command is also not supported.

```
enoft> LISTATTRIBUTE DETAIL
******** List of Common File Attributes

Name:                   $guest.bn.aomevtct
File Format:            ELF64-bit, Big_Endian, IA64
Type:                   PIC_Program (loadfile)
Debugging Symbols:      Yes
> Nbr SubProgs:         7476
> Nbr Variables:        24550
> Nbr SrcFiles:         66
> Compiler(s):          Cobol85 C89 C++
Float-Point Type:       TANDEM_FLOAT
---------------------------------------------------------------------
Float-overrule:         No
System Type:            Guardian
Creation Timestamp:     2012-10-08 08:56:22
Process Subtype:        0
Highrequestors:         No
Runnamed:               Yes
Highpin:                Yes
Saveabend:              Yes
PRIV or CALLABLE Main:  No
CALLABLE Procs:         No
Default Debugger:       Visual Inspect
```

```
---------------------------------------------------------------------
(build proc symbols table... done)
Entry Point:              _MAIN
Unresolved References:    ERROR
C++ Dialect:              V3
Maximum Heap Size:        0x00000000
Main Stack Size:          0x00000000
Space Guarantee:          0x00000000
Fingerprint:              0000-0000-0000-0000
Fingerprint Version:      0
Ctors_vaddr:              0x0815b5f0
Dtors_vaddr:              0x00000000
Initz_vaddr:              0x0815b740
Termz_vaddr:              0x00000000
User Library Name:        (none)
Interpose User Library:   Off
Globalized Symbols:       Yes
Limit Runtime Paths:      No
MCB address:              0x08001170
Nbr Procedures:           7489 (excl comp/linker generated procs)
Nbr Ptal AltEntrPts:      0
Languages and Dialects:   Asm C C++V3 Cobol Ptal
Instrumented File:        No
Ansistreams:              Yes
```

LISTCOMPILERS or LC

`[ DETAIL | D ]`

This command lists version information about the compiler components and object file linker used to create the target object file.

`DETAIL` or `D` provides the toolset for each source file in the object.

The following shows three source files built from two toolsets. Note: header files are not listed and the file numbers are from `LISTSOURCE`.

```
enoft> listcompilers detail


******** Compiler Information


Linker Vproc: TNS/E Linker Internal Build Date April 8, 2004.
```

```
Compiler: C89
Descript: T0549H01_01OCT2004_CCOM_22Mar2004_GRD 3.3 TOOLSY02 Release

Compiler: C89
Descript: T0549H01_01OCT2004_CCOM_22Mar2004_WIN32 3.3 TOOLSY02 Release
```

LISTDATA or LD

This is an alias for SYMTAB DATA that lists all data symbols from sections .dynsym and .dynsym.gblzd in loadfiles and import libraries, and section .symtab from linkfiles.

The display shows a subset of the applicable elf symbols sections. See SYMTAB for the full listing of all symbols from the dynamic symbols table.

```
enoft> listdata


******** Section .dynsym (data symbols only)
SymNdx Address/Value      Symbol_Name
       Bind Type EM Lang  FDescAddr/Size    Section
---------------------------------------------------------------------
    37 0x08000008         b
       Lcl  Data    C     1                 .data
    40 0x00000000         errno
       Glob Data    C     0                 UNDEF
    42 0x080001c0         _initz
       Lcl  Data    Asm   16                .rdata
    43 0x700007d0         _CTORS
       Lcl  Data    Asm   8                 .rconst
    44 0x700007d8         _DTORS
       Lcl  Data    Asm   8                 .rconst
    45 0x00000000         environ
       Glob Data    C     0                 UNDEF
    46 0x700007e0         _termz
       Lcl  Data    Asm   8                 .rconst
    47 0x08000010         _MCB
```

```
      Lcl  Data    C     400                .data
   49 0x700007d8         _dtors
      Lcl  Data    Asm   8                  .rconst
   50 0x08000000         hw
      Lcl  Data    C     4                  .data
   51 0x700007d0         _ctors
      Lcl  Data    Asm   8                  .rconst
   54 0x08000004         a
      Lcl  Data    C     1                  .data
Number of symbols matching scope: 12
 ('Bind' tells if the symbol is local or global)
 ('Type' tells if it is code or data or some special kind of entry)
 ('E' = STO_EXPORT set, 'M' = STO_MULTIPLE_DEF_OK set)
******** Section .dynsym.gblzd not found.
```

LISTDEBUG or LDE

```
[ * | PROC | P | DATA | D ] [ DETAIL | D ]
```

New to `eNOFT`, this command lists all names in the .debug_info symbols table that meet the variable (data) or subprogram (subprogram, subroutine, entry point) criteria.

`DETAIL` or `D` provides more information, such as type of the symbol.

```
enoft> listdebug * detail

******** List of Debugging Symbols

Count Src:Fil  Line Edit_Line Type Start_Addr
      Symbol_Name (mangled)
---------------------------------------------------------------------
    1   0:001    40    44.    Proc 0x70049200
      __INIT__1_C

    2   0:001    45    55.    Proc 0x700494a0
      _MAIN

    3   1:001     8     8.    Proc 0x70049680
      T8432H01_01OCT2004_CCPLMAIN

    4   2:001    10    10.    Proc 0x700496a0
      T8432H01_10OCT2004_ETK_EAP

    5   3:001  2531           Proc 0x700496c0
      get_8__FRPc

    6   3:002   494           Proc 0x00000000
      __ct__13DynArrayErrorFi

    7   3:002  1697           Proc 0x00000000
      Data__16Dw_File_Info_RepCFv

    8   3:002  1880           Proc 0x00000000
      __ct__8Dw_ErrorFQ2_8Dw_Error5Codes
. . .
```

LISTEXPORTS or LE

This is an alias for `SYMTAB EXPORT` and is applicable to loadfiles and import libraries.

This command lists all exported symbols that are global and defined from sections .dynsym and .dynsym.gblzd in loadfiles and import libraries. Symbols available only to other linkfiles and local to loadfiles (not exported) are not listed.

The display shows a subset of the .dynsym and .dynsym.gblzd sections whereby only global symbols that are defined are shown. See `SYMTAB` for the full listing of all symbols from the dynamic symbols table.

```
enoft> listexports


******** Section .dynsym (exported symbols only)


SymNdx Address/Value      Symbol_Name
       Bind Type EM Lang  FDescAddr/Size    Section
---------------------------------------------------------------------
    30 0x780006e0         PROC1B
       Glob Code    Asm   0x78010000         .text

    32 0x78000740         ENTRY1B
       Glob Code    Asm   0x78010010         .text

    38 0x780005e0         PROC1A
       Glob Code    Asm   0x78010020         .text

 ('Bind' tells if the symbol is local or global)
 ('Type' tells if it is code or data or some special kind of entry)
 ('E' = STO_EXPORT set, 'M' = STO_MULTIPLE_DEF_OK set)
```

LISTOPTIMIZE or LO

`[ * | 0 | 1 | 2 | EXCLUDE | E | BRIEF | B ]`

This command lists procedures based on the optimization level of 0, 1, or 2.

The default display shows all procedures in the object file, sorted by optimization level.

`EXCLUDE` or `E` removes display of symbols that are generated by the compiler or symbols not found in the .debug_info section.

`BRIEF` or `B` limits display to counts of symbols matching scope.

```
enoft> listoptimize

******** Optimization of Procedures

UwNdx Opt Procedure Name
----------------------------------------------------------------------
    2  2  T8432G08_01FEB2001_CRTLMAIN
    0  2  __INIT__1_C
    1  2  _MAIN
    3  1  main
```

LISTPROC or LP

```
{ * | <proc_spec> }
[ EXCLUDE | E | SUBPROC | SP | NOSUBPROC | NSP ] [ DETAIL | D ]
```

This command lists procedures and subprocedures, as determined by the current scope. All procedures listed are defined.

Without any local or global scope setting, the default display shows all available procedure and applicable subprocedure items. If procedure P contains subprocedure S, a `LISTPROC P` command line lists S, as it is contained within P.

`EXCLUDE` or `E` removes display of symbols that are generated by the compiler or symbols not found in the .debug_info section.

`SUBPROC` or `SP` removes display of parent procedures. If procedure P contains subprocedure S, a `LISTPROC S SUBPROC` command line lists only S (and all its duplicates if available) and not P even though P encompasses S.

`NOSUBPROC` or `NSP` removes display of subprocedures. If procedure P contains subprocedure S, a `LISTPROC P NOSUBPROC` command line lists only P and not S even though S is within P.

The following shows all defined procedures for a loadfile. Note for C++ procedures, "mangled" names are shown by default.

```
enoft> listproc

******** List of Procedures

UwNdx Proc_Addr  Proc_Name
--------------------------------------------------------------------
    1 0x70000880 __INIT__1_C
    2 0x70000ae0 _MAIN
    3 0x70000cc0 T8432H01_01OCT2004_CCPLMAIN
    4 0x70000ce0 myproc1
    5 0x70000d00 main
```

DETAIL or D provides detailed information about procedures and subprocedures. For C++ procedures, DETAIL provides the "demangled" (original) names as well as the "mangled" internal equivalents. New to eNOFT, this command displays the specified procedure (or subprocedure) in the DETAIL format if <proc-spec> scoping is specified here or globally.

```
enoft> listproc * detail

******** List of Procedures

UwNdx Proc_Addr  Proc_Name
      InfoPtr_Addr       ProcSz SrNdx Opt Parent Attributes Section
---------------------------------------------------------------------
    1 0x70000880 __INIT__1_C
      0x700006c8            608 0     0    none             .text

    2 0x70000ae0 _MAIN
      0x700006e8            480 0     0    none       M     .text

    3 0x70000cc0 T8432H01_01OCT2004_CCPLMAIN
      0x70000708             32 1     0    none             .text

    4 0x70000ce0 myproc1
      0x70000720             32 2     0    none             .text

    5 0x70000d00 main
      0x70000738            624 2     0    none             .text

Key: SrNdx = Src Index  Opt = optimization level
     A = alt entry point  C = callable  E = extensible
     G = Gateway    K = kernel callable M = main N = Cobol nonstop
     P = privileged R = resident        S = shell
     pTAL subprocs and Cobol nested procs have a 'parent'.
```

LISTSOURCE or LS

```
[ * | <source-spec> ] [ DETAIL | D ]
```

This command lists all source files in an object file, as determined by the current scope.

The default display shows all available source files.

New to `eNOFT`, the display will be in the `DETAIL` format if `<source-spec>` or `<proc-spec>` is specified here or globally.

The following shows two source files used to build a linkfile. The `NOFT` attribute "Address" is not supported.

```
enoft> listsource


******** List of Source Files (Compilation Units)


SrNdx NoSrc Source_and_Header_Files
---------------------------------------------------------------------
    0     1 \SPEEDY.$DATA06.T8432H01.CPLMAINC
            \SPEEDY.$DATA06.T8432H01\CPLMAINC
            \SPEEDY.$DATA01\#0062090
            \SPEEDY.$DATA01.TOOLSY02\STDIOH
            \SPEEDY.$DATA01.TOOLSY02\SYSTYPEH
            \SPEEDY.$DATA01.TOOLSY02\ERRNOH
    1     1 \SPEEDY.$DATA06.T8432H01.VERSNMNC
            \SPEEDY.$DATA06.T8432H01\VERSNMNC
    2     1 d:\temp\hello.c
            d:\temp\hello.c
```

`DETAIL or D` displays detailed information about the specified source file.

The attributes "Size", "Address of First Procedure", "Optimization Level Default", and "Symbols" in the `NOFT` command are not supported.

```
enoft> listsource * detail
******** List of Source Files (Compilation Units)

Source:   0 \SPEEDY.$DATA06.T8432H01.CPLMAINC
Copies:   1
Compiler: C89
Descript: T0549H01_01OCT2004_CCOM_22Mar2004_GRD 3.3 TOOLSY02 Release

Header:   \SPEEDY.$DATA06.T8432H01\CPLMAINC

Header:   \SPEEDY.$DATA01\#0062090

Header:   \SPEEDY.$DATA01.TOOLSY02\STDIOH

Header:   \SPEEDY.$DATA01.TOOLSY02\SYSTYPEH

Header:   \SPEEDY.$DATA01.TOOLSY02\ERRNOH

Source:   1 \SPEEDY.$DATA06.T8432H01.VERSNMNC
Copies:   1
Compiler: C89
Descript: T0549H01_01OCT2004_CCOM_22Mar2004_GRD 3.3 TOOLSY02 Release

Header:   \SPEEDY.$DATA06.T8432H01\VERSNMNC

Source:   2 d:\temp\hello.c
Copies:   1
Compiler: C89
Descript: T0549H01_01OCT2004_CCOM_22Mar2004_WIN32 3.3 TOOLSY02 Release

Header:   d:\temp\hello.c
Time:     2004 May 6, 13:44:02
Size:     402
```

LISTUNREFERENCED or LUR

```
[ * | PROC | P | DATA | D ] [ DETAIL | D ]
```

This command is similar to that in `NOFT` except the asterisk "*" is optional.

This command lists all the names that are undefined and unreferenced in this object file and need to be linked in before it is executable.

```
enoft> listunreferenced * detail


******** List of Unreferenced Symbols


SymNdx Address/Value      Symbol_Name
       Bind Type EM Lang  FDescAddr/Size    Section
       Calling_Procedure
---------------------------------------------------------------------
    26 0x00000000         BLAHZERO
       Glob Data          0                 UNDEF


    27 0x00000000         BLAH1
       Glob Data          0                 UNDEF


    28 0x00000000         BLAH2
       Glob Data          0                 UNDEF


    29 0x00000000         BLAH3
       Glob Data          0                 UNDEF
```

LISTUNRESOLVED or LU

```
[ * | PROC | P | DATA | D ] [ EXCLUDE | E ]
```

This command is similar to that in NOFT in that it lists all the names that are undefined (yet referenced) in this object.

However the asterisk "*" is optional and this command also lists "unresolved" data symbols.

```
enoft> listunresolved * detail


******** List of Unresolved (undefined) Symbols


SymNdx Address/Value      Symbol_Name
       Bind Type EM Lang  FDescAddr/Size    Section
       Calling_Procedure
---------------------------------------------------------------------
    26 0x00000000         BLAHZERO
       Glob Data          0                 UNDEF

    27 0x00000000         BLAH1
       Glob Data          0                 UNDEF

    28 0x00000000         BLAH2
       Glob Data          0                 UNDEF

    29 0x00000000         BLAH3
       Glob Data          0                 UNDEF

    30 0x00000000         STOP
       Glob Code          0                 UNDEF
```

The `NOFT` argument "EXCLUDE" is not supported.

XREFPROC or XP

```
[ * | <proc-spec> ] [ CALLEDBY | CALLS | BOTH ] [ DETAIL | D ]
```

This command displays an alphabetical cross-reference listing of procedures.

`CALLEDBY` option lists each procedure and the procedures it is called by (default).

`CALLS` option lists each procedure and the procedures it calls.

`BOTH` option gives both sets of information, first `CALLEDBY`, then `CALLS`.

If `<proc-spec>` is specified, the display will be restricted to the specified procedure and the procedures that calls it else to the procedures that it calls if `CALLS` option is used.

`DETAIL` or `D` option lists the called or calling procedures referenced by the indicated procedures and the addresses where the calls are made.

The following shows a listing of procedures and the procedures that they call.

```
enoft> xrefproc * both detail


******** List of Cross-Referenced Symbols


 Called Procedures
    UwNdx Calling Procedures
          Address(es)
---------------------------------------------------------------------
 C_INT_INIT_COMPLETE_
    2 _MAIN
          0x70000c80
 C_INT_INIT_START_
    1 __INIT__1_C
          0x70000b70
 exit
    2 _MAIN
          0x70000d50
 main
    2 _MAIN
          0x70000d20
 malloc
    5 main
          0x70000df0
 printf
    5 main
```

```
            0x70000e50
         6 proc1
            0x70000ec0
     proc1
         5 main
            0x70000e60
     strcpy
         5 main
            0x70000e30

    Number of Called (callee) procedures: 8

    UwNdx Calling Procedures
             Called Procedures
                Address(es)
    ---------------------------------------------------------------------
        1 __INIT__1_C
             0x70000b70 C_INT_INIT_START_
        2 _MAIN
             C_INT_INIT_COMPLETE_
                0x70000c80
             exit
                0x70000d50
             main
                0x70000d20
        5 main
             malloc
                0x70000df0
             printf
                0x70000e50
             proc1
                0x70000e60
             strcpy
                0x70000e30
        6 proc1
             0x70000ec0 printf
```

File Handling Commands

The following user interface commands retain the styles and content of `NOFT` as much as practical to maintain continuity for users.

<Break Key>

This feature terminates the output of the current command and if `eNOFT` is running in command-line mode, terminates the `eNOFT` program itself.

On some emulators, the control key <CTL> is required to be pressed concurrently with the break key <BR>.

CD

```
[pathname]
```

This command sets the current working directory.

New to `eNOFT`, this command is an alias for the `VOLUME` command in Guardian. See that command for syntax requirements when using this command in the Guardian environment.

For OSS, the target file may be a Guardian subvolume by use of "/G/vol/subvol" path name. For files across a node, prefix the path name with the node name "/E/node".

`pathname` may be fully qualified or relative, with forward slashes for OSS names and reverse slashes for the PC environment.

Without any option, this command reverts to the default directory (directory at the time `eNOFT` was invoked.)

No validation of specified pathname is performed; validation is only performed while attempting to open the target file.

Relative path names are accepted with the default directory being the current directory from which `eNOFT` command is typed.

COMMENT

This command adds the remainder of the command input line as comment to the designated output. Start each successive comment line with this option.

COMP

```
[ ref-objfile ] target-objfile [ DETAIL | D ]
```

New to `eNOFT`, this command allows comparison between two object files for major differences, including file headers and program headers.

`ref-objfile` is the reference object file. If not specified, the current object file is used (if it exists).

`Target-objfile` is the target object file.

As `COMP` processes each section/header, it displays a message each time it encounters a difference between the two objects. If there are no differences in the sections/headers, then a message is displayed stating that the sections/headers compared identically.

When `COMP` is done processing all the sections/headers, it displays the final result of the comparison. If there were any differences encountered, then the result is that the two objects are not identical. If no differences are encountered then the result is that the two objects are identical.

In brief mode (when `DETAIL` is not specified), only differences and the final result of the comparison are displayed.

The COMP command compares the following headers and sections:

File Header

Program Headers

Section Headers

Tandem Info Section

Data Sections

Gateway Sections

RTDU Sections

Relocation Sections

PIC Sections

Procedure Sections

Symbol Table Sections

Unwind Sections

The `COMP` command does not compare DWARF Symbol Tables.

DEMANGLE or DE

<proc_spec>

New to `eNOFT`, this command displays C++ symbol names in demangled format. An object file need not be opened prior to use of this command.

```
enoft> demangle __ct__7CMRWOUTFUiT1

CMRWOUT::CMRWOUT(unsigned int, unsigned int)
```

DIR or FILES

`<pathname>`

New to `eNOFT`, this command lists all entries in the specified directory.

`pathname` may be fully qualified or relative, with forward slashes for OSS names and reverse slashes for the PC environment.

Without any option, this command reverts to the default directory (directory at the time `eNOFT` was invoked.)

No validation of specified pathname is performed; validation is only performed while attempting to open the target file.

Relative path names are accepted with the default directory being the current directory from which `eNOFT` command is typed.

ENV

This command displays the current values of the `eNOFT` program environment and that of the target object file. Use `SHOW` to see all globally `SET` commands.

```
enoft> env
Object File:     c:\idis\comptest.exe
Environment:     PC
Out File:        (none)
Log File:        (none)
Obey File:       (none)
Current Path:    .
```

EXIT or E or QUIT or Q

These commands terminate `eNOFT` with return code "0" (EXIT_SUCCESS).

FC and !

```
[ <history-num>  | -<history-offset> | text ]
```

The ! (exclamation point) command executes a previously executed command line and is applicable in an interactive session only. The FC performs the same task except it first echoes a previous command and is therefore convenient for editing.

By default, both commands revert to the previous command, as applicable.

`<history-num>` is the ordinal value from the HISTORY command.

`-<history-offset>` is a negative offset from the current command; for example, the command before is FC -1.

`text` corresponds to the last command starting with that text.

FILE or F

`objectfile`

This command opens the specified target object file. The `NOFT` option "?" is not supported. Use ENV to view the current file settings.

`objectfile` must match the `eNOFT` product input requirements. If the new object file is valid, the prior object file, if any, is closed and `SET CASE` is set to the applicable sensitivity of the new object file. Invalid arguments to this command will generate a syntax error and reset the currently opened object file, if any, and its case sensitivity to their default settings ("none"). Regardless, its associated global scope settings `SET SCOPEPROC` and `SET SCOPESOURCE` are set to their default settings, as applicable.

The file name must be specified in the format of the host platform; for example, the Guardian file format must be used when running `eNOFT` in the Guardian environment. For OSS, the target file may be a Guardian object file by use of "/G/vol/subvol" path name. For files across a node, prefix the path name with the node name "/E/node".

Specifying fully qualified path may not be required. If the specified file name is not fully specified, `eNOFT` first attempts to resolve the file name to the current location established by `CD` or `VOLUME` if different from the default path location (path location at the time `eNOFT` was invoked). If the specified file is not found, `eNOFT` next attempts to resolve the file name using the default path location.

```
enoft> f c:\comptest.exe
Object File:    c:\comptest.exe
File Format:    ELF64-bit, Big_Endian, IA64
Current Scope:  (none)
Case:           Sensitive
```

HELP or ?

`[ command | help-topic ]`

This command displays a one-line description of each command and available help-topics in `eNOFT`. This supercedes the requirement to type option `ALL` in `NOFT`. The `NOFT` option `UNDOCUMENTED` is not supported.

`command` presents the user with detailed help on the command in question, including syntax and other information. The content is similar to the data provided on the OSS manual page as well as the written manual, although certain exceptional behavior may be left out of the online documents.

`help-topic` gives detailed information on specific topics about `eNOFT` or the ELF object file format. The `NOFT` help topic shortcuts is not supported.

```
enoft> help

Type "HELP <topic>" or "? <topic>" for details on
syntax of individual commands:

******** SET and RESET Commands
RESET            resets one or more set-cmds to default values
SET              sets or show one or more set-cmds in current session
SET CASE SC      sensitivity for source, procedure, and path names
SET DEMANGLE SDE sets C++ symbols name displays to DEMANGLE
SET DISPLAY SD   sets the display format to BRIEF or DETAIL
SET FORMAT SF    formatting of DUMP commands set
SET HISTORYBUFFER SHB size of previous commands stored
SET HISTORYWINDOW SHW number of previous commands seen
SET LINES SL     number of lines displayed before pause
SET SCOPEPROC SSP limits to looking at individual procedures
SET SCOPESOURCE SSS limits to looking at individual source files
SET SORT ST      sorting of LIST commands set

******** DUMP Display Commands
DUMPALL ALL      all non-zero sections + file, prgrm, & section hdrs
DUMPADDRESS DA   program text or data range (executables)
DUMPCODE DC      all code in object file, optionally disassembled
 . . . and so on . . .
```

HISTORY or H

```
[ <num> ]
```

This command displays the list of previous commands.

`<num>` is the number of command lines to be displayed. The default is 10 lines.

This parameter is useful only in an interactive session, because options given in the command line are not stored in the history buffer.

```
enoft> history
     1> f c:\comptest.exe
     2> env
     3> history
```

LOG and OUT

```
[ OFF | outfile [ ASCII ] ]
```

The `LOG` command echoes a copy of the current session's input and output to a specified file. The `OUT` command redirects the output listings from the standard terminal to a specified file; the input remains being displayed to the standard terminal. The `NOFT` option "?" is not supported.

`OFF` resets to not logging. New to `eNOFT`, this command also resets to not logging if no option is entered.

`outfile` defaults to the EDIT file type in the Guardian environment and ASCII text file type in OSS and PC Windows.

`ASCII` specifies the file type will be ASCII text mode with file code "180". This option replaces option "BINARY" in the `NOFT` command and is not applicable to OSS or PC Windows.

If specified outfile does not exist, `eNOFT` creates it. If the specified file name is not fully specified, the log file is created in the current path set by `CD` or `VOLUME`. Specifying a partial path location is accepted (for example, only specify subvolume without the node and volume) with the current path as the default path from which the file is created.

If specified outfile is an existing file, `eNOFT` appends the log output to the file.

If logging is already in progress, `eNOFT` closes the previous log file and begins logging to the new file. If the file is the same as the previous log file, `eNOFT` ignores this command and continues logging to the same file.

The alias `NOFT` commands `SET LOG` and `SET OUT` are not supported.

NOEXIT

After executing prior listed commands on the command line, reverts program to Interactive mode and generates `eNOFT` prompt.

Note commands listed after this command on the command line are ignored. Also commands that are not applicable to Command-Line mode (for example, `SET LINES`) will be ignored if specified prior to this command.

This program is only applicable to Command-Line mode.

OBEY

```
infile
```

This command directs `eNOFT` to read from the specified command script file.

`infile` is mandatory; a syntax error will generate if no file name is given and a data error will generate if `eNOFT` cannot open the specified file.

In the Guardian environment, infile must be of the EDIT file type. A data error will be generated if the file type is not of this code.

The command files may be nested to any depth but cannot be circularly linked; for example, recursive. Opening a currently opened OBEY command file will result in an error.

The commands listed in this file must follow the rules specified for command-line processing.

SHOW

```
[ * | set-cmd ]
```

This command is an alias for the command `SET`, except that option <argument> of `SET` is not available; that is, `SHOW` cannot be used to set a <set cmd>.

Unlike `NOFT`, attributes associated with the `ENV` command are not shown with output from this command. Use that command to show name of current target object file and its environment.

This command replaces `NOFT` commands `SET <set-cmd>?` and `OPTIONS`.

VOLUME or CD

```
[ \<node> ] [ .$<volume> ] [ .<subvolume> ]
```

This command changes the default node, volume or subvolume and is applicable to the Guardian environment only. Use `CD` and its syntax for the PC Windows and OSS personalities.

New to `eNOFT`, `CD` is an alias for this command on the Guardian environment.

Without any option, this command reverts to the default directory (directory at the time `eNOFT` was invoked.)

Specifying a partial file name is acceptable (for example, only specify subvolume without the node and volume) with the default path being the current location from which `eNOFT` command is typed.

No validation of specified node name, volume, or subvolume; file validation is performed while attempting to open the target file.

The alias `NOFT` command `SYSTEM` is not supported.

3 The ar Utility

The `ar` utility creates and maintains archives composed of groups of object files. You can mix PIC and non-PIC files in an archive. After an archive has been created, new files can be added and existing files can be extracted, deleted, or replaced.

The `ar` utility runs in the following environments:

- Guardian
- Open System Services (OSS)
- The following PC platforms:

	Operating System			
Platform	**Windows 98**	**Windows NT**	**Windows 2000**	**Windows XP**
TDS[1]	Yes	Yes	No	No
ETK[2]	No	Yes	Yes	Yes

1. HP Tandem Development Suite

2. HP Enterprise Toolkit—NonStop Edition

To run the `ar` utility, use the following syntax. The syntax is the same in every environment in which `ar` runs:

```
ar action-option [modifier-option ...] [position_name]
   archive [file ...]
```

action-option

is an `ar` option that specifies the action to be performed. The action options are as follows:

Name	**Function** (page 1 of 2)
`-d`	Delete the specified files from the archive.
`-m`	Move the specified files. The `-a`, `-b`, or `-i` option with the *position-name* operand indicates the destination of the move; otherwise, move the files to the end of the archive.
`-p`	Write the contents of the specified files from the archive to the standard output. If no files are specified, write the contents of all files in the archive, from first to last.
`-q`	Quickly append the specified files to the end of the archive file. In this case, `ar` does not check whether the added members are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive file piece by piece.

Name	**Function** (page 2 of 2)
`-r`	Replace or add files to the archive. If the archive specified by *`archive`* does not exist, `ar` creates a new archive and writes a diagnostic message to standard error (unless you specify the `-c` option). If no files are specified and the archive exists, no changes are made to that archive. Files that replace existing files do not change the order of the archive; files that do not replace existing files are appended to the archive.
`-t`	Write a table of contents of *`archive`* to the standard output. The specified files are included in the written list. If no *`file`* operands are specified, all files in the archive are included in the list, in the order in which they occur in the archive.
`-Wobey` *`obey-file`*	Indicates that an option and a list of files to be processed should be read from the file *`obey-file`* rather than from the command line. The `-Wobey` option cannot be used on the command line when any other option is used on the command line. Use the `-Wobey` option to speed up execution of the `ar` command when more than one file must be processed. The file *`obey-file`* must be either a Guardian EDIT file or a OSS text file. In the *`obey-file`*, you must specify one and only one option from the *`required-flag`* set `dmpqrtx`. You can also specify any number of optional flags from the set `abcilsuvCT`. If you select a *`modifier-option`* (`a`, `b`, or `i`), you must also specify the name of a file within the library (*`position_name`*) immediately following the option list and separated from it by a space.
`-x`	Extract the specified files from the archive. The contents of the archive file are not changed. If no *`file`* operands are specified, `ar` extracts all files in the archive. `ar` sets the modification time of each extracted file to the time at which the file is extracted from the archive. When `ar` is running in the Guardian environment, it gives file code 700 to extracted TNS/R files and file code 800 to extracted TNS/E files; it gives file code 180 to all other extracted files. If the filename of a file to be extracted is longer than that supported in the directory to which it is being extracted, an error occurs and `ar` does not extract the file unless the `-T` option is specified, in which case `ar` extracts the file and renames it with the truncated filename. If the name of a file to be extracted is not valid on the platform where `ar` is running, `ar` does not extract the file but issues a diagnostic instead.

modifier-option

is an `ar` option that gives instructions for the operation of the *action-option*. The *modifier-option*s are as follows:

Name	**Function** (page 1 of 2)
`-a`	Position new files in the archive after the file specified by the *position-name* operand.
`-b`	Position new files in the archive before the file specified by the *position-name* operand.
`-c`	Suppress the diagnostic message that would be written to standard error by default when the archive file is created.
`-C`	Prevent extracted files from replacing like-named files in the file system. This option is useful when `-T` is also used, to prevent truncated filenames from replacing files with the same prefix.
`-i`	Position new files in the archive before the file specified by the *position-name* operand (same function as the `-b` option).
`-l`	In the OSS environment, create temporary files in the local current working directory instead of the directory specified by the environmental variable TMPDIR. In the Guardian environment and on Windows platforms, this option is ignored; temporary files are always created in the default subvolume in the Guardian environment and in the current folder on platforms running Windows.
`-s`	Force regeneration of the archive symbol table even if `ar` is not invoked with an option that modifies the archive file contents.
`-T`	Allow filename truncation of extracted files having archive names that are longer than the file system supports. By default, an error occurs when attempting to extract a file with a name that is too long; `ar` writes a diagnostic message and does not extract the file.
`-u`	Update older files. When used with the `-r` option, this option causes `ar` to replace a file within the archive only if the corresponding file has a modification time that is at least as new as the modification time of the file within the archive.

Name	**Function** (page 2 of 2)
`-v`	Give verbose output. ● When used with the `-d`, `-r`, or `-x` options, this option causes `ar` to write the name of each file involved in archiving operations. ● When used with `-p`, this option causes `ar` to write the name of each file to the standard output before writing the file itself to the standard output. ● When used with `-t`, this option causes `ar` to include a long listing of information about each file within the archive, including access, ownership, size, and date-and-time information. The specific content of the listing is as follows: *`access info, user ID, group ID, member size, month, day, hour, minute, year, Filename`* When used with `-t` on Windows platforms, the ownership fields are shown (although shown as zero), because Windows provides for ownership, but access information for group and other is shown as "no access," because these fields are not relevant under Windows. `ar` makes no use of the file access information saved for archive members.
`-Wfiletype`	Display the file type. When this option is used with the `-tv` option, `ar` displays `[elf]`, `[tns]`, or nothing after the filename. This can be useful to discover the cause of the problem when `ar` fails to generate a symbol table because the archive contains a mix of TNS, TNS/R, or TNS/E files. This option is available in the OSS and Guardian environments.

When modifier options are used in combination, the preceding hyphen can be omitted for all but the first option specified.

`position-name`

is the name of a file in the archive that is used for relative positioning. See the descriptions of the `-m` and `-r` options.

archive

depends on the platform, as follows:

Environment	**Archive**
Guardian	is the pathname of the archive file to be created or modified. Archives created in the Guardian environment are given file code 700 on TNS/R, and file code 800 on TNS/E platforms.
OSS	is the filename of the archive file to be created or modified.
PC	is the filename of the archive file to be created or modified. It can be partially or fully qualified, and can include system and folder names.

The maximum size of an archive file in the Guardian environment is 128,073,728 bytes. If operations on an archive file cause it to exceed that size, `ar` returns an error and the archive file becomes corrupted.

file

can be PIC or non-PIC and depends on the platform as follows:

Environment	**File**
Guardian	is the fully qualified or partially qualified filename of a file whose file code is 100, 180, 700, or 800. If partially qualified, the value of the #DEFAULTS DEFINE supplies the missing components of the filename.
OSS	is the pathname of a filename.
PC	is a fully qualified or partially qualified pathname, which can include system or folder names. Wild-card characters (`?` and `*`) can be used in the filename portion of the path, but cannot be used in folder names.

The possible combinations of options and operands are:

```
ar -d [-v] [-l] archive file ...
ar -m [-abilv] [position-name] archive file ...
ar -p [-v] [-s] archive [file ...]
ar -q [-clv] archive [file ...]
ar -r [-cuv] [-abil] [position-name] archive [file ...]
ar -t [-v] [-s] [-Wfiletype] archive [file ...]
ar -x [-v] [-sCT] archive [file ...]
```

`ar` accepts many kinds of files as archive members. `ar` recognizes three kinds of HP object files: TNS, TNS/R, and TNS/E. All other files, including text files, are considered target-independent files. Only archives composed entirely of the same kind of HP object file and target-independent files contain an archive symbol table and are suitable for use by Binder or the linkers`(nld, ld or eld)`.

If `ar` detects mixing of the kinds of HP object files, it generates the archive but does not generate a symbol table, issuing an appropriate warning message instead.

When an archive contains a mix of TNS, TNS/R, or TNS/E object files, it is not usable by either Binder or the linkers `(nld, ld, or eld)`because no symbol table is generated. When such an archive is generated on Windows platforms, however, no error message is displayed because TNS files are not recognized as object files in that environment.

An archive symbol table is created as the first file member of the archive file for a successful archive operation when there is at least one object file in the archive. The symbol table is maintained by `ar` and is used by Binder or the linkers`(nld, ld or eld)` to search the archive. Whenever `ar` is used to create or update the contents of such an archive, `ar` rebuilds the symbol table. The `-s` option of `ar` forces the symbol table to be rebuilt.

An archive file embedded as a member of another archive file is not usable by Binder or the linkers`(nld, ld or eld)`.

A file within an archive is named by a filename, which is the last component of the pathname used when the file was entered into the archive. The comparison of a *`file`* operand to the name of a file in an archive is performed by comparing the last component of the operand to the name of the archive file. In the Guardian environment and on platforms running Windows, this comparison is case-insensitive.

Multiple files in an archive can have the same name. In such a case, however, each *`file`* and *`position-name`* operand matches only the first archive file having a name that is the same as the last component of the operand.

In the OSS environment, `ar` accepts OSS files as archive members. Archive libraries built by `ar` in any environment can be used for linking in any environment where Binder or the linkers`(nld, ld or eld)` run, provided the archive contains a symbol table and the appropriate kind of HP object file for the linker used.

It is your responsibility to ensure that archive members are appropriate for the target environment; for example, archive members must be compiled as OSS targets when they are to be used to construct an application that will run in the OSS environment.

For more information on the `ar` utility, see the Open System Services Shell and Utilities Manual.

4 eNOFT Diagnostic Messages

`eNOFT` sends all information to the standard output and does not differentiate error messages from its standard output when redirection of output is specified. A return code of "1" is generated on fatal termination and "0" (EXIT_SUCCESS) otherwise.

In interactive mode, messages that appear in the output listing fall into one of four severity levels:

Fatal Errors on page 4-1

Data Errors on page 4-1

Syntax Errors on page 4-1

Warnings on page 4-2.

Fatal Errors

Fatal errors are generated when memory cannot be allocated, most likely due to an internal problem with the program; for example, illegal access into memory.

```
enoft> FATAL ERROR *** [code]
<description of error>
```

where code is a value from 1 to 999.

Memory allocated data are destroyed and the program aborts with a return code of "1" (EXIT_FAILURE).

Data Errors

This type of error is generated when `eNOFT` cannot continue processing because the object file is incomplete or damaged or if the specified command is not applicable to the target object type.

```
enoft> DATA ERROR *** [code]
<description of error>
```

where code is a value from 1000 to 1999.

In interactive mode, `eNOFT` returns a prompt after the error message is displayed. In command-line mode, `eNOFT` continues with the next command after the error message is displayed.

Syntax Errors

This type of error is generated when `eNOFT` cannot recognize the entered command or its syntax.

```
enoft> SYNTAX ERROR *** [code]
<description of error>
```

where code is a value from 3000 to 3999.

In interactive mode, `eNOFT` returns a prompt after the error message is displayed. In command-line mode, `eNOFT` continues with the next command after the error message is displayed.

Warnings

`eNOFT` generates a recovery mode message and course of action.

```
enoft> WARNING *** [code]
<description of error and corrective action>
```

where code is a value from 2000 to 2999.

`eNOFT` continues processing the command in accordance to its understanding of the user intent.

5 ar Diagnostic Messages

The `ar` utility produces messages when errors occur in command input or in data on which the utility is operating. The following messages are in alphabetic order.

```
ar: archive: Guardian or User Defined Error 43
             Unable to obtain disk space for file extent.
```

Cause. The maximum size of an archive file is 128,073,728 bytes. If operations on the archive file cause the file to exceed that size limit, the `ar` command returns an error message and the archive file becomes corrupted because `ar` cannot create the symbol table information.

Recovery. See the Guardian Procedure Errors and Messages Manual.

```
ar: archive: Guardian or User Defined Error 45
             The resulting file size exceeds 128,073,728
             bytes and the file is not a valid archive
             file.
```

Cause. The maximum size of an archive file is 128,073,728 bytes. If operations on the archive file cause the file to exceed that size limit, the `ar` command returns an error message and the archive file becomes corrupted because `ar` cannot create the symbol table information.

Recovery. See the Guardian Procedure Errors and Messages Manual.

```
ar: cannot create archive symbol table due to ELF/TNS mix.
```

Cause. This is an advisory message. Object files for the NonStop servers are linked together using the linking program Binder. TNS/R object files are linked together using the linking utility `nld`. TNS/E object files are linked using `eld`. `ar` attempts to build a symbol table for the archive so that the linking program can know the archive contents. A different symbol table is required for each linking program, so `ar` cannot build a single symbol table when both types of object files are represented in the archive. If the archive is not to be used for linking, no action is required. The archive contains all the members specified.

Recovery. If the archive is to be used for linking, separate the members into separate archives, one containing TNS object files and the others containing either TNS/R or TNS/E object files. Note that you may mix PIC and non-PIC TNS/R object files in an archive, but you may not have TNS/R and TNS/E PIC files in the same archive.

```
ar: member: encountered error errnum.
```

Cause. When building an archive in the Guardian environment, `ar` encountered a File System error identified by the number *`errnum`* when processing the specified member. The archive should be intact except for the member that caused the error.

Recovery. Correct the error condition and add the member to the archive using the `-r` option, also specifying `a` or `b` if the position of the member is relevant.

```
ar: member: file too large.
```

Cause. When building an archive in the Guardian environment, the maximum size for a single member is approximately 128,073,728 bytes.

Recovery. Rebuild the archive, excluding the file that is too large.

```
ar: member: invalid file type.
```

Cause. When building an archive in the Guardian environment, each member file must be an odd, unstructured disk file. In particular, Edit-format (code 101) files are not allowed in archives.

Recovery. Rebuild the archive, excluding files that are of invalid types.

```
archive, cannot close to reposition for symbol table.
```

Cause. An I/O error occurred when the specified archive was being closed prior to constructing the archive symbol table.

Recovery. Correct the condition that caused the I/O error, then build the symbol table by using `ar` with the `-ts` options.

```
archive, cannot open.
```

Cause. An error occurred when `ar` attempted to open the specified archive.

Recovery. Identify the error and remedy the situation, then rebuild the archive.

```
archive, cannot open archive to set permissions.
```

Cause. An I/O error occurred in the Guardian environment when `ar` attempted to open the specified archive to set the file permissions.

Recovery. Use the FUP utility to look at the permissions on the archive and make any corrections needed.

```
archive, cannot reopen to add symbol table.
```

Cause. An I/O error occurred when the specified archive was being reopened prior to constructing the archive symbol table.

Recovery. Correct the condition that caused the I/O error, then build the symbol table by using `ar` with the `-ts` options.

```
archive, cannot set archive permissions.
```

Cause. An I/O error occurred in the Guardian environment when `ar` attempted to set the file permissions on the specified archive.

Recovery. Use the FUP utility to look at the permissions on the archive and make any corrections needed.

```
archive, error on close.
```

Cause. When running in the Guardian environment, `ar` encountered an error closing the specified archive.

Recovery. Correct the error condition and rebuild the archive.

```
archive, error on reopen.
```

Cause. When running in the Guardian environment, `ar` encountered an error reopening the specified archive.

Recovery. Correct the error condition and rebuild the archive.

```
archive, not a valid archive file.
```

Cause. In the Guardian environment, a file specified as an existing archive must be an odd, unstructured file having file code 700 or 800.

Recovery. If `archive` refers to a file that is not currently in archive format, delete that file and reenter the `ar` command. If `archive` refers to an archive that was built by the `ar` utility in another environment, check the file code of that file using the FUP utility; if the file code is not 700 or 800, change it using FUP.

```
cannot malloc space for symbol table.
```

Cause. `ar` cannot obtain additional memory space for the symbol table. The archive file, if created, will not be usable by `nld or eld`.

Recovery. Retry the command when there are fewer active processes in the system or break up a large archive file into smaller archive files.

```
cannot realloc space.
```

Cause. `ar` cannot obtain additional memory space for the symbol table. The archive file, if created, will not be usable by `nld` or `eld`.

Recovery. Break the archive into several smaller archives.

```
corrupted object file <filename>.
```

Cause. `ar` detected inconsistencies in the object file member and cannot finish its operation. The archive symbol table is unusable by `nld or eld`.

Recovery. Either remove the object file member from the archive file or obtain a valid copy of the object file and repeat the `ar` operation.

```
error during operation, archive archive may not contain
correct symbol table usable by the binder/nld/eld.
```

Cause. An error occurred during the operation; for example, a member file cannot be found. The archive symbol table contained in the archive may have been corrupted or may not contain up-to-date information required by `nld` or `eld` for resolving external references.

Recovery. Identify the source of the error and remedy the situation so that the archive operation can be finished normally. The `-s` option can also be used to restore or regenerate the symbol table.

```
file name filename: filename too long for filesystem.
```

Cause. During an extract (`-x`) operation, the filename of the specified file is longer than the maximum supported by the File System; the specific member is not extracted.

Recovery. Use the `-T` option to truncate the filename to the maximum length allowed by the File System during extraction.

```
illegal option combination for option.
```

Effect. The identified option combination is not allowed.

Recovery. Reenter the command correctly.

```
member already exists.
```

Cause. This is an advisory message. It warns when an archive member is not extracted because a file of that name already exists.

Recovery. To overwrite the existing file, either purge it before using `ar` or be sure not to use the `C` option on the `ar` command.

```
member: archive: bad file format.
```

Cause. The specified archive has an invalid file format.

Recovery. Rebuild the archive.

```
member, cannot extract because destination is not in Guardian
filespace.
```

Cause. When running in the Guardian environment, `ar` does not allow files to be extracted to the OSS file space.

Recovery. Specify a location in the Guardian file space where members can be extracted.

```
member: not found in archive.
```

Cause. The file *member* specified in the file operand is not in the archive. The `-t` option can be used to find out what members exist in the archive.

Recovery. Reenter the command with correct member names.

```
no position operand specified.
```

Cause. A positional option must be followed by a position operand preceding the name of the archive.

Recovery. Reenter the command correctly.

```
no archive members specified.
```

Cause. An `ar` command with the operational option specified requires file-member operands.

Recovery. Reenter the command correctly.

```
no archive specified.
```

An archive file is not specified in the command. Usually this indicates that either no operand or one operand (when a position operand is required) is specified in the command. Reenter the command with the correct number of operands.

```
one of the options -dmpqrtx is required.
```

Cause. One of the listed options is required.

Recovery. Reenter the command correctly.

```
only one of -a and -[bi] options allowed.
```

Cause. Only one positional option is allowed in an `ar` command.

Recovery. Reenter the command correctly.

```
posname: archive member not found.
```

Cause. The specified position operand is not in the archive. The `-t` option can be used to find out what members exist in the archive.

Recovery. Reenter the command with correct member names as the position operand.

```
W option is not a recognized flag.
```

Cause. The flag specified with the `-W` option is not recognized by `ar`.

Recovery. Reenter the command correctly.

A TNS/E Native Object Files

This appendix contains the following information:

The Object File Format - the types of object files and their content.

Code and Data Sections - the "ordinary" code and data sections that come from application source code, possibly with additions by the compiler or linker.

Relocation Tables - when code is relocated, who resolves the address and prepares relocation tables?

The DWARF Symbol Table - this table contains information used by debuggers and the Cobol compiler.

Archives - contains an extension of material covered in a previous section of this manual.

Tools That Work With Object Files - a quick look at which NSK tools use object files.

The Object File Format

This general information may also be found in the *eld Manua*l.

Basic Properties of Object Files

User versions of TNS/E tools may run in the following places:

- All TNS/E versions of the NSK operating system, including both the Guardian and OSS “personalities” of NSK.
- Some TNS/R versions of the NSK operating system, at least in the Guardian personality.
- Appropriate versions of the Windows operating system on PC’s.

TNS/E object files only run on TNS/E.

On Guardian, and in Guardian subvolumes of OSS, object files are unstructured files that are “odd unstructured”, the same as in TNS/R.

On Guardian, and in Guardian subvolumes of OSS, TNS/E object files have the file code 800.

TNS/E object files use the 64-bit version of the ELF file format.

TNS/E object files are big endian. This means that all their data is big endian. Code on the IPF (Itanium Processor Family) platform is always little endian.

Types of TNS/E Object Files

There are the following four types of TNS/E object files.

Table A-1. Types of TNS/E Object Files

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

Collectively, programs and DLL's are called *loadfiles*. Loadfiles and import libraries are built by the linker.

This appendix describes all four types of object files. The main distinctions occur between linkfiles and loadfiles. There is little difference between a program and a DLL as far as the file format is concerned, and an import library is a subset of what is in a DLL.

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

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

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

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

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

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

How to Distinguish the Different Types of Object Files

The first four bytes of an ELF file (in the ELF header) identify the file as an ELF file.

The fifth byte, named *e_ident [EI_CLASS]*, tells if it is the 32-bit or 64-bit version of ELF. This distinguishes between TNS/R and TNS/E object files.

The *e_machine* field of the ELF header identifies the target platform. This also distinguishes between TNS/R and TNS/E object files.

The *e_type* field of the ELF header distinguishes among the four types of TNS/E object files described in this section, except that the same value, *ET_DYN,* is used both for DLL's and import libraries.

When *e_type* = *ET_DYN*, the *EF_TANDEM_IMPORT_LIB* bit of the *e_flags* field tells if it is a DLL or an import library. When it is an import library, the *EF_TANDEM_IMP_LIB_COMPLETE* bit tells if it is complete or incomplete.

The *EF_TANDEM_IMPLICIT_LIB* bit of the *e_flags* field tells if this DLL is one of the implicit libraries, and is also set in the import library that represents the implicit libraries. The import library that represents the implicit libraries is also identified by the DLL name "__IMPLICIT_LIB__" found in the *DT_SONAME* record of the *.dynamic* section.

Summary of the Contents of an Object File

This appendix does not specify the ordering of sections within linkfiles. Compilers and the assembler are free to arrange sections as they wish, and so can the linker when it creates a linkfile with the *-r* option. The following is a list of the things that may exist in linkfiles:

ELF Header

Stack Unwinding Information (*.IA_64.unwind* and *.IA_64.unwind_info*)

Text Sections (sections whose names begin *.text* or *.restext*)

User Data Sections (*.data, .sdata, .bss, .sbss*, *.rdata*, *.srdata*, and *.rconst*)

A *.tandem_info* section (possibly abbreviated to four bytes)

The *.procinfo* and *.procnames* Sections

DWARF Symbol Table Sections

Relocation Table Sections (*.rela.x*, where *.x* could be any of the section names listed above)

ELF Symbol Table Sections (*.symtab* and *.strtab*)

Source RTDU Sections (*.rtdu.index, .rtdu.names,* and *.rtdu.data*)

ELF Section Headers and the *.shstrtab* Section

This appendix does, however, specify the ordering of sections within loadfiles and import libraries, as shown in Contents of a Loadfile or Import Library on page A-5.

It is also possible for the compilers or assembler to create sections of names not listed here. The characteristics of such sections, as listed in their ELF section headers, would tell the linker what to do with them, and they would be propagated by the linker into its output file.

Linkfiles also contain a section named *.comment*, but it is discarded by the linker.

In a loadfile, some of the sections are organized into segments. There is always a text segment, which comes at the beginning of the file. There may be a gateway segment. There may be either one or two data segments. When there are two data segments, they are called the *data constant* segment, followed by the *data variable* segment.

The first column in the table on the following page lists the items that may be found in a loadfile, in the order they would exist. Note that the text segment is always the first segment in the file, and that there may be one or two data segments. Note that the placement of the *.gateway* section (equivalent to the gateway segment) depends on whether the loadfile is a program or a DLL, and that the placement of the *.data* section depends on whether there are one or two data segments. The last two columns have an “X” next to those sections that may be referenced with 22-bit global pointer (GP) -relative addressing, or that may be found in import libraries, respectively.

The segments are loaded into virtual memory. The layout in virtual memory is the same as in the file within each segment, but there are choices for where each segment is placed into virtual memory.

The *.sbss* and *.bss* sections don’t actually take up any space in loadfiles. The table only shows where they would be placed in virtual memory.

Table A-2. Contents of a Loadfile or Import Library

Loadfile Contents		GP-Relative	Import LIbrary
ELF Header			X
ELF Program Headers			X
.tandem_info			X
.lic			
.dynamic			X
.liblist			X
.dynsym.gblzd			X
.hash.gblzd			X
.hashval.gblzd			
.rela.gblzd			
.dynstr2			X
.IA_64.unwind			
.IA_64.unwind_info			
.IA_64.unwind.strings			
.rconst			
.plt			
.restext			
.text			
.hash			X
.dynsym			X
.dynstr			X
.hashval			
.rela.dyn			
.gateway	(for a program)		
.data	(can have more than one data segment)		
.rdata			
.fptr			
.srdata		X	
.got		X	
.IA_64.pltoff		X	
.sdata		X	
.sbss		X	
.bss			

Table A-2. Contents of a Loadfile or Import Library

Loadfile Contents		**GP-Relative**	**Import LIbrary**
.gateway	(for a DLL)		
DWARF Symbol Table Sections			X
.source.rdtu	(if present, there are three of them.)		
.object.rdtu	(if present, there are three of them.)		
.shstrtab			X
ELF Section Headers			X

Note that the sections from *.got* through *.sbss* are purposely kept together as much as possible, since they are all referenced with GP-relative addressing. However, when there are two data segments, the *.data* section is allowed to intrude among these sections.

Both the data constant segment and data variable segment can have data that requires modification by *rld* when loaded into memory. The difference is that the data constant segment cannot be modified thereafter, while the data variable segment can.

The following is a brief description of each of the items that can occur in a linkfile or loadfile. Unless otherwise stated, a section is not required to be present if, based on its description, it would not contain any useful information for a given object file.

`ELF Header`

This contains header information for the entire file. It is always found at the start of an ELF file.

`ELF Program Headers`

These contain information that summarizes the main parts of the object file required for loading into memory. Program headers are required in loadfiles and import libraries.

`.tandem_info` `Section`

This contains more information of interest to the operating system. It is required in loadfiles and import libraries. It also exists in linkfiles because some of its fields are also meaningful there.

`.lic` `Section`

This contains information about the DLL's that were used to preset this loadfile. It is required in a loadfile, as a placeholder even if the loadfile is not preset.

`.dynamic` `Section`

This contains information needed by the runtime loader, such as the addresses of the *.liblist* through *rela.dyn* sections. It is required in loadfiles and import libraries.

.liblist Section

In a loadfile, this tells the names of the DLL's that were in the linker command stream when the linker built this loadfile. In an import library that represents a single DLL it contains the same information as in that DLL.

.dynsym.gblzd Section

This is a symbol table section, similar to the *.dynsym* section (see below), but just for globalized symbols. It may be present in loadfiles and import libraries.

.hash.gblzd Section

This is a hash table section, similar to the *.hash* section (see below), but for looking up symbols in the *.dynsym.gblzd* section.

.hashval.gblzd Section

This is similar to the *.hashval* section (see below), but providing information about the symbols in the *.dynsym.gblzd* section.

.rela.gblzd Section

This is similar to the *.rela.dyn* section (see below), but for the relocation sites whose targets are globalized symbols.

.dynstr2 Section

This is a string space that is pointed at from the *.dynamic*, *.liblist*, and *.dynsym.gblzd* sections.

Stack Unwinding Sections

These contain information for stack unwinding. Note that there are two such sections in a linkfile (not counting the relocation table section named *.rela.IA_64.unwind*), and three such sections in a loadfile.

.rconst Section

This contains application-defined initialized data that does not get modified at runtime, and does not contain addresses that might need modification when the loadfile is first brought into memory. This may never be created by a compilation or assembly, but when the linker sees an input section named *.rdata* that contains no relocation sites it renames the section to *.rconst*.

.plt Section

This section contains *import stubs*. An import stub is created by the linker in a loadfile when the linker cannot guarantee that the target of an IP-relative procedure call will be resolved within the same loadfile.

Text Sections

Text sections contain application-defined executable code (procedures). The object file design also allows them to contain data, but that is not expected to happen. In linkfiles, there can be any number of text sections. Their names must begin either *.text* or *.restext*, corresponding to whether they contain non-resident or resident text, respectively. In loadfiles, all the sections that had names beginning *.text* are combined into a single section named *.text*, and similarly for *.restext*, and the *.restext* section (if it exists) comes before the *.text* section. A text section is required in a program, because there must be a main entry point. Text sections in a loadfile can contain *branch stubs*, which are generated by the linker when a procedure call would need to jump farther than its instruction format allows.

.hash Section

This is a hash table for looking up symbols in the *.dynsym* section. It is required in loadfiles and import libraries.

.dynsym Section

This is the *dynamic symbol table*. It contains information about symbols referenced in this loadfile or exported from this loadfile, other than globalized symbols. It is required in loadfiles and import libraries.

.dynstr Section

This is a string space that is pointed at from the *.dynsym* section. It is required in loadfiles and import libraries.

.hashval Section

This contains precomputed hash values for the symbols listed in the *.dynsym* section. It is required in loadfiles.

.rela.dyn Section

This is the *dynamic relocation table.* It contains descriptions of the relocation sites within this loadfile whose targets are the symbols listed in the *.dynsym* section.

.gateway Section

This contains *gateways*. A gateway is created for each procedure entry point that has the *CALLABLE* or *KERNEL_CALLABLE* attribute.

.data Section

This contains application-defined initialized data, but doesn't have either of the restrictions that make it possible to put data into the *.rdata* or *.sdata* section.

`.rdata Section`

This contains application-defined initialized data that does not get modified at runtime (although the initial values may be addresses that need modification when the loadfile is first brought into memory).

`.fptr Section`

This section contains *official function descriptors*. An official function descriptor contains the address and GP value for a procedure that exists in this loadfile. Procedure pointers point at official function descriptors. An official function descriptor is only created for a procedure if the address of that procedure is taken in the same loadfile, or if the procedure is exported from the loadfile.

`.srdata Section`

This contains application-defined initialized data that does not get modified at runtime (although the initial values may be addresses that need modification when the loadfile is first brought into memory), and that furthermore is “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference will be in the same loadfile.

`.got Section`

This is the *global offset table*, which contains addresses of data items that are referenced indirectly, as well as the addresses of official function descriptors and EnterPriv labels. The linker creates entries in the *.got* section as necessary. The entries in the *.got* section are found by 22-bit GP-relative addressing.

`.IA_64.pltoff Section`

This section contains *local function descriptors*. A local function descriptor contains the address and GP value for a procedure that is referenced from this loadfile. Direct procedure calls (that is, not involving procedure pointers) use these local function descriptors. The linker creates entries in the *.IA_64.pltoff* section as necessary. The entries in the *.IA_64.pltoff* section are found by 22-bit GP-relative addressing.

`.sdata Section`

This contains application-defined initialized “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference will be in the same loadfile.

`.sbss Section`

This contains application-defined uninitialized “small” data for which 22-bit GP-relative addressing is used because the compiler or assembler can guarantee that the target of the reference will be in the same loadfile. This section occupies no space in an object file, but rather reserves memory space that is automatically initialized to zero. The object file design supports such sections, although compilers might not use them.

`.bss` Section

This contains application-defined uninitialized data, but this section doesn't have the restriction that makes it possible to put data into the *.sbss* section. It occupies no space in an object file, but rather reserves memory space that is automatically initialized to zero. The object file design supports such sections, although compilers might not use them. The linker allocates *.bss* sections in loadfiles to contain what the compiler called common data.

`.rela.x` Sections

These sections describe relocation sites within linkfiles. Relocation sites can be within code or data sections, including unwind function sections, the *.procinfo* section, and the DWARF sections. A *.rela.x* section is required in a linkfile for each section named *.x* that has relocation sites. For example, *rela.data* describes the relocation sites in the *.data* section.

`.symtab` Section

This is the *ELF symbol table*. It is required in linkfiles. It contains information about symbols whose names are meaningful to the linker.

`.strtab` Section

This is a string space that is pointed at from the *.symtab* section. It is required in linkfiles.

`.procinfo` Section

This section provides information about procedures and subprocedures.

`.procnames` Section

This is a string space pointed at by the *.procinfo* section.

DWARF Symbol Table Sections

These sections contain information for the debugger and for the COBOL compiler. There are several sections that collectively form the DWARF symbol table.

Source RTDU Sections

These are sections that represent source RTDU's, which are part of the SQL/MP implementation. These can exist only in linkfiles and programs. In a linkfile that is created by compiling a source file with embedded SQL/MP, the set of source RTDU's is represented by three sections. In a program, the set of source RTDU's is also represented by three sections, although not with the same section names as in a linkfile.

`Object RTDU Sections`

An object RTDU, which is part of the SQL/MP implementation, can be placed into a program by a tool named SQLCOMP. The object RTDU is represented by three sections.

.shstrtab `Section`

This is a string space that is pointed at from the ELF section headers. It is required.

`ELF Section Headers`

These contain header information to describe everything in the object file, except for the ELF header, the program headers, the section headers themselves, and possibly unused space within the object file.

A general principle behind the loadfile design is that the sections up through *.dynstr2* are expected to be small, and it can therefore be more efficient to have them all near the front. That is the reason that the *.dynstr2* section was invented, that is, to segregate out the strings needed by other small sections near the front of the file.

Another general principle is that, after all the things that are “small”, all the things that might need to be resident come next. More specifically, the *.restext* section needs to be resident (by definition), and if it is present then some other sections also need to be resident, and some don’t. All the other things that would also need to be resident are placed before the *.restext* section, so that the *.restext* section (if present) marks the end of the portion of the text segment that needs to be resident.

Note 1:

The way NSK arranges all the relocation table entries of a loadfile into sections named *rela.dyn* and *rela.gblzd* is not the standard way to do it for IPF, at least as followed by both Intel and HP.

In the Intel and HP implementations there is a different relocation table section for each of the sections of the loadfile that have relocation sites, which can include the *.got* section, *.IA_64.pltoff* section, and various user data sections. Each of these relocation tables has its own section header. In effect, that means that the relocation sites are sorted according to their locations, or at least according to which sections they are in. But then, most of these relocation table sections are placed consecutively in memory, with no rounding up of the space between them, so that you can think of them as one relocation table section, and that is what it looks like when you find them via the same entries in the *.dynamic* section that we use to find the dynamic relocation table. The exception to this in the Intel and HP implementations is the relocation table section for the *.IA_64.pltoff* section, which is found via an additional group of *.dynamic* section entries.

Instead of following this approach, the NSK implementation is based on the strategy used by SGI. Namely, relocation table entries are sorted by the target symbol. So, they could not be segregated by section as is done by Intel and HP.

NSK has also invented the *.rela.gblzd* section to handle globalized symbols in our implementation of C++. Other implementations take different approaches, not just to handle this specific feature of C++ but with regard to the issue of preemption in general. This invention of the *.rela.gblzd* section again follows the same strategy for NSK of segregating relocation table entries based on the target symbol, not based on the address of the relocation site.

The reason that Intel and HP separate out the relocation table entries for the *.IA_64.pltoff* section is related to the feature of "lazy evaluation", which NSK does not support (and which is not described in this appendix).

Note 2:

The *.got* is used to make indirect references to data, while the *.IA_64.pltoff* section is used to make indirect references to procedures. Instead of these two names, the names used by HP are more sensible, namely, *.dlt* ("data linkage table") and *.plt* ("procedure linkage table"), respectively. HP has a different name for the section of import stubs, which NSK calls the *.plt* section.

Also, note that the *.plt* section of import stubs makes references to the *.IA_64.pltoff* section of local function descriptors. That seems backwards, because you would think that a section named *pltoff* would contain "offsets" into a section named *plt*.

The strange names that NSK uses are the ones found in the IPF standards documents.

The 32-Bit and 64-Bit Programming Models

According to the *IPF-Specific ABI Document* there is a choice of two programming models, named *ILP32* and *LP64.* In the standard, this means two different things. On the one hand, it tells something about your C compiler, namely, whether the sizes of pointers and the predefined type *long* are 32-bits or 64-bits. On the other hand, it also tells whether 32-bit or 64-bit addresses are stored in object files.

At the present time, the NSK C compiler supports the 32-bit model. In the future, NSK will also support the 64-bit model. When we do that, it will not be possible for general users to mix the two within the same loadfile, but it will be possible to mix different types of loadfiles in the same process. Regardless of the data model supported by the compiler, however, NSK always use the 64-bit format for object files. That allows us to internally put 64-bit code into loadfiles that are otherwise 32-bit. When TNS/E object files contain values that are 32-bit addresses, they correspond to the actual 64-bit values supported by the underlying hardware via sign extension.

Code and Data Sections

This subsection discusses the "ordinary" code and data sections that come from application source code, possibly with some things added by the compiler or linker. Special types of data sections, such as the stack unwinding information, the .procinfo and .procnames sections, the DWARF sections, and the various linker-created sections in loadfiles, are not detailed here.

User Code

In linkfiles there can be many text sections. The sections whose names begin .text contain procedures and subprocedures that are not resident. The sections whose names begin .restext contain procedures and subprocedures that are resident.

When the linker is building a new linkfile it concatenates each of the text sections from the various input files into a section of the same name in the output file. On the other hand, in loadfiles, all the non-resident code is combined into a single .text section, and all the resident code into a single .restext section. The text sections of a loadfile may also contain linker-generated branch stubs, which are not present in linkfiles.

Some procedures are global, which means their names are meaningful across separate compilations. All references to global procedures must be marked with relocation table entries. When there are duplicate copies of global procedures, the linker picks one to use, and the relocation table entries are used by the linker to make sure all references go to the copy that was picked.

If a procedure is in a section whose name begins either ".text." or ".restext.", and the rest of the name is the same as that of the procedure, this is an indication by the compiler that, if this is an unused copy of the procedure, then in fact the entire section containing it may be ignored by the linker. In that case, the linker ignores that input section, thus making the resulting code space smaller.

Text sections are allowed to contain data, such as branch tables. This should not happen in sections that are marked for omission as in the previous paragraph.

The .procnames and .procinfo sections provide additional information about procedures and subprocedures in linkfiles.

The size of executable code is always a multiple of 16 bytes, because instructions are grouped into 128-bit bundles. However, even when a text section contains data, its total size must still be a multiple of 16 bytes. (Actually, NSK compilers usually say that text sections must be aligned on 32-byte boundaries, and similarly each procedure within a code section starts at an offset within that section that is a multiple of 32 bytes. Larger alignments can also be specified in assembler source files. When space is wasted between procedures, the assembler fills that space with no-ops.)

The total size of a text section in any linkfile must not exceed 16 MB, so that the linker can add branch stubs to the section if necessary. Also, it is suggested that compilers not put all the code of a compilation into one code section, but rather divide it into multiple code sections, such as by putting each procedure into its own section. That is a way to avoid running into the 16 MB limit, either directly as the result of a compilation, or later after the linker has combined many separate compilations into a single linkfile with the -r option, since the linker will concatenate input sections that have the same name.

Certain procedures may be included just to identify an object file. Such a procedure is called a VPROC ("version procedure"). The names of such procedures would always be found in the .procinfo section of a linkfile or in the stack unwinding information of a loadfile. Depending on whether a VPROC was visible outside its compilation, or exported from its loadfile, it might also be found in the ELF symbol table of a linkfile, or the dynamic symbol table of a loadfile or import library.

User Data

The .data (and .sdata) sections are for initialized data, while .bss (and .sbss) are for uninitialized data. However, if a data item is initialized to all zeros, the compiler may treat it as uninitialized data. That is possible, because all uninitialized data is automatically initialized to zeroes by NSK.

When the linker combines a set of linkfiles into a new file it usually concatenates each of the user's data sections from the various input files into a section of the same name in the output file. For example, some of the input files may have a section named .data, and then the output file would also have a section named .data, and it would be the concatenation of the .data sections that existed in the input files. The names of typical user data sections, and what each one means, were listed near the beginning of this document. Like text sections, sizes of data sections must be multiples of 16 bytes.

The exception to the general rule given in the previous paragraph is that, if an input section has the name .rdata, but doesn't contain any relocation sites, then the linker changes its name to .rconst for the output file. Note that a similar optimization is not done for .srdata because that is GP-addressable.

The sections named .sdata, .srdata, and .sbss are called small data sections with the meaning that the compiler might choose to put "small" data items into them (that is, data items whose sizes are no larger than 8 bytes). However, these sections actually have no such requirement. The real meaning of these sections is that the items placed here can be referenced directly by 22-bit GP-relative addressing, rather than getting their addresses out of the .got section. That is only correct to do if the compiler or assembler programmer can guarantee that the symbol cannot be preempted.

Linkfiles also have common data, which has not been allocated to any section. When the linker builds a loadfile it allocates common data in the .bss section.

The following is how the C compiler works:

Data that is global or large, and initialized to a non-zero value, is placed into .data.

Data that is global or large, and initialized to a zero value, is placed into .bss.

Data that is local and small, and initialized to a non-zero value, is placed into .sdata.

Data that is local and small, and initialized to a zero value, is placed into .sbss.

Data that is uninitialized is called common data.

Character strings are called local data items and placed into .rdata.

The MCB (Master Control Block)

The linker adds the MCB to the .data section of a program (or creates a section of this name if there was none before). The MCB is a data item that can be referenced by the name _MCB within the program. The linker only creates the MCB in programs (not DLL's), and only if the program makes a reference to the symbol named _MCB.

This is a description of the fields that are nonzero in the MCB of an object file.

The Check_quad field is an 8-byte string, where the first two and last two bytes each contain the value 0xAA and the middle four bytes contain the ASCII string "MCB ".

The Version_item field currently contains 0, but presumably could contain a different value in the future.

The Standard_C_streams bit is set to 1, rather than 0, to indicate that the program should use code 180 files for C text files, rather than code 101 files. The linker sets this bit to 1 when it creates the program if the -ansistreams option is specified or if the target platform is OSS.

The C_std_files_open bit is set to 1, rather than 0, to indicate that this program should automatically open the standard C/C++ I/O files. This linker sets this bit to 1 if the program contains a main procedure that is written in C or C++ and the -nostdfiles option is not specified.

The FP_format field is set to indicate the floating point type assumed by this program, repeating the information also found in the file header. 0 indicates that the Tandem floating point is required. 1 indicates that the IEEE floating point is required. 2 indicates neutral.

Relocation Tables

It is possible that the contents of one place in the code or data of an object file need to be filled in with the address of another place in the code or data, or in some other way based on such an address. If the compiler or assembler knows what needs to go there, without later modification by the linker or runtime loader, then that's the end of the story. But, if the linker or runtime loader will need to be involved, the compiler or assembler must indicate that location accordingly, by creating relocation tables in linkfiles to provide such information. Similarly, the linker must put relocation tables into loadfiles if there is still work for the runtime loader to do.

The place that needs to be filled in is called the relocation site. It would either be an operand within an executable instruction, which come in various sizes, or a data item, which would be a 32-bit or 64-bit integer. The place whose address needs to be calculated is called the target of the relocation. The relocation site is also said to be a reference to the target symbol.

The target of a relocation site is described by giving an offset relative to a symbol that is listed either in the .symtab section (in the case of a linkfile) or the .dynsym or .dynsym.gblzd section (in the case of a loadfile). If the symbol is of type STB_LOCAL then it must be defined with an address in this object file, and that is the address that is used for the symbol. If the symbol is of type STB_GLOBAL then the definition of the symbol that is used to resolve the reference might exist in this object file or in another object file.

The process of figuring out the target address is called resolving the reference. After a reference has been resolved, the proper way to fill in the contents of the relocation site depends on the site's relocation type.

The relocation types that can occur in linkfiles and loadfiles are different, and the names of the relocation table sections are different. In linkfiles, for each code or data section named .x that contains relocation sites there is a relocation table section named .rela.x that describes the relocation sites in that section. This also includes relocation tables needed to describe relocation sites in the .procinfo section, the unwind function sections, and the DWARF symbol table sections. In loadfiles there are relocation table sections named .rela.dyn and .rela.gblzd that describe all the relocation sites in the data segment of the loadfile. Loadfiles never have relocation sites in the text segment. The entries in .rela.dyn are for relocation sites whose target symbols are in .dynsym, while the entries in .rela.gblzd are for relocation sites whose target symbols are the globalized symbols listed in .dynsym.gblzd.

The format of the relocation information is the same in all cases. The ELF section type is SHT_RELA, and the format of a relocation table entry is the following:

```
typedef struct ELF64_Rela {
                        ELF64_Addr          r_offset;
                        ELF64_Xword         r_info;
                        ELF64_Xword         r_addend;
}Elf64_Rela
```

The size of this structure is 24 bytes.

In linkfiles, relocation table entries always completely describe what needs to be filled in at the corresponding relocation sites. So, it doesn't matter what is actually in the operands at the relocation sites. In fact, what is there should be zero, with the following two exceptions:

The value "-1" is filled in for relocation sites that point from DWARF information at executable code, when they correspond to unused copies of procedures.

Relocation sites that point from one DWARF section into another, that is, giving a section offset rather than an address, are also fixed up in linkfiles created by the linker.

For loadfiles the relocation types whose names begin R_IA_64_REL make use of the contents of the relocation site, rather than pointing at a target symbol These relocation table entries say that the contents of the relocation site need to be updated at runtime, or by the -alf option of the linker, based on how much the segment pointed at by the relocation site is rebased.

In loadfiles, the relocation sites whose targets were STB_LOCAL would only need to be updated if the loadfile was rebased. This can happen for DLL's, but not for main programs (not even by the linker's -alf or -alfp options). So, such relocation table entries may be omitted by the linker when it creates a main program.

In loadfiles, the elements of the .rela and .rela.gblzd sections are sorted by target symbol index. In particular, that means that all the entries with the same target symbol are consecutive. This includes the case of relocation types whose names begin R_IA_64_REL, which don't have a target symbol, so that the target symbol index is 0.

The DWARF Symbol Table

The DWARF symbol table contains information used by debuggers and by the COBOL compiler, whereas the .symtab, .dynsym, and .dynsym.gblzd sections contain information used by the linker and runtime loader.

The DWARF symbol table information in an import library that represents a single DLL is the same as the DWARF symbol table information that is present in the corresponding DLL. There is no DWARF symbol table information in the import library that represents the implicit libraries.

A file may be "stripped", meaning that it doesn't have debugging information in it. This means that the DWARF symbol table is not present. Note that it is even possible for a linkfile to be stripped. In other words, even after being stripped, a linkfile can still be processed by the linker, because the DWARF symbol table does not contain any information that is required by the linker. An import library can be stripped even if the corresponding DLL is not stripped.

DWARF information is updated by the linker corresponding to the effect of its -rename option. That is, the DWARF information does not look like what the compiler or assembler originally generated, but rather reflects how the symbol table information was changed by the -rename option.

DWARF Object File Sections

Here is a summary of the purposes of the DWARF sections that we use:

.debug_info

This is the main section of DWARF information. It is a tree of nodes, each node contains various attributes.

.debug_abbrev

This section provides additional information required to decode the information in the .debug_info section, including information about implementation-defined material.

.debug_line

This section contains information that tells how to map things to source line numbers.

.debug_line_nsk

This has a format similar to .debug_line, but to represent EDIT line numbers rather than sequential line numbers.

.debug_relocs

This section describes the places in DWARF sections of DLL's that contain code and data addresses, so that they can be updated by the -alf option of the linker when that option is used to rebase the DLL.

Archives

An archive is a single file that contains within it copies of other files, called the "members" of the archive. Archives are created by the tool named ar. An archive may be used for various purposes, one of which is to be an input for the linker. The linker uses archives as a source of linkfiles. Archives are not used at load time.

The format described here, used for TNS/E archives differs in various ways from what was used in the TNS/R implementation.

An archive contains "symbol table" information that tells which linkfile within the archive, if any, provides a definition for a given symbol. These would be the symbols defined in that linkfile and visible outside, that is, their binding is STB_GLOBAL and their st_scndx field is not SHN_UNDEF in the ELF symbol table.

The first eight bytes of an archive contain the string "!<arch>", followed by a newline character (ASCII LF). This identifies the file as an archive. After that the archive is a concatenation of "pieces", each of which contains the following items, which always begin at file offsets that are multiples of 2 bytes.

an ar_hdr structure

the contents of this piece

The first one or two pieces of the archive may be special. The first special piece is the archive symbol table, which is present if the archive contains any linkfiles. The other special piece is the "long member name string space", which is present if any of the names of the members of the archive are longer than 16 characters. The contents of the remaining pieces are the members of the archive.

Here is the declaration for the ar_hdr structure:

```
typedef struct ar_hdr {
char      ar_name [16];
char      ar_date [12];
char      ar_uid [6];
char      ar_gid [6];
char      ar_mode [8];
char      ar_size [10];
char      ar_fmag [2];
} ar_hdr;
```

The size of this structure is 60 bytes.

The ar_size field tells the size of the contents of this piece of the archive, and the ar_name field tells its name. When the name is less than 16 characters long, the rest of the field is filled with blanks. The other fields of the ar_hdr are all readable ASCII character fields.

In the ar_hdr for the symbol table piece, the ar_name is a single slash ("/").

The contents of the symbol table piece are the following (in this order):

a four-byte integer that tells the number of symbols in the symbol table piece
an array of four-byte integers
a string space (see below)

The integers mentioned above are binary integers (big endian).

The string space is a concatenation of strings, telling the names of the symbols in the symbol table piece. Each name is terminated by a zero byte. If the total size is odd, an extra zero byte at the end makes it even. These strings are in the same order as the previous array of four-byte integers. For each name, the corresponding four-byte integer tells the file offset within the archive for the ar_hdr of the member that defines that symbol. Symbols are only listed in the symbol table if they are defined somewhere. A symbol may be defined in more than one member, but the symbol table only points at one place.

In the ar_hdr for the long member name string space, the ar_name is two slashes ("//").

The long member name string space is a concatenation of strings, telling the names of the members whose names are longer than 16 bytes. Each name is terminated by a slash ("/") and a newline character. If the total size is odd, an extra newline character at the end makes it even.

In the ar_hdr for an archive member, the ar_name tells the name of the file that was placed into the archive. If the name is longer than 16 bytes then it is stored instead in the long member name string space and the ar_name field for the member consists of a slash ("/") followed by an ASCII string for the integer value that is the byte offset of the member's name in the long member name string space. Leading zeroes are removed from this string, and it is blank filled on the right.

The following is a summary of what is in an archive. Horizontal lines separate pieces of the archive. This example shows the case when there is a symbol table and a long member name string space.

!<arch>

---

ar_hdr for the symbol table
the number of symbols in the symbol table

file offset for the member that defines the first symbol
file offset for the member that defines the second symbol
...

---

name of the first symbol
name of the second symbol
...

---

ar_hdr for the long member name string space
the string space of long member names

---

ar_hdr for the first member
contents of the first member

---

ar_hdr for the second member
contents of the second member

...

Tools That Work With Object Files

Here is a list of some of the tools (that is, customer products) that read or write object files (or archives):

- Compilers and the assembler create object files.
- The linker (*eld*) reads and writes object files, and reads archives.
- `eNOFT` reads object files to display their contents.
- VPROC can read object files to print out version procedures (that is, a very special case of what `NOFT` does).
- The NSK operating system, including the runtime loader (*rld*), reads object files to bring them into memory.
- Debuggers read object files as well as their memory images, and can modify the memory images.
- The archive creation tool (its standard name is *ar*) reads object files, and reads and writes archives.
- SQLCOMP can read and write object files in order to create or update their object RTDU’s.

B

Differences Between eNOFT and NOFT

Architecture

NOFT supports TNS/R object files which include ELF and COFF object file structures. eNOFT supports TNS/E architecture, which is exclusively ELF.

All TNS/E object files are big endian files with DWARF2 debugging symbol tables. Code on the TNS/E platform is always little endian.

When accessing a code area of the object file (for example, .text), eNOFT displays in 16-byte "bundles" whereas NOFT displays in units of 4-byte "words". However eNOFT will display virtual addresses in the same format as NOFT: 32-bit hexadecimal values.

Debugging

NOFT uses the Third Eye symbol table where some of its tables are used for linking; eNOFT uses the DWARF2 symbol table which does not contain such information used by the object file linker "eld" or runtime loader "rld" . One consequence of this difference is in the behavior of eNOFT on stripped files. Commands that require the use of the debugging information are not supported on stripped files. Because the meaning of stripping is different between TNS/R and TNS/E architectures, eNOFT does not support the same commands that NOFT supports. See SET and RESET Commands on page B-1 and subsequent tables for details.

Displays

NOFT typically displays listings in a center-justified format. eNOFT displays are typically left-aligned with the object file offset value for the specified target section in the heading.

Summary of eNOFT Commands

The following tables list eNOFT commands and their NOFT equivalents.

Table B-1. SET and RESET Commands

NOFT	eNOFT	Alias	Options
RESET <set-cmd>			
SET <set-cmd> OFF	RESET	-	[* | set-cmd]
SET CASE	SET CASE	SC	{OFF | ON}

Table B-1. SET and RESET Commands

NOFT	eNOFT	Alias	Options
None	SET CPPNAME	SN	[MANGLE | DEMANGLE]
None	SET DISPLAY	SD	[BRIEF | B | DETAIL | D]
SET FORMAT	SET FORMAT	SF	{READABLE | R | ASCII | A | DECIMAL | D | HEX | H | ICODE | IC | INNERLIST | IN}
SET HISTORYBUFFER	SET HISTORYBUFFER	SHB	<num>
SET HISTORYWINDOW	SET HISTORYWINDOW	SHW	<num>
SET LINES	SET LINES	SL	<num>
SET SCOPEPROC			
PROC	SET SCOPEPROC	SSP	<proc-spec>
SET SCOPESOURCE			
SOURCE	SET SCOPESOURCE	SSS	<source-spec>
SET SORT			
ALPHA			
LOCATION	SET SORT		
SET SORT ALPHA			
SET SORT LOC	ST	{NONE | N| ALPHA | A | LOC | L}	
NUMBER	RESET SORT	-	-

Table B-2. Dump Commands

NOFT	eNOFT	Options	Set format	Set scope	Set sort
DUMPADDRESS					
<address>	DUMPADDRESS	<scope-range> [IN <format-specifier>]	Y	N	N
ALL					

Table B-2. Dump Commands

NOFT	eNOFT	Options	Set format	Set scope	Set sort
HEADERS	DUMPALL	[* | LIST]			
ALLTEXT					
RESTEXT					
TEXT					
USERGATE	DUMPCODE				
(default ICODE)	DC	Y	Y	N	
DATA					
LARGEDATA					
READONLY					
SMALLDATA	DUMPDATA (default HEX)	[IN <format-specifier>]	Y	Y	Y
DUMPOFFSET	DUMPOFFSET	<scope-range> [IN <format-specifier>]	Y	N	N
DUMPPROC	DUMPPROC	<proc-spec> [<scope-range>][IN <format-specifier>]	Y	N	N
ALL	DUMPSECTION	[* | section-name | <section-num>] [IN <format-specifier>]	Y	Y/N	N/Y
AUXSYMTBL					
EXTSYMTBL					
FILETBL					
INDFILETBL					
LINBRTBL					
LOCSYMTBL					
PROCTBL					
SYMHDR					

Table B-2. Dump Commands

NOFT	eNOFT	Options	Set format	Set scope	Set sort
SYMBOLS	DWARF	[INFO | ABBREV | LINE [ORDINAL]]	Y	Y	N
DYNAMIC	DYNAMIC	-	Y	N	N
FILEHDR	FILEHDR				
not applicable	FUNCDESC	-	Y	Y	Y
GOT	GOT	-	Y	Y	Y
HASH	HASH	-	N	Y	N
MSYM	HASHVAL	-	N	Y	Y
LISTSRLINFO	LIBLIST	-	Y	N	N
not applicable	LIC	-	Y	N	Y
not applicable	PROCINFO	-	Y	Y	Y
PROGHDRS	PROGHDRS				
RELOC					
DYNREL	RELOC	-	N	Y	Y
not applicable	RTDU	-	Y	N	Y
SECTHDRS	SECTHDRS				
DYNSTR					
DYNSTR2	STRTAB	[*| DYNSTR | DYNSTR2 | PROCNAMES | RTDU | SHSTRTAB | STRTAB | UNWIND]	N	Y	Y
DYNSYM					
ELFSYMTBL	SYMTAB	[* | EXPORT | E | PROC | P | DATA | D]	N	Y	N
TANDEMINFO					
SRLDIGEST	TANDEMINFO	-	Y	N	N
RUNTIMEPROC	UNWIND	-	Y	Y	Y
-	UNWINDINFO				

Table B-3. List Commands

NOFT	eNOFT	Options	Set format	Set scope
LAYOUT	LAYOUT	[* | CODE | DATA]	N	N
LISTATTRIBUTE	LISTATTRIBUTE	[DETAIL | D]	N	N
LISTCOMPILERS	LISTCOMPILERS	[DETAIL | D]	N	N
-	LISTDATA			
-	LISTDEBUG	[* | PROC | P | DATA | D] [DETAIL | D]		
LISTEXPORTS	LISTEXPORTS	-	N	N
LISTOPTIMIZE	LISTOPTIMIZE	[* |0|1|2]	N	Y
LISTPROC	LISTPROC	[* | <proc-spec>] [NOSUBPROC | NSP] [DETAIL | D]	N	Y
LISTSOURCE	LISTSOURCE	[* |<source-spec>] [DETAIL | D]	N	Y
LISTUNREFERENCED	LISTUNREFERENCED	[* | PROC | P | DATA | D] [DETAIL | D]	N	Y
LISTUNRESOLVED	LISTUNRESOLVED	[* | PROC | P | DATA | D]	N	Y
XREFPROC	XREFPROC	[* | <proc-spec>] [CALLEDBY | CALLS | BOTH] [DETAIL | D]	N	Y

Table B-4. File User Interface Commands

NOFT	eNOFT	Options
!		
FC	!	
FC	-	
Break Key	Break Key	-
CD	CD	[pathname]
COMMENT	COMMENT	-

Table B-4. File User Interface Commands

NOFT	eNOFT	Options
ENV	ENV	-
EXIT		
QUIT	EXIT	
QUIT	E	
Q	-	
FILE	FILE	objectfile
HELP		
HELP ALL		
HELP UNDOCUMENTED	HELP	[command | help-topic]
HISTORY	HISTORY	[<num>]
LOG		
SET LOG		
OUT		
SET OUT	LOG	
OUT	-	
OBEY	OBEY	infile
SHOW		
SET <set-cmd> ?		
OPTIONS	SHOW	[* | set-cmd]
SYSTEM		
VOLUME	VOLUME	[\<node>] [.$<volume>] [.<subvolume>]

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 or runtime. It is therefore a loadfile that offers functions or data for use by other loadfiles. For TNS/E, DLLs replace SRLs commonly associated with the TNS/R architecture. The object file linker `eld` generates DLLs for TNS/E (as does `ld` for the TNS/R DLLs). In UNIX, this type of file is known as a shared object file or dynamic shared object (DSO).

Dynamic loading. Loading and opening DLLs under programmatic control after the program is loaded and execution has begun.

EDIT Line Number. The conventional source line numbering convention is where the source lines are numbered sequentially using integers starting at 1. The Guardian EDIT text file (file code "101") uses a source line number convention where the lines are assigned numbers that have three places after the decimal point, and can be sparse within all such possible numbers.

ELF. This term stands for "executable and link format" and describes an extensible file structure that can deal with various target platforms. Like TNS/R, TNS/E uses the ELF file structure with Tandem extensions. However TNS/E is ELF all-inclusive whereas TNS/R uses both ELF and COFF file structures. All TNS/E compiler/assemblers, linkers, and loaders generate object files with this file structure.

Explicit library. Any library that is named in the libList of any client loadifle or is a user library of a client program.

Export. To provide a symbol definition for use by other loadfiles. A loadfile offers for export a symbol definition for use by other loadfiles that need a data item or function having that symbolic name.

Gateway. For every callable function there is a gateway; all calls to the function jump first to the gateway, which effects the transition to privileged state if the caller is not already privileged. There are two types of gateway pages, those that promote to kernel and those that promote to executive level.

Gblzd. globalized [symbol]

Globalized import. The import-control characteristic of a loadfile that allows it to import symbols from any loadfile in the loadList of the program with which it is loaded. When those loadfiles offer multiple definitions of the same symbol, those loadfiles are searched in loadList sequence and the first definition found takes precedence. See also searchList.

Globalized symbol. An exported symbol generated by the C++ compiler that may have multiple definitions, of which the linker and loader must assure only one is used throughout the process.

Hybrid file. This term describes a 'pseudo-DLL' that contains non-PIC text to allow a PIC process to call (as inputs) when building or relinking a program or DLL file. Hybrids do not exist in TNS/E.

Implicit library. A library supplied by HP that is available in the read-only and execute-only globally mapped address space shared by all processes without being specified to the linker or loader. The public libraries on TNS/E that replace System Code, System Library, and millicode. These libraries are called implicit because every loadfile is implicitly a user of them. Contrast with public DLLs, which are explicit because a loadfile explicitly asks to use a public DLL, although it does not specify where to find the public DLL. See also System library. and Public Libraries.

Implicit library import library (imp-imp). An import library that can be used by the Linker as a proxy for a set of implicit libraries. See Import library and Zimpimp file.

Import. To refer to a symbol definition from another loadfile. A loadfile imports a symbol definition when it needs a data item or function having that symbolic name.

Import control. The characteristic of a loadfile that determines from which other loadfiles it can import symbol definitions. The programmer sets a loadfile’s import control at link time. That import control can be localized, globalized, or semiglobalized. A loadfile’s import control governs the way the linker and loader construct that loadfile’s searchList and affects the search only for symbols required by that loadfile.

Import library. This term describes one type of a loadfile whereby only enough parts of the file are contained therein to allow the linker to resolve references, but not enough to expose its source code; that is, exports the symbols of the DLL . It is a file that can be used by the Linker as a proxy for one or more DLLs, but that cannot actually be loaded and run. It is useful in cross-linking. See Implicit library import library (imp-imp) and Zimpimp file.

Indirect reference (of a loadfile). A library in a loadfile’s searchList that is not named in its libList.

Instance. A particular case of a class of items, objects, or events. For example, a process is defined as one instance of the execution of a program; multiple processes might be executing the same program simultaneously. Also, instance data refers to global data of a program or library; each process has its own instance of this data.

Library. Generically, a collection of functions and data offered for use by clients. Libraries can exist as source files, linkable object files, archives (aggregated of linkfiles), and loadable object files. See also Loadable Library..

LibList. The list of libraries to be loaded along with a loadfile. However, it may not be the complete list of loadfiles that must be loaded; see loadList definition below.When linking the loadfile, the linker constructs the libList from the names of libraries specified in the linker's command stream; it stores the libList within the loadfile.

Libname. An attribute of a program loadfile, which can be set by the linker, specifying the name of a user library to be loaded with this program.

Linker. A utility whose basic function is to process one or more linkfiles to create a loadfile.

Linker platform. The system on which the linker executes. Also called *host* or *host platform*.

LIC. Library Import Characterization: A data string that characterizes the information used by a linker or loader to bind the global symbols of a particular loadfile. If the same loadfile is bound on two occasions, and its LIC has not changed, the two bindings will be the same. Thus it is possible to reuse a set of bindings if it has the same LIC as that determined for this loadlfile in the presence of the other loadfiles with which it is being loaded.

Linkfile. This term describes the output of the compiler and input to the linker. This object file has accompanying tables required to build it into a PIC loadfile and can be all or part of a loadfile. The code of a linkfile is not executable until linked. In the default mode, the linker process one or more linkfiles to produce a loadfile. This term is synonymous with the term "relinkable" in TNS/R .

Loader. A programming utility that transfers a program into memory so it can run. The mechanism that brings loadfiles into memory for execution, maps them into virtual address space, and resolves symbol references among them. Synonyms include run-time loader and run-time linker. The loader for TNS and for TNS/R native programs and libraries that are not position-independent code (PIC) is part of the operating system. For PIC loadfiles and all TNS/E native programs, the loader called `rld` works with the operating system to load programs and libraries.

Loadfile. hThis term describes the input to the runtime loader and default output of the linker. This object file may contain name references to symbols that exist in other loadfiles in the same process. Such references are typically resolved when the loadfiles are brought into memory by the runtime loader `rld` . This term is synonymous with the term "executable" file. An executable object code file is one that is ready for loading into memory and executing on the computer. Loadfiles are further classified as executable programs (containing a main routine at which to begin execution of that program) or executable libraries (supplying routines or variables to multiple programs or separately loaded libraries). A TNS code file might be both a loadfile and a linkfile. Native code files are never both. Contrast with Linkfile.

LoadList. A list of all the libraries that must be loaded for a given loadfile to execute. A loadfile's loadList includes all the libraries in the given loadfile's libList plus all the libraries in those loadfiles' libLists, and so on. It does not include the implicit libraries. The loadList order is the sequence in which these loadfiles are to be loaded when they are not already loaded by a previous operation. The loadList of the program includes all the loadfiles present in the process, in the order they were loaded.

Loadable Library. A loadfile that offers functions and data to other loadfiles. In this document, DLLs are such libraries. A library cannot be invoked externally, for example, by a RUN command; instead, it is invoked by calls or data references from client loadfiles. In TNS/E, functions and data can also be obtained from the system library and millicode.

Loader Library. A public library for loading PIC programs and libraries. It works in close cooperation with the operating system. It is called "`rld`" when loading a program and its libraries at process creation time. It also exports a set of functions for dynamic loading.

Localized. The import-control characteristic of a loadfile that allows it to import symbols only from the loadfile itself followed by the libraries in its libList, libraries that those libraries re-export, and from these, any successions of re-exported libraries.

MCB. The Master Control Block. This contains global information such as the product version number, valid file types, language dialects and floating point types that may be used.

Millicode library. Low-level library routines. Although separate from it, the millicode can be considered an adjunct of the system library.

Presetting. This is the process of resolving references to DLLs at linktime.

PIC. This term stands for 'position independent code' and describes a nomenclature associated with DLLs whereby PIC text contains references do not have to be resolved at link time. Executable code that need not be modified to run at different virtual addresses. External reference addresses appear only in a data area that can be modified by the loader; they do not appear in PIC code. PIC code is even more position independent than one might imagine from the term; it can be simultaneously mapped to different addresses for different processes in the same CPU. PIC introduces several new elements into ELF files, some of which are adapted from the Intel LP64 ELF structure. TNS/E supports only PIC files. TNS/R supports PIC and non-PIC file types.

Program. This term describes one type of loadfile that is capable of being run on the system. This is the main program and there can only be one program associated with a process.

Public Libraries. A set of libraries (offering widely-used functions) that are managed as part of the system, available to all users of the system, and in large part supplied by HP, although it is possible for customers and third parties to provide DLLs to be added to the public DLLs. A loadfile must explicitly reference a public library in order to access it.

Preempt. When the linker’s binding of a symbolic reference to a symbol defined in the same DLL is rebound by the loader to a definition in another loadfile.

Process. An instance of the execution of a program.

Re-exported library. A library whose symbols are made available by another DLL to any localized client of that DLL. Re-export is an attribute of the DLL's libList entry for that library. This attribute is specified by the DLL's programmer and recorded by the linker as a DLL is built. It affects only localized clients of the DLL. This feature allows a symbol to be moved from one DLL to another without relinking clients of the original DLL.

Re-exporting is transitive; that is, if A re-exports B and B re-exports C, then A re-exports C. Thus, re-exported libraries can re-export other libraries to form a succession of re-exported libraries of arbitrary length.

Region. The Itanium® architecture divides the address space into eight regions, indexed by the high-order three bits of the 64-bit address. TNS/E initially implements just two, regions 0 and 7: region 0 is mapped per-process; region 7 is shared by all processes. Sign extension places “negative” 32-bit addresses in region 7. Note that the high bit of the 32-bit address on TNS/E determines global addressing, and privilege is an attribute of the page; the MIPS architecture on TNS/R is just the opposite.

Relocation. the process of assigning load addresses to the different parts of a program, adjusting the code and data in the program to reflect the assigned addresses.

SearchList. For each loadfile, a list that specifies which libraries to examine, and in which order, to locate symbol definitions needed by that loadfile. The linker and loader construct the loadfile's searchList in accordance with that loadfile's import control, which is set at link time. The system library and millicode are appended to every searchList. A loadfile's searchList is unaffected by the import control of any other loadfile.

Sections and Segments. The TNS/E object file is organized into contiguous items called sections. There is an array of ELF section headers that contains the type and name of each of these section items. A section is not required to be present if it would not contain any useful information for a given object file. In loadfiles, some of the sections are further organized in segments that get loaded into virtual memory.

Strip file. These are files do not have debugging information; that is, DWARF symbol table, in it. Stripping can be done on any object file. It is still possible for the linker to process a linkfile that has been stripped because the DWARF symbol table does not contain any essential information to it. An import library can be stripped even if the corresponding DLL is not stripped.

Symbol Resolution. When a program is built from multiple subprograms, the references from one subprogram to another are made using symbols. For example a main program might use a square root routine called `sqrt` and the math library defines `sqrt`. A linker resolves the symbol by noting the location assigned to `sqrt` in the math library and patches the caller's object code so the call instruction refers to that location.

Semi-globalized. An import control characteristic of a loadfile that allows the loadfile first to obtain symbols from its own definitions and then to obtain others as for a globalized loadfile. Thus, a semi-globalized loadfile cannot have its symbol references to itself preempted. See also SearchList..

Symbol. The symbolic name of a function or data item. Symbols are defined in loadfiles and referenced in the same or other loadfiles.

Symbol definition. a function or data item whose name is the symbol.

Symbol value. the address of a definition of that symbol.

Symbolic reference. An occurrence in code or data of a symbol that is or must be bound to a definition of that symbol. The symbolic reference is bound (resolved and made usable) by assigning to it the value of a definition of that symbol.

System library. TNS/E library routines required to access TNS/E operating system functions. (Similar for TNS/R.) The loader automatically searches the system library for definitions that satisfy a loadfile's unresolved symbols after searching all the loadfiles in the loadfile's searchList.

TNS/E. The hardware platform based on the Itanium™ architecture and the HP NonStop operating system and software that are specific to that platform. All code is PIC.

TNS/R. The hardware platform based on the MIPS™ architecture and the HP NonStop operating system and software that are specific to that platform. Code may be PIC or non-PIC.

TLB. Translation Lookaside Buffer: a cache of page table entries, where each entry designates the physical memory page corresponding to a range of virtual addresses. Information within the entry can make the translation unique to the accessing process. Unless the appropriate TLB entry is present, the page cannot be accessed; typically the processor generates a fault to allow software to find and load the missing entry from a memory-management structure.

TNS/E object file format. This object file format is an amalgam of Intel IA-64 code architecture and the HP NonStop operating system extensions.

TNS/E object files are categorized into three types of files: linkfiles, loadfiles, and import libraries. The following are key differences between TNS/R and TNS/E platforms:

Platform	TNS/R	TNS/E
Processor	MIPS RISC	Itanium
Architecture	SGI	Intel IA-64
Programming model	32-bit (ILP32)	32-bit (ILP32) and in future: 64-bit LP64
Object type	ELF and COFF	ELF exclusive
Debugging symbols	Third-Eye	DWARF2
Compiler Backend	SGI w/ HP extensions	Intel w/ HP extensions
Linker, PIC	`ld`	`eld`

User library. A loadable library; primarily a legacy feature for NonStop systems. For PIC programs, a user library is a DLL treated as if it were the first library in the program's libList and therefore is searched first for symbols required by the program. However, a user library does not appear in the program's libList; instead, its name is recorded in the program's loadfile as the libname attribute. A program can be associated with at most one user library; the association can be specified using the linker at link time or in a later change command, or at run time using the process creation interfaces. (The /LIB.../ option to the RUN command in TACL uses these interfaces.)

VHPT. Virtual Hash Page Table: an Itanium® architecture feature that can supply missing TLB entries without generating faults.

VPROC. The version procedure number used to identify which version of the product you are using.

Zimpimp file. The internal name of the imp-imp file. Also called the "import library that represents the implicit DLL's", it is the file that tells which symbols are available in the set of implicit DLL's, which collectively correspond to what was previously called the system library. See also Implicit library import library (imp-imp).

Zreg file. This is the internal name of the public DLL registry file, which lists the names of all the public DLL's.

Index