TNS/E Native Application Conversion Guide

Abstract

This manual introduces the TNS/E native development and execution environments and explains how to convert existing TNS applications to TNS/E native applications.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports H06.03 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
529659-003	August 2010

Document History

Part Number	Product Version	Published
529659-002	N.A.	July 2005
529659-003	N.A.	August 2010

TNS/E Native Application Conversion Guide

3. C and C++ Conversion Tasks (continued)

4. Converting COBOL Programs

5. Converting TAL to pTAL

6. Converting a TNS User Library

7. Converting Data Definition Language (DDL)

8. Converting Programs That Run in the Common Run-Time Environment

Examples

Figures

Tables

What's New in This Manual

Manual Information

Abstract

This manual introduces the TNS/E native development and execution environments and explains how to convert existing TNS applications to TNS/E native applications.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports H06.03 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
529659-003	August 2010

Document History

Part Number	Product Version	Published
529659-002	N.A.	July 2005
529659-003	N.A.	August 2010

New and Changed Information

New in the H06.21/J06.10 revision:

- Added information throughout the manual to include c99 compiler support.

About This Manual

This manual introduces the TNS/E native development and execution environments and explains how to convert existing TNS applications to TNS/E native applications.

This manual applies to all TNS programs; those that run on a TNS/R system and those that run on a TNS/E system. The information in this manual applies only to conversion to TNS/E native mode, although converting a TNS program to TNS/R native mode is similar. For details on converting a TNS program to TNS/R native mode, see the *TNS/R Native Application Migration Guide*.

This manual describes changes required to convert a program to TNS/E native mode. It does not describe changes for other products and subsystems such as NonStop SQL/MP, HP NonStop TS/MP, and HP NonStop TCP/IP, which must be made regardless of whether a program runs in TNS mode or native mode. Depending on the RVU from which you are migrating your program, additional changes might be required. For more information, see a product or subsystem's documentation set and the RVU documentation.

Audience

This manual is intended for those who manage or write applications for HP NonStop systems. The reader is assumed to be familiar with the HP documentation for the languages in which the programs are written, which are:

HP Document	Compiler	T Number
C/C++ Programmer's Guide	TNS C	T9255
	TNS C++	T9541
	TNS `c89`	T8629
	TNS/E `c89,c99`	T8164
	TNS/E CCOMP	T0549
	TNS/E CPPCOMP	T0549
COBOL Manual for TNS/E Programs	TNS COBOL85	T9257
	TNS `cobol`	T8498
	TNS/E ECOBOL	T0356
	TNS/E `ecobol`	T0356
Data Definition Language (DDL) Reference Manual	DDL	T9100
pTAL Reference Manual	TNS/E EpTAL	T0561
TAL Reference Manual	TNS TAL	T9250
TAL Programmer's Guide		

Purpose

This document is designed to help you perform these tasks:

- Learn about the TNS/E native execution and development environment and how it differs from the TNS environment.
- Determine which programs can be converted and the effort required to convert them.
- Plan a conversion strategy.
- Convert programs written in C and C++ from TNS to native mode.
- Convert programs written in COBOL from TNS to native mode.
- Convert Transaction Application Language (TAL) programs to pTAL (portable TAL) programs.
- Convert user libraries.
- Use the Data Definition Language (DDL) to support TNS and native programs.
- Make changes in the Guardian application program interface (API), Open System Services (OSS) API and utilities, and Common Run-Time Environment (CRE) API required to convert programs to TNS/E native mode.
- Enable TNS and native programs to share data.

The manual assumes that you are familiar with the programming languages, compilers, and tools used to create the programs you plan to convert to TNS/E native mode.

This manual does not describe RVU installation, configuration, and conversion or migration issues that are not related to converting an existing application to TNS/E native mode. For information on these issues, see:

- *H06.nn Release Version Update Compendium*
- *H06.nn Software Installation and Upgrade Guide* for a given RVU

Organization

Section	Explains
Section 1, Introduction to Native Mode	TNS/E native development and execution environments, features of TNS/E native architecture, and benefits and constraints of TNS/E native mode
Section 2, Developing a Conversion Strategy	How to prepare programs for conversion, plan system resources, determine which programs to convert, maintain common TNS and native source files, and maximize the performance of TNS/E native programs
Section 3, C and C++ Conversion Tasks	How to convert NonStop C and NonStop C++ programs to TNS/E native mode
Section 4, Converting COBOL Programs	How to convert NonStop COBOL programs to TNS/E native mode
Section 5, Converting TAL to pTAL	How to convert TAL programs to pTAL
Section 6, Converting a TNS User Library	How to convert a TNS user library to a TNS/E native user library
Section 7, Converting Data Definition Language (DDL)	How to use DDL to generate files for TNS and TNS/E native compilers
Section 8, Converting Programs That Run in the Common Run-Time Environment	CRE API changes required to convert a program to TNS/E native mode
Section 9, Converting Programs That Share Data	How TNS and native programs can share data
Section 10, Converting Programs With Guardian API Calls	Guardian API changes required to convert a program to TNS/E native mode
Section 11, OSS API and Utilities Conversion Tasks	OSS API and utilities changes required to convert an OSS program to TNS/E native mode

For More Information

Manual	Explains (page 1 of 2)
Object Code Accelerator (OCA) Manual	How to improve performance of TNS programs running on TNS/E systems (converting them to TNS/E native mode is preferable)
Binder Manual	How to use the stand-alone Binder product to bind compilation units (or modules) that were compiled with TNS compilers
C/C++ Programmer's Guide	The NonStop C and NonStop C++ programming languages, compilers, and run-time libraries
COBOL Manual for TNS/E Programs	The COBOL programming language, compilers, and run-time libraries
CRE Programmer's Guide	The Common Run-Time Environment (CRE), a set of run-time services that enable mixed-language programming and support the language-specific run-time libraries
Data Definition Language (DDL) Reference Manual	How to use the Data Definition Language (DDL) to define data objects and translate them into source code
eld Manual	How to use the `eld` utility to link TNS/E native object files.
eNOFT Manual	How to use the `enoft` utility to view TNS/E native object files
Guardian Procedure Calls Reference Manual	The syntax and semantics of most Guardian procedure calls
Guardian Programmer's Guide	How to write programs for the Guardian environment
Guardian C Library Calls Reference Manual	The syntax and semantics of the Guardian TNS HP C run-time library
Guardian Native C Library Calls Reference Manual	The syntax and semantics of the Guardian TNS/E native HP C run-time library
H06.nn Release Version Update Compendium	New features, migration issues, and fallback considerations for RVU H06.*nn*
H-Series Application Migration Guide	How to migrate TNS and TNS/R programs from G-series systems to H-series systems.
H06.nn Software Installation and Upgrade Guide	How to install an H06.*nn* RVU of the HP NonStop operating system on an HP Integrity NonStop server
Native Inspect Manual	How to debug programs using the Native Inspect symbolic debugger

Manual	Explains (page 2 of 2)
KMSF Manual	How to use NSKCOM utility to configure and manage the swap volumes used by the Kernel-Managed Swap Facility (KMSF)
Inspect Manual	How to debug programs using the Inspect source-level and machine-level interactive debugger
Open System Services System Calls Reference Manual	The syntax and semantics of part of the HP NonStop Open System Services (OSS) API, including the ISO/ANSI NonStop C run-time library calls
Open System Services Porting Guide	How to port C programs from other UNIX vendors to the OSS environment
Open System Services Programmer's Guide	How to write programs for the OSS environment
Open System Services Shell and Utilities Reference Manual	The OSS shell and utilities
Open System Services System Calls Reference Manual	The syntax and semantics of part of the OSS API
pTAL Conversion Guide	How to convert TAL code to pTAL code
pTAL Guidelines for TAL Programmers	How to write TAL code that can be converted later to pTAL with as few changes as possible
pTAL Reference Manual	The syntax and semantics of the pTAL language and how to run the pTAL compiler
Software Internationalization Guide	Software internationalization standards and facilities available on NonStop systems
TAL Reference Manual	Syntax descriptions and error messages of TAL
TAL Programmer's Guide	How to write TAL programs
TACL Reference Manual	The HP Tandem Advanced Command Language (TACL), which provides an interface to the NonStop operating system
Visual Inspect Online Help	How to use the Visual Inspect PC-based symbolic debugger

Notation Conventions

Hypertext Links

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

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

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 that you have concerning this document.

1 Introduction to Native Mode

TNS/E native mode enables you to write programs that are fully optimized for Integrity NonStop servers that use the TNS/E architecture. The term *TNS/E native* means the program uses the process, memory, and instruction set architectures that are native to Intel® Itanium® processors. Throughout the rest of this manual, the terms *native* and *native mode* are used to mean *TNS/E native* and *TNS/E native mode*, respectively. The *TNS/E* qualifier is used only when necessary to contrast or compare with *TNS/R native mode*.

This manual applies to all TNS programs; those that run on a TNS/R systems and those that sun on a TNS/E system. The information in this manual applies only to conversion to TNS/E native mode, although converting a TNS program to TNS/R native mode is similar. For details on converting a TNS program to TNS/R native mode, see the *TNS/R Native Application Migration Guide*.

Native compilers and tools are used to generate native programs. Other tools have been enhanced to support native programs. Native programs can be written in pTAL (a variant of TAL), C, C++, and COBOL. Native programs consist entirely of Intel® Itanium® instructions and do not have TNS architecture-specific attributes.

This section discusses:

Summary of Execution Modes

A TNS process—a process that runs in TNS interpreted mode or TNS accelerated mode—is initiated by running a TNS interpreted or accelerated object file.

A native process—a process that runs in native mode—is initiated by running a native object file. Native object files consist entirely of Intel® Itanium® instructions. Native processes do not maintain TNS architecture-specific constructs.

TNS/E systems support these execution modes:

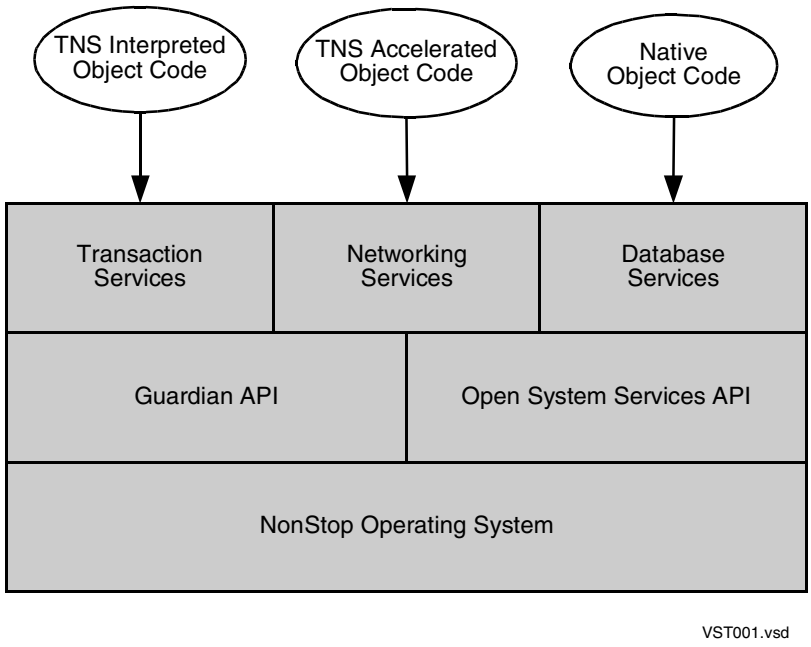
Mode	Characteristics of Mode
TNS interpreted mode	● Programs generated by TNS compilers. ● Programs use TNS process and memory architecture. ● Programs consist of TNS object code. ● Programs consist of TNS instructions. Millicode routines implement TNS instructions on Itanium processors.
TNS accelerated mode	● Programs generated by TNS compilers and processed by the Object Code Accelerator (OCA). ● Programs use TNS process and memory architecture. ● Programs consist of TNS object code and accelerated object code. ● Programs consist of TNS instructions and equivalent OCA-generated Itanium instructions. Programs execute Itanium instructions directly on Itanium processors. Programs transition to TNS mode when OCA is unable to generate equivalent Itanium instructions.
Native mode	● Programs generated by native compilers. ● Programs use native process and memory architecture. ● Programs consist of native object code. ● Programs consist of Itanium instructions which run directly on Itanium processors.

Because of architectural differences between the execution modes, TNS interpreted object code, accelerated object code, and native object code cannot be mixed in one program file. A native program can contain only native object code.

Underlying Native Mode Structure for All Programs

As shown in Figure 1-1, in H-series RVUs, HP has converted nearly all system code, system library routines, and other products to run in native mode. (Shaded regions in the figure indicate code that runs in native mode.) Therefore, interpreted and accelerated object code benefit from the increased performance of the native architecture without a single line of source change or recompilation.

Figure 1-1. Native Mode Benefits All Programs



While native mode offers many benefits, you are not required to convert your programs to native mode. H-series RVUs continue to support the TNS compilers and tools. You can continue to create and run TNS interpreted and accelerated object code and gain the performance benefits provided by HP software that has been converted to native mode.

Differences Between Accelerated and Native Object Code

While both accelerated and native object code execute Itanium instructions, most native object code has a significant performance advantage over accelerated object code. The OCA cannot produce Itanium instructions for TNS instruction sequences whose exact meaning cannot be determined until run time. In such cases, a process makes a transition into TNS code and executes the TNS instructions through millicode routines. To enable this transition to occur, the accelerated object code maintains TNS architecture-specific constructs, such as the P and ENV registers.

Native object code consists entirely of Itanium instructions. Transitions into TNS code do not occur, so TNS architecture-specific constructs are not maintained. Additionally, OCA must base its Itanium instruction sequences, data layout, and code optimizations on object code. The native compilers base their Itanium instructions sequences, data layout, and code optimizations on source code. For more information on OCA, see the *Object Code Accelerator (OCA) Manual*.

Native Development Environment

A development environment comprises the tools used to compile, link, optimize, and debug a program and the run-time libraries available to a program. The native development environment includes:

- pTAL Compiler on page 1-5
- Native C Compiler on page 1-6
- Native C++ Compiler on page 1-6
- Native COBOL Compiler on page 1-7
- Native C Run-Time Library on page 1-8
- Native Linker (eld Utility) on page 1-8
- Native Object File Tool (enoft Utility) on page 1-9
- ETK on page 1-9
- Native Mode Debugging Tools on page 1-10
- SQL Compiler on page 1-12
- Data Definition Language (DDL) on page 1-12

Table 1-1 compares the tools used in the TNS and native development environments:

Table 1-1. Development Environment Comparison

TNS Development Environment	Native Development Environment
TNS C compiler (c)	Native C compiler (CCOMP)
TNS `c89` utility (TNS/R systems only)	Native `c89` utility
	Native `c99` utility (TNS/E only)
cprep and cfront for C++	Native C++ compiler (CPPCOMP)
TNS COBOL compiler (COBOL85)	Native COBOL compiler (ECOBOL)
TNS `cobol` utility (TNS/R systems only)	Native `ecobol` utility
TAL compiler	pTAL compiler (EPTAL)
Binder	Native linker (`eld` utility) Native object file tool (`enoft` utility)
Object Code Accelerator	Not needed because native compilers produce optimized Itanium object code
Inspect and Visual Inspect symbolic debuggers	Native Inspect and Visual Inspect symbolic debuggers
Debug machine-level debugger (TNS/R systems only)	Debug not supported
SQL compilers: SQLCOMP for SQL/MP, MXCOMP for SQL/MX	SQL compilers: SQLCOMP for SQL/MP, MXCOMP for SQL/MX
DDL	DDL
CROSSREF	Not needed because native compilers have directives or pragmas to produce listings, and `enoft` utility produces cross-reference listings

The following subsections describe the components of the native development environment.

pTAL Compiler

The Portable Transaction Application Language (pTAL) is a dialect of TAL. pTAL does not depend on TNS architecture-specific constructs. pTAL introduces new constructs that replace TNS architecture-specific TAL constructs. The pTAL compiler reads pTAL source code and creates native object code. The command line syntax of the pTAL compiler is similar to that of the TAL compiler.

The pTAL compiler syntax-checking mode helps you convert TAL to pTAL. In this mode, the compiler identifies most source code changes and suggests a method to recode in pTAL.

You can run the pTAL compiler in the Guardian environment and on the PC as part of the HP Enterprise Toolkit - NonStop Edition (ETK) (the HP Tandem Development Suite is not supported on H-series systems). The EPTAL command runs the compiler in the

Guardian environment. The EPTAL compiler command line syntax is similar to that of the TAL compiler command. The `eptal` utility runs the pTAL compiler on the PC.

ETK enables you to compile and link programs on a PC (for details, see the *pTAL Reference Manual*). You can copy the object files to a TNS/R or TNS/E system and execute them in the Guardian and OSS environments.

The TNS/E pTAL compiler provides a syntax-checking mode that helps you convert TAL to pTAL. In this mode, the compiler identifies most source code changes and suggests a method to recode pTAL.

Native C Compiler

The native C compiler accepts C language source files that comply with either the ISO/ANSI C language standard (ISO/IEC 9899:1990), ISO/ANSI C language standard (ISO/IEC 9899:1999), or Common Usage C (sometimes called Kernighan and Ritchie C or K&R C). The native C compiler also accepts HP language extensions for NonStop systems.

You can run the native C compiler in the Guardian and OSS environments and on the PC as part of ETK (the Tandem Development Suite is not supported on H-series systems). The CCOMP command runs the compiler in the Guardian environment. The native C compiler command line syntax is similar to that of the TNS C compiler. The native `c89` or native `c99` utility runs the compiler in the OSS environment. The native `c89` or native `c99` utility syntax is similar to that of the TNS `c89` utility or the TNS/E `c99` utility.

ETK provides cross compilers that enable you to compile and link programs on a PC under the Microsoft Windows operating system for execution on the Integrity NonStop server. You can compile programs targeted for either the TNS/R or TNS/E system, copy the object files to the appropriate Integrity NonStop server, and execute them in the Guardian and OSS environments.

The native C compiler supports programs that define the size of pointers and type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C language programs that define pointers or type `int` as 16 bits must be changed. Few other C language source code changes are required to use the native C compiler.

The H-series RVUs do not support the NMCMT native mode conversion tool, which is available on TNS/R systems for converting programs to TNS/R native mode.

Native C++ Compiler

The native C++ compiler accepts C++ language source files that comply with the ISO/ANSI C++ language standard (ISO/IEC 14882:1998). The native C++ compiler also accepts HP language extensions for NonStop systems.

You can run the native C++ compiler in the Guardian and OSS environments and on the PC as part of ETK (the Tandem Development Suite is not supported on H-series systems). The CPPCOMP command runs the compiler in the Guardian environment.

The native C++ compiler command line syntax is similar to that of the TNS C compiler and Cfront. The native `c89` utility runs the compiler in the OSS environment and on the PC. The native `c89` utility syntax is similar to that of the TNS `c89` utility.

ETK provides cross compilers that enable you to compile and link programs on a PC under the Windows operating system for execution on the Integrity NonStop server. You can compile programs targeted for either the TNS/R or TNS/E system, copy the object files to the appropriate Integrity NonStop server, and execute them in the Guardian and OSS environments.

The native C++ compiler supports programs that define the size of type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C++ language programs that define the type `int` as 16 bits must be changed. Few other C++ language source code changes are required to use the native C++ compiler.

The native C++ compiler provides a more powerful and simplified development environment than TNS Cfront. For example, you must run the C preprocessor, Cfront, the TNS C compiler, Binder, and OCA to create an executable accelerated C++ program. In comparison, you run only the native C++ compiler and `eld` utility to create an executable native C++ program.

The C and C++ native mode conversion tool, NMCMT, is not supported in H-series RVUs.

Native COBOL Compiler

The native COBOL compiler accepts COBOL language source files that comply with the ISO/ANSI COBOL85 Standard. The native COBOL compiler also accepts HP language extensions for NonStop systems.

You can run the native COBOL compiler in the Guardian and OSS environments and on the PC as part of ETK (the Tandem Development Suite is not supported in H-series RVUs). The ECOBOL command runs the compiler in the Guardian environment. The ECOBOL compiler command line syntax is similar to that of the TNS COBOL85 compiler command. The native `ecobol` utility runs the compiler in the OSS environment and on the PC. The native `ecobol` utility syntax is similar to that of the TNS `cobol` utility.

ETK provides cross compilers that enable you to compile and link programs on a PC under the Windows operating system for execution on the Integrity NonStop server. You can compile programs targeted for either the TNS/R or TNS/E system, copy the object files to the appropriate Integrity NonStop server, and execute them in the Guardian and OSS environments.

Native C Run-Time Library

The native C run-time library provides functions conforming to the ISO/ANSI C Standard. It also contains functions conforming to the X/OPEN UNIX 95 specification and HP extensions for NonStop systems to these standards.

The native C run-time library supports Guardian and OSS processes. The native C run-time library is nearly identical for the Guardian and OSS environments and therefore increases the interoperability between environments.

The native C run-time library does not have many of the nonstandard functions in the Guardian TNS C run-time library. However, the native C library does have additional functions from the X/OPEN UNIX 95 specification that are absent from the Guardian and OSS TNS C libraries. The native C run-time library also provides additional local sensitive functions and algorithmic code-set converters for use in internationalized OSS applications. For details, see the *Software Internationalization Guide*.

The TNS and native C run-time libraries return the same error messages. The native C run-time library returns additional `errno` return values.

Native Linker (eld Utility)

The native linker, `eld`, links one or more native position-independent code (PIC) linkfiles (object files generated by the native compilers or by `eld`) to produce either a loadfile or a linkable native object file. The loadfile is either a program or a dynamic-link library (DLL) that can be loaded into memory and executed. (For more information, see Native Object File Format on page 1-16.) `eld` can also modify process attributes, such as HIGHPIN, of executable native object files and strip nonessential information from native object files.

`eld` is used instead of Binder for native object files. Binder and `eld` have a different syntax and operate on different object file types, but perform essentially the same operations.

The TNS/E environment also provides a run-time loader, `rld`, that works with the operating system to dynamically link and load PIC loadfiles and their requisite DLLs into memory at execution time. You do not call `rld` directly (at the command prompt), but you can access it programmatically through the `rld` run-time linking functions. For more information about `eld` and `rld`, see the *eld Manual* and the *rld Manual*.

Unlike Binder, `eld` cannot replace individual procedures and data blocks in an object file or build an object file from individual procedures and data blocks. `eld` operates on procedures and data blocks, but only in terms of an entire object file.

`eld` does not support the Binder SELECT SEARCH behavior. In most cases, you can use archive files (files created by the `ar` utility) to replace this behavior. For more information on the differences between Binder and `eld`, see the *eld Manual*.

`eld` runs in the Guardian and OSS environments and on the PC, either at the command prompt or as part of ETK. `eld` syntax and capabilities are nearly identical in each environment.

Native Object File Tool (enoft Utility)

The native object file tool, `enoft`, reads and displays information about native object files. You can use `enoft` to:

- Determine the optimization level of procedures
- Display object code with corresponding source code
- List object file attributes
- List unresolved references

`enoft` runs in the Guardian and OSS environments. The `enoft` syntax and capabilities are nearly identical in each environment.

ETK

The Enterprise Toolkit - NonStop Edition (ETK) is an integrated development environment that enables you to build NonStop applications, on a PC running the Windows operating system, for execution on a Integrity NonStop server. ETK is an extension package to Visual Studio.NET and provides a graphical user interface (GUI) in which you use menus and dialog boxes to select tools and options. ETK was introduced with the G06.20 RVU and coexists with the Tandem Development Suite (TDS) in subsequent G-series RVUs. In the H-series RVUs, only ETK is supported. You must convert TDS projects to ETK projects when converting G-series TNS programs to H-series native mode as described under ETK Migration Tool.

ETK supports the native C/C++, pTAL, and COBOL cross compilers and related tools. The cross compilers can compile programs for the Guardian and OSS environments. TNS compilers and tools are not available on the PC.

ETK Migration Tool

If you are still using TDS on a G-series system, you must convert to ETK when converting to an TNS/E native mode. A migration tool is available to help you convert TDS projects to equivalent ETK projects. The tool plugs into TDS. It can be obtained from IPM Scout at no charge.

Note. The TDS-to-ETK migration tool runs as part of TDS; therefore, you must install the tool and perform the migration on a G-series system with TDS installed. The migration tool does not require ETK to perform the migration.

The terminology used by TDS and ETK differs somewhat. TDS works with “projects,” which contain multiple “targets.” ETK works with “solutions”, which contain multiple “projects.” TDS projects are mapped to ETK solutions, and TDS targets are mapped to ETK projects.

The migration procedure involves converting TDS targets to ETK projects, then adding the projects to an ETK solution. The ETK solution could be a newly created solution or an existing solution.

As part of the migration process, the tool ensures that all files pertaining to the TDS target remain part of the ETK project. However, file properties for file types that are unknown to ETK are not migrated. For every such file type, users must explicitly add these file properties in ETK. The migration tool supports the following file extensions: `.cpp`, `.css`, `.tal`, `.cob`, and `.cbl`.

Note. In the TDS project system, a file type can be independent of the file extension. For example, a `file1.cpp` file can have the file type set as `.cob`. In this case, although the file as a `.cpp` extension, it behaves as a COBOL file and is recognized as such by the COBOL cross compiler. In the ETK project system, file type information is based solely on the file extension; for example, ETK will always treat `file1.cpp` as a C++ file. For the migration tool to preserve file semantics while migrating TDS targets, there cannot be any file in the TDS target that has a mismatch between its extension and its type. On encountering the first such file, the tool generates an error and the TDS target is not migrated.

Native Mode Debugging Tools

The TNS/E native environment offers two symbolic debugging tools: Native Inspect and Visual Inspect. The TNS/R machine-level Debug facility is not supported on TNS/E systems. However, both Native Inspect and Visual Inspect have been enhanced with machine-level debugging capabilities and can be used as replacements for Debug.

The Inspect debugger is available on TNS/E systems, but can be used only for debugging TNS processes. After converting a program to native mode, you must use either Visual Inspect or Native Inspect.

Visual Inspect

Visual Inspect supports high-level symbolic debugging of TNS (interpreted and accelerated) and native processes through a PC-based (GUI). Visual Inspect can also be used for debugging TNS and native snapshot files. Most Visual Inspect commands apply to both TNS and native processes. You can use Visual Inspect to debug programs created by the PC cross compilers (C, C++, COBOL, and pTAL). For native mode debugging, you can invoke Visual Inspect from within ETK.

Visual Inspect is the preferred application debugging tool in the TNS/E native environment, and you are encouraged to do as much as possible of your native mode development and debugging on the PC platform. Visual Inspect offers a simpler and more intuitive user interface and more capabilities than the command line debuggers. For example, working from a PC, you can use Visual Inspect to debug multiple processes residing on the same or on different nodes in a network. The processes can be TNS, TNS/R native, or TNS/E native. Inspect and Native Inspect do not provide this capability.

The H-series product version of Visual Inspect has been enhanced for machine-level debugging; you can use it for low-level debugging tasks on the PC that needed to be done in previous RVUs on the Integrity NonStop server using Debug or Inspect.

For code compiled at optimization level 0 (no optimization) or optimization level 1 (intermediate optimization), you can perform the same operations for native processes (using Visual Inspect) as for TNS process (using Inspect), including:

- Step through code
- Set breakpoints
- Display source
- Display variables

In addition, H-series Visual Inspect provides these machine-level capabilities:

- Set instruction breakpoints
- Display instruction code
- Display and modify data using a numeric (nonsymbolic) address
- Modify, format, and monitor registers
- Display and format data buffers as SPI or EMS buffers

Differences in the TNS and native process architectures result in differences in registers and address ranges. Unlike TNS interpreted and accelerated code, TNS architecture-specific constructs, such as TNS environment registers, do not exist in native mode code.

At optimization level 2 (full optimization), machine-level debugging might be necessary because of the effects of optimization. For example, variables might remain in registers and never be written to memory. For more details on compiler optimization, see Determining Optimization Levels on page 2-5.

The Inspect and Visual Inspect symbolic debuggers differ in these respects:

- User interface

  The Inspect debugger uses a line-oriented command interpreter, while the Visual Inspect debugger uses a GUI environment consisting of menus and icons that you can select and click to perform tasks.

- Features and functions

  Visual Inspect supports many but not all Inspect capabilities. Many commands have become part of the GUI (for example, the ENV command).

For details on Visual Inspect capabilities, see the Visual Inspect online help.

Native Inspect

Native Inspect is a command-line symbolic debugging tool that can be used for debugging TNS/E native processes and snapshot files. It can be used for source statement level debugging as well as machine-level debugging. Native Inspect is intended as a replacement for the G-series Debug facility and the G-series Inspect debugger for native mode debugging. The command name for Native Inspect is `eInspect`.

Note. Native Inspect cannot currently be used to debug COBOL programs. The only debugger available to H-series COBOL programs is Visual Inspect.

Native Inspect provides most of the functionality of Inspect and Debug. However, the Native Inspect command syntax differs from that of Inspect and Debug. The Native Inspect syntax is based on the Open Source Foundation GDB debugger, a tool that is widely used throughout the industry and is familiar to many application developers. In most cases, you are encouraged to use Visual Inspect as your primary application debugger. The primary advantage of Native Inspect is that it provides enhanced scripting support in the form of the Tool Command Language (TCL), a widely used scripting language, which enables you to automate many debugging tasks.

Note that any Inspect command files that you are currently using to automate debugging operations must be converted to Native Inspect syntax.

See the *Native Inspect Manual* for details.

SQL Compiler

The HP NonStop SQL/MP compiler supports embedded SQL in TNS and native C and COBOL programs. The HP NonStop SQL/MX compiler, available in the OSS and PC environments, supports embedded SQL in native C, C++, and COBOL programs. You cannot use embedded SQL in pTAL source code. For more information, see the NonStop SQL/MP and NonStop SQL/MX manual sets.

Data Definition Language (DDL)

The H-series product version of the DDL compiler generates host-language source files that can be used with both TNS and native programs. The DDL compiler inserts pragmas in C and directives in pTAL and COBOL host-language source files to ensure that the same data alignment is generated, regardless of whether a TNS or native compiler is used. For details, see Section 7, Converting Data Definition Language (DDL).

Native Architecture Features

The native architecture introduces these new features:

Native Process Environment

A process that runs in TNS interpreted mode or TNS accelerated mode—a TNS process—consists entirely of TNS instructions or both TNS instructions and Object Code Accelerator-generated Itanium instructions. A TNS process is initiated by executing a TNS interpreted or TNS accelerated program.

A process that runs in native mode—a native process—consists entirely of native-compiled Itanium instructions. A native process is initiated by executing a native program. Unlike TNS processes, native processes do not use or emulate TNS architecture-specific constructs, such as TNS registers or 16-bit addressing.

Differences between native and TNS processes are discussed next:

Process Attributes

Native processes have the process attributes HIGHPIN ON, HIGHREQUESTERS ON, and INSPECT ON by default. For many TNS processes, these process attributes are set to OFF by default.

Process Organization

Executable code for a TNS/E native process is contained in these objects:

- The initial program of the process, called user code. This code is read from the program file.
- Dynamic-link libraries (DLLs). These include:
  - The system library, which contains system-related procedures and operating system code that is accessible by the process using system procedure calls. The system library consists of a set of implicit DLLs.
  - Other DLLs supplied by HP, such as the C run-time library.
  - User-created DLLs.

When a process is created, it occupies space in virtual memory. The basic organization of a process is discussed next in terms of code spaces, which are associated with the objects in the preceding list, and data spaces.

Code Segments

A process has distinct code segments that contain executable code. *(Segments* is the term used for TNS/R and TNS/E native processes. In the TNS environment, the term *code spaces* is used.) This table compares the code segments for TNS and TNS/E processes:

Process Type	Code Segments
TNS process on TNS/R processor	UC (user code) UL (user library) SC (system code) SL (system library) SCr (system code RISC) SLr (system library RISC
TNS process on TNS/E processor	UC (user code) UL (user library) SC (system code) SL (system library) **i**mplicit DLLs
TNS/E native process	UC (user code) UL (DLL user library) Implicit DLLs (correspond to TNS SCr and SLr on TNS/R system, SC and SL on TNS/E system) Public DLLs Ordinary DLLs

Data Segments

When a process is created, several data segments are allocated for its use. This table compares the data spaces for TNS and TNS/E processors:

Process Type	Data Segments
TNS process on TNS/R processor	A user data segment, containing global data (for TAL, COBOL85, and small memory-model C programs) and the user data stack for TNS procedures
	An automatic (compiler-generated) extended data segment, containing extended global data and local data, and optionally a heap (for C and C++ programs)
	A main RISC stack segment, containing the stack for nonprivileged native procedures
	A privileged RISC stack segment, containing the stack for privileged native procedures
	A process file segment (PFS), used by the operating system
	Optional program-allocated extended data segments (selectable or flat segments)
TNS process on TNS/E processor	A user data segment, containing global data (for TAL, COBOL85, and small memory-model C programs) and the user data stack for TNS procedures
	An automatic (compiler-generated) extended data segment, containing extended global data and local data, and optionally a heap (for C and C++ programs)
	A main memory stack for nonprivileged TNS/E native procedures.
	A privileged memory stack for privileged procedures.
TNS/E native process	A globals-heap segment, containing program global data and, optionally, a heap
	A main memory stack for nonprivileged TNS/E native procedures.
	A privileged memory stack for privileged procedures.
	A main register stack engine (RSE) backing store for nonprivileged procedures
	A privileged RSE backing store for privileged procedures
	Zero or more DLL data segments
	A process file segment (PFS), used by the operating system
	Optional program-allocated extended data segments (selectable or flat segments)

The globals-heap segment in native processes is comparable to the user data segment and the automatic extended data segment in TNS processes.

For native C and C++ programs, the native Common Run-Time Environment (CRE) automatically manages a heap in the globals-heap segment. The heap is optional for other programs.

TNS and Itanium stack growth is as follows:

- TNS stacks grow upwards (from lower to higher addresses).
- The Itanium RSE backing store grows upwards.
- Itanium memory stacks grow downwards.

On TNS/E processors, the main memory stack and the heap grow automatically as needed, to a maximum size. The maximum size of each can be specified when a process is created. The default limit for the TNS/E main memory stack is 2 MB. You can increase the maximum stack size via an `eld` or PROCESS_LAUNCH_ parameter up to a limit of 32 MB.

The heap can grow to the maximum size of the globals-heap segment less the size of the global data. The maximum globals-heap size is 1.5 GB.

See the *Guardian Programmer's Guide* for more information on TNS/E native processes.

Native Object File Format

Native object files use a different file format from that of TNS interpreted or accelerated object files. Native object files are in 64-bit executable and linking format (ELF), a standard format used for UNIX object files, with HP extensions for NonStop systems. The native object file format is the same in the Guardian and OSS environments and on the PC as part of ETK. In the Guardian environment, native object files are type 800 files.

Native object files are either relinkable or executable, but not both. (TNS interpreted and accelerated object files, by contrast, can be both relinkable and executable.) As the name implies, a relinkable object file can be linked to produce an executable object file, but it cannot be run. Likewise, an executable object file can be run, but it cannot be linked to produce another executable object file.

The native compilers produce only relinkable object files. The `eld` utility can produce either relinkable or executable object files. Relinkable object files can be used as `eld` input again. Executable object files can be used as `eld` input for modifying executable object file attributes only. Both relinkable and executable object files can be used as `noft` input.

For details on the structure of native object files, see the *eld Manual*.

Signals Facility

Certain critical error conditions occurring during process execution prevent normal process execution. Most of these error conditions are unrecoverable. In TNS processes, these errors cause the process to receive a trap. In native processes, these errors cause the process to receive a signal. Native processes do not receive traps.

Signals are software interrupts that provide a way of handling asynchronous events, such as timer expiration, detection of a hardware fault, abnormal termination of a process, or any trap condition normally detectable by a TNS process. Each TNS trap has a corresponding signal, although the trap number and the signal number are different.

The Debug (TNS/R systems only) and Inspect debuggers display a debugging prompt when a TNS process receives a trap. The Native Inspect and Visual Inspect debuggers display a prompt when a native process receives a signal only if the process has previously entered Native Inspect or Visual Inspect (for example, with a TACL RUND or RUNV command). Otherwise, a termination message with the signal name and number is displayed. Native Inspect and Native Inspect can be used to display and set signal information when debugging. For details, see the *Native Inspect Manual* or the Visual Inspect online help.

For information on signal behavior, see:

- ARMTRAP on page 10-2
- *Guardian Programmer's Guide*

DLLs

All libraries in the TNS/E native environment are dynamic-link libraries (DLLs). A DLL is a type of library that is constructed of position-independent code (PIC). PIC is code that can be relocated in virtual memory at load time without alteration. All references in PIC files to global or external symbols are made indirectly through addresses stored in a data area so that the loader can find and bind them to reflect their virtual-memory location at load time without modifying code. Shared run-time libraries (SRLs), by contrast, are constructed of non-PIC and are bound to fixed virtual addresses for execution. They cannot have their addresses modified at load time.

An important attribute of DLLs is that they can be dynamically loaded; that is, a running program can load a DLL and gain access to its symbols. This capability means that you do not need to load infrequently loaded DLLs when the application is loaded. Instead, you can load and use these DLLs when required during execution and unload them when they are no longer needed.

Like other types of libraries, DLLs provide functions and data needed by a program or other DLLs. A DLL might be any of these:

- A library that supports a single program
- A library that is available to a project or a group with common computational needs
- A library that is available to all users

PIC and DLLs were introduced in the G06.20 RVU. Non-PIC and SRLs continue to be supported in the G06.2x RVUs. On H-series systems, all native code is PIC, and all native libraries are DLLs. Non-PIC and SRLs are not supported. DLLs on H-series systems include:

- The system library, which is packaged as a set of implicit DLLs
- Other libraries supplied by HP, such as compiler run-time libraries
- All user-created libraries.

The TNS/E native compilers generate PIC object files (linkfiles), the TNS/E linker `eld` creates executable loadfiles (programs or DLLs), and the loader and operating system load and execute the results.

For more information about building and using DLLs, see the *DLL Programmer's Guide for TNS/E Systems*.

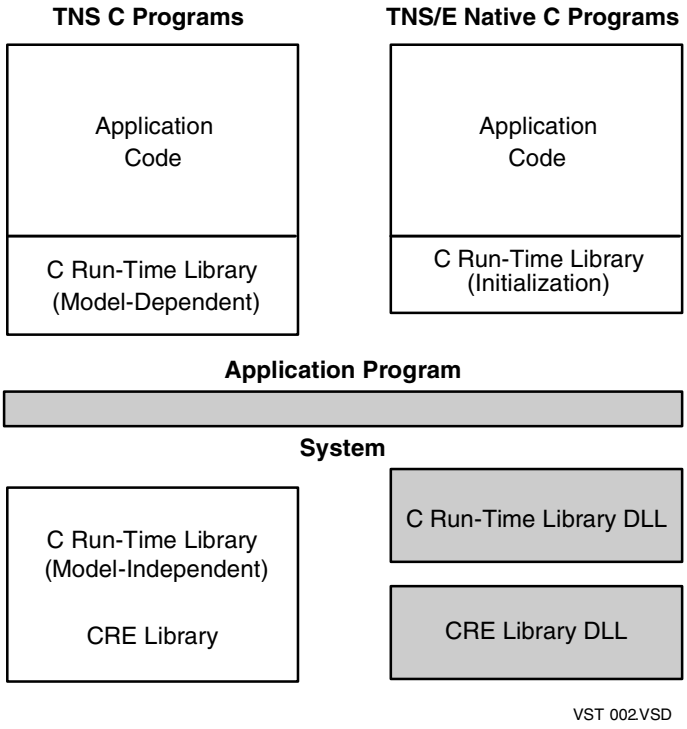
Native Mode Conversion Considerations

Code that is configured in the system library for TNS processes is packaged in DLLs for TNS/E native processes. Public libraries, such as the C run-time library, the COBOL run-time library, the TCP/IP sockets library, the Tools.h++ class library, and the OSS API are also packaged as DLLs for TNS/E native processes. No code changes are required to use these DLLs. Certain HP-supplied libraries that have been repackaged as DLLs have new names, which means that build scripts that reference these libraries will need to change.

If your application uses a TNS user library, that library must be rebuilt as a DLL. See Section 6, Converting a TNS User Library, for more information.

Figure 1-2 on page 1-19 illustrates how code that is bound into TNS C programs or configured in the system library is configured in DLLs for TNS/E native C programs.

Figure 1-2. Run-Time Library Organization



As shown in Figure 1-2, the TNS C and CRE libraries are located in the system library. The native C and CRE libraries are DLLs.

The memory and data-model dependent TNS C run-time library code is linked into the program. Native programs support only one memory model and one data model, so linking in run-time library code is unnecessary. Native C programs do require C run-time library initialization code, which is located in the CRTLMAIN file (in the Guardian environment) or the `crtlmain.o` file (in the OSS environment and on the PC). For details on linking C programs, see the *C/C++ Programmer's Guide*.

KMSF

A swap file is a disk file used for copying data between physical memory and disk storage. Pages of memory are swapped to disk when physical memory is needed, and swapped back to physical memory when the data is needed.

Beginning with the D40 RVU and continuing with the G-series and H-series RVUs, the Kernel-Managed Swap Facility (KMSF) manages virtual memory using swap files under its control. Each processor in a node has one or more kernel-managed swap files that provides the swap space needed by TNS and native processes running on the processor.

KMSF manages:

- The globals-heap segment and DLL instance data segments (variable and constant) for TNS/E native processes
- The memory stack segment, RSE backing store segment, and privileged backing store segment for TNS/E native processes
- The main stack segment and the privileged stack segment for TNS processes
- The user data segment for TNS processes
- The default extended data segment for TNS processes, unless it is explicitly specified not to be managed by KMSF
- Program-allocated extended data segments (selectable or flat segments), such as those allocated with the SEGMENT_ALLOCATE_ procedure, for TNS and native processes, unless you explicitly specify them not to be managed by KMSF

For details on how programs can specify that their extended data segments not be managed by KMSF, see the SEGMENT_ALLOCATE_ procedure in the *Guardian Procedure Calls Reference Manual*.

KMSF benefits include speeding up process creation and deletion and reducing the total size of swap files on disks.

You configure and manage the swap volumes used by KMSF with the NSKCOM utility. KMSF emits EMS warnings when swap space is running low. For more information on KMSF, see the *KMSF Manual*.

KMSF changes the control you have over process swap files. You can still specify the space you need, but you cannot decide at process creation where the data is swapped. As a result of this change, commands and procedures related to swap files might have reduced or no effects. Swap file information from procedures and commands might have different meanings for native processes. Table 1-2 provides an overview of these changes. For more details, see the specific procedure or command.

Table 1-2. Comparing Swap for TNS and Native Processes

How Specified	Effect on TNS Processes	Effect on Native Processes
RUN command SWAP option SWAP DEFINEs Swap file parameters in process creation procedures, such as NEWPROCESS, PROCESS_CREATE_, and PROCESS_LAUNCH_	Managed by KMSF. Ignored. Value passed for informational purposes In some cases, specifies the volume for temporary files created by a process	Managed by KMSF. Ignored. Value passed for informational purposes In some cases, specifies the volume for temporary files created by a process
RUN command EXTSWAP option Extended swap file parameters in process creation procedures, such as PROCESS_CREATE_ and PROCESS_LAUNCH_	Specifies the volume or file for a process's default extended data segment Managed by KMSF unless an option, value, or file is specified	Ignored. Native process architecture does not have extended swap space.
RUN command MEM option ?DATAPAGES directive Memory pages parameters in process creation procedures, such as NEWPROCESS, PROCESS_CREATE_, and PROCESS_LAUNCH_	Specifies the number or maximum number of data pages to be allocated for a process' user data stack	Managed by KMSF. Ignored. Similar heap attribute values set with `nld` utility or PROCESS_LAUNCH_ procedure
STATUS commands and process information procedures, such as PROCESSINFO	Returns the default or specified swap file name for a process' user data. Actual swap file managed by KMSF. If managed by KMSF, returns the volume name and #0 for a process' compiler-generated (default) extended data segment. Actual swap file managed by KMSF.	Returns the default or specified swap file name for a process' user data. Actual swap file managed by KMSF. Returns the volume name and #0 for a process' compiler-generated (default) extended data segment. Actual swap file managed by KMSF.
ALLOCATESEGMENT or SEGMENT_ALLOCATE_ procedure with volume or file name specified	Swap managed by process	Swap managed by process
ALLOCATESEGMENT or SEGMENT_ALLOCATE_ procedure without volume or file name specified	Managed by KMSF	Managed by KMSF

Benefits of Native Mode

- General
  - Native code often runs significantly faster than TNS interpreted or accelerated code.
  - Native object code does not need to be accelerated.
  - Native processes support more global variables than TNS processes.
  - Native processes support a default stack of 2 MB (expandable to 32 MB), which is significantly larger than the 64 KB stack limit for TNS processes.
- C and C++
  - The native C and C++ compilers and the `eld` utility run on the PC as part of ETK. These native cross compilers generate code that runs on NonStop systems.
  - The native Guardian and OSS C run-time library functions provide greater correspondence and interoperability than the TNS C library functions.
  - The native Guardian and Open System Services C run-time library provide much of the X/OPEN UNIX 95 application program interface (API).
  - The native C++ compiler generates code that is easier to debug than the TNS C++ preprocessor, Cfront.
- COBOL
  - The code space limit for a native COBOL program is 32 MB, compared to 128 KB for a TNS COBOL program.
  - The data space limit of approximately 60 KB for the sum of all the Working-Storage Sections and File Sections of a TNS process does not apply to native COBOL programs. The Working-Storage Section and the Extended-Storage Section are the same in native COBOL, and there is no distinction between user data space and user extended space.
  - Native dynamic-link libraries (DLLs) are consulted automatically.

    If a TNS COBOL program calls utility routines, it must put the libraries that contain those routines (one or more of COBOLLIB, CBL85UTL, and CLULIB) on search lists (using the compiler directives SEARCH, LIBRARY, and CONSULT).

    For a native COBOL program, search lists are optional. If a program does not have search lists, or if the compiler cannot find an external reference in the files on the search lists, the compiler automatically searches the DLLs ZCOBDLL and ZCREDLL and then the file ECOBEXT (the native equivalent of COBOLEXT).

ZCOBDLL contains the COBOL utility routines which, for TNS COBOL, reside in the system library, COBOLLIB, and CBL85UTL. ZCREDLL contains the COBOL run-time routines which, for TNS COBOL, reside in the system library or in CLULIB.

Constraints of Native Mode

Before you start converting applications to native mode, be aware of the constraints of the native environment:

- pTAL, unlike TAL, cannot contain embedded SQL statements.
- C and COBOL programs that contain embedded SQL statements must use the SQL release 2 feature set.
- TNS interpreted object code, TNS accelerated object code, and native object code cannot be mixed in one program file.
- A program and its user library must both be native object files.
- Mixed-language programs can consist only of C, C++, COBOL, and pTAL.
- TNS/E native compilers and linkers are not hosted on TNS/R systems.
- TNS/E native object files cannot be executed on TNS/R systems.
- TNS/R native object files cannot be executed on TNS/E systems.

2 Developing a Conversion Strategy

This section describes the decisions you must make to convert a program to native mode, including:

Determining Which Programs to Convert

To determine which TNS programs to convert to native mode, follow these guidelines:

- If CPU performance is not an issue (for example, the program is I/O-bound), you gain some but not much measurable performance by native-compiling the program instead of accelerating the program with OCA. Where CPU performance is an issue, the great advantage to native-compiling is that programs usually run significantly faster.
- If your program consists mainly of calls on system code, you do not gain much additional performance by native-compiling the program itself. Much of the performance-critical and heavily-used HP system code has been native-compiled.
- If your program uses large amounts of memory, is very recursive, or makes many function calls with a large number of local variables, you often gain additional performance by native compiling the program instead of accelerating the program. The native process architecture supports a much larger and more efficient process heap and stack.
- If you want to compile and link your program on a PC using ETK, you must convert to native mode.
- The OSS environment on TNS/E systems does not support TNS development and execution. Therefore, if you are migrating a OSS TNS application to a TNS/E system, you must convert the application to native mode. You can either convert the application to native mode on the TNS/R system, and then migrate the converted application to the TNS/E system (in most cases, a simple process) or you can migrate and convert the application in a single step as described in this manual.

You can use the Measure system performance-analysis tool to determine which programs can be significantly improved by native-compiling. When measuring program performance, select a measurement window that coincides with some representative portion of the system workload, usually the system's peak time. For further verification, take and compare multiple measurements to create a level of confidence with the data collected.

Another issue to consider when converting to native mode is the effort required to convert the program. It is not a trivial matter to convert programs that rely significantly on TNS hardware and process architecture. For example, a program that explicitly manipulates the P-register probably requires significant changes to convert to native mode.

Preparing Programs for Conversion

If you are migrating your TNS programs from a TNS/R system, you must complete these tasks before converting a program to native mode:

- Compile programs with a D20 or later TNS compiler version and resolve any C-series to D-series conversion issues.
- Run your programs successfully on a system running the current version of the operating system and resolve any migration issues.
- Run your program successfully on an Itanium-based NonStop system and resolve any hardware variances between TNS and TNS/E systems. For information on hardware variances, see the *Object Code Accelerator (OCA) Manual*. For general information on migrating a Guardian TNS program to a TNS/E system, see the *H-Series Application Migration Guide*.
- Convert your C or C++ program to use the 32-bit or wide data model if it does not do so already. For details, see Converting Code to Use 32-Bit Pointers and Integers on page 3-3.

This manual assumes that you have completed these tasks. For information on C-series to D-series migration, see:

- *Guardian Application Conversion Guide*
- *D-Series System Migration Planning Guide*

Planning System Resources

In general, native object files use the same or slightly fewer disk resources than accelerated object files (object files produced by a TNS compiler and processed by the accelerator or OCA). Likewise, native processes use comparable processor memory to TNS processes running in accelerated mode.

The TNS/E native environment is available only on systems running H-series versions of the operating system. These systems include the NonStop NS-series servers.

Maintaining Common Source Code for TNS and TNS/E Native Compilers

If your program requires few changes to convert to native mode, you can often maintain common source code for the TNS and native compilers. If your program requires extensive or complex changes to convert to native mode, maintaining common source code is impractical.

By using D3x or later versions of the TNS compilers and H-series versions of the native compilers, you can maintain common source code for the D3x and H-series operating system RVUs. (Because of differences between the D20 and D30 TNS compilers, it is too difficult to maintain a common source between the D20 and H-series operating system RVUs.)

To maintain a common source, use features available in the D3x or later RVUs and features available to TNS and TNS/E native programs. Features that you cannot use include:

- C++ language features that the cfront C++ preprocessor cannot process
- pTAL language features that the TAL compiler does not support
- The small-memory model in C and the 16-bit data model in C and C++
- COBOL language features that are not supported by both the TNS COBOL and native COBOL compilers

To maintain a common source, you must use:

- Separate build scripts to run the TNS and native compilers and other tools
- D3x versions of the TNS C compiler
- D32 or later versions of the TNS COBOL compiler
- D31 or later versions of the TAL compiler
- Versions of the Guardian procedure external declaration files (EXTDECS and CEXTDECS) and the C header files that correspond to the RVU
- COBOL dummy files (COBOLEX0, COBOLEX1, COBOLEXT, ECOBEX0, ECOBEX1, and ECOBEXT)

In some cases, two logically equivalent but syntactically different fragments of code might be required: one for TNS compilers and one for native compilers. Such code can be isolated within a source file for conditional compilation:

Language	For conditional compilation, use:	For more information, see:
C C++	#ifdef macros	*C/C++ Programmer's Guide*
COBOL	IF, IFNOT, ENDIF, SETTOG, and RESETTOG directives	*COBOL Manual for TNS/E Programs*
pTAL	IF directives	*pTAL Reference Manual*

For source code compiled with both the pTAL and TAL compilers, you must first check the syntax of the program with the pTAL compiler's syntax checking option and then compile the program with the TAL compiler. The D31 TAL compiler supports but does not check most pTAL language features.

Adjusting for Increased DCT Limits

The Destination Control Table (DCT) contains entries for logical device numbers and named processes. The DCT limit refers to the maximum number of logical device numbers and named processes that the operating system can accommodate. As of the G06.23 RVU, the size of the DCT can optionally be increased from its previous limit of 32,767 (a logical device number can have at most 15 bits) to 65,376 (a logical device number can have up to 16 bits). This change can affect programs that you are migrating to an H-series system and that call any of these C-series procedures:

C-Series Procedure	Extended DCT limits affect calls that:
FILEINFO	Use the optional *ldevnum* parameter
GETDEVNAME	(Affects all calls)
GETPPDENTRY	(Affects all calls)
GETSYSTEMNAME	Use the return value as an ldev or check for specific error codes
LOCATESYSTEM	Use the return value as an ldev or check for specific error codes
LOOKUPPROCESSNAME	Pass a DCT index in the *ppd* parameter

In G06.23 and later G-series RVUs, the default setting for extended DCT limits is OFF (extended limits are not in effect). In H-series RVUs, the default setting for extended DCT limits is ON (extended limits are in effect). Therefore, if you have not yet changed any affected applications to allow for increased limits, you must do one of the following:

- Ensure that the system default for or DCT limits extension is reset to OFF (enter an SCF command).
- Change your program to allow for the increased DCT limits.

The recommended solution is to replace the affected procedures with updated procedures that can handle increased DCT limits:

C-Series Procedure	Replacement Procedure
FILEINFO	FILE_GETINFOLIST_
GETDEVNAME	DEVICE_GETINFOBYLDEV_ CONFIG_GETINFOBYLDEV_ CONFIG_GETINFOBYLDEV2_ FILENAME_FINDSTART_ FILENAME_FINDNEXT_
GETPPDENTRY	PROCESS_GETPAIRINFO_
GETSYSTEMNAME	NODENUMBER_TO_NODENAME_
LOCATESYSTEM	NODENAME_TO_NODENUMBER_
LOOKUPPROCESSNAME	PROCESS_GETPAIRINFO_

For more details using the replacement procedures, see the *Guardian Procedure Calls Reference Manual*.

Determining Optimization Levels

The native C, native C++, COBOL, and pTAL compilers support three levels of optimization:

Optimization Level	Characteristics
0 (No optimization)	● Slower execution ● Supports symbolic debugging ● Data always in memory ● Useful during development and migration; not intended for production
1 (Intermediate optimization)	● Faster execution ● Supports symbolic debugging ● Data not always in memory ● Useful in production
2 (Full optimization)	● Fastest execution (on average, a 15 percent reduction in pathlength over level 1) ● Limited support for symbolic debugging ● Data not always in memory ● Useful in production

Use optimization level 0 to debug a program, and then use optimization level 1 or 2.

The `optimize` pragma (in the Guardian environment), the `-Woptimize c89` flag (in the OSS environment), and the `-O`*`optlevel`* `c99` flag (in the OSS environment) set the optimization level for the native C and C++ compilers. The OPTIMIZE directive sets

the optimization level for the native COBOL and pTAL compilers. The native compilers' default optimization level is 1.

For C, C++, and pTAL, optimization can be set on a module or procedure basis. Therefore, native programs can contain modules or procedures compiled at different optimization levels. This hybrid approach can be used to improve program performance while maintaining as much symbolic debugging support as possible. With this approach, you compile performance-critical code at optimization level 2 and the remainder at optimization level 1. The Measure PROCESS entity can be used to select the performance-critical procedures.

Use a hybrid approach (optimization level 2 and level 1):

- If performance is important
- IF symbolic debugging is important for analyzing failures

Use full optimization (optimization level 2):

- If performance is critical
- If failures can often be reproduced in a development environment using a less optimized version of the program

Use intermediate optimization (optimization level 1):

- If performance is rarely critical
- If failures are difficult to reproduce in a development environment with a less optimized version of the program
- If a hybrid approach is too complicated

For performance reasons, optimization level 0 is rarely used outside of a development environment.

You can run Native Inspect on code compiled at optimization level 2. However, the ability to set breakpoints, step through code, and display variables is limited. For more details on how optimization impacts symbolic debugging, see the *Native Inspect Manual*.

For additional performance information, see the performance document for a given RVU.

Determining Data Alignment

By default, the native compilers align data and generate code assuming that data misalignment traps do not occur. If a program is not sensitive to how a compiler allocates consecutive fields in structures, use the default data alignment. Data alignment is an issue for the COBOL compiler only when level-01 or level-77 data items are used.

If a program is sensitive to how a compiler allocates structures, specify the `FIELDALIGN CSHARED2` and `REFALIGNED 2` pragmas in C and C++ and the

FIELDALIGN SHARED2 and REFALIGNED 2 directives in pTAL. These pragmas and directives ensure that the native compilers use the same data alignment as TNS compilers.

Another strategy is to set the `FIELDALIGN CSHARED2` and `REFALIGNED 2` pragmas or the FIELDALIGN SHARED2 and REFALIGNED 2 directives when you first start converting a program. After the program has been converted to native mode and runs correctly, you can remove the pragmas or directives and recompile the program. You can then resolve any problems caused by data misalignment.

The alignment of level-01 and level-77 data items is different in TNS and TNS/E native COBOL. In TNS COBOL, level-01 and level-77 data items are aligned on a 2-byte boundary. In TNS/E native COBOL, these items are aligned on an 8-byte boundary. For consistency with TNS COBOL, you can direct the TNS/E native COBOL compiler to align level-01 and level-77 items on a 2-byte boundary by adding the SYNCHRONIZED clause to their declarations.

The data alignment you select depends on whether data is shared. See Section 9, Converting Programs That Share Data, if data is shared between:

- TNS programs and native programs
- pTAL programs and native C or C++ programs

For more information on how the native compilers align data, see:

- *C/C++ Programmer's Guide*
- *pTAL Conversion Guide*

Converting Programs With Misaligned Data

Programs compiled with the TNS and native compilers align data items according to certain rules. TNS compilers align data on even-byte boundaries, while TNS/R and TNS/E compilers align 4-byte data items on boundaries whose addresses are a multiple of 4. TNS/E compilers also align 8-byte data items on 8-byte boundaries. Additionally, TNS/E native GETPOOL and malloc procedures allocate buffers aligned on 16-byte boundaries. Occasionally, however, a programming error or a run-time event causes a data item to violate these alignment rules.

In TNS interpreted or accelerated programs on a TNS/R system, these misaligned addresses are sometimes rounded down to the next lower even-byte address. This "round-down" behavior can result in unexpected program behavior, including run-time errors or incorrect results.

On a TNS/E system, in both TNS interpreted or accelerated programs and in native programs, misaligned addresses are never rounded down. As a result, TNS program with misaligned addresses that are rounded down on a TNS/R system might behave differently when converted to TNS/E native mode (because the misaligned address will not be rounded down). For this reason, you should ensure that TNS programs on TNS/R systems do not contain data misalignments before converting them to native mode.

Both TNS/R and TNS/E systems offer facilities for detecting misaligned addresses and for controlling program action if a misalignment is detected. For more information on detecting and correcting data misalignments, see:

- *Binder Manual*
- *C/C++ Programmer's Guide*
- *pTAL Reference Manual*
- *TAL Programmer's Guide Data Alignment Addendum*

Tuning the Performance of Native Programs

To get the maximum performance from a native program, make sure that your program does not spend excessive time in compatibility traps caused by data misalignment.

Detecting Compatibility Traps

Compatibility traps can be detected by using the Measure PROCESS entity during a representative run of your program. The PROCESS entity COMP-TRAP counter contains the number of compatibility traps.

If you detect significant compatibility traps, use the EPTRACE tool against the program during another representative run. EPTRACE reports the type of each compatibility trap and the code address of where each trap occurred. Next, use the `enoft` utility to map the code address from EPTRACE to the source code location in the program.

Measure and EPTRACE run only in the Guardian environment. Use an OSS run `gtacl` command to run them from the OSS environment.

Eliminating Compatibility Traps

In most cases, data misalignment in data objects and reference misalignment in pointers cause traps. To eliminate compatibility traps caused by data misalignment in data objects, specify a FIELDALIGN SHARED2 pragma or directive for the misaligned data object.

By default, the native compilers generate code for pointer dereferencing operations that expects the pointer to contain an address that satisfies the alignment requirements of the data object being pointed to. For example, a 4-byte object should have an address which is a multiple of four. If the data object is at an address that does not satisfy its alignment requirements, a compatibility trap occurs. To avoid this compatibility trap, specify a REFALIGNED 2 pragma or directive on the pointer to the object. This directs the native compilers to generate code that assumes the dereferenced object is not properly aligned and compensates for the improper alignment. While REFALIGNED 2 always generates a few extra instructions for each dereferencing operation, a compatibility trap results in hundreds of additional instructions.

These directives and pragmas ensure that compatibility traps do not occur. For additional information on data alignment, see:

- *C/C++ Programmer's Guide*
- *pTAL Conversion Guide*

3 C and C++ Conversion Tasks

This section describes how to convert TNS C and C++ programs to TNS/E native mode.

The C and C++ compilers to which this section applies are:

Compiler	T Number
TNS C	T9255
TNS C++	T9541
TNS `c89`	T8629
TNS/E `c89`, `c99`	T8164
TNS/E CCOMP	T0549
TNS/E CPPCOMP	T0549

This section discusses:

This section assumes your program can be compiled by the current TNS C compiler or TNS C++ preprocessor. It also assumes your program runs on the current version of the operating system. See the *C/C++ Programmer's Guide* for information on converting C programs to use the current TNS C compiler.

Using the Native C and C++ Compilers

Both the TNS and native C compilers conform to the following ISO/ANSI C language standards: ISO/IEC 9899:1990 for C, ISO/IEC 9899:1999 for C, and ISO/IEC 14882:1998 for C++. Source code that compiles without warnings or errors with the TNS C compiler or C++ preprocessor might get warnings or errors using the native C and C++ compilers. (In most cases, the native compilers are better than the TNS compilers in detecting violations to the ISO/ANSI C standard.) For example, the native C and C++ compilers detect these errors that the TNS compilers do not detect:

- Characters trailing on a `#include` line, except `nolist`.
- Incorrect use of a NULL pointer. In C, NULL is defined as zero. In TNS programs, address 0 exists. If you call a function, such as `strlen()` with a NULL pointer, the function does not trap but returns an answer (typically 1). In native C programs, address 0 does not exist. Such function calls fail, usually with a SIGSEGV. This user bug is undetected on the TNS architecture.

It is possible, but highly unlikely, that you will need to make changes caused by differences in the translation limits of the TNS and native compilers.

Program logic or behavior that depends on the knowledge of the underlying machine architecture or uses undocumented features (mainly privileged features) might require changes to compile and run correctly using the native compilers.

There is no correlation between either the text or the number of diagnostic messages produced by the TNS and native compilers.

The TNS C compiler and C++ preprocessor support HP extensions for NonStop systems to the C and C++ languages by default. The native C and C++ compilers do not support these HP extensions by default. Specify the `EXTENSIONS` pragma or the `-Wextensions c89` or `c99` flag to direct the native compilers to support these HP extensions.

In the Guardian environment, the CCOMP command runs the native C compiler, and the CPPCOMP command runs the native C++ compiler. In the OSS and PC environments, the native `c89` utility runs the native C and C++ compilers, and the native `c99` utility runs the native C compiler . On the PC, the `c89` utility can be run from within ETK.

The `c89` and `c99` commands for the PC and OSS environments provide an option to specify the target platform for the compilation: specify `-Wtarget=TNS/R` to generate RISC code or `-Wtarget=TNS/E` to generate Itanium code. The default for the PC is `-Wtarget=TNS/R`, and the default for OSS is the same as the host platform. Note that

TNS/R code cannot be executed on a TNS/E platform, and TNS/E code cannot be executed on a TNS/R platform.

For `c89` and `c99` conversion information, see Section 11, OSS API and Utilities Conversion Tasks.

All code generated by the TNS/E native compilers is position-independent code (PIC). PIC is code that need not be modified to run at different virtual addresses and is used to create dynamic-link libraries (DLLs).

For details on compiling and linking native C and C++ programs, see the *C/C++ Programmer's Guide*.

Converting Code to Use 32-Bit Pointers and Integers

The memory model determines the size of pointers. In the TNS C environment, there are two memory models: the small-memory model (16-bit pointers) and the large-memory model (32-bit pointers).

The data model determines the size of type `int`. In the TNS C and C++ environments, there are two data models: the 16-bit data model and the 32-bit (or wide) data model.

In the native C and C++ environments, there are only the large-memory model and the 32-bit data model.

You must convert existing Guardian C programs that use the small-memory model to the large-memory model. You must also convert existing Guardian C and C++ programs that use the 16-bit data model to the 32-bit data model. (You do not need to convert existing OSS C and C++ programs because OSS supports only the large-memory model and the 32-bit data model.)

The memory and data models a program uses are determined by the environment in which you run the compiler (Guardian or OSS), the version of the compiler, the `SYSTYPE` pragma setting, and the explicit `XMEM`, `NOXMEM`, `WIDE`, or `NOWIDE` pragmas in the source code. D-series versions of the Guardian TNS C compiler and D20 versions of the Guardian C++ preprocessor generate programs that use the large-memory model and the 16-bit data model by default. See the *C/C++ Programmer's Guide* for further details.

Before compiling small-memory model or 16-bit data model programs with the native compilers, it is often easier to first convert them to the large-memory model and 32-bit data model using the TNS C compiler. Follow these guidelines for doing the conversion:

- Specify the `XMEM` and `WIDE` pragmas in your source code.
- Use the TNS C compiler with the `STRICT` pragma.

- Ensure that the type of a function call argument matches the defined type of its associated parameter. The TNS C compiler issues this warning message

  for argument-parameter mismatches:

  ```
  Warning 86: argument "name" conflicts with formal definition
  ```

- Write function prototypes for all user-written functions without prototypes. The TNS C compiler issues this warning message for function calls without corresponding function prototypes:

  ```
  Warning 95:  prototype function declaration not in scope:
  "function-name"
  ```

- Ensure that the formal and actual parameters of pointer types are matched. The TNS C compiler issues this warning message if pointers do not match:

  ```
  Warning 30:  pointers do not point to same type of object
  ```

  For example:

  ```
  int func1(short *);
  ```

  In the 16-bit data model and the large-memory model, you can pass to `func1` a pointer of type `short` or `int` and get correct results. In the 32-bit data model, you can pass to `func1` only a pointer of type `short`; a pointer of type `int` generates incorrect results.

  Parameter mismatch is most often an issue for Guardian system procedures and external TAL and pTAL procedures.

- Ensure that literals do not cause type mismatches, as illustrated in this example:

  ```
  #include <cextdecs(MONITORCPUS)>
  ...
  short get_cpu_number;
  MONITORCPUS(0x8000 >> get_cpu_number);
  ```

  In the 32-bit data model, if `get_cpu_number` is equal to zero, an arithmetic overflow occurs because the compiler generates code to convert an unsigned 32-bit integer to a 16-bit signed integer. Declarations in the `cextdecs` header file do not use the type `unsigned short`.

- Avoid using the type `int` in your program if possible. Use type `long` or `short` instead. However, if you want to keep your program data-model independent, you cannot avoid using type `int` completely. For example, C library calls, bit fields, TCP/IP sockets library functions, and Guardian system procedures might require type `int`.

Using IEEE Floating Point Format

TNS C and C++ programs use the HP proprietary TNS floating-point format. The native C and C++ compilers provide the option of using either the TNS or the IEEE floating-point format. The default option is IEEE format. Programs that depend on TNS format must specify the TANDEM_FLOAT pragma (or `-Wtandem_float` flag) when converted to native mode. Differences between IEEE and TNS floating-point formats include:

- Results of IEEE floating-point operations might differ slightly from those of TNS floating-point operations.
- IEEE floating-point values can include not-a-number (NaN) and infinity.
- The sign of 0.0 (zero) in IEEE format can be either positive or negative.

Replacing Obsolete External Function Declarations

External function declarations declare functions not written in the C language. Current TNS C compilers recognize correctly, but issue warnings for, C-series external function declaration syntax with newer replacements. The native compilers do not recognize the C-series syntax. If you use `FUNCTION` pragma syntax or declarations with _language name keywords (such as `_tal` and `_c`), no changes are required. See the *C/C++ Programmer's Guide* for details.

Replacing Obsolete Keywords

Current TNS C compilers recognize, but issue warnings for, certain C-series keywords with newer replacements. The native C compiler does not recognize the C-series keywords. Replace the following C-series keywords with the equivalent replacement keywords:

C-Series Keyword	Replacement
`cc_status`	`_cc_status`
`extensible`	`_extensible`
`extptr`	See the following paragraph
`lowmem`	`_lowmem`
`tal`	`_tal`
`variable`	`_variable`
`myproc ="`*`alias-name`*`"`	`_alias("`*`alias-name`*`") myproc` (or use the FUNCTION pragma syntax instead—see the *C/C++ Programmer's Guide* for more information)

The `extptr` keywords identify 32-bit pointers in external function declarations for TAL. Native programs support only 32-bit pointers, so this keyword is unnecessary. Delete the `extptr` keyword.

Changing Use of _cc_status for Return Values

`_cc_status` is used for calls to Guardian and TAL procedures that set a condition-code register instead of returning a value. For TNS C and C++ programs, the CCL, CCE, and CCG macros examine the results of a function declared with the `_cc_status` type specifier. For native C and C++ programs, you must replace these macros with the `_status_lt(x)`, `_status_eq(x)`, and `_status_gt(x)` macros. These new macros are defined in the `tal.h` header and can be used in TNS and native programs. These macros are:

- For native C and C++:

```
#define _status_lt(x) ((x) < 0)
#define _status_eq(x) ((x) == 0)
#define _status_gt(x) ((x) > 0)
```

- For TNS C and C++:

```
#define _status_lt(x) ((x) == 2)
#define _status_eq(x) ((x) == 1)
#define _status_gt(x) ((x) == 0)
```

Example 3-1 shows the difference between using the new macros and the previous macros to examine _cc_status.

Example 3-1. Examining _cc_status

Previous macros (TNS Programs Only)

```
_tal _extensible _cc_status READX ( ... );
#include <tal.h>
...
_cc_status CC;

CC = READX ( ... );

if (CC == CCL) {
...
} else if (CC == CCG) {
...
}
```

Current macros (TNS and Native Programs)

```
_tal _extensible _cc_status READX ( ... );
#include <tal.h>
...
_cc_status CC;

CC = READX ( ... );

if (_status_lt(CC)) {
...
} else if (_status_gt(CC)) {
...
}
```

Replacing Calls to Obsolete C Library Supplementary Functions

The native C run-time library does not support the C library functions listed in Table 3-1. None of the obsolete functions are specified in the ISO/ANSI C Standard, the X/OPEN XPG4 Specification, or the X/OPEN UNIX Specification. In most cases, you can replace the obsolete function with another function and a few additional lines of code.

Table 3-1. Obsolete C Supplementary Functions (page 1 of 2)

Obsolete Function	Suggested Replacement
`_is_system_trap()`	Delete `_is_system_trap()`. Trap handling mechanism is replaced with signals in native processes. See the *Guardian Programmer's Guide* for details.
`iscsym()`	Write your own function `iscsym()` to check whether a character is a valid character in a C identifier.
`iscsymf()`	Write your own function `iscsymf()` to check whether a character is a valid first character in a C identifier.
`memswap()`	Replace `memswap()` with a series of calls to `memcpy()` to swap the blocks of memory using a temporary buffer. The order and type of parameters for these two functions are different.
`movmem()`	Replace `movmem()` with a call to `memmove()`. The order and type of parameters for these two functions are different.
`repmem()`	Replace `repmen()` with a series of calls to `memcpy()` to copy the block of memory the required number of times. The order and type of parameters for these two functions are different.
`setmem()`	Replace `setmem()` with a call to `memset()`. The order and type of parameters for these two functions are different.
`setnbuf()`	Replace `setnbuf()` with a call to `setbuf()` with a buffer parameter set to NULL.
`stcarg()`	Replace `stcarg()` with a call to `strcspn()`. The return values for these two functions are different.
`stccpy()`	Replace `stccpy()` with a call to `strncpy()`. The type of parameters and the return value for these two functions are different.
`stcd_i()`	Replace `stcd_i()` with a call to `strtol()`. The order and type of parameters and the return value for these two functions are different.
`stcd_l()`	Replace `stcd_l()` with a call to `strtol()`. The order and type of parameters and the return value for these two functions are different.
`stch_i()`	Replace `stch_i()` with a call to `strtol()`. The order and type of parameters and the return value for these two functions are different.
`stci_d()`	Replace `stci_d()` with a call to `sprintf()` with a `%d` conversion specifier.
`stcis()`	Replace `stcis()` with a call to `strspn()`.
`stcisn()`	Replace `stcisn()` with a call to `strcspn()`.
`stclen()`	Replace `stclen()` with a call to `strlen()`.

Table 3-1. Obsolete C Supplementary Functions (page 2 of 2)

Obsolete Function	Suggested Replacement
`stcpm()`	Replace `stcpm()` with a call to `strstr()`. The order and type of parameters and the return value for these two functions are different. Additional code might be required to support pattern matching.
`stcpma()`	Replace `stcpma()` with a call to `strstr()`. The order and type of parameters and the return value for these two functions are different. Additional code might be required to support pattern matching.
`stcu_d()`	Replace `stcu_d()` with a call to `sprintf()` with a `%u` conversion specifier.
`stpblk()`	Replace `stpblk()` with a call to `strspn()` with a string of space characters.
`stpbrk()`	Replace `stpbrk()` with a call to `strpbrk()`.
`stpchr()`	Replace `stpchr()` with a call to `strchr()`. The order and type of parameters and the return value for these two functions are different.
`stpsym()`	Replace `stpsym()` with a call to `strspn()` with a string containing alphanumeric characters. Pass the return value of `strspn()` to `strncpy()` to copy the correct string length.
`stptok()`	Replace `stptok()` with a series of calls to `strtok()` to find a token and `strncpy()` to append a token to another string.
`stscmp()`	Replace `stscmp()` with a call to `strcmp()`.
`trap_overflows()`	Delete `trap_overflows()`. The trap handling mechanism is replaced by signals in native processes.

Replacing Calls to Obsolete C Library Guardian Alternate-Model I/O Functions

The TNS Guardian C library alternate-model I/O functions provide an I/O facility similar to but not identical to the UNIX file descriptor model. To enable improved interoperability between the Guardian and OSS environments, the native environment does not support the Guardian alternate-model I/O functions. Three options are available to programs that use the alternate-model I/O functions:

- Rewrite your code to use the OSS versions of these functions.
- Rewrite your code to use ANSI-model I/O functions.
- Rewrite your code to call Guardian system procedures.

Using the OSS versions of these functions is usually the easiest. Instead of specifying files in the Guardian file system using Guardian file name syntax ($VOL.SUBVOL.FILE), you specify files in the Guardian file system using OSS pathname syntax (`/G/VOL/SUBVOL/FILE`). To use this option, OSS must be installed

on the system running the program. For details, see the function's reference page online or in the *Open System Services Library Calls Reference Manual*.

The obsolete Guardian alternate-model I/O functions and replacement options are listed in Table 3-2.

Table 3-2. Obsolete Guardian Alternate-Model I/O Functions

Obsolete Function	Suggested Replacement
`close()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`creat()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`edlseek()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`fcloseall()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`fcntl()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`fdopen()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`fdtogfn()`	Replace with the `gfileno()` function.
`fileno()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`lastreceive()`	Recode using Guardian FILE_GETRECEIVEINFO_ procedure.
`lseek()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`open()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`read()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`readupdate()`	Recode using Guardian READUPDATEX procedure.
`receiveinfo()`	Recode using Guardian FILE_GETRECEIVEINFO_ procedure.
`reply()`	Recode using Guardian REPLYX procedure.
`unlink()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`write()`	Recode using ANSI-model I/O functions or Guardian system procedures. Use OSS versions of function.
`writeread()`	Recode using Guardian WRITEREADX procedure.

Checking Calls to Changed C Library Functions

Functions Having Different Behavior

The following functions have changes in ISO/ANSI C standard implementation-defined behavior. These changes affect only Guardian C functions and were made to match the behavior of the TNS OSS environment and the UNIX computer industry

Changed Function	Changed Behavior	Action Required
`exit()`	In the TNS Guardian environment, a nonzero status parameter value indicates abnormal termination. In the native Guardian environment, any status parameter value indicates normal termination.	Change calls to the `terminate_program()` function to use the status parameter to indicate abnormal termination.

`fscanf()`
`scanf()`
`sscanf()`
	In the TNS Guardian environment, the hyphen character in a scan set denotes the hyphen character. In the native Guardian environment, the hyphen character might denote a range of characters.	Make the hyphen the first or last character in the scan set to maintain the TNS Guardian environment behavior.

`remove()`	In the TNS Guardian environment, the `remove()` function returns Guardian error codes. In the native Guardian environment, the function sets standard `errno` values.	Change error handling code following calls to the `remove()` function to check `errno` values.

For more information, see the function's reference page in the *Guardian Native C Library Calls Reference Manual*.

The `ecvt()` function has changed to match the XPG4 Version 2 (X/OPEN UNIX) specification.

Changed Function	Changed Behavior	Action Required
`ecvt()`	In the TNS Guardian and OSS environments, the second, third, and fourth parameters are type `short`. In the native Guardian and OSS environments, these parameters are type `int`.	Change calls to the `ecvt()` function to use type `int` instead of type `short`.

For more information, see the `ecvt(3)` reference page online, in the *Open System Services Library Calls Reference Manual*, or in the *Guardian Native C Library Calls Reference Manual*.

Using the setjmp() and longjmp() Functions

Calling the `setjmp()` function in a function marked as inline and then doing a subsequent `longjmp()` to the location of the calling function is a practice that should be avoided. Programs that use `setjmp()` and `longjmp()` in this way generally expect that functions marked as inline are actually inlined, and that the `longjmp()` call will restore the execution context of the function that called `setjmp()`. In TNS programs compiled at optimization level 0, this is sometimes the case and the `longmp()` call might work as expected. But in a TNS/E native program compiled at optimization level 0, functions marked as inline are never actually inlined, and the `longjmp()` call will not work as expected. A preferred practice is to use a macro instead of an inline function.

Using the semctl() Function

The `semctl()` function has an optional fourth parameter that is required in certain cases. As described in the *Open System Services System Calls Reference Manual*, the value passed in that parameter must be defined in the calling program as a semun union, as in this example:

```
union semun {
      int                 val;
      struct semid_ds     *buf;
      unsigned short int *array;
}arg

union semun semopts;
semopts.val = 1;
if (semctl(semid, 0, SETVAL, semopts) ==-1)
```

On TNS/R systems, you can also pass a value directly, without the use of the semun union (for example, as a simple scalar), as in this example:

```
if (semctl(semid, 0, SETVAL, 1) ==1)
```

A value passed in this way gives the expected results.

However, on TNS/E systems, you must define the parameter value as a semun union. Code that does not do so will not function as expected.

Changing Programs That Use Guardian and OSS Environment Interoperability

Because of differences in the Guardian and OSS file systems, six functions require environment-specific parameters: `fopen()`, `freopen()`, `remove()`, `rename()`, `tmpnam()`, and `tmpfile()`. Each of these base functions has a Guardian variant and an OSS variant, such as `fopen_guardian()` and `fopen_oss()`.

In TNS programs, the environment-specific parameters expected by these base functions are determined at compile time. That is, the functions expect Guardian parameters if called from a module compiled with SYSTYPE GUARDIAN and expect OSS parameters if called from a module compiled with SYSTYPE OSS. The run-time environment of the process the module is bound into does not affect the call. For example, an `fopen()` call in a module compiled with SYSTYPE GUARDIAN and bound into a program that runs as an OSS process expects a Guardian parameter at run time.

In native programs, the environment-specific parameters expected by these base functions are determined at run time. That is, the functions expect Guardian parameters if called from a Guardian process and OSS parameters if called from an OSS process. The compilation environment is ignored. This change was made to simplify and enhance the usability of the Guardian and OSS environment interoperability model.

In modules compiled with SYSTYPE OSS and linked into a program that runs as a Guardian process, replace calls to the base function with calls to the `_oss` variant. Likewise, in modules compiled with SYSTYPE GUARDIAN and linked into a program that runs as an OSS process, replace calls to the base function with calls to the `_guardian` variant. For example, replace calls to `fopen()` with either `fopen_oss()` or `fopen_guardian()`, as appropriate.

Changing Code That Relies on Arithmetic Overflow Traps

Guardian TNS C library functions, such as the `tan()` function, generate a trap on arithmetic overflow. In the Guardian native C library, such functions generate neither a trap nor a signal on arithmetic overflow. In most cases, the functions set `errno` to `[ERANGE]` or `[EDOM]` to indicate arithmetic overflow. The behavior of Guardian native C library functions now matches the behavior of the TNS and native OSS C library functions. Change your code that relies on arithmetic overflow to evaluate either `errno` or the parameters to the function call.

By default, the native C and C++ compilers do not generate code that traps on arithmetic overflow. Specify an `OVERFLOW_TRAPS` pragma to generate code that traps (raises a signal) on arithmetic overflow. The overflow traps occur in code generated by the compiler, not code linked into the program (for example, C library functions).

Using Active Backup Programming in C

To convert a TNS program that uses the active backup programming functions to native mode:

- Replace the header file `nonstop.h` with `crtlns.h`.
- Instead of binding in `cnonstop`, link the active backup support library:

Environment	Active Backup Support Library
Guardian	CRTLNS
OSS or PC	`crtlns.o`

For more details on writing programs that use the active backup functions, see the *Guardian Programmer's Guide*.

Replacing Obsolete C++ Library Operations

The TNS C++ run-time library uses the TNS Guardian C library alternate-model I/O functions to provide an I/O facility similar to but not identical to the UNIX file descriptor model. To improve interoperability between the Guardian and OSS environments, the native environment does not support the Guardian alternate-model I/O functions.

Because of this change, the Guardian native C++ run-time library `filebuf` and `fstream` classes cannot be used to specify an existing open file stream with the `attach` function. Two options are available to programs that use the alternate-model I/O functions:

- Rewrite your code to use the OSS versions of these classes.
- Open the file stream instead of using a file descriptor to attach to an existing open stream.

The `attach` function's use of file descriptors is nonstandard behavior.

Using the Tools.h++ Class Library

Two product versions of Tools.h++ (version 6.1 and version 7) are available since the D45.00 and G05.00 RVUs. Only version 7 is available in H-series RVUs. If you are converting a TNS program that uses an earlier version of Tools.h++ to TNS/E native mode, you must convert to Tools.h++ version 7. Version 7 offers many new features over earlier versions, but converting to version 7 requires extensive source code changes. The *C/C++ Programmer's Guide* provides an overview of Tools.h++ version 7 along with a summary of conversion tasks. See the *Tools.h++ 7.0 User's Guide* for detailed conversion information.

Specifying Pragmas or Flags

The native C and C++ compilers are multipass, multicomponent compilers. As such, they require that the following pragmas be specified on the C or C++ compiler Guardian RUN command line or as flags to the `c89` or `c99` utility. You cannot specify these pragmas in source code files:

ANSISTREAMS	[NO]INSPECT	SQL
CPATHEQ	LINES	[NO]STDFILES
ERRORS	OPTIMIZE	[NO]SUPPRESS
HEAP	RUNNABLE	[NO]SYMBOLS
HIGHPIN	RUNNAMED	SYNTAX
HIGHREQUESTERS	[NO]SAVEABEND	SYSTYPE
[NO]INLINE	SEARCH	

For the `c89` or `c99` flag that corresponds to each pragma, see the `c89` or `c99` reference page online or in the *Open System Services Shell and Utilities Reference Manual*. Note that `c89` and `c99` do not support all these pragmas.

Checking Changed Pragmas

For the pragmas listed in Table 3-3, the native C and C++ compilers produce results different from those of the TNS compilers. Verify that programs do not rely on any of the changed behaviors and make appropriate changes.

Table 3-3. Changed Pragmas

Changed Pragma	Changed Behavior	Action Required
`[NO]INLINE`	Native C and C++ compilers do not generate inline code for standard C function calls. Native C++ compiler generates inline code for functions with an `INLINE` specified.	Remove pragma.
`OPTIMIZE`	Native compilers perform a different optimization than the TNS compilers for the same optimization level.	Remove pragma while converting program. After program has been converted, specify desired optimization level.
`SEARCH`	The `eld` utility is invoked instead of the Binder. `eld` links the entire object file (similar to a Binder ADD * command) instead of selectively linking only portions of the object file (as in a Binder SELECT SEARCH command).	None, unless the way in which you build your program requires the select search behavior. See the *eld Manual* for details.
`SQL`	`RELEASE1` is not supported as an option. Native C does not support NonStop SQL/MP RVUs prior to Release 2.	Replace the `RELEASE1` option with `RELEASE2` and convert your application to use Release 2 or later of NonStop SQL/MP.
`SSV`	Native compilers do not ignore `SSV` pragmas following a skipped `SSV` pragma number.	Change `SSV` pragma use if you rely on the changed behavior.
`[NO]WARN`	Native compilers generate different warnings than the TNS compilers for the same warning number.	Remove pragma or replace with native compiler warning number that corresponds to TNS compiler warning. See the *C/C++ Programmer's Guide* for a list of compiler messages.

Removing Obsolete Pragmas

The native C and C++ compilers do not support the pragmas in Table 3-4. In most cases, you must remove the pragma from the source code. In a few cases, you might need to write additional code.

Table 3-4. Obsolete Pragmas (page 1 of 2)

Pragma	Reason pragma is obsolete	Action Required
ANSICOMPLY	Pragma has been replaced by NOEXTENSIONS pragma.	None. The NOEXTENSIONS pragma is set by default.
[NO]CHECK	The native C run-time library does not support the additional parameter checking provided by the TNS C run-time library.	Remove pragma.
CSADDR	Native process memory architecture does not require use of pragma.	Remove pragma.
LARGESYM	Native compilers generate complete symbols information. Pragma is unnecessary.	Remove pragma.
[NO]LMAP	Native compilers do not generate load map information.	Remove pragma.
[NO]NEST	Native compilers do not support nested comments. The ISO/ANSI C Standard does not support nested comments.	Remove pragma and nested comments.
OLDCALLS	Native compilers do not support B-series C language function calling behavior.	Remove pragma and change code to eliminate B-series C language function calling behavior.
SQLMEM	Native process memory architecture does not require use of pragma.	Remove pragma.
STRICT	Native compilers perform similar syntactic and semantic checking by default.	Remove pragma.
TRIGRAPH	Native compilers always translate trigraph characters.	Remove pragma and change code that treats each character in a trigraph as a C token.
VERBOSE	Native compilers in OSS environment do not write compiler banners.	Remove pragma.

Table 3-4. Obsolete Pragmas (page 2 of 2)

Pragma	Reason pragma is obsolete	Action Required
[NO]WIDE	Native compilers generate programs that use only the 32-bit (or wide) data model.	Remove pragma and recode programs that specify NOWIDE to use the 32-bit (or wide) data model.
[NO]XMEM	Native compilers generate programs that use only the large-memory model.	Remove pragma and recode programs that specify NOXMEM to use the large-memory model.
[NO]XVAR	Native process memory architecture does not require use of pragma.	Remove pragma.

4 Converting COBOL Programs

This section describes how to convert TNS COBOL programs to TNS/E native mode.

The compilers to which this section applies are:

Compiler	T Number
TNS COBOL85	T9257
TNS `cobol`	T8498
TNS/E ECOBOL	T0356
TNS/E `ecobol`	T0356

This section discusses:

COBOL Compiler Overview

Both the TNS COBOL compiler and the TNS/E native COBOL compiler comply with the 1985 COBOL85 standard.

The TNS and TNS/E native COBOL support the same optimization levels:

OPTIMIZE 0	No optimization
OPTIMIZE 1	Optimizations that do not interfere with debugging
OPTIMIZE 2	Full optimization

Key differences between the TNS and native COBOL compilers include:

- The native COBOL compilers have new command names. The ECOBOL command calls the Guardian native COBOL compiler, and the `ecobol` command calls the OSS native and PC COBOL compilers.
- The TNS COBOL compilers produce TNS object code, and the TNS/E native compilers produce Itanium object code.
- Code produced by the native compilers is position-independent code (PIC), which can be used to create dynamic-link libraries (DLLs).
- The code space limit for a TNS COBOL program is 128 KB. The code space limit for any single native COBOL program is 16 MB. This limit does not include any contained programs, each of which has its own 16 MB limit. The object code space limit for the combined program file and ordinary DLLs is 256 MB (this does not include public DLLs, such as ZCOBDLL and ZCREDLL).

- The data space limit of approximately 60 KB for the sum of all the Working-Storage Sections and File Sections of a TNS process does not apply to native COBOL programs. The Working-Storage Section and the Extended-Storage Section are the same in native COBOL.
- In TNS/E native COBOL, there is no distinction between the small data area and the large data area.

Converting COBOL Programs

The recommended procedure for conversion from TNS COBOL to native COBOL is:

1. Change your program so that it runs in the Common Run-Time Environment (CRE), if it does not already. (See the *COBOL Manual for TNS/E Programs*.)
2. Verify that you have adequate system resources for the converted programs.

   Native object files use approximately the same amount of disk space as accelerated object files. Native processes use approximately the same amount of processor memory as TNS processes running in accelerated mode.
3. If necessary, change your source program (see Changing the Source Program on page 4-4).

   To learn whether you must change your source program, compile it with the native COBOL compiler, which issues warnings when it encounters source constructs that it does not accept. Running the native COBOL compiler with the DIAGNOSE-85 directive is especially recommended. This directive causes the native COBOL compiler to issue warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the TNS COBOL compiler. For a complete description of the DIAGNOSE-85 directive, see the *COBOL Manual for TNS/E Programs*.
4. If your program calls TNS programs, convert them to native mode. Native programs cannot call TNS programs. For the following list of languages, follow these instructions. The last one, Data Alignment on page 4-3, applies to several languages.

   - C/C++

     Recompile C and C++ programs with the native C and C++ compilers, respectively. Specify the SYMBOLS pragma when recompiling a C or C++ program that your TNS COBOL program references in a CALL or ENTER statement. This generates symbols for use by a symbolic debugger.

   - TAL

     Convert TAL programs to pTAL. (See the *pTAL Conversion Guide*.) Some CRE library routines might no longer exist, so you might have to change calls to them. Also, if you want to reference an object in a CALL or ENTER statement, you must compile your pTAL program with symbols.

If any of your TAL programs use the FORTRAN convention for determining the length of a string parameter (that is, if they use the ENTER routine and do not specify the language TAL), convert them to use another method. (For example, use the *string*:*length* convention or pass the length as a separate parameter.)

For information on tools that can help you convert TAL programs to pTAL programs, see Section 5, Converting TAL to pTAL.

- FORTRAN and Pascal

  There are no native FORTRAN or Pascal compilers. Rewrite FORTRAN and Pascal programs in native C, native C++, native COBOL, or pTAL. You need not use the same language for all of them.

- Data Alignment

  The alignment of certain data items differs between TNS COBOL and native COBOL. The native COBOL compiler aligns each level-01 and each level-77 item on a physical 8-byte boundary, whereas the TNS COBOL compiler aligns these items on a physical 2-byte boundary. Offsets from the containing level-01 or level-77 item are the same in both compilers. The difference in alignment of level-01 and level-77 items will not affect a program unless the program depends on the relative placement in memory of particular level-01 or level-77 items. Any programs that have this dependency should be changed to remove the dependency.

  By default, the TNS compilers for C, C++, and TAL generate code with different data alignment than the corresponding native compilers for C, C++, and pTAL. To convert a TNS C, TNS C++, or TAL program to native mode, you must do one of the following:

  - If DDL was not used, use SHARED2 pragmas in pTAL programs and CSHARED2 pragmas in native C programs.
  - If DDL was used, regenerate DDL source files that TNS and native programs share.

    DDL adds pragmas to ensure that all compilers generate code with the same data alignment. For more information, see Section 7, Converting Data Definition Language (DDL).

5. If you want to put the routines that you converted to native mode in Step 4 in a DLL (instead of in a TNS user library, which your program can no longer use), follow the directions in the *COBOL Manual for TNS and TNS/R Programs*.

6. Compile your source program with the native COBOL compiler. For instructions, see the *COBOL Manual for TNS/E Programs*.

   The native COBOL compiler needs more symbol table space than the TNS COBOL compiler does. If the native COBOL compilation fails because of dictionary overflow, use the PARAM SYMBOL-BLOCKS command to increase the space available for the symbol table, local label table, and embedded SQL statements and then recompile. The maximum value of the PARAM SYMBOL-BLOCKS command's *count* parameter is 25 for the native COBOL compiler (compared to 14 for the TNS COBOL compiler).

7. Run the COBOL program that you compiled in Step 6.
8. If necessary, debug the program.

The H-series native environment provides different debugging tools than the TNS environments. See Native Mode Debugging Tools on page 1-10 for an overview of TNS/E native debugging tools.

Note. The only debugger currently available to TNS/E native COBOL programs is Visual Inspect.

Changing the Source Program

Source program changes fall into these categories:

- General Conversion Tasks
- Removal Required
- Possible Changes Required
- Removal Optional
- New Features

General Conversion Tasks

If your TNS COBOL program calls obsolete or changed Guardian procedures, replace them. Change calls to procedures affected by either the Kernel-Managed Swap Facility (KMSF) or the native process architecture (for example, process creation calls). For more information on obsolete or changed procedures, see Section 10, Converting Programs With Guardian API Calls.

Removal Required

Remove the following directives, statements, and library references from your TNS COBOL source program before compiling it with the native COBOL compiler:

- ENV OLD directive

  If you specify the ENV OLD directive, the native COBOL compiler reports an error. Native COBOL programs always run in the CRE.

- SQL directive

  If your program contains SQL statements, include the SQL directive in the native ECOBOL compiler command line. Do not use the SQL option RELEASE1.

- USE DEBUGGING statement

  The 1985 COBOL standard classifies the USE DEBUGGING statement as obsolete, so you are advised not to use it even in TNS COBOL programs. The native COBOL compiler does not recognize the USE DEBUGGING statement, and the TNS COBOL compiler no longer recognizes the names of the debug items, which are:

  - DEBUG-CONTENTS
  - DEBUG-ITEM
  - DEBUG-LINE
  - DEBUG-NAME
  - DEBUG-SUB-1
  - DEBUG-SUB-2
  - DEBUG-SUB-3

Remove references to these TNS libraries, which native COBOL cannot use and does not need.

- CBL85UTL
- COBOLLIB
- CLULIB

Make these substitutions:

Replace	With
COBOLEX0	ECOBEX0
COBOLEX1	ECOBEX1
COBOLEXT	ECOBEXT

References to the preceding libraries could appear in these contexts:

- CONSULT directive
- LIBRARY directive
- SEARCH directive
- FILE-MNEMONIC clause of the SPECIAL-NAMES paragraph
- OF or IN clause of the CALL or ENTER statement

Possible Changes Required

The TNS and native COBOL compilers handle the following directives, the ENTER statement, floating-point arithmetic, and checkpointing differently. Make any necessary changes to them before compiling your COBOL source program with the native COBOL compiler. For complete descriptions of the directives, see the *COBOL Manual for TNS/E Programs*.

Directives

- BLANK and NOBLANK

  For the TNS COBOL compiler, BLANK is the default. For the native COBOL compiler, NOBLANK is the default.

- CONSULT

  For the native COBOL compiler, each *object-name* in a CONSULT directive must designate a native object file (otherwise the native COBOL compiler reports an error). If a native COBOL program references the object in a CALL or ENTER statement, the object must have been compiled with symbols.

- LARGEDATA

  For the TNS COBOL compiler, the LARGEDATA directive determines whether individual data items are located in the user data space or the user extended space. The default value for the LARGEDATA directive's parameter is 256.

  The native COBOL compiler does not distinguish between user space and extended space. The LARGEDATA directive is ignored, and a warning is reported.

- LIBRARY

  The native COBOL compiler issues a warning for the LIBRARY directive.

  In native COBOL, a user library is implemented as a dynamic-link library (DLL). Instead of the LIBRARY directive, specify the name of your DLL as explained in the *COBOL Manual for TNS/E Programs*.

- RUNNAMED

  The RUNNAMED directive works in the native COBOL compiler only if you specify the new RUNNABLE directive (see RUNNABLE directive on page 4-11).

- SAVEABEND and NOSAVEABEND

  The SAVEABEND and NOSAVEABEND directives work in the native COBOL only if you specify the new RUNNABLE directive (see RUNNABLE directive on page 4-11).

- SEARCH

  For the native COBOL compiler, each *object-name* in a SEARCH directive must designate a native object file (otherwise the native COBOL compiler reports an error). If a native COBOL program references the object in a CALL or ENTER statement, the object must have been compiled with symbols.

- SUBTYPE

  The SUBTYPE directive works in the native COBOL compiler only if you specify the new RUNNABLE directive (see RUNNABLE directive on page 4-11).

ENTER Statement

The *language* parameter of the ENTER statement is unnecessary, because the native COBOL compiler can determine the language in which the program is written. If you do specify *language*, it must be C or TAL. If you specify TAL, the native COBOL compiler requires a pTAL program (it does not accept TAL programs). If you specify FORTRAN or Pascal, the native COBOL compiler issues an error message. (You must convert to native languages any FORTRAN, Pascal, or TAL programs that your COBOL program calls, as Step 4 of Converting COBOL Programs on page 4-2 explains.) Although a native COBOL program can use the ENTER statement to call C++ or pTAL programs, *language* cannot have the value C++ or pTAL.

Floating-Point Arithmetic

The TNS COBOL and native COBOL compilers can produce slightly different results for floating-point arithmetic, partly because of different floating-point formats used by the TNS and native COBOL compilers. TNS COBOL uses the HP proprietary TNS format, while TNS/E native COBOL uses the IEEE format. This format difference can cause problems for exponentiation with a negative or fractional exponent (such as 10**-3 or 2**0.3). To avoid these problems, include the ROUNDED phrase in statements that perform floating-point arithmetic so that both compilers produce the same results. For more information on the ROUNDED phrase, see the *COBOL Manual for TNS/E Programs*.

Working-Storage Limits

TNS COBOL enforces a limit of slightly less than 64 KB on data defined in the WORKING-STORAGE section and a limit of 127.5 KB on data defined in the EXTENDED-STORAGE section. Native COBOL does not enforce a specific limit.

Checkpointing

The TNS COBOL compiler automatically checkpoints data items that are stored directly on the stack in two or fewer bytes (that is, if they are level-01 or level-77 items, in the Working-Storage Section, and fewer than three characters long). The native COBOL compiler checkpoints only those data items that one or more CHECKPOINT statements specify explicitly. For more information on the CHECKPOINT statement, see the *COBOL Manual for TNS/E Programs*.

Use of RENAMES Clause

As documented in both the *COBOL Manual for TNS and TNS/R Programs* and *COBOL Manual for TNS/E Programs*, the RENAMES clause must not rename a level-01 data item. However, if a TNS COBOL program uses a RENAMES clause in this way, the error is not diagnosed and, in fact, the program executes normally. The TNS/E COBOL compiler does detect this error. Therefore, if a TNS COBOL program uses a RENAMES clause to rename a level-01 data item, you must change the source code as follows before recompiling it with the TNS/E native COBOL compiler.

If the level-01 data item is elementary, change the RENAMES clause to a REDEFINES clause. For example, change:

```
01 CARD-COUNTER  PIC 9(6).
66 ITEM-COUNT RENAMES CARD-COUNTER.
```

To:

```
01 CARD-COUNTER  PIC 9(6).
66 ITEM-COUNT    PIC 9(6) REDEFINES CARD-COUNTER.
```

If the level-01 data item is a structure, rename the first subordinate data item through the last subordinate data item. For example, change:

```
01 CARD-REC.
   05 REFERENCE-NUMBER  PIC 9(6).
   05 CARD-CODES.
      10 STORE-CODE     PIC 9.
      10 STATE-CODE     PIC 9(4).
   05 ACCOUNT-NUMBER    PIC 9(6).
   05 CHECK-DIGIT       PIC 9.
   66 CARD-DATA RENAMES CARD-REC.
```

To:

```
01 CARD-REC.
   05 REFERENCE-NUMBER  PIC 9(6).
   05 CARD-CODES.
      10 STORE-CODE     PIC 9.
      10 STATE-CODE     PIC 9(4).
   05 ACCOUNT-NUMBER    PIC 9(6).
   05 CHECK-DIGIT       PIC 9.

   66 CARD-DATA RENAMES REFERENCE-NUMBER THRU CHECK-DIGIT
```

Use of PARAM SYMBOL-BLOCKS Command

The TNS/E COBOL compiler requires more space to describe a symbol that does the TNS/R COBOL compiler; thus, the symbol dictionary requires more space on a TNS/E system. If you use a PARAM SYMBOL-BLOCKS command to specify the amount of space to allocate for the symbol dictionary in a TNS/R compilation, you might need to increase the amount specified when compiling with the TNS/E compiler.

The default value for PARAM SYMBOL-BLOCKS has been increased for the TNS/E COBOL compiler, so if you are not specifying it for a TNS/R compilation, you probably will not need to specify it for a TNS/E compilation. See the *COBOL Manual for TNS and TNS/R Programs* and the *COBOL Manual for TNS/E Programs* for details on the amount of space allocated by the PARAM SYMBOL-BLOCKS command.

Removal Optional

The native COBOL compiler ignores the following items, so you can (but need not) remove them from your source program. The native COBOL compiler issues a warning when it finds these items, except as noted.

- CODE and NOCODE directives

  The native COBOL compiler does not produce an octal code listing. If you need to display an object file, use the `enoft` utility. (See the *eNOFT Manual*.)

- COMPACT and NOCOMPACT directives

  These directives determine whether BINSERV attempts to compact the code space of the target file. The native COBOL compiler does not use BINSERV.

- CROSSREF and NOCROSSREF

  The native COBOL compiler does not produce a cross-reference listing. If you need one, use the `enoft` utility with the XREFPROC flag. (See the *eNOFT Manual*.)

- ENV COMMON directive

  Native COBOL programs always run in the CRE. The native COBOL compiler does not issue a warning if you use this directive.

- FLOAT and NOFLOAT directives

  The native COBOL compiler determines whether to use floating-point arithmetic for certain complex expressions. (The FLOAT option is always in effect.)

- LESS-CODE directive

  Native COBOL does not support the option to use a system call to initialize the Extended-Storage Section.

- ENV LIBRARY

Instead of using the ENV LIBRARY directive to build a TNS user library, use the UL or SHARED directive to build a PIC object file, which can be used by the linker to create a DLL (see the *COBOL Manual for TNS/E Programs*).

- EXTENDED-STORAGE SECTION header

  Native COBOL does not need an Extended-Storage Section. The native COBOL compiler handles data items that are described in the Extended-Storage Section as if they were described in the Working-Storage Section. The native COBOL compiler does not issue a warning if you use the EXTENDED-STORAGE SECTION header.

- HEAP directive
- HIGHPIN directive

  Native COBOL programs always run at a high PIN.

- HIGHREQUESTERS directive

  Native COBOL programs can always run as servers that communicate with requesters running at high PINs.

- ICODE and NOICODE

  The native COBOL compiler ignores these directives, warning you that it has done so. Use the INNERLIST and NOINNERLIST directives instead (see INNERLIST and NOINNERLIST directives on page 4-11).

- LMAP and NOLMAP directives

  The LMAP and NOLMAP directives determine which load maps the compiler obtains from BINSERV. The native COBOL compiler does not use BINSERV.

- NOCONSULT
- NOSEARCH
- NOSQL

  This is the default for native COBOL programs.

- SQLMEM

  The concept of extended memory does not exist in native mode.

- TRAP2 and NOTRAP2

  Native COBOL85 programs have traps set by default.

- TRAP2-74 and NOTRAP2-74

  Native COBOL programs cannot call COBOL 74 programs.

New Features

You can (but need not) add these new features to your COBOL source program before you compile it with the native COBOL compiler. For more information on these features, see the *COBOL Manual for TNS/E Programs*.

- DIAGNOSE-85 directive

  The DIAGNOSE-85 directive directs the native COBOL compiler to issue warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the TNS COBOL compiler.

- FMAP

  The FMAP directive directs the native COBOL compiler to produce a source file map, which shows the fully qualified name and timestamp of the IN file and each file specified by a SOURCE directive or COPY statement.

- INNERLIST and NOINNERLIST directives

  The INNERLIST and NOINNERLIST directives determine whether the compiler lists the mnemonic version of each source statement immediately after that source statement.

- RUNNABLE directive

  The RUNNABLE directive causes the native COBOL compiler to use the `eld` utility to produce an executable object file if there were no compilation errors, but that action is not the default. If you run the native COBOL compiler without the RUNNABLE directive and no compilation errors occur, you can produce an executable object file by running the `eld` utility separately. If you do, you must specify the COBOL DLL (ZCOBDLL) and CRE DLL (ZCREDLL). ZCOBDLL and ZCREDLL reside in the active subvolume $SYSTEM.SYS*nn*. For instructions on determining the active subvolume, see the *eld Manual*.

  You must compile your native program with the RUNNABLE directive if you use the SAVEABEND, NOSAVEABEND, or SUBTYPE directive.

  For information on running the `eld` utility, see the *eld Manual*.

5 Converting TAL to pTAL

HP TAL is a higher-level, block-structured language used to write system software and transaction-oriented applications for NonStop systems.

pTAL is a dialect of TAL with these differences:

- pTAL does not depend on architecture-specific characteristics of NonStop processors.
- pTAL has constructs that replace architecture-specific TAL constructs and that take advantage of TNS/E processors.
- pTAL also enforces stricter rules on using certain TAL constructs and operations.

The compilers to which this section applies are:

Compiler	T Number
TNS/E pTAL	T0561
TNS TAL	T9250

This section discusses:

Using the pTAL Compiler

The `EPTAL` command runs the TNS/E pTAL compiler. The TNS/E pTAL compiler generates Itanium code. Object code produced by the TNS/E pTAL compiler is position-independent code (PIC), which can be used to create DLLs.

TNS/E pTAL provides a PC-based cross compiler, which is supported on the Windows platform. You can run the cross compiler at the Microsoft Windows command prompt (`eptal` command) or through ETK (`eptal.exe` command).

Required Changes

Some of the changes required to convert from TAL to pTAL include:

- Replace INT and INT(32) declarations holding addresses with new address types WADDR, BADDR, and EXTADDR.
- Test condition codes, such as with IF $OVERFLOW, immediately following the statement returning the condition.
- Add explicit returns at the bottom of functions.
- Declare procedure pointers using new pTAL syntax.
- Specify a RETURNSCC attribute in procedures that return condition codes.

- Remove CODE, STACK, and STORE statements.
- Rewrite code that uses G, S, or L-relative addresses.

pTAL does not support embedded SQL statements. To convert a TAL program with embedded SQL to pTAL, write C functions that contain the embedded SQL statements and call the C functions from pTAL.

Many TAL programs require few changes to create valid pTAL programs. However, TAL programs that make use of low-level TNS architecture features might require significant changes.

Use the pTAL compiler's syntax checking option to produce a detailed list of TAL constructs that must be changed to convert TAL code to pTAL code. In Example 5-1, the variable `I` must be declared as a WADDR.

Example 5-1. pTAL Compiler Listing With Syntax Checking

```
                      PROC myproc;
                      BEGIN
                      INT I;
*** Error:                ^
--> I is declared to have type INT(16), but is used later as WADDR
[error 01023].
                      INT .p1;
                      I := @p1;
```

In some cases, you can maintain one set of source code for both TAL and pTAL. See Maintaining Common Source Code for TNS and TNS/E Native Compilers on page 2-3 for details.

This manual does not describe the language-specific tasks required to convert from TAL to pTAL. For detailed TAL-to-pTAL conversion information, see the appropriate manual in the pTAL documentation set:

Manual	Description
pTAL Guidelines for TAL Programmers	Describes how to write TAL code that can be converted to pTAL with minimal changes. Lists the major differences between TAL and pTAL.
pTAL Conversion Guide	Describes the differences between TAL and pTAL. Explains in detail how to covert TAL programs to pTAL.
pTAL Reference Manual	Describes the syntax of the pTAL language. Explains how to run the pTAL compiler.

As an alternative to reading the lengthy *pTAL Reference Manual* and *pTAL Conversion Guide*, you can convert the majority of TAL programs by following these steps:

1. Read *pTAL Guidelines for TAL Programmers* to familiarize yourself with the difference between TAL and pTAL.
2. Process your source files with the pTAL compiler syntax checking option set. The syntax is:

   ```
   EPTAL / IN source-filename, OUT listing-filename / ;SYNTAX
   ```

   See the *pTAL Reference Manual* for details on running the pTAL compiler.
3. Examine the pTAL compiler output listings to determine the changes that you must make to the source files. Look up the items that require changes in the *pTAL Reference Manual* and the *pTAL Conversion Guide*.

6 Converting a TNS User Library

This section explains how to convert TNS user libraries to TNS/E native user libraries.

A user library is an object file that the operating system links to a program file at run time. C, C++, COBOL, TAL, and pTAL programs can have a user library. There are two types of user libraries on H-series systems: TNS user libraries and TNS/E native user libraries. A TNS user library is available to TNS processes in the Guardian environment. A TNS/E native user library is available to TNS/E native processes in the Guardian and OSS environments.

A TNS/E native user library is a DLL that has a special relationship to a program. A TNS/E native user library is architecturally identical to, and in most respects is treated the same as, any other DLL. A program can have only one TNS/E native user library, although it can load multiple ordinary DLLs.

This section discusses:

User Library Differences

The following table summarizes the differences between TNS and TNS/E native user libraries:

TNS User Libraries	TNS/E Native User Libraries
Either or both the program file and user library file can be accelerated (processed by AXCEL, OCA, or both).	Both the program file and user library file must be compiled with TNS/E native compilers.
Can contain embedded SQL statements.	Cannot contain embedded SQL statements. Move functions with embedded SQL statements to user code before converting a TNS user library to a TNS/E native user library.
Can call a limited subset of the C run-time library.	Can call any function in the C run-time library.
A COBOL user library specified at compilation time can be overridden at run time.	A COBOL user library specified at compilation time cannot be overridden at run time.

Building a User Library

Building a TNS/E native user library is similar to building a TNS user library. To build a TNS/E native user library:

1. Remove pragmas and directives:
   - In C and C++, remove ENV LIBRARY pragmas from source code files and the compiler RUN command line.
   - In COBOL and pTAL, remove ENV LIBRARY directives from source code files.
2. Add pragmas, directives, and flags:
   - In C and C++:
     - In the Guardian environment, add a CALL_SHARED pragma to the compiler RUN command line.
     - In the OSS or PC environment, add the `-Wcall_shared` flag to the `c89` or `c99` command.
   - In COBOL:
     - In the Guardian environment, add a SHARED directive to the ECOBOL command line or the source code.
     - In the OSS or PC environment, add a `-Wshared` flag to the `c89` or `c99` command, or add a SHARED directive to the source code.
   - In pTAL:
     - In the Guardian environment, add a CALL_SHARED directive to the EPTAL command line or the source code.
     - On the PC, add a `-Wcall_shared` flag to the `c89` or `c99` command, or add a CALL_SHARED directive to the source code.
3. Compile the source files as relinkable files, not as executable files. Do not specify the COBOL RUNNABLE directive or the C/C++ RUNNABLE pragma.
4. Link the object files using `eld` with the -`shared` and `-ul` and flags. The -`shared` flag directs the linker to build a DLL, and the `-ul` flag causes `eld` to link the object files into a native user library. If the user library calls functions in another DLL, you must specify the DLL when linking. (This is true for any DLL, not just user libraries.) Do not link the CRTLMAIN file (in the Guardian environment) or the `crtlmain.o` file (in the OSS environment and on the PC) to a user library.

The preceding steps create a DLL in two steps: a compilation step and a linker step. Alternatively, you can create a DLL in a single step by specifying appropriate compiler directives to invoke the linker automatically after compilation is complete.

See the following manuals for more details on building TNS/E native user libraries and DLLs:

- Compiler manuals
- *eld Manual*
- *DLL Programmer's Guide for TNS/E Systems*

Specifying a User Library

Only one user library can be associated with a program file at any time. Associating a user library with a program file is nearly identical for TNS user libraries and TNS/E native user libraries. This table describes how to specify a user library for TNS processes and TNS/E native processes:

When	TNS Processes	TNS/E Native Processes
Compile time	TAL programs: LIBRARY directive at compile time.	LIBRARY directive not available COBOL programs: Specify the TNS/E native user library in a CONSULT directive
Bind and link time	Binder SET LIBRARY command C/C++ programs: TNS `c89` utility `-Wrunlib=` *`pathname`* flag (for Guardian programs compiled using `c89`) C/C++ programs: TNS/E `c99` utility `-Wrunlib=` *`pathname`* flag (for Guardian programs compiled using `c99`)	`eld` utility `-libname` flag `eld` utility `-set libname` and `-change libname` flags TNS/E native `c89` utility `-Weld= "-libname` *`library`*`"` flag TNS/E native `c99` utility `-Weld= "-libname` *`library`*`"` flag
Run time	TACL command interpreter RUN command LIB option OSS `run gtacl` command `-lib` flag	TACL command interpreter RUN command LIB option OSS `run gtacl` command `-lib` flag COBOL programs: Specify the same TNS/E native user library that you specified at compile time, using the LIB option of the RUN command (in the Guardian environment) or the `-lib` flag of the `ecobol` utility (in the OSS or PC environment)

Unlike TAL, you cannot specify a user library in pTAL source code with the LIBRARY directive. Remove LIBRARY directives from pTAL programs.

User libraries specified at run time override those specified at link time.

For more information on TNS user libraries, see the *Binder Manual*. For more information on TNS/E native user libraries and DLLs, see the *eld Manual* and the *DLL Programmer's Guide for TNS/E Systems*.

7 Converting Data Definition Language (DDL)

This section describes how to convert Data Definition Language (DDL) host language source files that:

- Are shared between TNS and TNS/E native programs written in C, C++, TAL, or pTAL

  and

- Were generated by a pre-D40 product version of the DDL compiler

If these criteria apply to your DDL host language source files, follow the instructions in this section to generate new host language source files and recompile your programs with the new files. If your DDL host language source files are not shared between TNS and native programs or were generated by a D4x, G-series, or H-series DDL compiler, or your applications are written in COBOL, no actions are required.

This section discusses:

Background Information

DDL defines data objects and translates object definitions into source code for programming languages and other products. Data objects can include parameters, structures, messages, database entries, and disk records. Data objects in host-language source code generated by DDL have the same physical layout, regardless of host language.

The native compilers align data for optimal performance on NonStop systems by default. This default alignment is the same on TNS/R and TNS/E systems. Except for the TNS COBOL and TNS/E native COBOL compilers, this default alignment is different from and incompatible with the default data alignment generated by the TNS compilers.

The TNS COBOL and TNS/E native COBOL compilers generate code with the same data alignment. You need not change any data alignment directives before converting a TNS COBOL program to native COBOL. You need not regenerate DDL source files that are shared only by TNS COBOL and native COBOL programs.

Because of the data alignment incompatibility for languages other than COBOL, the D4x, G-series, and H-series DDL compilers have been enhanced to generate host-language source code that produces the same data alignment, whether the TNS

compilers or native compilers are used. To ensure the same data alignment, the D4x, G-series, and H-series DDL compilers emit `FIELDALIGN SHARED2` pragmas for C and FIELDALIGN SHARED2 directives for TAL and pTAL.

Except for COBOL, host-language source files used by TNS/E native programs and shared with TNS programs must be generated using the D4x, G-series, or H-series DDL compiler. Host-language source files supplied by HP for H-series systems have already been generated by the H-series DDL compiler.

Host-language source files used exclusively by TNS/E native programs or TNS programs do not require changes. Only shared host-language source files must be generated using the H-series DDL compiler.

If you deliver host-language source files to your customers or use a significant number of host-language source files, plan and test your files carefully to ensure that you generate new source files for all files shared by TNS and native programs.

The remainder of this section applies only to C, C++, TAL, and pTAL programs; it does not apply to COBOL programs.

Generating New Host-Language Source Code Files

If a TNS program uses host-language source files that HP supplies, you do not need to generate new host-language source files. HP products that supply DDL files have been generated by H-series DDL for you. Proceed to Compiling With New Host-Language Source Code Files on page 7-3.

If a TNS program uses host-language source files that you created, you must generate new files with the H-series DDL compiler. The DDL compiler requires a DDL source schema file or a DDL dictionary to generate host-language source files. If you think your DDL dictionary differs from your DDL source schema files, generate a new DDL dictionary. Before generating a new DDL dictionary:

1. Make a complete backup of the existing DDL dictionary.
2. Generate new source schema files from the existing DDL dictionary.
3. Compare the existing source schema files with the new source schema files and resolve any differences by making the appropriate changes in the source schema files.
4. Generate the new DDL dictionary using the corrected source schema files.

To identify those DDL files that you have generated with the H-series DDL compiler, you might want to add a comment to the DDL source schema files similar to that used in HP source schema files:

```
DDL output recompiled with H-series DDL to allow native and non-
native applications to share structures.
```

See the *Data Definition Language (DDL) Reference Manual* for complete DDL usage information.

△ **Caution.** Do not attempt to regenerate a DDL dictionary installed by the Pathmaker product from DDL source code. Pathmaker application design information will be lost.

Compiling With New Host-Language Source Code Files

HP products that supply DDL host-language files have been generated with H-series DDL.

You can recompile programs using the TNS/E pTAL, TNS/E native C, TNS/E native C++, and TAL compilers. You must recompile programs using a D-series, G-series, or H-series version of the TNS C compiler that supports the `FIELDALIGN` pragmas emitted by DDL.

This table shows the product version updates (PVUs) that must be used with D2x and D3x TNS C compiler versions. For D4x, G-series, and H-series compilers, the base product version supports the new pragmas and no PVU is required.

For this TNS C compiler product version:	Use an PVU version of at least:
T9255D20	T9255ABN
T9255D30	T9255ABM
T8377D30	T8377AAA

Compile and (if necessary) accelerate your programs, specifying the new host-language source code files. No other actions are required.

For details on how the native compilers align data, see the *C/C++ Programmer's Guide* and the *pTAL Reference Manual*.

8

Converting Programs That Run in the Common Run-Time Environment

The Common Run-Time Environment (CRE) coordinates many run-time tasks on behalf of the language-specific run-time libraries to provide a common environment for all routines in a program, regardless of their languages. Most programs use CRE services implicitly by calling language specific run-time libraries, such as the C run-time library, which in turn call the CRE. You do not need to change TNS programs that use CRE services implicitly to convert them to native mode.

This section describes the changes required to convert applications that make explicit use of CRE services; that is, programs that make explicit calls to the CRE_, RTL_, and CLU_ functions. TAL main routines that run in the CRE make explicit use of CRE services and require changes. (Most programs with a TAL 'main' routine do not run in the CRE.) This section discusses:

Converting pTAL Programs to Run in the CRE

A program with a TAL main routine that runs in the CRE calls the TAL_CRE_INITIALIZER_ procedure to initialize and establish the CRE. There is no equivalent procedure for pTAL. Therefore, a pTAL procedure cannot be the main procedure in a program that runs in the CRE. To convert a program that runs in the CRE from TAL to pTAL:

- Delete the call to TAL_CRE_INITIALIZER_.
- Delete the MAIN keyword from the main procedure.
- Write a simple C main function that calls the pTAL procedure that previously was declared as main.

The C main function automatically initializes the CRE. Because the program now has a C main function, you must link to the SRLs and CRTLMAIN module required by native C programs. See the *CRE Programmer's Guide* and *C/C++ Programmer's Guide* for details.

Unlike the TAL compiler, which requires an ENV COMMON or ENV NEUTRAL directive for programs that run in the CRE, the pTAL compiler does not require ENV directives. The pTAL compiler issues a warning for any ENV directives that it finds. Remove ENV COMMON and ENV NEUTRAL directives from pTAL programs.

Specifying Header Files

Separate TAL and pTAL header files describe the external declarations of CRE functions. Change external declaration file references in SOURCE directives to use the pTAL declaration files, as follows:

For function names beginning with	Use this TAL declaration file	Use this pTAL declaration file
CRE_	CREDECS	CRERDECS
RTL_	RTLDECS	RTLRDECS
CLU_	CLUDECS	CLURDECS

Replacing Obsolete CRE Functions

The obsolete CRE functions include:

In many cases, the obsolete functions have nearly identical replacements or, for COBOL, equivalent intrinsic functions. The replacements match the functions in the native C run-time library. To call these functions from pTAL, you must write your own pTAL function declarations. See the C header files and the *Guardian Native C Library Calls Reference Manual* for a description of each function's definition and behavior. See the *pTAL Reference Manual* for information on writing pTAL declarations for C functions. See the *COBOL Manual for TNS/E Programs* for information on COBOL intrinsic functions.

CRE functions that can generate arithmetic traps in TNS processes do not generate arithmetic traps in native processes. In the native environment, the trap handling facility has been replaced with a signal handling facility. Arithmetic overflow is detected dynamically, and a default signal handler is provided. Programs must now evaluate `errno` or the parameters to a function call. For native programs, the CRE provides a default signal handler instead of a default trap handler.

Standard Math Functions

These CRE standard math functions cannot be called by native programs. Table 8-1 lists the obsolete functions and suggests replacement functions. COBOL programs can call either the COBOL intrinsic functions or the other suggested replacement functions.

Table 8-1. Obsolete Standard Math Functions (page 1 of 3)

Obsolete Function	Suggested Replacement	
	COBOL Only	**All Languages (including COBOL)**
CRE_Arccos_Real32_ RTL_Arccos_Real32_ CRE_Arccos_Real64_ RTL_Arccos_Real64_	ACOS	`acos()`
CRE_Arcsin_Real32_ RTL_Arcsin_Real32_ CRE_Arcsin_Real64_ RTL_Arcsin_Real64_	ASIN	`asin()`
RTL_Arctan_Real32_ RTL_Arctan_Real64_	ATAN	`atan()`
CRE_Arctan2_Real32_ RTL_Arctan2_Real32_ CRE_Arctan2_Real64_ RTL_Arctan2_Real64_		`atan2()`
RTL_Cos_Real32_ RTL_Cos_Real64_	COS	`cos()`
CRE_Cosh_Real32_ RTL_Cosh_Real32_ CRE_Cosh_Real64_ RTL_Cosh_Real64_		`cosh()`
CRE_Exp_Real32_ RTL_Exp_Real32_ CRE_Exp_Real64_ RTL_Exp_Real64_		`exp()`
CRE_Ln_Real32_ RTL_Ln_Real32_ CRE_Ln_Real64_ RTL_Ln_Real64_	LOG	`log()`
CRE_Log10_Real32_ RTL_Log10_Real32_ CRE_Log10_Real64_ RTL_Log10_Real64_	LOG10	`log10()`
RTL_Lower_Real64_		`floor()`

Table 8-1. Obsolete Standard Math Functions (page 2 of 3)

Obsolete Function	Suggested Replacement	
	COBOL Only	**All Languages (including COBOL)**
CRE_Mod_Int16_ RTL_Mod_Int16_ CRE_Mod_Int32_ RTL_Mod_Int32_ CRE_Mod_Int64_ RTL_Mod_Int64_ RTL_Mod_Real32_ RTL_Mod_Real64_	MOD	`modf()`
RTL_Normalize_Real64_		`frexp()`
RTL_Odd_Int32_		Write your own function that determines whether a value is even or odd.
RTL_Positive_Diff_Int16_ RTL_Positive_Diff_Int32_ RTL_Positive_Diff_Int64_ RTL_Positive_Diff_Real32_ RTL_Positive_Diff_Real64_		Write your own function that returns the arithmetic difference between two numbers.
CRE_Power_Int16_to_Int16_ RTL_Power_Int16_to_Int16_ CRE_Power_Int32_to_Int16_ RTL_Power_Int32_to_Int16_ CRE_Power_Int64_to_Int16_ RTL_Power_Int64_to_Int16_ CRE_Power_Real32_to_Int16_ RTL_Power_Real32_to_Int16_ CRE_Power_Real32_to_Real32_ RTL_Power_Real32_to_Real32_ CRE_Power_Real64_to_Int16_ RTL_Power_Real64_to_Int16_ CRE_Power_Real64_to_Real64_ RTL_Power_Real64_to_Real64_		`pow()`
CRE_Power2_Real64_ RTL_Power2_Real64_		`ldexp()`
CRE_Random_Next_	RANDOM	`rand()`
CRE_Random_Set_		`srand()`
RTL_Round_Real32_ RTL_Round_Real64_		Write your own function that returns the nearest whole number.

Table 8-1. Obsolete Standard Math Functions (page 3 of 3)

Obsolete Function	Suggested Replacement	
	COBOL Only	**All Languages (including COBOL)**
RTL_Sign_Int16_ RTL_Sign_Int32_ RTL_Sign_Int64_ RTL_Sign_Real32_ RTL_Sign_Real64_		Write your own function that returns its first parameter with the sign set according to its second parameter.
RTL_Sin_Real32_ RTL_Sin_Real64_	SIN	`sin()`
CRE_Sinh_Real32_ RTL_Sinh_Real32_ CRE_Sinh_Real64_ RTL_Sinh_Real64_		`sinh()`
RTL_Split_Real64_	`MOD`	`modf()`
CRE_Sqrt_Real32_ RTL_Sqrt_Real32_ CRE_Sqrt_Real64_ RTL_Sqrt_Real64_	`SQRT`	`sqrt()`
RTL_Tan_Real32_ RTL_Tan_Real64_	`TAN`	`tan()`
CRE_Tanh_Real32_ RTL_Tanh_Real32_ CRE_Tanh_Real64_ RTL_Tanh_Real64_		`tanh()`
RTL_Truncate_Real32_ RTL_Truncate_Real64_		Write your own function that returns the nonfractional part of a number.
RTL_Upper_Real64_		`ceil()`

String Functions

These CRE string functions cannot be called by native programs. Table 8-2 lists the obsolete functions and suggests replacement functions.

Table 8-2. Obsolete String Functions (page 1 of 3)

Obsolete Function	Suggested Replacement
RTL_Atof_	`atof()` in the native C run-time library
RTL_Atoi_	`atoi()` in the native C run-time library
RTL_Atol_	`atol()` in the native C run-time library
RTL_Stcarg_ RTL_StcargX_	`strcspn()` with a string of bytes representing the characters for which to search

Table 8-2. Obsolete String Functions (page 2 of 3)

Obsolete Function	Suggested Replacement
RTL_Stccpy_ RTL_StccpyX_	`strcpy()`
RTL_Stcd_I_ RTL_Stcd_IX_	`strtol()` in the native C run-time library
RTL_Stcd_L_ RTL_Stcd_LX_	`strtol()` in the native C run-time library
RTL_Stch_I_ RTL_Stch_IX_	`strtol()` in the native C run-time library
RTL_Stci_D_ RTL_Stci_DX_	`sprintf()` with a `%d` conversion specifier. `sprintf()` is in the native C run-time library.
RTL_Stcpm_ RTL_StcpmX_	`strstr()`. Additional code might be required to support pattern matching.
RTL_Stcpma_ RTL_StcpmaX_	`strstr()`. Additional code might be required to support pattern matching.
RTL_Stcu_D_ RTL_Stcu_DX_	`sprintf()` with a `%u` conversion specifier. `sprintf()` is in the native C run-time library.
RTL_Stpblk_ RTL_StpblkX_	`strpbrk()` with a string of bytes representing the nonspace characters
RTL_Stpsym_ RTL_StpsymX_	`strspn()` with a string containing alphanumeric characters. Pass the return value of `strspn()` to `strncpy()` to copy the correct string length.
RTL_Stptok_ RTL_StptokX_	Series of calls to `strtok()` to find a token and `strcat()` to append a token to another string
RTL_Strcat_ RTL_StrcatX_	`strcat()`
RTL_Strchr_ RTL_StrchrX_	`strchr()`
RTL_Strcmp_ RTL_StrcmpX_	`strcmp()`
RTL_Strcpy_ RTL_StrcpyX_	`strcpy()`
RTL_Strcspn_ RTL_StrcspnX_	`strcspn()`
RTL_Strlen_ RTL_StrlenX_	`strlen()`
RTL_Strncat_ RTL_StrncatX_	`strncat()`
RTL_Strncmp_ RTL_StrncmpX_	`strncmp()`

Table 8-2. Obsolete String Functions (page 3 of 3)

Obsolete Function	Suggested Replacement
RTL_Strncpy_ RTL_StrncpyX_	`strncpy()`
RTL_Strpbrk_ RTL_StrpbrkX_	`strpbrk()`
RTL_Strrchr_ RTL_StrrchrX_	`strstr()`
RTL_Strspn_ RTL_StrspnX_	`strspn()`
RTL_Strstr_ RTL_StrstrX_	`strstr()`
CRE_Strtod_ CRE_StrtodX_	`strtod()` in the native C run-time library
CRE_Strtol_ CRE_StrtolX_	`strtol()` in the native C run-time library
CRE_Strtoul_ CRE_StrtoulX_	`strtoul()` in the native C run-time library
RTL_Substring_Search_	`strstr()`

Memory Block Functions

These CRE memory block functions cannot be called by native programs. Table 8-3 lists the obsolete functions and suggests replacement functions:

Table 8-3. Obsolete Memory Block Functions

Obsolete Function	Suggested Replacement
RTL_Memory_Compare_ RTL_Memory_CompareX_	`memcmp()`
RTL_Memory_Copy_ RTL_Memory_CopyX_	`memcpy()`
RTL_Memory_Findchar_ RTL_Memory_FindcharX_	`memchr()`
RTL_Memory_Move_ RTL_Memory_MoveX_	`memmove()`
RTL_Memory_Repeat_ RTL_Memory_RepeatX_	Series of calls to `memcpy()` to copy the block of memory the required number of times
RTL_Memory_Set_ RTL_Memory_SetX_	`memset()`
RTL_Memory_Swap_ RTL_Memory_SwapX_	Series of calls to `memcpy()` to swap the blocks of memory

Exception-Handling Functions

The CRE_Stacktrace_ function has separate TAL and pTAL declarations to support the TNS and native stack architectures. See the *CRE Programmer's Guide* for details.

Sixty-Four-Bit Logical Operation Functions

You cannot call these CRE functions from native programs:

- RTL_Shift_Left_Int64_
- RTL_Shift_Right_Int64_
- RTL_Complement_Int64_
- RTL_And_Int64_
- RTL_Or_Int64_
- RTL_Xor_Int64_
- RTL_Remainder_Int64_

There are no equivalent replacements. You must write your own code to replace these functions.

Decimal-Conversion Functions

You cannot call these CRE functions from native programs:

- RTL_Decimal_to_Int16_
- RTL_Decimal_to_Int32_
- RTL_Decimal_to_Int64_
- RTL_Int16_to_Decimal_
- RTL_Int32_to_Decimal_
- RTL_Int64_to_Decimal_

There are no equivalent replacements. You must write your own code to replace these functions.

9

Converting Programs That Share Data

This section describes the conversion tasks related to:

- Sharing Data Between TNS and TNS/E Native Programs
- Sharing Data Between pTAL Programs and Native C or C++ Programs on page 9-2

Shared data objects can include parameters, structures, messages, database entries, and disk records.

Note. If DDL is used to define shared data objects and generate source code, see Section 7, Converting Data Definition Language (DDL).

Sharing Data Between TNS and TNS/E Native Programs

The native compilers align data for optimal performance on the TNS/E platform by default. This default alignment is different and incompatible with the default data alignment generated by the TNS compilers.

Because of this incompatibility, data objects shared between native programs and TNS programs require `FIELDALIGN SHARED2` pragmas for C and C++ and FIELDALIGN SHARED2 directives for TAL and pTAL. These pragmas and directives cause both TNS and native compilers to generate code with the same alignment.

FIELDALIGN SHARED2 produces code that is not optimally aligned for the TNS/E platform. Therefore, specify FIELDALIGN SHARED2 only for shared data objects. Otherwise, use the default native compiler alignment.

See the *C/C++ Programmer's Guide* or the *pTAL Reference Manual* for more information on controlling data alignment.

D-series, G-series, and H-series versions of the TAL compiler support FIELDALIGN directives. D4x, G-series, and H-series versions of the TNS C compiler support `FIELDALIGN` pragmas. Pre-D40 product versions of the TNS C compiler require these PVUs to support `FIELDALIGN` pragmas:

For this TNS C compiler product version	Use a PVU version of at least
T9255D20	T9255ABN
T9255D30	T9255ABM
T8377D30	T8377AAA

Sharing Data Between pTAL Programs and Native C or C++ Programs

The native C and C++ compilers align data for the optimal performance of C and C++ programs on the TNS/E platform by default (pragma `FIELDALIGN AUTO`). The pTAL compiler aligns data for the optimal performance of pTAL programs on the TNS/E platform by default (directive FIELDALIGN AUTO). Although the default setting of the pTAL, C, and C++ compilers is FIELDALIGN AUTO, the compilers generate different and incompatible alignments for FIELDALIGN AUTO.

Because of this data alignment incompatibility, data objects shared between pTAL programs and native C or C++ programs require `FIELDALIGN SHARED8` pragmas for C and C++ and FIELDALIGN SHARED8 directives for pTAL. This pragma and directive cause the native compilers to generate code with the same alignment.

Instead of using FIELDALIGN SHARED8, you can also use FIELDALIGN PLATFORM for programs that do not contain pTAL WADDR, BADDR, or EXTADDR address types in structures.

If the data objects are also shared with TNS programs, use the pragmas and directives described under Sharing Data Between TNS and TNS/E Native Programs on page 9-1 instead.

10

Converting Programs With Guardian API Calls

This section describes the changes you must make to Guardian application program interface (API) calls in programs you convert to TNS/E native mode.This section discusses:

Replacing Obsolete Procedures

These Guardian procedures cannot be called by native programs:

The following subsections describe how to replace these obsolete functions.

ADDRTOPROCNAME

The ADDRTOPROCNAME procedure takes a P register value and stack marker ENV value and returns the associated symbolic procedure name and various optional items that describe the procedure in detail.

Native processes cannot call the ADDRTOPROCNAME procedure because native processes do not have P or ENV registers. Remove calls to ADDRTOPROCNAME from your program. No direct replacement is available, but a comparable service is provided for accelerated and native programs by the HIST_INIT_, HIST_FORMAT_, and HIST_GETPRIOR_ procedures.

These three procedures display process state, including register contents and procedure activation history or stack traces. The HIST_INIT_ procedure called with the HO_Init_Address option provides results comparable to the ADDRTOPROCNAME procedure. See the *Guardian Procedure Calls Reference Manual* for details.

ARMTRAP

Native processes cannot call the Guardian procedure ARMTRAP. For native processes, the trap facility based on the TNS architecture has been replaced with a different yet comparable signals facility. Native processes receive signals when run-time events occur that require immediate attention; they cannot receive traps. Signals are software interrupts that provide a way for handling asynchronous events, such as timer expiration, detection of a hardware fault, abnormal termination of a process, or any trap condition normally detectable by a TNS process. Each TNS trap has a corresponding signal.

Guardian system procedures that previously trapped on error conditions emit signals in native processes. For details, see the *Guardian Procedure Calls Reference Manual*.

If a program has trap handlers to handle trap conditions, you must write signal handlers to handle the equivalent signals.

Native programs in the Guardian environment can use these functions and procedures to receive and handle signals:

- Signals functions in the POSIX.1 standard.

  These are the signals functions in the OSS API. All of these functions can be called in C and C++, and most can be called in pTAL.

  These functions include `longjmp()`, `raise()`, `setjmp()`, `signal()`, and `sigaction()` in C and LONGJMP_, RAISE_, SETJMP_, SIGNAL_, and SIGACTION_ in pTAL.

- HP signals extensions to the POSIX.1 standard.

  These extensions are especially written for applications that focus on handling signals indicating conditions known as traps in TNS processes (those applications that call ARMTRAP). These procedures can be called in pTAL, C, and C++.

These procedures include `sigactioninit()`, `sigactionrestore()`, and `sigactionsupplant()` in C and SIGACTION_INIT_, SIGACTION_RESTORE_, and SIGACTION_SUPPLANT_ in pTAL.

The HP signals extensions are provided as convenience tools that allow native processes to catch signals corresponding to trap conditions in TNS processes. The HP signals extensions provide shortcuts to the same base functions provided by the standard signals API.

If you are concerned about conforming to the POSIX.1 standard and application portability, use the standard functions. If you are mainly interested in the performance gains of converting from TNS to native processes but want to focus on handling those signals known as trap conditions in TNS processes, use the signals extensions.

For more information on writing signal handlers, see the *Guardian Programmer's Guide*.

CHECKPOINT

The CHECKPOINT procedure is called by a primary process to send information about its current executing state to its backup process.

Native processes cannot call the CHECKPOINT procedure. Replace calls to the CHECKPOINT procedure with calls to the CHECKPOINTX procedure. CHECKPOINTX is the same as CHECKPOINT, except for an additional optional parameter to allow checkpointing data in extended data segments.

Because of differences between TNS and TNS/E native stack architecture, additional changes might be required. For details, see the CHECKPOINTX procedure in the *Guardian Procedure Calls Reference Manual*.

CHECKPOINTMANY

The CHECKPOINTMANY procedure is called by a primary process to send information about its current executing state to its backup process. The CHECKPOINTMANY procedure is used in place of the CHECKPOINT procedure when more than 13 pieces of information need to be sent.

Native processes cannot call the CHECKPOINTMANY procedure. Replace calls to the CHECKPOINTMANY procedure with calls to the CHECKPOINTMANYX procedure. CHECKPOINTMANYX is the same as CHECKPOINTMANY, except for an additional optional parameter to allow checkpointing data in extended data segments.

Because of differences between the TNS and native stack architecture, additional changes might be required. For details, see the CHECKPOINTMANYX procedure in the *Guardian Procedure Calls Reference Manual*.

CURRENTSPACE

The CURRENTSPACE procedure returns the ENV register (as saved in the stack marker) and the space ID of the caller. Native processes do not have the same register and space ID architecture as TNS processes. Remove calls to CURRENTSPACE from your program. If your program's logic relies on TNS process architecture, significant recoding of your application to support native process architecture is required.

FORMATDATA

The FORMATDATA procedure converts data item values between internal and external representations. The FORMATDATA procedure requires that all its reference parameters be 16-bit addresses. The native architecture does not support 16-bit addresses. Replace calls to FORMATDATA procedure with calls the FORMATDATAX procedure. The FORMATDATAX procedure requires that all of its reference parameters be 32-bit addresses. For details, see the *Guardian Procedure Calls Reference Manual*.

LASTADDR

The LASTADDR procedure returns the 'G'[0] relative address of the last word in the application process' data area. Native processes do not support G-relative addressing. Replace calls to the LASTADDR procedure with calls to the ADDRESS_DELIMIT_ procedure. For details, see the *Guardian Procedure Calls Reference Manual*.

LASTADDRX

The LASTADDRX procedure allows user programs to check stack limits or parameter addresses. LASTADDRX returns the last extended address available in the specified relative segment. Replace calls to the LASTADDR procedure with calls to the ADDRESS_DELIMIT_ procedure. For details, see the *Guardian Procedure Calls Reference Manual*.

XBNDSTEST

The XBNDSTEST procedure enables programs to check stack limits and parameter addresses. To check parameter addresses, replace calls to the XBNDSTEST procedure with calls to the REFPARAM_BOUNDSCHECK_ procedure. The REFPARAM_BOUNDSCHECK_ procedure checks the validity of parameter addresses passed to the procedure that calls it. Primarily, it verifies that a specified memory area is valid for a specified type of access, such as read only or read/write. For details, see the *Guardian Procedure Calls Reference Manual*.

XSTACKTEST

The XSTACKTEST procedure ensures that adequate stack space is available and returns a set of constants to be used with the XBNDSTEST procedure.

If XSTACKTEST is called to return constants passed to the XBNDSTEST procedure, delete the XSTACKTEST procedure, because native programs cannot call the XBNDSTEST procedure. For details, see XBNDSTEST on page 10-4 for more details.

If XSTACKTEST is called to ensure that adequate stack space is available, replace XSTACKTEST with calls to the HEADROOM_ENSURE_ procedure. The HEADROOM_ENSURE_ procedure enables you to make sure that the current main stack or privileged stack has enough room for the needs of your process. For details, see the *Guardian Procedure Calls Reference Manual*.

Using the INITIALIZER Procedure

The INITIALIZER procedure reads the startup message, and optionally requests receipt of the ASSIGN and PARAM messages sent by the starting process (which is often a TACL process). The INITIALIZER procedure optionally initializes file control blocks (FCBs) with the information read from the startup and ASSIGN messages.

You must change calls to the INITIALIZER procedure that pass the first parameter (the RUCB, run-unit control block). Such programs must specify two additional parameters. You need not change other callers to the INITIALIZER procedure. For details, see the *Guardian Procedure Calls Reference Manual*.

Using Sequential I/O Procedures

Many of the sequential I/O (SIO) procedures accept file control blocks (FCBs) as parameters. TNS and native programs have FCBs of different sizes. These SIO procedures have separate TAL and pTAL declarations to support FCBs of different sizes:

CLOSE^WRITE
OPEN^FILE
READ^FILE
WAIT^FILE
WRITE^FILE

You do not need to change your programs for different FCB sizes. The TAL and pTAL compilers automatically select the correct procedure version.

Callers to the CHECK^FILE and SET^FILE SIO procedures must make the changes described next.

CHECK^FILE

The CHECK^FILE procedure retrieves the file characteristics of a specified file. There are two versions of the CHECK^FILE procedure: one for TNS programs and one for native programs. Separate TNS and native versions are required because pTAL uses separate data types for passing and returning integer and address parameter values.

For TNS programs, the procedure passes integer and address values through a type INT parameter. For native programs, the procedure returns integer values through a type INT parameter and address values through a new optional parameter of type WADDR. pTAL type WADDR is equivalent to C type `short`. When converting to native mode, change your programs to use the native version of CHECK^FILE.

To maintain common TNS and native source code, use the CALL_CHECK^FILE_ADDRESS_ DEFINE in the GPLDEFS declarations file. This DEFINE calls the correct version of CHECK^FILE, depending on which environment the program is compiled in.

For more information about the CHECK^FILE procedure, see the *Guardian Procedure Calls Reference Manual*.

SET^FILE

The SET^FILE procedure alters file characteristics and checks the old values of the characteristics being altered. There are two version of the SET^FILE procedure: one for TNS programs and one for native programs. Separate TNS and native versions are required because pTAL uses separate data types for passing and returning integer and address parameter values.

For TNS programs, the procedure returns integer and address values through a pointer of type INT. For native programs, the procedure returns integer values through a type INT parameter and address values through a new optional parameter of type WADDR. pTAL type WADDR is equivalent to C type `short`. When converting to TNS/E native mode, change your programs to use the native version of SET^FILE.

To maintain common TNS and native source code, use the CALL_SET^FILE_ADDRESS_ DEFINE in the GPLDEFS declarations file. This DEFINE calls the correct version of SET^FILE, depending on which environment the program is compiled in.

For more information about the SET^FILE procedure, see the *Guardian Procedure Calls Reference Manual*.

Using Procedures Enhanced to Support the Native Architecture

These procedures have been enhanced to support native processes, native object files, and DLLs:

NEWPROCESS
NEWPROCESSNOWAIT
OBJECTFILE_GETINFOLIST_
PROCESS_CREATE_
PROCESS_GETINFOLIST_
PROCESSINFO
PROCESS_SETINFO_
PROCESS_SPAWN_

Depending on the operations performed by these procedures, you might need to specify new or different parameters. For details, see the procedure's description in the *Guardian Procedure Calls Reference Manual*.

Additionally, the NEWPROCESS, NEWPROCESSNOWAIT, and PROCESS_CREATE_ procedures have been superseded by the PROCESS_LAUNCH_ procedure. The PROCESSINFO procedure has been superseded by the PROCESS_GETINFOLIST_ procedure. You might need to replace calls to these superseded procedures with calls to the replacement procedures to specify parameters specific to the TNS/E native architecture.

Note. Because the PROCESS_LAUNCH_ procedure is the only procedure to fully support native process creation, you should replace calls to the NEWPROCESS, NEWPROCESSNOWAIT, and PROCESS_CREATE_ procedures with calls to the PROCESS_LAUNCH_ procedure when converting to native mode.

Using Procedures Affected by KMSF

KMSF manages the swap space for native processes and, to a lesser extent, TNS processes. For more information on this facility, see KMSF on page 1-19. KMSF affects procedures that specify and return information on swap space.

Use the following procedures to specify or return the swap volume or file of the user data segment for a TNS process:

NEWPROCESS
NEWPROCESSNOWAIT
PROCESS_CREATE_
PROCESSINFO
PROCESS_GETINFO_

Native processes do not use swap volume or file values specified by these procedures, so those input values are ignored. The PROCESSINFO and PROCESS_GETINFO_ procedures return the volume and file names (if specified). The values returned are the names specified when the process was created, not the actual swap volume managed by KMSF.

Note. You can continue to use the NEWPROCESS, NEWPROCESSNOWAIT, and PROCESS_CREATE_ procedures to specify the volume for temporary files created by TNS and native processes.

Use these procedures to specify or return the number or maximum number of data pages to be allocated for the user data stack for a TNS process:

CHECKMONITOR
GETSYNCINFO
NEWPROCESS
NEWPROCESSNOWAIT
PROCESS_CREATE_
PROCESS_SPAWN_
SETSYNCINFO

Native processes do not use these data page values, so they are ignored. Similar heap attribute values for native processes can be set with `eld` utility or the PROCESS_LAUNCH_ procedure.

Use these procedures to specify or return the volume or file of the default extended data segment for a TNS process:

PROCESS_CREATE_
PROCESS_GETINFOLIST_
PROCESS_SPAWN_

Native processes do not use these volume or file values, so they are ignored. The PROCESS_GETINFOLIST_ procedure returns the volume name (if specified) and #0. The value returned is the name specified when the process was created, not the actual swap volume managed by KMSF. Depending on the operations performed by these procedures, you might need to specify new or different parameters. See the procedure's description in the *Guardian Procedure Calls Reference Manual* for details.

Using Procedures With pTAL Address Types

These procedures use pTAL data types that support the address types WADDR, BADDR, EXTADDR, and PROCPTR:

ADDRESS_DELIMIT_	INITIALIZER	SORTERROR
AWAITIO	MEASWRITE_DIFF_	SORTERRORDETAIL
AWAITIOX	NUMIN	SORTERRORSUM
CHECK^FILE	POOL_CHECK_	SORTMERGEFINISH
DNUMIN	PROCESS_SPAWN_	SORTMERGERECEIVE
ENFORMSTART	SEGMENT_ALLOCATE_	SORTMERGESEND
GETPOOL	SEGMENT_GETINFO_	SORTMERGESTART
HEAPSORT	SEGMENT_USE_	SORTMERGESTATISTICS
HEAPSORTX	SET^FILE	

When converting TAL programs, use the pTAL compiler SYNTAX directive to enable syntax checking. That detects whether the program requires changes.

C and C++ programs do not require changes.

For more details, see the procedure's description in the *Guardian Procedure Calls Reference Manual*.

Writing Multithreaded Programs

Native processes do not maintain the TNS register and stack architecture. Therefore, user-written multithreaded programs that directly manipulate TNS registers and the stack require changes to be converted to native mode. Programs must use a new set of multithreaded support procedures. The multithreaded support procedures enable programs to save and restore thread context and to create the context for new threads. Both TNS and native programs can use the multithread support procedures. See Support Note S96001, "T9050: User-Level TNS/R Native Thread Primitives" for details.

Calling Code You Add to the System Library

If a TNS program calls code that you have added to the system library, the code added to the system library must be converted to TNS/E native code when you convert the program.

If both TNS processes (running in interpreted mode or accelerated mode) and native processes call code you add to the system library, you need two versions of the code: one that has been accelerated and one that has been compiled with a TNS/E native compiler. The accelerated and TNS/E native versions can contain the same procedure

names. See the *H06.nn Software Installation and Upgrade Guide* for installation details.

Adjusting for Increased DCT Limits

This change can affect TAL programs that you are converting to TNS/E pTAL.

The destination control table (DCT) contains entries for logical device numbers and named processes. The DCT limit refers to the maximum number of logical device numbers and named processes that the operating system can accommodate. As of the G06.23 RVU, the size of the DCT can optionally be increased from its previous limit of 32,767 (a logical device number can have at most 15 bits) to 65,376 (a logical device number can have up to16 bits). This change can affect TAL programs that call any of these C-series procedures:

C-Series Procedure	Extended DCT limits affect calls that:
FILEINFO	Use the optional *ldevnum* parameter.
GETDEVNAME	(Affects all calls.)
GETPDENTRY	(Affects all calls.)
GETSYSTEMNAME	Use the return value as an ldev or check for specific error codes.
LOCATESYSTEM	Use the return value as an ldev or check for specific error codes.
LOOKUPPROCESSNAME	Pass a DCT index in the *ppd* parameter.

In G06.23 and later RVUs, the default setting for extended DCT limits is OFF; that is, the extended limits are not in effect. In H-series RVUs, the default setting for extended DCT limits is ON; that is, the extended limits are in effect. Therefore, if you have not yet changed any affected applications to allow for the increased limits, you must do one of the following:

- Ensure that the system default DCT limits extension is reset to OFF (do this through an SCF command).

  or

- Change your program to allow for the increased DCT limits.

The recommended solution is to replace the affected procedures with updated procedures that can handle the increased DCT limits. The recommended replacement procedures are:

C-Series Procedure	Replacement Procedure
FILEINFO	FILE_GETINFOLIST_
GETDEVNAME	DEVICE_GETINFOBYLDEV_ CONFIG_GETINFOBYLDEV_ CONFIG_GETINFOBYLDEV2_ FILENAME_FINDSTART_ FILENAME_FINDNEXT_
GETPDENTRY	PROCESS_GETPAIRINFO_
GETSYSTEMNAME	NODENUMBER_TO_NODENAME_
LOCATESYSTEM	NODENAME_TO_NODENUMBER_
LOOKUPPROCESSNAME	PROCESS_GETPAIRINFO_

For more details on the use of these replacement procedures, see the *Guardian Procedure Calls Reference Manual.*

11

OSS API and Utilities Conversion Tasks

The OSS environment provides an industry-standard API and set of utilities for Integrity NonStop servers..

This section describes the OSS API changes required to convert TNS programs to TNS/E native mode. It also describes changes required to the `cobol`, `c89` and `c99` utility command lines to run the TNS/E native COBOL and C/C++ compilers.

Make the changes described in Section 3, C and C++ Conversion Tasks or Section 4, Converting COBOL Programs before making the changes described in this section. This section discusses:

Specifying Compilation System Flags

The TNS and TNS/E native compilation systems use different components, as described in Native Development Environment on page 1-4. Following are descriptions of the differences in the compilation system flags.

COBOL Compilation System

To use the TNS/E native COBOL compilation system components, use the `ecobol` utility instead of the `cobol` utility. You must also change certain flags. Table 11-1 on page 11-2 shows the compilation system flags that must be changed when moving from TNS mode to TNS/E native mode.

Table 11-1. COBOL Flag Changes Required: TNS to TNS/E

Changed TNS `cobol` Utility Flag	Reason for Change	Action Required
`-Waxcel`	Accelerator unnecessary because TNS/E native compilers generate Itanium instructions.	Remove flag.
`-Wbind`	`eld` utility used instead of `Binder` for TNS/E native programs.	Remove flag. Some `-Wbind` arguments (such as for setting object file attributes) have corresponding `ecobol` or `eld` flags. Specify corresponding `ecobol` flags or pass corresponding arguments to `eld` using `-Weld` flag. See `eld(1)` and `ecobol(1)` reference pages for details.
`-Wrunlib`	TNS/E native compilers cannot specify user library.	Specify user library with a `-Weld="-libname`*`library`*`"` flag.

The `ecobol` utility provides new flags that are not supported by the `cobol` utility. For details, see the *Open System Services Shell and Utilities Reference Manual* or the `ecobol(1)` reference pages.

Native C Compilation System

Table 11-2 shows the c89 flags that must be changed when moving from TNS mode to TNS/E native mode.

Table 11-2. c89 Flag Changes Required: TNS to TNS/E Native (page 1 of 2)

Changed TNS c89 Utility Flag	Reason for Change	Action Required
-O	Accelerator unnecessary because TNS/E native compilers generate Itanium instructions. Flag now specifies native compiler optimization level.	Remove flag. Use default native compiler optimization level of 1 during conversion.
-Waxcel	Accelerator unnecessary because TNS/E native compilers generate Itanium instructions.	Remove flag.
-Wbind	eld utility used instead of Binder for TNS/E native programs.	Remove flag. Some -Wbind arguments (such as for setting object file attributes) have corresponding native c89, c99, or eld flags. Specify corresponding TNS/E native c89 or c99 flags or pass corresponding arguments to eld using -eld flag. See eld(1), c89(1), and c99(1) reference pages for details.
-Wccom	All compiler pragmas and arguments must be specified using c89 or c99 flags so that they can be validated.	Remove flag. Most -Wccom arguments (such as pragmas) have corresponding native c89 or c99 flags. Specify corresponding c89 or c99 flags. See c89(1) or c99(1) reference page for details.
-Wcfonly	cfront function performed by component of TNS/E native compilers.	Replace with -WP flag.

Table 11-2. c89 Flag Changes Required: TNS to TNS/E Native (page 2 of 2)

Changed TNS c89 Utility Flag	Reason for Change	Action Required
`-Wcfront`	`cfront` function performed by component of TNS/E native compilers.	Replace with `-WP` flag. No arguments can be passed. Most `-Wcfront` arguments (such as pragmas) have corresponding native `c89` or `c99` flags. Specify corresponding `c89` or `c99` flags. See `c89(1)` or `c99(1)` reference page for details
`-Wcprep`	`cprep` function performed by component of TNS/E native compilers.	Replace with `-WP` flag. No arguments can be passed. Most `-Wcprep` arguments (such as pragmas) have corresponding native `c89` or `c99` flags. Specify corresponding `c89` or `c99` flags. See `c89(1)` or `c99(1)` reference page for details
`-Wnobind`	`eld` utility used instead of Binder for TNS/E native programs.	Replace with `-Wnolink` flag.
`-Wrunlib`	TNS/E native compilers cannot specify user library.	Specify user library with a `-Weld="-libname`*`library`*`"` flag.
`-Wsql`	`-Wsql` now implements `SQL` pragma (`c89`). `-Wsql` is not supported (`c99`).	Replace with `-Wsqlcomp` flag (`c89`). See Specifying SQL Compilation on page 11-6 for details.

Using System Calls Enhanced to Support the Native Architecture

These OSS system calls have been enhanced to support native processes, native object files, and DLLs:

```
tdm_execve
tdm_execvep
tdm_fork
tdm_spawn
tdm_spawnp
```

Depending on the operations that your TNS programs perform with these functions, you might need to specify new or different parameters. Also, any returned error values should not be depended upon. For details, see the function's reference page online or in the *Open System Services System Calls Reference Manual*.

Specifying Compiler Pragmas

On G-series systems, there are two versions of the OSS `c89` utility: one version for the TNS compilation system and one version for the native compilation system. H-series systems support only the native compilation system, either the native `c89` utility or the native `c99` utility (beginning with H06.21/J06.10 and later versions). The native `c89` utilities are nearly identical on G-series and H-series systems.

The TNS `c89` utility has two flags that support compiler pragmas, `-Wsystype` and `-Wverbose`. For all other pragmas, you either place pragmas in the source text or pass pragmas to compilation system components by using the `-Wccom` flag and an argument string, such as:

```
-Wccom="runnamed,nomap,inline"
```

The native `c89` and `c99` utilities have flags that support most compiler pragmas because native compilers require most pragmas to appear on the command line. The flags also enable `c89` and `c99` to validate pragmas before invoking compilation system components. For example, the TNS `c89 -Wccom` flag in the preceding example is replaced with these flags:

```
-Wrunnamed -Wnomap -Winline
```

Flags that support compiler pragmas begin with *`-W`* to identify them as HP extensions for NonStop systems. For information on mapping pragmas to native c`89` or `c99` flags, see the native `89` or `c99` reference page online or in the *Open System Services Shell and Utilities Reference Manual*.

Specifying Files in the Guardian File System (/G)

To specify files in the Guardian file system, use OSS pathname syntax (`/G/volume/subvol/file`). Product versions of the TNS `c89` utility prior to D40 do not require files in the Guardian file system to be identified with a suffix. (The OSS file system requires files to be identified with a suffix.) D40 and later versions of the TNS `c89` utility and the H-series native `c89` and `c99` utilities require a suffix. For example,

```
/G/MYVOL/MYSUBVOL/FILE.c
```

identifies the Guardian source file FILEC (in the Guardian environment, the suffix becomes the last character of the filename).

Therefore, when converting TNS programs written prior to D40, you must add the correct suffix to files in the Guardian file system (if it is not already present). For a list of valid file suffixes, see either the `c89(1)` or `c99(1)` reference page online or in the *Open System Services Shell and Utilities Reference Manual*.

Specifying SQL Compilation

As noted in Table 11-2, the TNS and TNS/E native `c89` utilities use different flags to run the SQL compiler. The required changes are shown in this example:

- To compile a program with embedded SQL using the TNS `c89` utility, specify the `SQL` pragma in the `-Wccom` flag and the `-Wsql` flag to run the SQL compiler, as follows:

  ```
  c89 -Wccom="sql(sqlmap,release2)"
      -Wsql="compile program" prog.c
  ```

- To compile a program with embedded SQL using the native `c89` utility, specify the `SQL` pragma with a `-Wsql` flag and the `-Wsqlcomp` flag to run the SQL compiler, as follows:

  ```
  c89 -Wsql="sqlmap,release2"
      -Wsqlcomp="compile program" prog.c
  ```

Compiling and Linking for Pthreads

When compiling a TNS program on a TNS/R system using T1248 pthreads and the C++ exception handling mechanism, you need to explicitly link the object `/usr/lib/sptcpp.o`. to ensure that exception handling continues to work with all C++ versions. However, for programming in native C++ on a TNS/E system, linking `/usr/lib/sptcpp.o` is not necessary.

Glossary

accelerate. To speed up emulated execution of a TNS object file by applying the Accelerator for TNS/R system execution or the Object Code Accelerator (OCA) for TNS/E system execution before running the object file.

accelerated mode. See TNS accelerated mode.

accelerated object code. The MIPS RISC instructions (in the MIPS region) that result from processing a TNS object file with the Accelerator, or the Intel® Itanium® instructions (in the Itanium region) that result from processing a TNS object file with the Object Code Accelerator (OCA).

accelerated object file. A TNS object file that, in addition to its TNS instructions and symbol information, has been augmented by either the Accelerator, with equivalent but faster MIPS RISC instructions, or the Object Code Accelerator (OCA), with equivalent but faster Intel® Itanium® instructions, or both.

Accelerator. A program optimization tool that processes a TNS object file and produces an accelerated object file that also contains equivalent MIPS RISC instructions (called the MIPS region). TNS object code that is accelerated runs faster on TNS/R processors than TNS object code that is not accelerated. See also Object Code Accelerator (OCA).

API. See application program interface (API).

application program interface (API). A set of services (such as programming language functions or procedures) that are called by an application program to communicate with other software components. For example, an application program in the form of a client might use an API to communicate with a server program.

Binder. A programming utility that combines one or more compilation units' TNS object code files to create an executable TNS object code file for a TNS program or library. Used only with TNS object files. See also nld utility, ld utility, and eld utility.

CISC. See complex instruction-set computing (CISC).

complex instruction-set computing (CISC). A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with reduced instruction-set computing (RISC) and Explicitly Parallel Instruction Computing (EPIC).

DLL. See dynamic-link library (DLL).

DWARF. An industry-standard format for symbol table information. It is added to TNS/E object files, and is used primarily for debugging (not for most linking activities).

dynamic-link library (DLL). A collection of procedures whose code and data can be loaded and executed at any virtual memory address, with run-time resolution of links to and

from the main program and other independent libraries. The same DLL can be used by more than one process. Each process gets its own copy of DLL static data. Contrast with shared run-time library (SRL). See also position-independent code (PIC).

eld utility. A utility that collects, links, and modifies code and data blocks from one or more TNS/E object files to produce a target TNS/E native loadfile. See also nld utility, ld utility, and Binder.

ELF. See executable and linking format (ELF).

emulate. To imitate the instruction set and address spaces of a different hardware system by means of software. Emulator software is compatible with and runs software built for the emulated system. For example, a TNS/R or TNS/E system emulates the behavior of a TNS system when executing interpreted or accelerated TNS object code.

enoft utility. A utility that reads and displays information from TNS/E native object files. See also noft utility.

EPIC. See Explicitly Parallel Instruction Computing (EPIC).

executable and linking format (ELF). A standard format used for POSIX object files. TNS/R and TNS/E native object files are in ELF format with HP extensions.

execution mode. The emulated or real instruction set environment in which object code runs. A TNS system has only one execution mode: TNS mode using TNS compilers and 16-bit TNS instructions. A TNS/R system has three execution modes: TNS/R native mode using TNS/R native compilers and RISC instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode. A TNS/E system also has three execution modes: TNS/E native mode using TNS/E native compilers and Intel® Itanium® instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode.

Explicitly Parallel Instruction Computing (EPIC). The technology that forms the basis for the Intel® Itanium® architecture. EPIC technology enables parallel processing opportunities to be explicitly identified by the compiler before the software code is executed by the processor.

Guardian. An environment available for interactive or programmatic use with the NonStop operating system. Processes that run in the Guardian environment use the Guardian system procedure calls as their application program interface. Interactive users of the Guardian environment use the HP Tandem Advanced Command Language (TACL) or another HP product's command interpreter. Contrast with Open System Services (OSS).

Guardian environment. The Guardian application program interface (API), tools, and utilities.

HP NonStop™ Open System Services (OSS). An open system environment available for interactive or programmatic use with the HP NonStop™ operating system. Processes

that run in the OSS environment usually use the OSS application program interface. Interactive users of the OSS environment usually use the OSS shell for their command interpreter. See also HP NonStop Open System Services (OSS) environment. Contrast with Guardian.

HP NonStop Open System Services (OSS) environment. The HP NonStop™ Open System Services (OSS) application program interface (API), tools, and utilities.

HP NonStop™ operating system. The operating system for HP NonStop systems.

HP Transaction Application Language (TAL). A systems programming language with many features specific to stack-oriented TNS systems.

hybrid shared run-time library (hybrid SRL). A shared run-time library (SRL) that has been augmented by the addition of a dynamic section that exports SRL symbols in a form that can be used by position-independent code (PIC) clients. A hybrid SRL looks like a dynamic-link library (DLL) to PIC clients (except it cannot be loaded at other addresses and cannot itself link to DLLs). The code and data in the SRL are no different in a hybrid SRL, and its semantics for non-PIC clients are unchanged.

Intel® Itanium® instructions. Register-oriented EPIC machine instructions in the Itanium instruction set that are native to and directly executed by a TNS/E system. Itanium instructions do not execute on TNS and TNS/R systems. Contrast with TNS instructions and RISC instructions.

TNS Object Code Accelerator (OCA)-generated Itanium instructions are produced by accelerating TNS object code. Native-compiled Itanium instructions are produced by compiling source code with a TNS/E native compiler.

ld utility. A utility that collects, links, and modifies code and data blocks from one or more position-independent code (PIC) object files to produce a target TNS/R native PIC object file. See also eld utility, ld utility, and Binder.

linker. (1) The process or server that invokes the message system to deliver a message to some other process or server. (2) A programming utility that combines one or more compilation units' linkfiles to create an executable loadfile for a native program or library. See also ld utility, nld utility, and eld utility.

linkfile. (1) A file containing object code that is not yet ready to load and execute. Linkfiles are combined by means of a linker to make an executable loadfile for a program or library. (2) For native C/C++ compilers in the Guardian environment, a command file for input to the `eld`, `ld`, or `nld` utility. Compiling creates one linkfile per independent source module. Contrast with loadfile.

linking. The operation of examining, collecting, linking, and modifying code and data blocks from one or more object files to produce a target object file.

loader. A programming utility that transfers a program into memory so that 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, a loader called `rld` works with the operating system to load programs and libraries.

loadfile. An executable object code file 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.

native linker. Often used to refer generically to the `ld`, `nld` or `eld` utility. In this manual, the term is used as shorthand for the `eld` utility. See also ld utility, nld utility, and eld utility.

native mode. Often used to refer generically to TNS/R or TNS/E native mode. In this manual, the term is used as shorthand for TNS/E native mode. See also TNS/R native mode and TNS/E native mode.

native object code. Often used to refer generically to TNS/R or TNS/E native object code. In this manual, the term is used as shorthand for TNS/E native object code. See also TNS/R native object code and TNS/E native object code.

native object file. Often used to refer generically to a TNS/R or TNS/E native object file. In this manual, the term is used as shorthand for TNS/E native object file. See also TNS/R native object file and TNS/E native object file.

native object file tool. Often used to refer generically to the `noft` or `enoft` utility. In this manual, the term is used as shorthand for `enoft`. See also noft utility and enoft utility.

native process. Often used to refer generically to a TNS/R or TNS/E native process. In this manual, the term is used as shorthand or TNS/E native process. See also TNS/R native process and TNS/E native process.

nld utility. A utility that collects, links, and modifies code and data blocks from one or more non-position-independent code (non-PIC) object files to produce a target TNS/R native non-PIC object file. See also ld utility, eld utility, and Binder.

noft utility. A utility that reads and displays information from TNS/R native object files. See also enoft utility.

NonStop™ Open System Services (OSS). See HP NonStop™ Open System Services (OSS).

NonStop™ operating system. See HP NonStop™ operating system.

Object Code Accelerator (OCA). A program optimization tool that processes a TNS object file and produces an accelerated file for a TNS/E system. OCA augments a TNS object

file with equivalent Itanium instructions. TNS object code that is accelerated runs faster than TNS object code that is not accelerated. See also Accelerator and Object Code Interpreter (OCI).

Object Code Interpreter (OCI). A program that processes a TNS object file and emulates TNS instructions on a TNS/E system without preprocessing the object file. See also Object Code Accelerator (OCA).

object file. A file generated by a compiler, Binder, or linker that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file might be a complete program that is ready for immediate execution, or it might be incomplete and require linking with other object files before execution.

OCA. (1) The command used to invoke the TNS Object Code Accelerator (OCA) on a TNS/E system. (2) See Object Code Accelerator (OCA).

OCI. See Object Code Interpreter (OCI).

Open System Services (OSS). See HP NonStop™ Open System Services (OSS)

operating system. See HP NonStop™ operating system.

OSS. See Open System Services (OSS).

PIC. See position-independent code (PIC).

position-independent code (PIC). 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. PIC makes it possible for programmers to create dynamic-link libraries, which can be loaded and unloaded by an executing program. See also dynamic-link library (DLL).

process. (1) A program that has been submitted to the operating system for execution or a program that is currently running in the computer. (2) An address space, a single thread of control that executes within that address space, and the system resources required by that thread of control.

program file. An executable object code file containing a program's main routine plus related routines statically linked together and combined into the same object file. Other routines shared with other programs might be located in separately loaded libraries. A program file can be named on a RUN command; other code files cannot. See also object file.

pTAL. Portable Transaction Application Language. A machine-independent system programming language based on Transaction Application Language (TAL). The pTAL language excludes architecture-specific TAL constructs and includes new constructs that replace the architecture-specific constructs. Contrast with HP Transaction Application Language (TAL).

pTAL compiler. An optimizing native-mode compiler for the pTAL language.

public dynamic-link library (public DLL). Optional native-mode executable code modules available to all native user processes. A TNS/E public library is specified in the public library registry, supplied by HP or optionally a user.

public library. A dynamic-link library (DLL) or shared run-time library (SRL) that is known to the operating system, available for execution by any process or user, and is not an implicit library.

public shared run-time library (public SRL). A TNS/R library supplied by HP.

reduced instruction-set computing (RISC). A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with complex instruction-set computing (CISC) and Explicitly Parallel Instruction Computing (EPIC).

RISC. See reduced instruction-set computing (RISC).

RISC instructions. Register-oriented 32-bit machine instructions in the MIPS-1 RISC instruction set that are native to and directly executed on TNS/R systems. RISC instructions do not execute on TNS systems and TNS/E systems. Contrast with TNS instructions and Intel® Itanium® instructions.

Accelerator-generated RISC instructions are produced by accelerating TNS object code. Native-compiled RISC instructions are produced by compiling source code with a TNS/R native compiler.

RISC word. An instruction-set-defined unit of memory. A RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. Contrast with TNS word. See also ld utility.

shared run-time library (SRL). A collection of procedures whose code and data can be loaded and executed only at a specific assigned virtual memory address (the same address in all processes). SRLs use direct addressing and do not have run-time resolution of links to and from the main program and other independent libraries. SRLs are not supported on TNS/E systems. Contrast with dynamic-link library (DLL). See also TNS shared run-time library (TNS SRL) and TNS/R native shared run-time library (TNS/R native SRL).

SRL. See shared run-time library (SRL).

TAL. See HP Transaction Application Language (TAL).

TAL compiler. The nonnative compiler that takes TAL or pTAL source code as input and generates TNS object code. Compare to pTAL compiler.

TNS. Fault-tolerant HP computers that support the HP NonStop™ operating system and are based on microcoded complex instruction-set computing (CISC) technology. TNS systems run the TNS instruction set. Contrast with TNS/R and TNS/E.

TNS accelerated mode. A TNS emulation environment on a TNS/R or TNS/E system in which accelerated TNS object files are run. TNS instructions have been previously translated into optimized sequences of RISC or Intel® Itanium® instructions. TNS accelerated mode runs much faster than TNS interpreted mode. Accelerated or interpreted TNS object code cannot be mixed with or called by native mode object code. See also Object Code Accelerator (OCA). Contrast with TNS/R native mode and TNS/E native mode.

TNS C compiler. The C compiler that generates TNS object files. Compare to TNS/R native C compiler and TNS/E native C compiler.

TNS COBOL compiler. The COBOL compiler that generates TNS object files. Compare to TNS/R native COBOL compiler and TNS/E native COBOL compiler.

TNS instructions. Stack-oriented, 16-bit machine instructions that are directly executed on TNS systems by hardware and microcode. TNS instructions can be emulated on TNS/E and TNS/R systems by using millicode, an interpreter, and either translation or acceleration. Contrast with RISC instructions and Intel® Itanium® instructions

TNS interpreted mode. A TNS emulation environment on a TNS/R or TNS/E system in which individual TNS instructions in a TNS object file are directly executed by interpretation rather than permanently translated into RISC or Itanium instructions. TNS interpreted mode runs slower than TNS accelerated mode. Each TNS instruction is decoded each time it is executed, and no optimizations between TNS instructions are possible. TNS interpreted mode is used when a TNS object file has not been accelerated for that hardware system, and it is also sometimes used for brief periods within accelerated object files. Accelerated or interpreted TNS object code cannot be mixed with or called by native object code. See also Object Code Interpreter (OCI). Contrast with TNS accelerated mode, TNS/R native mode, and TNS/E native mode.

TNS object code. The TNS instructions that result from processing program source code with a TNS language compiler. TNS object code executes on TNS, TNS/R, and TNS/E systems.

TNS object file. An object file created by a TNS compiler or the Binder. A TNS object file contains TNS instructions. TNS object files can be processed by the Accelerator or by the Object Code Accelerator (OCA) to produce accelerated object files. A TNS object file can be run on TNS, TNS/R, and TNS/E systems.

TNS process. A process whose main program object file is a TNS object file, compiled using a TNS compiler. A TNS process executes in interpreted or accelerated mode while within itself, when calling a user library, or when calling into TNS system libraries. A TNS process temporarily executes in native mode when calling into native-compiled parts of the system library. Object files within a TNS process might be accelerated or

not, with automatic switching between accelerated and interpreted modes on calls and returns between those parts. Contrast with TNS/R native process and TNS/E native process.

TNS shared run-time library (TNS SRL). An SRL available to a TNS process in the OSS environment on TNS/R systems. A TNS process can have only one TNS SRL. A TNS SRL is implemented as a special user library that allows shared global data.

TNS user library. A user library available to TNS processes in the Guardian environment. See also user library.

TNS word. An instruction-set-defined unit of memory. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. See also RISC word.

TNS/E. Fault-tolerant HP computers that support the HP NonStop™ operating system and are based on the Intel® Itanium® processor. TNS/E systems run the Itanium instruction set and can run TNS object files by interpretation or after acceleration. TNS/E systems include all HP NonStop™ systems that use NSE-*x* processors. Contrast with TNS and TNS/R.

TNS/E native C compiler. The C compiler that generates TNS/E object files. Compare to TNS C compiler and TNS/R native C compiler.

TNS/E native COBOL compiler. The COBOL compiler that generates TNS/E object files. Compare to TNS COBOL compiler and TNS/R native COBOL compiler.

TNS/E native mode. The primary execution environment on a TNS/E system, in which native-compiled Itanium object code executes, following TNS/E native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage. Contrast with TNS interpreted mode and TNS accelerated mode. See also TNS/R native mode.

TNS/E native object code. The Intel® Itanium® instructions that result from processing program source code with a TNS/E native compiler. TNS/E native object code executes only on TNS/E systems, not on TNS systems or TNS/R systems.

TNS/E native object file. An object file created by a TNS/E native compiler that contains Intel® Itanium® instructions and other information needed to construct the code spaces and the initial data for a TNS/E native process.

TNS/E native process. A process initiated by executing a TNS/E native object file. Contrast with TNS process and TNS/R native process.

TNS/E native user library. A user library available to TNS/E native processes in the Guardian and OSS environments. A TNS/E native user library is implemented as a dynamic-link library (DLL).

TNS/E pTAL compiler. An optimizing native-mode compiler for the TNS/E pTAL language. Compare to TNS/R pTAL compiler and TAL compiler.

TNS/R. Fault-tolerant HP computers that support the HP NonStop™ operating system and are based on 32-bit reduced instruction-set computing (RISC) technology. TNS/R systems run the MIPS-1 RISC instruction set and can run TNS object files by interpretation or after acceleration. TNS/R systems include all HP systems that use NSR-*x* processors. Contrast with TNS and TNS/E.

TNS/R native C compiler. The C compiler that generates TNS/R object files. Compare to TNS C compiler and TNS/E native C compiler.

TNS/R native COBOL compiler. The COBOL compiler that generates TNS/E object files. Compare to TNS COBOL compiler and TNS/E native COBOL compiler.

TNS/R native mode. The operational environment in which native-compiled RISC instructions execute.

TNS/R native object code. The RISC instructions that result from processing program source code with a TNS/R native compiler. TNS/R native object code executes only on TNS/R systems, not on TNS or TNS/E systems.

TNS/R native object file. A file created by a TNS/R native compiler that contains RISC instructions and other information needed to construct the code spaces and the initial data for a TNS/R native process.

TNS/R native process. A process initiated by executing a TNS/R native object file. Contrast with TNS process and TNS/E native process.

TNS/R native shared run-time library (TNS/R native SRL). A shared run-time library (SRL) available to TNS/R native processes in both the Guardian and HP™ NonStop Open System Services (OSS) environments. TNS/R native SRLs can be either public or private. A TNS/R native process can have multiple public SRLs but only one private SRL.

TNS/R native user library. A user library available to TNS/R native processes in both the Guardian and HP NonStop™ Open System Services (OSS) environments. A TNS/R native user library is implemented as a special private TNS/R native shared run-time library (TNS/R native SRL).

TNS/R pTAL compiler. An optimizing native-mode compiler for the TNS/R pTAL language. Compare to TNS/E pTAL compiler and TAL compiler.

user library. An object file that the operating system links to a program file at run time. A program can have only one user library. See also TNS user library and TNS/R native user library.

Index

Numbers

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Special Characters