

HP pTAL Reference Manual



HP Part Number: 523746-009

Published: Feb 2012

Edition: D44.00 and all subsequent D-series RVUs, all J-series, H-series, and G-series RVUs

Legal Notice

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks, and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. The OSF documentation and the OSF software to which it relates are derived in part from materials supplied by the following: © 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation. OSF software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.

Contents

About This Document.....	16
Supported Release Version Updates (RVUs).....	16
Intended Audience.....	16
New and Changed Information.....	16
New and Changed Information for 523746-009.....	16
New and Changed Information for 523746-008.....	18
New and Changed Information for 523746-007.....	19
Document Organization.....	19
Notation Conventions.....	20
Syntax Diagram Conventions.....	20
General Syntax Notation.....	23
Notation for Messages.....	25
Notation for Management Programming Interfaces.....	26
Related Information.....	26
Publishing History.....	29
HP Encourages Your Comments.....	29
1 Introduction to pTAL.....	30
pTAL and TAL Compatibility.....	30
EpTAL, pTAL, and TAL Compilers.....	30
pTAL Applications.....	31
pTAL Features.....	32
Procedures.....	32
Subprocedures.....	32
Private Data Area.....	32
Recursion.....	33
Parameters.....	33
Data Types.....	33
Data Grouping.....	33
Pointers.....	34
Data Operations.....	34
Bit Operations.....	34
Built-in Routines.....	34
Compiler Directives.....	34
Modular Programming.....	34
System Services.....	34
System Procedures.....	34
pTAL and the CRE.....	34
2 Language Elements.....	36
Character Set.....	36
Keywords.....	37
Delimiters.....	38
Operators.....	39
Base Address Symbols.....	40
Indirection Symbols.....	41
Declarations.....	41
Identifiers.....	42
Variables.....	43
Scope.....	43
Typed Integer Constants.....	44
Statements.....	45

3 Data Representation.....	46
Data Types.....	46
Specifying Data Types.....	47
Data Type Aliases.....	48
Operations by Data Type.....	48
Address Types.....	49
Storing Addresses in Variables.....	51
Converting Between Address Types and Numeric Data Types.....	51
Converting Between Address Types.....	52
Using Indexes to Access Array Elements.....	54
Incrementing and Decrementing Addresses (Stepping Pointers).....	54
Constants.....	57
Character String.....	57
STRING Numeric.....	58
INT Numeric.....	58
INT(32) Numeric.....	59
FIXED Numeric.....	61
REAL and REAL(64) Numeric.....	62
Constant Lists.....	63
Constant List Alignment Specification.....	64
4 Data Alignment.....	66
Misalignment Tracing Facility.....	66
Misalignment Handling.....	67
5 Expressions.....	69
Data Types of Expressions.....	70
Operator Precedence.....	70
Arithmetic Expressions.....	72
Signed Arithmetic Operators.....	73
Scaling of FIXED Operands.....	74
Using FIXED(*) Variables.....	74
Unsigned Arithmetic Operators.....	75
Bitwise Logical Operators.....	76
Using Bitwise Logical Operators and INT(32) Operands.....	76
Comparing Addresses.....	77
Extended Addresses.....	77
Nonextended Addresses.....	78
Constant Expressions.....	81
Conditional Expressions.....	81
NOT, OR, and AND Operators.....	82
Relational Operators.....	83
Special Expressions.....	85
Assignment.....	85
CASE.....	86
IF.....	87
Group Comparison.....	88
Bit Operations.....	92
Bit Extractions.....	93
Bit Shifts.....	94
6 LITERALS and DEFINES.....	97
Declaring Literals.....	97
Declaring DEFINES.....	98
Calling DEFINES.....	100
How the Compiler Processes DEFINES.....	100

Passing Actual Parameters to DEFINES.....	100
7 Simple Variables.....	103
Declaring Simple Variables.....	103
Specifying Simple Variable Address Types.....	105
Initializing Simple Variables With Numbers.....	105
Initializing Simple Variables With Character Strings.....	105
Examples.....	105
8 Arrays.....	108
Declaring Arrays.....	108
Declaring Read-Only Arrays.....	111
Using Constant Lists in Array Declarations.....	113
9 Structures.....	114
Structure Layout.....	115
Overview of Structure Alignment.....	116
Structures Aligned at Odd-Byte Boundaries.....	117
Overview of Field Alignment.....	117
SHARED2.....	117
SHARED8.....	118
PLATFORM.....	118
AUTO.....	118
Differences Between PLATFORM and AUTO.....	119
Field and Base Alignment.....	119
Base Alignment.....	119
Structure Alignment Examples.....	120
Array Alignment in Structures.....	122
Structure Alignment.....	123
Substructure Alignment.....	124
Alignment Considerations for Substructures.....	126
FIELDALIGN Clause.....	127
FIELDALIGN Compiler Directive.....	127
SHARED2 Parameter.....	128
SHARED8 Parameter.....	129
Alignment of Fields.....	131
Optimizing Structure Layouts.....	131
Structure Length.....	132
Alignment of UNSIGNED(17-31) Fields.....	133
Reference Alignment With Structure Pointers.....	134
REALIGNED Clause.....	134
Default Reference Alignment.....	135
REALIGNED(2).....	135
REALIGNED(8).....	136
Code Generation for Structure References.....	137
STRUCTALIGN (MAXALIGN) Attribute.....	137
VOLATILE Attribute.....	138
Declaring Definition Structures.....	138
Declaring Template Structures.....	139
Declaring Referral Structures.....	141
Declaring Simple Variables in Structures.....	142
Declaring Arrays in Structures.....	143
Declaring Substructures.....	144
Definition Substructures.....	144
Referral Substructures.....	146
Declaring Filler.....	147

Declaring Simple Pointers in Structures.....	148
Using Simple Pointers.....	149
Assigning Addresses to Pointers in Structures.....	150
Declaring Structure Pointers in Structures.....	151
Declaring Redefinitions.....	153
Simple Variable.....	153
Array.....	154
Definition Substructure.....	155
Referral Substructure.....	157
Simple Pointer.....	158
Structure Pointer.....	159
10 Pointers.....	161
Overview of Pointer Declaration.....	161
Declaring VOLATILE Pointers.....	163
Simple.....	163
Structure.....	164
Address Types.....	164
BADDR and WADDR	167
SGBADDR, SGWADDR, SGXBADDR, and SGXWADDR (System Globals)	167
PROCADDR, PROC32ADDR, and PROC64ADDR (Procedures, Procedure Pointers, and Procedure Entry Points)	168
Subprocedures, Subprocedure Entry Points, Labels, and Read-Only Arrays (CBADDR and CWADDR Address Types).....	169
EXTADDR, EXT32ADDR, and EXT64ADDR (Extended Addresses).....	169
Declaring Simple Pointers.....	170
Initializing Simple Pointers.....	172
Declaring Structure Pointers.....	173
Initializing Structure Pointers.....	174
Declaring System Global Pointers.....	176
11 Equivalenced Variables.....	177
Declaring Equivalenced Variables.....	178
Memory Allocation.....	179
Declaring Nonstructure Equivalenced Variables.....	180
Memory Usage for Nonstructured Equivalenced Variables.....	181
Equivalenced Arrays.....	181
Indirect Arrays.....	182
Equivalenced Simple Variables.....	182
Equivalenced Simple Pointers.....	183
Equivalencing Procedure Addresses (PROCADDR, PROC32ADDR, and PROC64ADDR) and Pointer Variables (PROCPTR, PROC32PTR, and PROC64PTR).....	187
Declaring Equivalenced Definition Structures.....	188
Structure Variants.....	191
Memory Usage for Structured Equivalenced Variables.....	192
FIELDALIGN Clause.....	193
System Global Equivalenced Variable Declarations.....	193
Equivalenced Simple Variable.....	193
Equivalenced Definition Structure.....	194
Equivalenced Referral Structure.....	195
Equivalenced Simple Pointer.....	196
Equivalenced Structure Pointer.....	197
12 Statements.....	199
Using Semicolons in Statements.....	199
Compound Statements.....	200

ASSERT.....	200
Assignment.....	201
Pointer Assignment.....	203
Assigning Numbers to FIXED Variables.....	203
Assigning Character Strings.....	203
Examples.....	203
Bit-Deposit Assignment.....	204
CALL.....	205
CASE.....	207
Empty CASE.....	207
Labeled CASE.....	207
Unlabeled CASE.....	209
DO-UNTIL.....	210
DROP.....	212
Dropping Labels.....	212
Dropping Temporary Variables.....	212
FOR.....	212
Nested.....	213
Standard.....	214
Optimized.....	214
GOTO.....	215
Local.....	215
Nonlocal.....	215
GOTO and Target Statements With Different Trapping States.....	216
IF.....	217
Testing Address Types.....	218
Testing Hardware Indicators.....	218
Move.....	218
Destination Shorter Than Source.....	222
\$FILL8, \$FILL16, and \$FILL32 Statements.....	222
RETURN.....	223
Functions.....	224
Procedures and Subprocedures.....	225
Condition Codes.....	225
SCAN and RSCAN.....	228
Determining What Stopped a Scan.....	230
Extended Pointers.....	230
Crossing Variable Boundaries.....	230
P-Relative Arrays.....	231
USE.....	232
WHILE.....	232
13 Hardware Indicators.....	234
Managing Overflow Traps.....	234
[NO]OVERFLOW_TRAPS Procedure Attribute.....	234
[EN DIS]ABLE_OVERFLOW_TRAPS Block Attribute.....	235
Hardware Indicators After Assignments.....	236
\$OVERFLOW.....	236
\$CARRY.....	236
Condition Codes.....	237
Hardware Indicators in Conditional Expressions.....	239
Nesting Condition Code Tests.....	242
Using Hardware Indicators Across Procedures.....	244
Testing a Hardware Indicator Set in the Calling Procedure.....	244
Returning a Condition Code to the Calling Procedure.....	244

Returning the Value of \$OVERFLOW or \$CARRY to the Calling Procedure.....	245
14 Procedures, Subprocedures, and Procedure Pointers.....	246
Procedure Declarations.....	246
Procedure Attributes.....	248
Parameters and VARIABLE and EXTENSIBLE Procedures.....	250
VARIABLE, EXTENSIBLE and RETURNSCC Procedures as Actual Parameters.....	251
Formal Parameter Specification.....	251
Using STRUCT as a Formal Parameter.....	255
Passing an Extended Address Parameter to a Non-EXTENDED Reference Parameter.....	255
Using the PROC Formal Parameter.....	256
Referencing Parameters.....	256
Procedure Body.....	256
Subprocedure Declarations.....	257
Subprocedure Body.....	259
Entry-Point Declarations.....	260
Procedure Entry-Point Identifiers.....	260
Subprocedure Entry-Point Identifiers.....	262
Procedure Pointers.....	263
Declaring Procedure Pointer Variables.....	266
Declaring Procedure Pointers in Structures.....	267
Declaring PROCPTRs as Formal Parameters.....	268
Assignments to Procedure Pointers.....	269
Dynamically Selected Procedure Calls.....	271
Labels in Procedures.....	273
15 Built-In Routines.....	274
Privileged Mode.....	274
Parameters.....	275
Addresses as Parameters.....	275
Expressions as Parameters.....	275
Hardware Indicators.....	276
Atomic Operations	276
\$ATOMIC_ADD.....	276
\$ATOMIC_AND.....	277
\$ATOMIC_DEP.....	278
\$ATOMIC_GET.....	279
\$ATOMIC_OR.....	280
\$ATOMIC_PUT.....	280
Nonatomic Operations	281
pTAL Privileged Routines.....	281
Type-Conversion Routines.....	282
Address-Conversion Routines.....	283
Character-Test Routines.....	284
Minimum and Maximum Routines.....	285
Arithmetic Routines.....	285
Carry and Overflow Routines.....	285
FIXED-Expression Routines.....	285
Variable-Characteristic Routines.....	285
Procedure-Parameter Routines.....	286
Miscellaneous Routines.....	286
\$ABS.....	291
\$ALPHA.....	291
\$ASCIITOFIXED.....	292
\$AXADR.....	293
\$BADDR_TO_EXTADDR.....	294

\$BADDR_TO_WADDR.....	294
\$BITLENGTH.....	295
\$BITOFFSET.....	296
\$CARRY.....	297
\$CHECKSUM.....	297
\$COMP.....	298
\$COUNTDUPS.....	299
\$DBL.....	300
\$DBLL.....	301
\$DBLR.....	301
\$DFIX.....	302
\$EFLT.....	302
\$EFLTR.....	303
\$EXCHANGE.....	303
\$EXECUTEIO.....	304
\$EXTADDR_TO_BADDR.....	305
\$EXTADDR_TO_WADDR.....	306
\$EXT64ADDR_TO_EXTADDR.....	306
\$EXT64ADDR_TO_EXT32ADDR.....	307
\$EXT64ADDR_TO_EXT32ADDR_OV	307
\$EXTADDR_TO_EXT64ADDR	308
\$FILL8, \$FILL16, and \$FILL32	308
\$FIX.....	309
\$FIXD.....	309
\$FIXED0_TO_EXT64ADDR.....	310
\$FIXEDTOASCII.....	310
\$FIXEDTOASCIIRESIDUE.....	311
\$FIXI.....	312
\$FIXL.....	312
\$FIXR.....	313
\$FLT.....	314
\$FLTR.....	314
\$FREEZE.....	315
\$HALT.....	315
\$HIGH.....	315
\$IFIX.....	316
\$INT.....	317
\$INT_OV.....	318
\$INTERROGATEHIO.....	318
\$INTERROGATEIO.....	320
\$INTR.....	321
\$IS_32BIT_ADDR	321
\$LEN.....	322
\$LFIX.....	323
\$LMAX.....	323
\$LMIN.....	324
\$LOCATESPTHDR.....	324
\$LOCKPAGE.....	325
\$MAX.....	326
\$MIN.....	327
\$MOVEANDCXSUMBYTES.....	327
\$MOVENONDUP.....	328
\$NUMERIC.....	329
\$OCCURS.....	330
\$OFFSET.....	332

\$OPTIONAL.....	333
\$OVERFLOW.....	335
\$PARAM.....	336
\$POINT.....	336
\$PROCADDR.....	337
\$PROC32ADDR.....	337
\$PROC64ADDR.....	338
\$READBASELIMIT.....	338
\$READCLOCK.....	339
\$READSPT.....	339
\$READTIME.....	340
\$SCALE.....	340
\$SGBADDR_TO_EXTADDR.....	341
\$SGBADDR_TO_SGWADDR.....	342
\$SGWADDR_TO_EXTADDR.....	342
\$SGWADDR_TO_SGBADDR.....	343
\$SPECIAL.....	343
\$STACK_ALLOCATE.....	344
\$TRIGGER.....	345
\$TYPE.....	345
\$UDBL.....	346
\$UDIVREM16.....	347
\$UDIVREM32.....	348
\$UFIX	349
\$UNLOCKPAGE.....	349
\$WADDR_TO_BADDR.....	350
\$WADDR_TO_EXTADDR.....	350
\$WRITEPTE.....	351
\$XADR.....	352
\$XADR32.....	352
\$XADR64.....	353
16 Compiling and Linking pTAL Programs.....	355
Compiling Source Files.....	355
Input Files.....	356
Output Files.....	356
Running the Compiler.....	357
Completion Codes Returned by the Compiler.....	358
Linking Object Files.....	358
Creating a Dynamic Linked Library (DLL).....	362
Compiling With Global Data Blocks.....	362
Declaring Global Data.....	362
Allocating Global Data Blocks.....	365
Address Assignments.....	365
Sharing Global Data Blocks.....	365
Compiling With Saved Global Data.....	366
Using the Code Profiling Utilities.....	366
17 Compiler Directives.....	367
Specifying Compiler Directives.....	367
Compilation Command.....	367
Directive Line.....	367
File Names as Compiler Directive Arguments.....	368
Directive Stacks.....	369
Pushing Directive Settings.....	369
Popping Directive Settings.....	369

Example.....	369
Toggles.....	370
Named Toggles.....	370
Numeric Toggles.....	370
Examples.....	371
Saving and Using Global Data Declarations.....	372
Saving Global Data Declarations.....	373
Retrieving Global Data Declarations.....	374
Examples.....	374
Migrating from TNS/R to TNS/E.....	375
Summary of Compiler Directives.....	377
ASSERTION.....	381
BASENAME.....	381
BEGINCOMPILE.....	382
BLOCKGLOBALS.....	382
CALL_SHARED.....	383
CHECKSHIFTCOUNT.....	384
CODECOV.....	385
COLUMNS.....	385
DEFEXPAND.....	386
DEFINETOG.....	388
DO_TNS_SYNTAX.....	389
ENDIF.....	390
ERRORFILE.....	391
ERRORS.....	393
EXPORT_GLOBALS.....	393
__EXT64.....	394
FIELDALIGN.....	395
FMAP.....	396
GLOBALIZED.....	396
GMAP.....	397
GP_OK.....	397
IF and IFNOT.....	398
INNERLIST.....	400
INVALID_FOR_PTAL.....	401
LINES.....	401
LIST.....	401
MAP.....	402
OPTIMIZE.....	404
OPTIMIZEFILE.....	404
OVERFLOW_TRAPS.....	406
PAGE.....	407
PRINTSYM.....	408
PROFDIR.....	408
PROFGEN.....	409
PROFUSE.....	409
REALIGNED.....	410
RESETTOG.....	411
ROUND.....	412
SAVEGLOBALS.....	413
SECTION.....	414
SETTOG.....	415
SOURCE.....	416
Section Names.....	417
Nesting Levels.....	418

Effect of Other Directives.....	418
Including System Procedure Declarations.....	419
Examples.....	419
SRL.....	420
SUPPRESS.....	420
SYMBOLS.....	421
SYNTAX.....	422
TARGET.....	423
USEGLOBALS.....	423
WARN.....	424
18 pTAL Cross Compiler.....	426
NonStop pTAL (ETK).....	426
pTAL or EpTAL (PC Command Line).....	427
Compilation and Linking.....	429
Debugging.....	429
Tools and Utilities.....	430
NonStop ar Utility.....	430
TACL DEFINE Tool (ETK).....	431
PC-to-NonStop-Host Transfer Tools.....	431
Documentation.....	431
A Syntax Summary.....	432
Data Types.....	432
Constants.....	432
Character String.....	432
STRING Numeric.....	432
INT Numeric.....	433
INT(32) Numeric.....	433
FIXED Numeric.....	433
REAL and REAL(64) Numeric.....	433
Constant List.....	434
Expressions.....	434
Arithmetic.....	434
Conditional.....	435
Assignment.....	435
CASE.....	435
IF.....	435
Group Comparison.....	435
Bit Extraction.....	436
Bit Shift.....	436
Declarations.....	436
LITERAL.....	436
DEFINE.....	436
Simple Variable.....	437
Array.....	437
Read-Only Array.....	438
Structures.....	438
Redefinition.....	442
Pointer.....	444
Equivalenced Variable.....	445
Procedure and Subprocedure.....	449
Statements.....	455
Compound.....	456
ASSERT.....	456
Assignment.....	456

Bit Deposit Assignment.....	456
CALL.....	457
Labeled CASE.....	457
Unlabeled CASE.....	457
DO-UNTIL.....	458
DROP.....	458
FOR.....	458
GOTO.....	458
IF.....	458
Move.....	459
RETURN.....	459
SCAN and RSCAN.....	459
USE.....	459
WHILE.....	460
Overflow Traps.....	460
OVERFLOW_TRAPS Directive.....	460
[EN DIS]ABLE_OVERFLOW_TRAPS Block Attribute.....	460
Built-in Routines.....	460
Atomic.....	460
Nonatomic.....	462
Compiler Directives.....	494
Directive Line.....	495
ASSERTION.....	495
BASENAME.....	495
BEGINCOMPILATION.....	496
BLOCKGLOBALS.....	496
CALL_SHARED.....	496
CHECKSHIFTCOUNT.....	497
CODECOV.....	497
COLUMNS.....	498
DEFEXPAND.....	498
DEFINETOG.....	499
DO_TNS_SYNTAX.....	500
ENDIF.....	500
ERRORFILE.....	500
ERRORS.....	500
EXPORT_GLOBALS.....	501
__EXT64.....	501
FIELDALIGN.....	502
FMAP.....	502
GLOBALIZED.....	502
GMAP.....	503
GP_OK.....	503
IF, IFNOT, and ENDIF.....	504
INNERLIST.....	505
INVALID_FOR_PTAL.....	505
LINES.....	506
LIST.....	506
MAP.....	506
OPTIMIZE.....	507
OPTIMIZEFILE.....	507
OVERFLOW_TRAPS.....	508
PAGE.....	508
PRINTSYM.....	509
PROFDIR.....	509

PROFGEN.....	509
PROFUSE.....	510
REFALIGNED.....	510
RESETTOG.....	511
ROUND.....	512
SAVEGLOBALS.....	512
SECTION.....	513
SETTOG.....	513
SOURCE.....	514
SRL.....	514
SUPPRESS.....	515
SYMBOLS.....	515
SYNTAX.....	516
TARGET.....	516
USEGLOBALS.....	516
WARN.....	517
B Disk File Names and HP TACL Commands.....	518
Disk File Names.....	518
Parts of a Disk File Name.....	518
Partial File Names.....	519
Logical File Names.....	520
Internal File Names.....	520
HP TACL Commands.....	520
DEFINE.....	521
PARAM SWAPVOL.....	522
ASSIGN.....	522
C Differences Between the pTAL and EpTAL Compilers.....	525
General.....	525
Data Types and Alignment.....	525
Routines.....	525
Compiler Directives.....	527
D RETURN, RETURNSCC, and C/C++ on TNS/E.....	528
E 64-bit Addressing Functionality.....	531
Address Types.....	531
EXT32ADDR.....	531
EXT64ADDR.....	531
PROC32ADDR.....	531
PROC64ADDR.....	531
Procedure Pointer Types.....	531
PROC32PTR.....	531
PROC64PTR.....	531
Indirection Symbols.....	532
.EXT32.....	532
.EXT64.....	532
Built-in Routines.....	532
\$EXT64ADDR_TO_EXTADDR.....	532
\$EXT64ADDR_TO_EXT32ADDR.....	532
\$EXT64ADDR_TO_EXT32ADDR_OV.....	532
\$EXTADDR_TO_EXT64ADDR.....	532
\$FIXED0_TO_EXT64ADDR.....	532
\$FIX.....	532
\$IS_32BIT_ADDR.....	532
\$PROCADDR.....	533

\$PROC32ADDR.....	533
\$PROC64ADDR.....	533
\$UFIX.....	533
\$XADR.....	533
\$XADR32.....	533
\$XADR64.....	533
Implicitly Defined Compilation Toggle __EXT64.....	534
Directives.....	534
__EXT64.....	534
DEFINETO, RESETTO, and SETTO.....	534
IF and IFNOT.....	534
Implicit Address Conversions.....	534
Index.....	536

About This Document

The Portable Transaction Application Language for HP NonStop systems (pTAL) is a high-level, block-structured language used to write systems software and transaction-oriented applications. This manual gives guidelines for using the pTAL language and the EpTAL and pTAL compilers, including:

- How to create, structure, compile, and run a pTAL program
- The process environment, addressing modes, and storage allocation
- How to declare and access procedures and variables

You can compile pTAL source programs with either the pTAL compiler or the EpTAL compiler (for their differences, see [Figure 16 \(page 361\)](#)).

In this manual:

Word	Meaning (unless otherwise specified)
compiler	The pTAL and EpTAL compilers
linker	The <code>nld</code> , <code>ld</code> , and <code>eld</code> linkers

Supported Release Version Updates (RVUs)

This manual supports D44.00 and all subsequent D-series RVUs, all J-series, H-series, and G-series RVUs, unless otherwise indicated by its replacement publication.

Intended Audience

This manual is intended for system programmers and application programmers familiar with NonStop systems.

New and Changed Information

Changes to this manual are itemized for each RVU.

New and Changed Information for 523746–009

- Added a new Appendix E, [64-bit Addressing Functionality \(page 531\)](#).
- Added the following new [Address Types](#):
 - EXT32ADDR
 - EXT64ADDR
 - PROC32ADDR
 - PROC64ADDR
- Added the following new [Procedure Pointers](#):
 - PROC32PTR
 - PROC64PTR
- Added a new 64-bit directive, [__EXT64 \(page 394\)](#) in the chapter “Compiler Directives”.
- Added the following 64-bit built-in routines:
 - [\\$EXT64ADDR_TO_EXTADDR \(page 306\)](#)
 - [\\$EXT64ADDR_TO_EXT32ADDR \(page 307\)](#)
 - [\\$EXT64ADDR_TO_EXT32ADDR_OV \(page 307\)](#)

- [\\$EXTADDR_TO_EXT64ADDR](#) (page 308)
- [\\$FIXED0_TO_EXT64ADDR](#) (page 310)
- [\\$IS_32BIT_ADDR](#) (page 321)
- [\\$PROC32ADDR](#) (page 337)
- [\\$PROC64ADDR](#) (page 338)
- [\\$UFIX](#) (page 349)
- [\\$XADR32](#) (page 352)
- [\\$XADR64](#) (page 353)
- Updated the following directives:
 - [DEFINETO](#) (page 388)
 - [ENDIF](#) (page 390)
 - [IF](#) and [IFNOT](#) (page 398)
 - [RESETTO](#) (page 411)
 - [SETTO](#) (page 415)
- Updated the following built-in routines:
 - [\\$INT](#) (page 317)
 - [\\$PROCADDR](#) (page 337)
 - [\\$XADR](#) (page 352)
- Updated the following sections with 64-bit addressing functionality:
 - [pTAL and TAL Compatibility](#) (page 30)
 - [Typed Integer Constants](#) (page 44)
 - [Converting Between Address Types and Numeric Data Types](#) (page 51)
 - [Using Arithmetic Operations to Adjust Addresses](#) (page 55)
 - [Comparing Addresses to Addresses](#) (page 56)
 - [Extended Addresses](#) (page 77)
 - [Initializing Simple Variables With Character Strings](#) (page 105)
 - [Assigning Addresses to Pointers in Structures](#) (page 150)
 - [Overview of Pointer Declaration](#) (page 161)
 - [PROCADDR, PROC32ADDR, and PROC64ADDR \(Procedures, Procedure Pointers, and Procedure Entry Points\)](#) (page 168)
 - [EXTADDR, EXT32ADDR, and EXT64ADDR \(Extended Addresses\)](#) (page 169)
 - [Declaring Simple Pointers](#) (page 170)
 - [EXTADDR, EXT32ADDR, and EXT64ADDR Declarations](#) (page 170)
 - [Equivalencing Procedure Addresses \(PROCADDR, PROC32ADDR, and PROC64ADDR\) and Pointer Variables \(PROCPTR, PROC32PTR, and PROC64PTR\)](#) (page 187)
 - [Extended Pointers](#) (page 230)
 - [Passing an Extended Address Parameter to a Non-EXTENDED Reference Parameter](#) (page 255)
 - [Procedure Pointers](#) (page 263)
 - [Declaring Procedure Pointer Variables](#) (page 266)

- [Declaring Procedure Pointers in Structures \(page 267\)](#)
- [Assignments to Procedure Pointers \(page 269\)](#)
- [Syntax Summary \(page 432\)](#)
- [Differences Between the pTAL and EpTAL Compilers \(page 525\)](#)
- Updated the following tables with 64-bit addressing functionality:
 - [Data Types \(page 33\)](#)
 - [Reserved Keywords \(page 37\)](#)
 - [Base Address Symbols \(page 40\)](#)
 - [Indirection Symbols \(page 41\)](#)
 - [Data Types and Their Address Types \(page 49\)](#)
 - [Valid Address Conversions \(page 53\)](#)
 - [Expressions \(page 69\)](#)
 - [Valid Address Expressions \(page 79\)](#)
 - [Signed Relational Operators \(page 83\)](#)
 - [Addresses in Simple Pointers \(page 149\)](#)
 - [Address Types \(page 165\)](#)
 - [Object Data Types and Their Addresses \(page 166\)](#)
 - [Valid Equivalenced Variable Declarations \(page 178\)](#)
 - [Data Types for Equivalenced Variables \(page 185\)](#)
 - [Formal Parameter Specification \(page 254\)](#)
 - [Type-Conversion Routines \(page 282\)](#)
 - [Built-In Address-Conversion Routines \(page 283\)](#)
 - [Built-In Routines for Nonatomic Operations \(page 286\)](#)
 - [Compiler Directives by Category \(page 377\)](#)
 - [Compiler Directives by Name \(page 379\)](#)
 - [Data Types and Alignment \(page 525\)](#)

New and Changed Information for 523746–008

- Added a caution under [Debugging \(page 429\)](#) on the manner in which the [CODECOV \(page 385\)](#) command line option interacts when you are debugging an instrumented application. Under CODECOV, placed a reference to this caution in the Debugging section.
- Changed [\\$INT \(page 317\)](#) to indicate that overflow can occur for INT(64).
- Added new [Document Organization \(page 19\)](#) section to the manual.
- Added J-series to [Supported Release Version Updates \(RVUs\) \(page 16\)](#).

New and Changed Information for 523746–007

- In [Chapter 16: Compiling and Linking pTAL Programs \(page 355\)](#), updated the overview of the Code Profiling Utilities to include the profile-guided optimization capability.
- In [Chapter 17: Compiler Directives \(page 367\)](#), added descriptions and syntax of the following directive:
 - BASENAME directive (Guardian) and -basename directive (Windows)
 - PROFDIR directive (Guardian) and -profdir directive (Windows)
 - PROFGEN directive (Guardian) and -profdir directive (Windows)
 - PROFUSE directive (Guardian) and -profuse directive (Windows)
- In [Appendix A: Syntax Summary \(page 432\)](#), added syntax descriptions of the preceding directives.

Document Organization

This document is organized as follows:

Table 1 Summary of Contents

Chapter	This chapter . . .
Chapter 1: Introduction to pTAL	Describes the differences between pTAL and TAL, and the applications, features, system services and procedures of pTAL.
Chapter 2: Language Elements	Describes pTAL language elements, such as character set, keywords, delimiters, operators, symbols, declarations, constants, and statements.
Chapter 3: Data Representation	Describes pTAL variables and constants, including data types and address types.
Chapter 4: Data Alignment	Describes how data items are aligned; covers the misalignment tracing facility and misalignment handling.
Chapter 5: Expressions	Describes expressions. An expression is a sequence of operands and operators that produces a single value. Operands in an expression include variables, constants, and routine identifiers. Operators in an expression perform arithmetic or conditional operations on the operands. pTAL supports arithmetic, address, constant, and conditional expressions.
Chapter 6: LITERALS and DEFINES	Describes how to declare LITERALS and DEFINES and refer to them throughout the program. A LITERAL declaration associates identifiers with constant values. A DEFINE declaration associates identifiers and parameters with text.
Chapter 7: Simple Variables	Describes the syntax for declaring simple variables. A simple variable is a single-element data item of a specified data type that is not an array, a structure, or a pointer.
Chapter 8: Arrays	Describes the syntax for declaring arrays. An array is a one-dimensional set of elements of the same data type.
Chapter 9: Structures	Describes structures. A structure is a collectively stored set of data items that you can access individually or as a group.
Chapter 10: Pointers	Describes the syntax for declaring and initializing pointers you manage yourself.

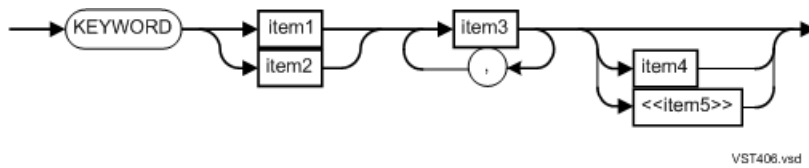
Table 1 Summary of Contents *(continued)*

Chapter	This chapter . . .
Chapter 11: Equivalenced Variables	Describes equivalenced variables. Equivalencing lets you declare more than one identifier and description for a location in a storage area.
Chapter 12: Statements	Describes statements. Statements — also known as executable statements — perform operations in a program. They can modify the program's data or control the program's flow.
Chapter 13: Hardware Indicators	Describes hardware indicators. Includes managing overflow traps, hardware indicators after assignments, hardware indicators in conditional expressions, nesting condition code tests, and using hardware indicators across procedures.
Chapter 14: Procedures, Subprocedures, and Procedure Pointers	Describes procedures, which are program units that contain the executable portions of a pTAL program and that are callable from anywhere in the program.
Chapter 15: Built-In Routines	Describes built-in routine calls whose results do not depend on the values of variables and can be used wherever constant values are allowed.
Chapter 16: Compiling and Linking pTAL Programs	Describes how to compile and link pTAL programs. Input to the compiler is a source file containing pTAL source text (such as data declarations, statements, compiler directives, and comments). Output from the compiler is a linkfile consisting of relocatable code and data blocks. To produce an executable pTAL program, you link one or more linkfiles into a single loadfile.
Chapter 17: Compiler Directives	Describes how to specify compiler directives. You can specify compiler directives either in the compilation command or in a directive line in the source code, unless otherwise specified. The compiler interprets and processes each directive at the point of occurrence.
Chapter 18: pTAL Cross Compiler	Describes the optional pTAL cross compiler that runs on PC platforms.
Appendix A: Syntax Summary	Provides a summary of syntax for data types, constants, expressions, declarations, statements, overflow traps, built-in routines, and compiler directives.
Appendix B: Disk File Names and HP TACL Commands	For Guardian platforms only, describes disk file names and HP TACL commands.
Appendix C: Differences Between the pTAL and EpTAL Compilers	Describes the differences between the pTAL and EpTAL compilers.
Appendix D: RETURN, RETURNSCC, and C/C++ on TNS/E	Describes RETURN, RETURNSCC, and C/C++ on TNS/E. Read this appendix if you write or call pTAL procedures that return both a traditional function value by means of the RETURN statement and an unrelated condition code value by means of the RETURNSCC attribute.

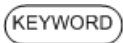


Notation Conventions

Syntax Diagram Conventions



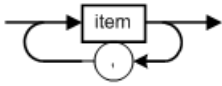
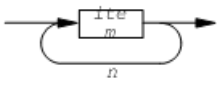
This manual presents syntax in railroad diagrams. Here is a generic railroad diagram:



To use a railroad diagram, follow the direction of the arrows and specify syntactic items as indicated by the diagram pieces:

Diagram Piece	Meaning
 VST412.vsd	Type KEYWORD as shown. You can type letters in uppercase or lowercase.
 VST413.vsd	Replace <i>item</i> with a value that fits its description, which follows the syntax diagram.
 VST414.vsd	Type content (punctuation mark, symbol, or letter) as shown. You can type a letter in uppercase or lowercase.

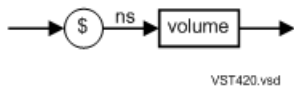
Some examples of the meanings of simple diagrams are:

Diagram Piece	Meaning
 VST407.vsd	Choose <i>item1</i> or <i>item2</i> .
 VST408.vsd	Choose <i>item1</i> , <i>item2</i> , or neither.
 VST409.vsd	Specify <i>item</i> one or more times, separating occurrences with commas.
 VST742.vsd	Specify <i>item</i> at most <i>n</i> times.

NOTE: To refer to a particular railroad diagram or figure when giving feedback to HP, use the number at the bottom right corner of that railroad diagram or figure (for example, VST742.vsd).

Spacing rules are:

- If the arrow between two diagram pieces is labelled “ns,” put no spaces between the syntactic items that they represent. For example:



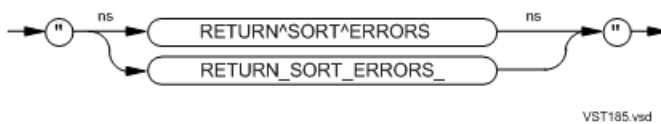
means that you type:

\$NEWVOL

not

\$ NEWVOL

- An “ns” on the top line of a choice structure applies to the lower lines in the choice structure as well. For example:



means that you type one of the following:

" ^RETURN^SORT^ERRORS "

"RETURN_SORT_ERRORS_ "

- If two diagram pieces are not separated by a separator character (such as a comma, semicolon, or parenthesis), separate the syntactic items that they represent by at least one space or a new line. For example:



means that you type:

MU<IPLY 3 4

not

MU<IPLY34

- If two diagram pieces are separated by a separator character, separating the syntactic items that they represent by spaces is optional. For example:



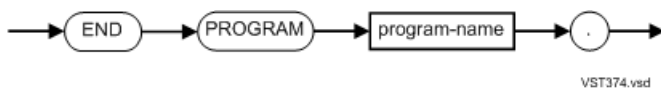
means that you type:

MU<IPLY 3,4

or

MU<IPLY 3, 4

- If a diagram piece is immediately followed by a period, putting spaces between the syntactic item and the period is optional. For example:



means that you can type:

END PROGRAM SORT.

or

END PROGRAM SORT .

- Diagram elements need not be on the same line. For example:



BLOCK DATA BEGIN

is equivalent to:

```
BLOCK DATA
BEGIN
```

- Explicit spacing rules given for individual railroad diagrams override the aforementioned rules.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS

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

MAXATTACH

Italic Letters

Italic letters, regardless of font, indicate variable items that you supply. Items not enclosed in brackets are required. For example:

file-name

Computer Type

Computer type letters indicate:

- C and Open System Services (OSS) keywords, commands, and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

Use the `cextdecs.h` header file.

- Text displayed by the computer. For example:

Last Logon: 14 May 2006, 08:02:23

- A listing of computer code. For example

```
if (listen(sock, 1) < 0)
{
    perror("Listen Error");
    exit(-1);
}
```

Bold Text

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

ENTER RUN CODE

?**123**

CODE RECEIVED: 123.00

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

[] Brackets

Brackets enclose optional syntax items. For example:

TERM [*\system-name.*]*\$terminal-name*

INT[ERRUPTS]

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

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

```
K [ X | D ] address
```

{ } Braces

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

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

```
ALLOWSU { ON | OFF }
```

| Vertical Line

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

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

... Ellipsis

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

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

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

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

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

Punctuation

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

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

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

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

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

Item Spacing

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

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

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

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

Line Spacing

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


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

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

!i and !o

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

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

!i,o

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

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

!i:i

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

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

!o:i

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

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

Notation for Messages

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

Bold Text

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

```
ENTER RUN CODE
```

```
?123
```

```
CODE RECEIVED:      123.00
```

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

Nonitalic Text

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

```
Backup Up.
```

Italic Text

Italic text indicates variable items whose values are displayed or returned. For example:

```
p-register
```

```
process-name
```

[] Brackets

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

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

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

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

{ } Braces

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

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

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

| Vertical Line

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

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

% Percent Sign

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

```
%005400
```

```
%B101111
```

```
%H2F
```

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

Notation for Management Programming Interfaces

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

UPPERCASE LETTERS

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

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

lowercase letters

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

```
token-type
```

!r

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

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

!o

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

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

Related Information

- [Table 2 \(page 27\)](#)
- [Table 3 \(page 27\)](#)

- [Table 4 \(page 27\)](#)
- [Table 5 \(page 28\)](#)

Table 2 Related Manuals

Manual	Description
<i>pTAL Conversion Guide</i>	Provides information needed to convert TAL programs to pTAL programs.
<i>pTAL Guidelines for TAL Programmers</i>	Gives guidelines for writing TAL code that you can migrate later to pTAL code with as few changes as possible.
<i>TAL Programmer's Guide</i>	Helps you get started in creating, structuring, compiling, running and debugging programs. Explains how to declare and access procedures and variables and how the TAL compiler allocates storage for variables.
<i>TAL Reference Manual</i>	Describes the syntax for declaring variables and procedures and for specifying expressions, statements, built-in routines, and compiler directives. Lists error and warning messages.
<i>TAL Reference Summary</i>	Summarizes the TAL syntax diagrams.

Table 3 System Manuals

Manual	Description
<i>D-Series System Migration Planning Guide</i>	Gives guidelines for migrating from a C-series system to a D-series system
<i>Introduction to D-Series Systems</i>	Provides an overview of D-series enhancements to the Guardian operating system
<i>Introduction to Tandem NonStop Systems</i>	Provides an overview of the system hardware and software
<i>TACL Reference Manual</i>	Describes the syntax of HP TACL

Table 4 Programming Manuals

Manual	Description
<i>C/C++ Programmer's Guide</i>	Contains information that you need about HP C and C++ for NonStop systems if you plan to call HP C and C++ routines from pTAL programs
<i>COBOL Manual for TNS and TNS/R Programs</i>	Contains information that you need about HP COBOL for TNS and TNS/R programs if you plan to call HP COBOL routines from pTAL programs
<i>COBOL Manual for TNS/E Programs</i>	Contains information that you need about HP COBOL for TNS/E programs if you plan to call HP COBOL routines from pTAL programs
<i>CRE Programmer's Guide</i>	Explains how to use the Common Runtime Environment (CRE) for running mixed-language programs
<i>Guardian Application Conversion Guide</i>	Gives guidelines for converting C-series TNS programs to D-series TNS programs, and for converting TNS programs to TNS/R programs
<i>Guardian Procedure Calls Reference Manual</i>	Describes the syntax and programming considerations for using system procedures
<i>Guardian Procedure Errors and Messages Manual</i>	Describes error codes, error lists, system messages, and trap numbers for system procedures

Table 4 Programming Manuals *(continued)*

Manual	Description
<i>Guardian Programmer's Guide</i>	Explains how to use the programmatic interface of the operating system
<i>H-Series Application Migration Guide</i>	Explains how to migrate programs from TNS/R to TNS/E
<i>TAL Programmer's Guide</i>	Contains information that you need about the HP Transaction Application Language (TAL) if you plan to call TAL routines from TNS HP COBOL programs
<i>TAL Programmer's Guide Data Alignment Addendum</i>	Documents the data alignment requirements of TAL

Table 5 Program Development Manuals

Manual	Description
<i>Accelerator Manual</i>	Explains how to accelerate TNS object files for a TNS/R system
<i>Accelerator Manual Data Alignment Addendum</i>	Documents the data alignment requirements of the Accelerator
<i>Binder Manual</i>	Explains how to bind TNS compilation units (or modules) using Binder
<i>Code Profiling Utilities Manual</i>	Explains how to use the Code Coverage Utilities to perform profile-guided optimization and to generate code coverage reports.
<i>CROSSREF Manual</i>	Explains how to collect cross-reference information using the stand-alone Crossref product
<i>Debug Manual</i>	Explains how to debug programs using the Debug machine-level interactive debugger
<i>DLL Programmer's Guide for TNS/E Systems</i>	Explains position-independent code (PIC) and dynamic-link libraries (DLLs) on TNS/E systems
<i>DLL Programmer's Guide for TNS/R Systems</i>	Explains position-independent code (PIC) and dynamic-link libraries (DLLs) on TNS/R systems
<i>Edit User's Guide and Reference Manual</i>	Explains how to create and edit a text file using the Edit line and virtual-screen text editor
<i>eld Manual</i>	Explains how to use the <code>eld</code> utility to link and change the attributes of TNS/E object files
<i>EMS Manual</i>	Describes the Event Management Service (EMS). The misalignment tracing facility generates EMS events (see Misalignment Tracing Facility (page 66))
<i>enofit Manual</i>	Explains how to use the <code>enofit</code> utility to display TNS/E object files
<i>Inspect Manual</i>	Explains how to debug programs using the Inspect source-level and machine-level interactive debugger
<i>ld Manual</i>	Explains how to use the <code>ld</code> utility to link and change the attributes of TNS/R PIC object files
<i>Native Inspect Manual</i>	Explains how to debug programs using the Native Inspect source-level and machine-level interactive debugger
<i>nld Manual</i>	Explains how to use the <code>nld</code> utility to link and change the attributes of TNS/R non-PIC object files and how the <code>ar</code> utility works

Table 5 Program Development Manuals *(continued)*

Manual	Description
<i>noft Manual</i>	Explains how to use the <code>noft</code> utility to display TNS/R object files (PIC and non-PIC)
<i>Object Code Accelerator Manual</i>	Explains how to accelerate TNS and TNS/R object files for a TNS/E system.
<i>PS Text Edit Reference Manual</i>	Explains how to create and edit a text file using the PS Text Edit full-screen text editor
<i>SCF Reference Manual for the Kernel Subsystem</i>	Describes the Subsystem Control Facility (SCF), whose user interface you can use to control tracing (see Misalignment Tracing Facility (page 66))
Visual Inspect Online Help	Explains how to debug programs using the Visual Inspect source-level and machine-level interactive debugger
<i>Code Coverage Tool Reference Manual</i>	

Publishing History

Part Number	Product Version	Publication Date
523746-005	pTAL D44, EpTAL H01	July 2005
523746-006	pTAL D44, EpTAL H01	November 2006
523746-007	pTAL D44, EpTAL H01	February 2007
523746-008	pTAL D44, EpTAL H01	May 2009
523746-009	pTAL D44, EpTAL H01	February 2012

HP Encourages Your Comments

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

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

1 Introduction to pTAL

pTAL is based on the HP Transaction Application Language (TAL). You can compile pTAL source code with either the pTAL or EpTAL compiler.

Topics:

- [pTAL and TAL Compatibility \(page 30\)](#)
- [EpTAL, pTAL, and TAL Compilers \(page 30\)](#)
- [pTAL Applications \(page 31\)](#)
- [pTAL Features \(page 32\)](#)
- [System Services \(page 34\)](#)
- [System Procedures \(page 34\)](#)
- [pTAL and the CRE \(page 34\)](#)

pTAL and TAL Compatibility

The pTAL language is a superset of the TAL language except that TAL supports constructs that depend on characteristics of the underlying TNS architecture, while pTAL (with a few exceptions) does not depend on the underlying TNS/R architecture or TNS/E architecture. For example, pTAL code cannot:

- Access a caller's stack marker
- Use CODE statements to execute instructions
- Build parameter masks for calls to VARIABLE and EXTENSIBLE procedures
- Embed SQL statements

Also, TAL code cannot use 64-bit addressing functionality added to TNS/E pTAL starting with SPR T0561H01^AAP . For more information, see ["64-bit Addressing Functionality" \(page 531\)](#).

Because pTAL uses few machine-dependent constructs, it works efficiently with the system hardware to provide optimal object program performance and is more portable than TAL.

Most pTAL and TAL declarations and executable statements have the same syntax and semantics. Minor semantic differences might affect your programs; for these differences, you must change your source code.

To accommodate migration from TAL to pTAL, pTAL retains some of TAL's operability.

For information about pTAL and TAL differences, see:

- *pTAL Conversion Guide*
- *pTAL Guidelines for TAL Programmers*

EpTAL, pTAL, and TAL Compilers

NOTE: This topic includes only enough information about the TAL compiler to compare it to the EpTAL and pTAL compilers. For complete information about the TAL compiler, see:

- *TAL Reference Manual*
- *TAL Programmer's Guide*
- *TAL Programmer's Guide Data Alignment Addendum*

You can compile pTAL source programs using either the pTAL compiler or the EpTAL compiler.

Table 6 EpTAL, pTAL, and TAL Compiler Characteristics

Compiler	Object Code Generated
EpTAL	TNS/E object code—PIC (position-independent code)
pTAL	TNS/R object code—Non-PIC (default) or PIC
TAL	TNS object code—Non-PIC

Difference between pTAL and EpTAL compilers:

pTAL Compiler	EpTAL Compiler
On Guardian platforms, object files have the file code 700	On Guardian platforms, object files have the file code 800
pTAL code cannot use 64-bit addressing functionality added to TNS/E pTAL starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).	EpTAL code can use 64-bit addressing functionality added to TNS/E pTAL starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

The compilers in Table 6 execute under control of the HP NonStop operating systems in Table 7.

Table 7 HP NonStop Operating Systems

Architecture	RVU
TNS/E	G06.20 and later H06.01 and later
TNS/R	D40 and later
TNS	C-series D-series

This manual indicates when pTAL behaves differently on TNS/E and TNS/R architectures. When no architecture is specifically mentioned, the syntax works the same way on TNS/E and TNS/R architectures.

For more information:

Topic	Source
Itanium® chips used in TNS/E systems	<i>Intel Itanium Architecture Software Developer's Manual</i>
RISC chips used in TNS/R systems	<i>MIPS RISC Architecture</i> by Gerry Kane and Joe Heinrich
TNS/R or TNS/E architecture	System description manual for your system
Compiling pTAL source programs	Chapter 16 (page 355)

pTAL Applications

The pTAL language is appropriate for writing applications where optimal performance has high priority, for example:

- Systems software
 - Operating system components
 - Compilers and interpreters
 - Command interpreters

- Special subsystems
- Special routines that support data communication activities
- Transaction-oriented applications
 - Server processes used with HP data management software
 - Conversion routines that allow data transfer between HP software and other applications
 - Procedures that are callable from programs written in other languages
 - Applications that require optimal performance

You cannot embed SQL/MP or SQL/MX statements in pTAL source code.

pTAL Features

- [Procedures \(page 32\)](#)
- [Subprocedures \(page 32\)](#)
- [Private Data Area \(page 32\)](#)
- [Recursion \(page 33\)](#)
- [Parameters \(page 33\)](#)
- [Data Types \(page 33\)](#)
- [Data Grouping \(page 33\)](#)
- [Pointers \(page 34\)](#)
- [Data Operations \(page 34\)](#)
- [Bit Operations \(page 34\)](#)
- [Built-in Routines \(page 34\)](#)
- [Compiler Directives \(page 34\)](#)
- [Modular Programming \(page 34\)](#)

Procedures

Each pTAL program contains one or more procedures. A procedure is a discrete sequence of declarations and statements that performs a specific task. A procedure is callable from anywhere in the program.

Each procedure executes in its own environment and can contain local variables that are not affected by the actions of other procedures. When a procedure calls another procedure, the operating system saves the caller's environment and restores that environment when the called procedure returns control to the caller.

Subprocedures

A procedure can contain subprocedures, callable only from within the same procedure. A subprocedure can have sublocal variables that are not affected by the actions of other subprocedures. When a subprocedure calls another subprocedure, the caller's environment remains in place. The operating system saves the location in the caller to which control is to return when the called subprocedure terminates.

Private Data Area

Each activation of a procedure or subprocedure has its own data area. Upon termination, each activation relinquishes its private data area, thereby minimizing the amount of memory that the program uses.

Recursion

Because each activation of a procedure or subprocedure has its own data area, a procedure or subprocedure can call itself or can call another procedure that in turn calls the original procedure.

Parameters

A procedure or subprocedure can have optional or required parameters. The same procedure or subprocedure can process different sets of variables sent by different calls to it.

Data Types

A pTAL program can declare and refer to the following types of data:

Data Type	Description
STRING	8-bit integer byte
INT, INT(16)	16-bit integer word
INT(32)	32-bit integer doubleword
FIXED, FIXED(0), INT(64)	64-bit integer quadrupleword
FIXED(-19 to -1)	Fixed-point quadrupleword
FIXED(1 to 19)	Fixed-point quadrupleword
REAL, REAL(32)	32-bit floating-point doubleword
REAL(64)	64-bit floating-point quadrupleword
UNSIGNED(n)	n-bit field, where $1 \leq n \leq 31$
BADDR	32-bit byte address
WADDR	32-bit 2-byte address
CBADDR	32-bit byte code word address
CWADDR	32-bit 2-byte code word address
SGBADDR	16-bit SG-relative byte address
SGWADDR	16-bit SG-relative 2-byte address
SGXBADDR	32-bit SG-relative 2-byte address
SGXWADDR	32-bit SG-relative 2-byte address
EXTADDR	32-bit byte address
EXT32ADDR*	Explicitly named 32-bit byte address
EXT64ADDR*	64-bit byte address
PROCADDR	32-bit code byte address
PROC32ADDR*	Explicitly named 32-bit code byte address
PROC64ADDR*	64-bit code byte address

* These data types are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Data Grouping

A pTAL program can declare and use groups of related variables, such as arrays and structures (records).

Pointers

A pTAL program can declare pointers (variables that can contain addresses) and use them to access locations throughout memory. You can store addresses in pointers when you declare them or later in your program.

Data Operations

A pTAL program can copy a contiguous group of words or bytes and compare one group with another. It can scan a series of bytes for the first byte that matches (or fails to match) a given character.

Bit Operations

A pTAL program can perform bit deposits, bit extractions, and bit shifts.

Built-in Routines

A pTAL program can use built-in routines to convert data types and addresses, test for an ASCII character, or determine the length, offset, type, or number of occurrences of a variable.

Compiler Directives

You can use directives to control a compilation. You can, for example, check the syntax in your source code or control the content of compiler listings.

Modular Programming

You can divide a large pTAL program into modules, compile them separately, and then link the resulting object files into a new object file.

System Services

Your program can ignore many things such as the presence of other running programs and whether your program fits into memory. For example, programs are loaded into memory for you and absent pages are brought from disk into memory as needed.

System Procedures

The file system treats all devices as files, including disk files, disk packs, terminals, printers, and programs running on the system. File-system procedures provide a file-access method that lets you ignore the peculiarities of devices. Your program can refer to a file by the file's symbolic name without knowing the physical address or configuration status of the file.

Your program can call system procedures that activate and terminate programs running on any processor on the system, and can also call system procedures that monitor the operation of a running program or processor. If the monitored program stops or a processor fails, your program can determine this fact.

For more information about system procedures see:

- *Guardian Procedure Calls Reference Manual*
- *Guardian Programmer's Guide*

pTAL and the CRE

pTAL does not have a run-time environment defined by a run-time library such as HP C and HP COBOL. The CRE provides a common foundation for language-specified run-time libraries that enables mixed-language programming.

A program with a pTAL main routine cannot run in the CRE because pTAL does not perform the necessary initialization of the run-time environment. pTAL routines can run in the CRE if they are

called from a program with an HP C main routine. There are additional restrictions on what operations can be performed in the pTAL routines. For complete details on writing pTAL routines that run in the CRE, see the *CRE Programmer's Guide*.

2 Language Elements

The elements that make up the pTAL language include:

- [Character Set \(page 36\)](#)
- [Keywords \(page 37\)](#)
- [Delimiters \(page 38\)](#)
- [Operators \(page 39\)](#)
- [Base Address Symbols \(page 40\)](#)
- [Indirection Symbols \(page 41\)](#)
- [Declarations \(page 41\)](#)
- [Typed Integer Constants \(page 44\)](#)
- [Statements \(page 45\)](#)

Character Set

pTAL supports the complete ASCII character set, which includes:

- Uppercase and lowercase alphabetic characters (A through Z.)
- Numeric characters (0 through 9)
- Special characters

Table 8 Special Characters

Character	Description	Character	Description
!	Exclamation point	"	Quotation mark
\$	Dollar sign	%	Percent sign
&	Ampersand	'	Apostrophe
(Opening parenthesis)	Closing parenthesis
*	Asterisk	+	Plus
,	Comma	-	Hyphen (minus)
.	Period (decimal point)	/	Right slash
:	Colon	;	Semicolon
<	Less than	=	Equals
>	Greater than	?	Question mark
@	Commercial at sign	[Opening bracket
\	Back slash]	Closing bracket
^	Circumflex	_	Underscore
`	Grave accent	{	Opening brace
	Vertical line	}	Closing brace
~	Tilde		

Keywords

Keywords have predefined meanings to the compiler when used as shown in the syntax diagrams in this manual.

Keyword Type	Description
Reserved	Reserved by the compiler. Do not use reserved keywords (shown in Table 9 (page 37)) for your identifiers.
Nonreserved	You can use nonreserved keywords anywhere identifiers are allowed except as noted in the Restrictions column of Table 10 (page 37) .

Table 9 Reserved Keywords

AND	ELSE	INTERRUPT	PROC32ADDR*	SGXBADDR
ASSERT	END	LABEL	PROC64ADDR*	SGXWADDR
BADDR	ENTRY	LAND	PROCPTR	UNTIL
BEGIN	EXTERNAL	LITERAL	PROC32PTR*	USE
BY	EXTADDR	LOR	PROC64PTR*	VARIABLE
CALL	EXT32ADDR*	MAIN	REAL	VOLATILE
CALLABLE	EXT64ADDR*	NOT	REFALIGNED	WADDR
CASE	FIELDALIGN	OF	RESIDENT	WHILE
CBADDR	FIXED	OR	RETURN	
CWADDR	FOR	OTHERWISE	RSCAN	
DEFINE	FORWARD	PRIV	SCAN	
DO	IF	PROCADDR	SGBADDR	
DOWNT0	INT	PROC	SGWADDR	

* These reserved keywords are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” ([page 531](#)).

Table 10 Nonreserved Keywords

Keyword	Restrictions
AT	Allowed in BLOCK declarations
AUTO	None
BELOW	Allowed in BLOCK declarations
BIT_FILLER	Not to be used as an identifier within a structure
BLOCK	Not to be used as an identifier in a source file that contains the NAME declaration
BYTES	Not to be used as an identifier of a LITERAL or DEFINE
C	None
ELEMENTS	Not to be used as an identifier of a LITERAL or DEFINE
EXT	None
EXTENSIBLE	None
FILLER	Not to be used as an identifier within a structure
LANGUAGE	None

Table 10 Nonreserved Keywords *(continued)*

Keyword	Restrictions
NAME	None
NODEFAULT	None
PRIVATE	Not to be used as an identifier in a source file that contains the NAME declaration
RETURNSCC	None
SHARED2	None
SHARED8	None
UNSPECIFIED	None
WORDS	Not to be used as an identifier of a LITERAL or DEFINE

Delimiters

Delimiters are symbols that begin, end, or separate fields of information. Delimiters tell the compiler how to handle the fields of information.

Table 11 Delimiters

Symbol	Character Representation	Uses
!	Exclamation mark	Begins and optionally ends a comment
--	Two consecutive hyphens	Begins a comment
,	Comma	Separates fields of information, such as in declarations, statements, directives, and constant lists
;	Semicolon	<ul style="list-style-type: none"> • Terminates data declarations • Separates statements • Separates declaration options
.	Period	Separates identifier levels in a qualified structure item identifier
< <i>n</i> : <i>n</i> >	Angle brackets	Delimits a bit field in a bit operation
:	Colon	<ul style="list-style-type: none"> • Denotes a statement label • Denotes a procedure entry point • Denotes an ASSERT statement assert level • Denotes a parameter pair
()	Parentheses	<ul style="list-style-type: none"> • Delimit subexpressions within an expression • Delimit the parameter list of a DEFINE, procedure, subprocedure, or CALL statement • Delimit the referral in a structure pointer declaration • Delimit the implied decimal point position in a FIXED variable
[<i>n</i> : <i>n</i>]	Square brackets	Delimit the bounds specification in the declaration of an array, structure, or substructure

Table 11 Delimiters (*continued*)

Symbol	Character Representation	Uses
->	Hyphen plus right angle bracket	<ul style="list-style-type: none"> • Begins one or more labels in a labeled CASE statement • Begins a <i>next-addr</i> clause in a SCAN or RSCAN statement • Begins a <i>next-addr</i> clause in a move statement • Begins a <i>next-addr</i> clause in a group comparison expression
"string"	Quotation marks	Delimit a character string
""	Consecutive quotation marks	The first quotation mark indicates that the second quotation mark is not a delimiter in a character string
=	Equal sign EQL	<ul style="list-style-type: none"> • Used in LITERAL declarations • Used in equivalence variable declarations • Used in redefinition declarations
= body #	Equal sign and hash mark	Delimit the body in a DEFINE declaration
','	Single quotation marks	Delimit a comma that is not a delimiter in a DEFINE parameter
\$	Dollar sign	Denotes a built-in routine (such as \$ABS) or a built-in routine (such as \$ASCIITOFIXED)
?	Question mark	Begins a directive line

Operators

Operators specify operations, such as arithmetic or assignments, that you want to perform on data items.

Table 12 Operators

Context	Operator	Description
Assignment	:=	Data declaration initialization; assignment statement, FOR statement, and assignment expression
Move statement	':='	Left-to-right move
	'=:'	Right-to-left move
	&	Concatenated move
Labeled CASE statement	..(two periods)	Describes a range of case alternatives
Remove indirection	@	Accesses the address contained in a pointer or the address of a nonpointer item
Repetition	* (asterisk)	Repetition factor in a constant list
Template structure	(*)	Template structure declaration
FIXED(*) parameter type	(*)	Value parameter to be treated as FIXED
Bit-field access	.(period)	Accesses a bit-deposit or bit-extraction field (<n> or <n:n>)
Bit shift	<<	Signed left shift
	>>	Signed right shift
	'<<'	Unsigned left shift

Table 12 Operators *(continued)*

Context	Operator	Description
Arithmetic expression	'>>'	Unsigned right shift
	+	Signed addition
	-	Signed subtraction
	*	Signed multiplication
	/	Signed division
	'+'	Unsigned addition
	'-'	Unsigned subtraction
	'*'	Unsigned multiplication
	'/'	Unsigned division
	'\'	Unsigned modulo division
	LOR	Logical OR bit-wise operation
	LAND	Logical AND bit-wise operation
	XOR	Exclusive OR bit-wise operation
Relational expression	<	Signed less than
	=	Signed equal to
	>	Signed greater than
	<=	Signed less than or equal to
	>=	Signed greater than or equal to
	<>	Signed not equal to
	'<'	Unsigned less than
	'='	Unsigned equal to
	'>'	Unsigned greater than
	'<='	Unsigned less than or equal to
	'>='	Unsigned greater than or equal to
	'<>'	Unsigned not equal to
	AND	Logical conjunction
	OR	Logical disjunction
	NOT	Logical negation

Base Address Symbols

Base address symbols let you declare pointers to specific data segments.

Table 13 Base Address Symbols

Symbol	Description
'P'	P-register addressing (read-only array declaration)
'SG'	Define base address equivalencies, system global space (privileged procedures only)
'SGX'	References data in the system data segment.

Table 13 Base Address Symbols *(continued)*

Symbol	Description
'G'	References data relative to the beginning of the Global data area (not supported by pTAL).
'L'	References data relative to the beginning of the Procedure (not supported by pTAL).
'S'	References data relative to the beginning of the Subprocedure (not supported by pTAL).

Indirection Symbols

Indirection symbols determine the address types of variables. Use indirection symbols when declaring formal parameters to cause them to be passed by reference (rather than by value).

Table 14 Indirection Symbols

Symbol	Declares ...
. (period)	<ul style="list-style-type: none"> • An array or structure as having standard direct addressing • A simple pointer or structure pointer
.EXT	<ul style="list-style-type: none"> • An array or structure as having extended 32-bit addressing • An extended (32-bit) simple pointer or structure pointer
.EXT32*	<ul style="list-style-type: none"> • An array or structure as having extended 32-bit addressing • An extended (32-bit) simple pointer or structure pointer
.EXT64*	<ul style="list-style-type: none"> • An array or structure as having extended 64-bit addressing • An extended (64-bit) simple pointer or structure pointer
.SG	A standard (16-bit) system global pointer
.SGX	An extended (32-bit) system global pointer.

* These indirection symbols are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Declarations

Declarations allocate storage and associate identifiers with declarable objects in a program; that is:

- Variables
- LITERALS and DEFINES (see [Chapter 6 \(page 97\)](#))
- Procedures (see [Chapter 14 \(page 246\)](#))
- Labels (see [Labels in Procedures \(page 273\)](#))
- Entry points (see [Entry-Point Declarations \(page 260\)](#))

Topics:

- [Identifiers \(page 42\)](#)
- [Variables \(page 43\)](#)
- [Scope \(page 43\)](#)

Identifiers

Identifiers must conform to these rules:

- Identifiers can have up to 132 characters. You can limit the identifier to 31 characters by setting the `DO_TNS_SYNTAX` (page 389).
- Identifiers begin with an alphabetic character, an underscore (`_`), or a circumflex (`^`).
- Identifiers contain alphabetic characters, numeric characters, underscores, or circumflexes.
- Identifiers contain lowercase and uppercase alphabetic characters. The compiler treats all characters as uppercase.
- Identifiers cannot be reserved keywords (see Table 9 (page 37)).
- Identifiers can be nonreserved keywords, except as noted in Table 10 (page 37).

In addition to the preceding rules, HP recommends that you:

- Use underscores rather than circumflexes to separate words in identifiers (for example, use `Name_Using_Circumflexes` rather than `Name^Using^Circumflexes`). This guideline reflects international character-set standards, which allow the character printed for the circumflex to vary by country.
- Do not end identifiers with an underscore. The trailing underscore is reserved for identifiers supplied by the operating system (such as `Name_Using_Trailing_Underscore_`).

Example 1 Correct Identifiers

```
a2
HP
_2345678012_31_characters
name_with_exactly_31_characters
```

Example 2 Incorrect Identifiers

Identifier	Problem
2abc	Begins with a number
ab%99	Contains % symbol
VARIABLE	Reserved word

Each identifier belongs to an identifier class. The compiler determines the identifier class based on how you declare the identifier.

Table 15 Identifier Classes

Class	Description
Block	Global data block
Code	Read only (P-relative) array
Variable	Simple variable, array, nonstructure pointer, structure pointer, structure, or structure data item
DEFINE*	Named text
Function	Procedure or subprocedure that returns a value
Label	Statement label
LITERAL	Named constant
PROC	Procedure or subprocedure that does not return a value

Table 15 Identifier Classes *(continued)*

Class	Description
Template	Template structure
* Available only on Guardian platforms.	

Variables

A variable is a symbolic representation of data. It can be a single-element variable or a multiple-element variable. You use variables to store data that can change during program execution. Before you can access data stored in a variable you must either:

- Initialize the variable with a value when you declare the variable
- Assign a value to the variable after you declare the variable

Table 16 Variable Types

Variable Type	Description
Simple variable	A variable that contains one element of a specified data type
Array	A variable that contains multiple elements of the same data type
Structure	A variable that can contain variables of different data types
Substructure	A structure nested within a structure or substructure
Structure item	A simple variable, array, simple pointer, substructure, or structure pointer declared in a structure or substructure; also known as a structure field
Nonstructure pointer	A variable that contains a memory address, usually of a simple variable or an array element, which you can access with this nonstructure pointer
Structure pointer	A variable that contains the memory address of a structure, which you can access with this structure pointer

Scope

Every declared item in a pTAL program has a scope that determines where in the program it is visible (after the point of declaration).

Scope	Declared in a ...	Visible ...
Global	Program	Everywhere in the program
Local	Procedure	Only in the procedure that declares it (including the subprocedures of that procedure)
Sublocal	Subprocedure	Only in the subprocedure that declares it

Formal parameters of procedures and subprocedures have local and sublocal scope, respectively.

Example 3 Scope of Declared Items

```

int i;           ! i has global scope and is visible everywhere
                ! from this point forward.
proc p;         ! p has global scope.  If p had formal
                ! parameters, they would have local scope.
begin
  int j := i;    ! j has local scope and is visible everywhere in
                ! procedure p from this point forward.
  subproc s;     ! s has local scope.  If s had formal parameters,
                ! they would have sublocal scope.
begin

```

```

    int k := i + j; ! k has sublocal scope and is visible only
                    ! in subprocedure s.
end;
! k is not visible here. It is created and destroyed every

! time subprocedure s is called.
end;
! j is not visible here. It is created and destroyed every time
! procedure p is called.
! i is accessible here and exists as long as the program is
! running.

```

Variables that have different scopes can have the same name, but they are different variables.

Example 4 Global and Local Variable With the Same Name

```

int(32) i;          ! This i is global
proc p;
begin
    int i;          ! This i is local to procedure p, different
                    ! from global variable i, and makes access to
                    ! global variable i impossible ("hides" it).
    i := i + 1D;    ! ERROR: local variable i is INT(16)
end;

```

For local and sublocal variables, the compiler generates code to evaluate and store an initialization expression. For example, for the expression

```
int k := i * j;
```

if *i*, *j*, and *k* are local or sublocal variables, the compiler generates code to multiply *i* by *j* and store the product in *k*.

For global variables, the compiler does not generate such initialization code. Initial values assigned to global variables are determined by the linker.

Typed Integer Constants

A constant is a value you can store in a variable, declare as a LITERAL, or use as part of an expression. Constants can be numbers or character strings. The following are examples of constants:

Constant Type	Example
Character string	"abc"
Numeric	654

You can specify numeric constants in binary, octal, decimal, or hexadecimal base, depending on the data type of the constant. The default number base in pTAL is decimal. The following are example constants in each number base:

Number Base	Example
INT(16) Decimal	-654
INT(32) Decimal	+654D or +654 D (% not allowed)
FIXED Decimal	654F or 654 F (% not allowed)
INT(16) Binary	%B101111
INT(32) Binary	%B101111D or %B101111%D or %B101111% D
FIXED Binary	%B101111F or %B101111°F or %B101111% F

Number Base	Example
INT(16) Octal	%57
INT(32) Octal	%57D or %57%D or %57 D
FIXED Octal	%57D or %57%F or %57 F
INT(16) Hexadecimal	%H2F
INT(32) Hexadecimal	%H2F%D or %H2F D (space or % required)
FIXED Hexidecimal	%H2F%F or %H2F F (space or % required)

Statements

A statement specifies operations to be performed on declared objects. Statements are discussed in [Chapter 12 \(page 199\)](#), and summarized in [Table 55 \(page 199\)](#).

3 Data Representation

A program operates on data—variables and constants—which it stores in the storage units that [Table 17 \(page 46\)](#) describes.

Table 17 Storage Units

Storage Unit	Number of Bits	Description
Byte	8	Smallest addressable unit of memory.
Word	16	2 bytes, with byte 0 (most significant) on the left and byte 1 (least significant) on the right
Doubleword	32	4 bytes
Quadword	64	8 bytes
<i>n</i> -bit field	1-16	Contiguous bit fields within 2 bytes
	17-31	Contiguous bit fields within 4 bytes

Topics:

- [Data Types \(page 46\)](#)
- [Address Types \(page 49\)](#)
- [Constants \(page 57\)](#)

Data Types

When you declare a variable, you specify its data type, which determines:

- Its storage unit
- The values that you can assign to it
- The operations that you can perform on it
- Its address type

Table 18 Data Types

Data Type	Storage Unit ¹	Values the Data Type Can Represent
STRING	Byte	<ul style="list-style-type: none">• ASCII character• Unsigned 8-bit integer in the range 0 through 255
INT INT(16) ²	Word	<ul style="list-style-type: none">• String of one or two ASCII characters• Unsigned 6-bit integer in the range 0 through 65,535• Signed 6-bit integer in the range -32,768 through 32,767
INT(32)	Doubleword	32-bit integer in the range -2,147,483,648 through +2,147,483,647
REAL REAL(32) ³	Doubleword	32-bit floating-point number in the range $\pm 8.6361685550944444\text{E-}78$ through $\pm 1.15792089237316192\text{E}$, precise to approximately 6.5 significant decimal digits.
FIXED FIXED(0) INT(64) ⁴ FIXED(-19 to -1) FIXED(1 to 19)	Quadword	64-bit fixed-point number. For FIXED, FIXED(0), FIXED (*), and INT(64) the range is -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.

Table 18 Data Types *(continued)*

Data Type	Storage Unit ¹	Values the Data Type Can Represent
REAL(64)	Quadrupleword	64-bit floating-point number in the same range as data type REAL but precise to approximately 16.5 significant decimal digits.
UNSIGNED	<i>n</i> -bit field	UNSIGNED(1-15) and UNSIGNED(17-31): Unsigned integer in the range 0 through (2 ^{<i>n</i>} - 1) UNSIGNED(16): <ul style="list-style-type: none"> Unsigned integer in the range 0 through 65,535 Signed integer in the range -32,768 through 32,767 UNSIGNED simple variable: The bit field can be 1 to 31 bits. UNSIGNED array: The element bit field can be 1, 2, 4, or 8 bits.

¹ Table 17 (page 46) describes storage units.

² INT and INT(16) are the same type.

³ REAL and REAL(32) are the same type.

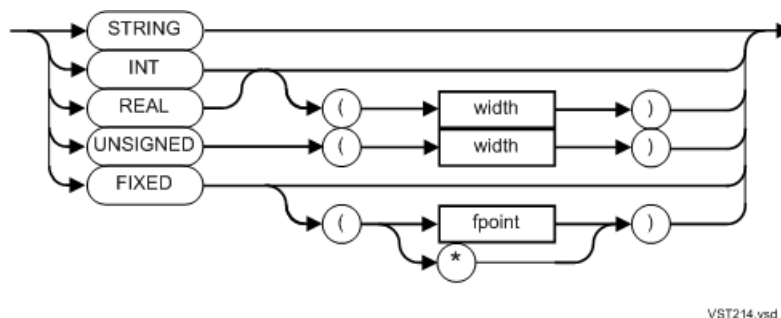
⁴ FIXED, FIXED(0), and INT(64) are the same type.

Topics:

- [Specifying Data Types \(page 47\)](#)
- [Data Type Aliases \(page 48\)](#)
- [Operations by Data Type \(page 48\)](#)

Specifying Data Types

The syntax for specifying the data type in a variable declaration is:



width

is a constant expression that specifies the width, in bits, of the variable. The value of *width* must be appropriate for the data type (see [Example 5 \(page 48\)](#)):

Data Type	Value of width
INT (16)*	16
INT (32)	32
INT (64)*	64
REAL(32)*	32
REAL(64)	64
UNSIGNED(<i>n</i>)	In the range 1 through 31

* Data type alias (see [Data Type Aliases \(page 48\)](#))

fpoint

is the implied fixed-point (decimal-point) setting. *fpoint* is an integer in the range -19 through 19. The default *fpoint* is 0 (no decimal places).

A positive *fpoint* specifies the number of places to the right of the decimal point:

```
FIXED(3) x := 0.642F; ! Stored as 642
```

A negative *fpoint* specifies a number of places to the left of the decimal point. When the value is stored, it is truncated leftward from the decimal point by the specified number of digits. When the value is accessed, zeros replace the truncated digits:

```
FIXED(-3) y := 642945F; ! Stored as 642; accessed as 642000
```

*(asterisk)

prevents scaling of the initialization values (for an explanation of scaling, see [Scaling of FIXED Operands \(page 74\)](#)).

Example 5 Constant Expressions in Data Type Specifications

```
LITERAL a = 2,  
        b = 35;  
INT(a + 30) aaa; ! OK: INT(32) is valid  
INT(b - 5) bbb; ! ERROR: expression must evaluate to valid  
                ! bit length (16, 32, or 64 for an INT)  
REAL (b - 19) ccc; ! ERROR: expression must evaluate to valid  
                ! bit length (32 or 64 for REAL)  
REAL (b + 29) ddd; ! OK: REAL(64) is valid  
UNSIGNED (a) eee; ! OK: UNSIGNED fields can be any number of  
                ! bits from 1 to 31.
```

Data Type Aliases

The compiler accepts these data type aliases:

Data Type	Aliases
INT	INT(16)
REAL	REAL(32)
FIXED	FIXED(0)INT(64)

The remainder of this manual avoids using data type aliases.

Operations by Data Type

The data type of a variable determines the operations you can perform on the variable.

Table 19 Operations by Data Type

Operation	STRING	INT or UNSIGNED (1-16)	INT(32) or UNSIGNED (17-31)	FIXED	REAL or REAL(64)
Unsigned arithmetic	Yes	Yes	Yes	No	No
Signed arithmetic	Yes	Yes	Yes	Yes	Yes
Logical operations	Yes	Yes	Yes	No	No
Relational operations	Yes	Yes	Yes	Yes	Yes

Table 19 Operations by Data Type (*continued*)

Operation	STRING	INT or UNSIGNED (1-16)	INT(32) or UNSIGNED (17-31)	FIXED	REAL or REAL(64)
Bit shifts	Yes	Yes	Yes	No	No
Byte scans	Yes	Yes	Yes	Yes	Yes

The data type of a variable also determines which built-in routines you can use with the variable (see [Chapter 15](#) (page 274)).

Address Types

Every identifier that you declare has both a data type and an address type. The data type describes the data item itself. The address type describes the address of the data item. If you declare a pointer to the data item, the value that you assign to the pointer must be of that address type.

You cannot explicitly declare the address type of a pointer. When you declare a pointer, the compiler determines its address type.

Table 20 Data Types and Their Address Types

Pointer Declaration		Data Type	Address Type	Storage Unit ¹
STRING	<code>.s i</code>	STRING	BADDR	Byte
INT	<code>.i i</code>	INT	WADDR	Word
INT(32)	<code>.j i</code>	INT(32)	WADDR	Word
REAL	<code>.r i</code>	REAL	WADDR	Word
REAL(64)	<code>.s i</code>	REAL(64)	WADDR	Word
FIXED	<code>.f i</code>	FIXED	WADDR	Word
STRUCT	<code>.t i</code>	none	WADDR	Word
SUBSTRUCT	<code>.v i</code>	none	BADDR	Byte
<i>addr-type</i> ²	<code>.a i</code>	<i>address_type</i> ³	WADDR	Word
STRING	<code>.EXT s i</code>	STRING	EXTADDR	Byte
INT	<code>.EXT i i</code>	INT	EXTADDR	Byte
INT(32)	<code>.EXT j i</code>	INT(32)	EXTADDR	Byte
REAL	<code>.EXT r i</code>	REAL	EXTADDR	Byte
REAL(64)	<code>.EXT s i</code>	REAL(64)	EXTADDR	Byte
FIXED	<code>.EXT f i</code>	FIXED	EXTADDR	Byte
STRUCT	<code>.EXT t i</code>	none	EXTADDR	Byte
SUBSTRUCT	<code>.EXT v i</code>	none	EXTADDR	Byte
<i>addr-type</i> ²	<code>.EXT a i</code>	<i>address_type</i> ³	EXTADDR	Byte
STRING	<code>.EXT32 s i</code>	STRING	EXT32ADDR ⁴	Byte
INT	<code>.EXT32 i i</code>	INT	EXT32ADDR ⁴	Byte
INT(32)	<code>.EXT32 k i</code>	INT(32)	EXT32ADDR ⁴	Byte
REAL	<code>.EXT32 r i</code>	REAL	EXT32ADDR ⁴	Byte
FIXED	<code>.EXT32 f i</code>	FIXED	EXT32ADDR ⁴	Byte

Table 20 Data Types and Their Address Types *(continued)*

Pointer Declaration		Data Type	Address Type	Storage Unit ¹	
STRUCT	.EXT32 t;	none	EXT32ADDR ⁴	Byte	
SUBSTRUCT	.EXT32 v;	none	EXT32ADDR ⁴	Byte	
addr-type	.EXT32 a;	address_type	EXT32ADDR ⁴	Byte	
STRING	.EXT64 s;	STRING	EXT64ADDR ⁴	Byte	
INT	.EXT64 i;	INT	EXT64ADDR ⁴	Byte	
INT(32)	.EXT64 k;	INT(32)	EXT64ADDR ⁴	Byte	
REAL	.EXT64 r;	REAL	EXT64ADDR ⁴	Byte	
FIXED	.EXT64 f;	FIXED	EXT64ADDR ⁴	Byte	
STRUCT	.EXT64 t;	None	EXT64ADDR ⁴	Byte	
SUBSTRUCT	.EXT64 v;	none	EXT64ADDR ⁴	Byte	
addr-type	.EXT64 a;	address_type	EXT64ADDR ⁴	Byte	
STRING	.SG s;	STRING	SGBADDR	Byte	
INT	.SG i;	INT	SGWADDR	Word	
INT(32)	.SG j;	INT(32)	SGWADDR	Word	
REAL	.SG r;	REAL	SGWADDR	Word	
REAL(64)	.SG s;	REAL(64)	SGWADDR	Word	
FIXED	.SG f;	FIXED	SGWADDR	Word	
addr-type ²	.SG a;	address_type ³	SGWADDR	Word	
STRING	.SGX s;	STRING	SGXBADDR	Byte	
INT	.SGX i;	INT	SGXWADDR	Word	
INT(32)	.SGX j;	INT(32)	SGXWADDR	Word	
REAL	.SGX r;	REAL	SGXWADDR	Word	
REAL(64)	.SGX s;	REAL(64)	SGXWADDR	Word	
FIXED	.SGX f;	FIXED	SGXWADDR	Word	
addr-type ²	.SGX a;	address_type ³	SGXWADDR	Word	
PROC p;		PROC	PROCADDR	Doubleword	
PROCPTR p(); END PROCPTR		PROCPTR	PROCADR	Doubleword	
PROC32PTR p(); END PROCPTR		PROC32PTR	PROC32PTR ⁴	Doubleword	
PROC64PTR p(); END PROCPTR		PROC64PTR	PROC64PTR ⁴	Quadword	
procedure e; ENTRY		PROC	PROCADDR	Doubleword	
STRING v = 'p' := "ab";		STRING	CBADDR	Byte	
INT v = 'p' := "ab";		INT	CWADDR	Word	
SUBPROC		SUBPROC	CWADDR	Word	
Subprocedure ENTRY e;			CWADDR	Word	
LABEL 1;		LABEL	CWADDR	Word	

¹ [Table 17 \(page 46\)](#) describes storage units.

² *addr-type* is any of the ten address types.

³ *address_type* is the same address type as specified in the declaration.

⁴ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see [Appendix E, “64-bit Addressing Functionality” \(page 531\)](#).

You can compare addresses using the relational operators described in [Table 12 \(page 39\)](#).

Topics:

- [Storing Addresses in Variables \(page 51\)](#)
- [Converting Between Address Types and Numeric Data Types \(page 51\)](#)
- [Converting Between Address Types \(page 52\)](#)
- [Using Indexes to Access Array Elements \(page 54\)](#)
- [Incrementing and Decrementing Addresses \(Stepping Pointers\) \(page 54\)](#)
- [Computing the Number of Bytes Between Addresses \(page 55\)](#)
- [Comparing Addresses to Addresses \(page 56\)](#)
- [Comparing Addresses to Constants \(page 56\)](#)
- [Comparing Procedure Addresses and Procedure Pointers \(page 56\)](#)
- [Testing a Pointer for a Nonzero Value \(page 56\)](#)

Storing Addresses in Variables

You can store an address into a variable when either of the following is true:

- The address is the same data type as the variable into which you are storing the address.
- The address is convertible to the data type of the variable into which you are storing the address.

Converting Between Address Types and Numeric Data Types

You can move any 16-bit integer value—a constant or variable—into any system global address type (SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR). Conversely, you can move the value of any system global data type into a 16-bit integer variable.

You can move an EXTADDR into an INT(32), or an INT(32) into an EXTADDR.

You can also move an EXTA32DDR into an INT(32), or an INT(32) into an EXT32ADDR.

Exceptions: You cannot convert the following address types to numeric data types:

- CBADDR
- CWADDR
- PROCADDR
- PROC32ADDR
- PROC64ADDR

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see [Appendix E, “64-bit Addressing Functionality” \(page 531\)](#).

Converting Between Address Types

You can convert an address from one address type to another using either:

- Built-in address-conversion functions (see [Table 61 \(page 283\)](#))
- Shift operations and low-level built-in routines

These conversions are supported for compatibility with TAL. HP recommends that you use the equivalent pTAL routine when you write pTAL code.

Expression	Operand Type	Equivalent Routine Call
$e \ll 1$	WADDR	<code>\$WADDR_TO_BADDR(<i>e</i>)</code>
$e \ll 1$	SGWADDR	<code>\$SGWADDR_TO_SGBADDR(<i>e</i>)</code>
$e \ll 1$	SGXWADDR	<code>\$SGWADDR_TO_SGBADDR(<i>e</i>)</code>
$e \gg 1$	BADDR	<code>\$BADDR_TO_WADDR(<i>e</i>)</code>
$e \gg 1$	SGBADDR	<code>\$SGBADDR_TO_SGWADDR(<i>e</i>)</code>
$e \gg 1$	SGXBADDR	<code>\$SGBADDR_TO_SGWADDR(<i>e</i>)</code>
<code>\$UDBL(<i>e</i>)</code>	BADDR	<code>\$BADDR_TO_EXTADDR(<i>e</i>)</code>
<code>\$DBLL(0,<i>e</i>)</code>	BADDR	<code>\$BADDR_TO_EXTADDR(<i>e</i>)</code>
<code>\$UDBL(<i>e</i>) << 1</code>	WADDR	<code>\$WADDR_TO_EXTADDR(<i>e</i>)</code>
<code>\$DBLL(0,<i>e</i>) << 1</code>	WADDR	<code>\$WADDR_TO_EXTADDR(<i>e</i>)</code>

The compiler generates code for implicit conversions for the following operations:

- Block moves and compares

The compiler automatically converts an address type if required for the source or destination pointer in a block move or block compare instruction.
- Call-by-reference actual parameters

The compiler automatically converts the address type of an actual parameter to the address type of a formal parameter if the conversion could not cause data loss. For example:

 - BADDR can be converted to EXTADDR
 - WADDR can be converted to EXTADDR
 - EXTADDR cannot be converted to WADDR
 - EXTADDR and EXT32ADDR can be converted to EXT64ADDR
 - PROCADDR and PROC32ADDR can be converted to PROC64ADDR
 - PROCPTR and PROC32PTR can be converted to PROC64PTR (if same parameter profile. For more information, see [“Procedure Pointers” \(page 263\)](#).)

Table 21 Valid Address Conversions

TO	FROM																F I X E D 0
	B A D D R	W A D D R	C B A D D R	C W A D D R	S G B A D D R	S G X B A D D R	S G W A D D R	S G X W A D D R	E X T A D D R	E X T 32 A D D R *	P R O C A D D R	P R O C 32 A D D R *	E X T 64 A D D R *	P R O C 64 A D D R *	INT	INT (32)	
BADDR	=	r7							r	r			c		r1		
WADDR	r9	=							r	r			c		r2		
CBADDR			=	9													
CWADDR			9	=													
SGBADDR					=	=	R7	r7					c		y3		
SGXBADDR					=	=	R7	r7					c		y3		
SGWADDR					r8	r8	=	=					c		y4		
SGXWADDR					r8	r8	=	=					c		y4		
EXTADDR	r	r	c11	c11	r	r	r	r	=	y			c			y	
EXT32ADDR*	r	r	c11	c11	r	r	r	r	y	=			c			y	
PROCADDR											=10	y10		c10			
PROC32ADDR*											r10	=10		c10			
EXT64ADDR*	r	r			r	r	r	r	y	y			=				y12
PROC64ADDR*											r10	r10		=10			
INT					y5	y5	y6	y6							=	c	c
INT(32)									y	y					c	=	c
FIXED(0)													c13		c	c	=

Key to Symbols:

= Same type, no conversion needed

y Implicit conversion for assignments, actual parameters (either by value or by reference) and function return statements

l Implicit conversion for assignments, actual parameters (passed either by value or by reference), and RETURN statements

r Implicit conversion for reference parameters; requires explicit conversion for other contexts

c Requires explicit conversion

1-13 See note (below)

Blank Always unsupported

* These address types are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, "64-bit Addressing Functionality" (page 531).

NOTE:

1. In assignment statements, only INT constants are allowed. They are interpreted as a 'G'-relative byte address.
 2. In assignment statements, only INT constants are allowed. They are interpreted as a 'G'-relative word address.
 3. Input INT interpreted as 'SG'-relative byte address.
 4. Input INT interpreted as 'SG'-relative word address.
 5. Output INT holds 'SG'-relative byte address.
 6. Output INT holds 'SG'-relative word address.
 7. Compiler assumes object is in lower half of the stack or in the lower half of the 'SG' segment (TNS only); there is no dynamic check for msb=0.
 8. The result is undefined if lsb=1; the round-down effect of TNS is not guaranteed. pTAL issues a warning if the BADDR or SGBADDR address is known to be an odd-byte offset from some word-addressed base. No warning is issued if pTAL cannot determine whether the offset is odd or even.
 9. Conversions between CWADDR and CBADDR are unsupported and illegal, because these conversions are probably unneeded, and because it is difficult to ensure that a word-addressed 'P'-relative structure is in the byte-addressable lower half of a TNS code segment.
 10. PROCPTR, PROC32PTR, and PROC64PTR variables and addresses of procedures are implicitly of type PROCADDR, PROC32ADDR, and PROC64ADDR, respectively, but are subject to matching of parameter profiles and procedure attributes. See ["Assignments to Procedure Pointers"](#) (page 269). Implicit conversions from PROC32ADDR to PROCADDR, PROC32ADDR to PROC64ADDR, and PROCADDR to PROC64ADDR are also allowed (again subject to the parameter and attribute matching rules, described in the section noted directly above).
 11. \$XADDR of a CWADDR and CBADDR yields an EXTADDR.
 12. In assignment statements, only FIXED(0) constants are allowed. They are interpreted as a byte address.
 13. \$FIX of an EXT64ADDR yields a FIXED(0).
-

Using Indexes to Access Array Elements

Indexing produces the correct result for all data types including structures. Use indexing wherever possible to adjust pointers.

Example 6 Using Indexing to Access an Array Element

```
int .p;  
@p := @p[2]      ! This statement is equivalent to  
@p := @p '+' 4; ! this statement
```

Incrementing and Decrementing Addresses (Stepping Pointers)

You can increment or decrement the value of a pointer (step a pointer) by:

- [Using Arithmetic Operations to Adjust Addresses \(page 55\)](#)
- [Computing the Number of Bytes Between Addresses \(page 55\)](#)
- [Comparing Addresses to Addresses \(page 56\)](#)
- [Comparing Addresses to Constants \(page 56\)](#)
- [Comparing Procedure Addresses and Procedure Pointers \(page 56\)](#)
- [Testing a Pointer for a Nonzero Value \(page 56\)](#)

Using Arithmetic Operations to Adjust Addresses

You can add an integer value to any address type except PROCADDR, PROC32ADDR, and PROC64ADDR. The address can be on either side of the operator.

Example 7 Adding Integer Values to Addresses

```
INT .p;
@p := @p '+' 4;    ! Increment WADDR pointer by four 16-bit words
@p := 4 '+' @p;    ! Increment WADDR pointer by four 16-bit words
@p := @p[2];
```

You can subtract an integer value from any address type except PROCADDR, PROC32ADDR, and PROC64ADDR. The address must be on the left side of the subtraction operator and the integer must be on the right.

Example 8 Subtracting Integer Values From Addresses

```
INT .p;
@p := @p '-' 4;    ! Decrement WADDR pointer by four 16-bit words
@p := 4 '-' @p;    ! ERROR: The address must be on the right,
                  ! the integer on the left
```

You must use signed operators for operations on EXTADDRs and unsigned operators for all other address types.

Example 9 Signed and Unsigned Operators in Address Arithmetic

```
INT .p;
INT .EXT e;
INT .EXT32 e32;
INT .EXT64 e64;

@p := @p '-' 4;    ! Unsigned arithmetic on WADDRs
@p := 4 '+' @p;    ! Unsigned arithmetic on WADDRs
@e := @e + 4D;     ! Signed arithmetic on EXTADDRs
@e32 := @e32 + 4D; ! Signed arithmetic on EXT32ADDRs
@e64 := @e64 + 8F; ! Signed arithmetic on EXT32ADDRs
```

If you increment or decrement a pointer, the number that you add to, or subtract from, a byte address (such as BADDR) is the number of bytes to move the pointer. Similarly, the number that you add to a word address (such as WADDR) is the number of 16-bit words to move the pointer, not the number of 32-bit words.

If you step a byte address (such as BADDR), the number you specify is added to, or subtracted from, the address in the pointer.

If you step a word address (such as WADDR), the address is incremented/decremented by twice the number you specify, because addresses on TNS/R and TNS/E architecture are represented as byte addresses.

Computing the Number of Bytes Between Addresses

You can subtract two addresses except PROCADDR, PROC32ADDR, and PROC64ADDR addresses. The address types of both operands must be the same except that SGBADDR and SGXBADDR are interchangeable, and SGWADDR and SGXWADDR are interchangeable.

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Comparing Addresses to Addresses

You can compare addresses only if both addresses are the same address type, except that:

- SGBADDR and SGXBADDR are interchangeable with one another
- SGWADDR and SGXWADDR are interchangeable with one another

You must use signed relational operators (<, =, >, <=, <>, >=) to compare EXTADDR, EXT32ADDR, and EXT64ADDR addresses. For all other address types, you must use unsigned relational operators ('<', '=', '>', '<=', '<>', '>='), or signed equal ('=') or signed not equal operators ('<>').

The result of comparing two addresses is an INT value that indicates whether the relationship is true (nonzero) or false (zero).

You can test the condition code after an IF statement that compares two addresses only if certain conditions are met. These conditions are described in [Chapter 13 \(page 234\)](#).

Comparing Addresses to Constants

You can compare a BADDR, WADDR, SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR address to a 16-bit constant value. The requirements for [Comparing Addresses to Addresses \(page 56\)](#) also apply to comparing addresses to constants.

Comparing Procedure Addresses and Procedure Pointers

You can compare PROCADDR, PROC32ADDR, and PROC64ADDR addresses with PROCPTR, PROC32PTR, and PROC64PTR addresses for equality and inequality. The result of comparing the addresses of two different procedures is always “not equal,” but the result of comparing the two addresses of the same procedure is not always “equal.”

NOTE: The address types and procedure pointers, PROC32ADDR, PROC64ADDR, PROC32PTR, and PROC64PTR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” ([page 531](#)).

Testing a Pointer for a Nonzero Value

You can test a pointer for a nonzero value without specifying the constant zero. For example, if *i* is declared:

```
int i;
```

Then these two statements are equivalent:

```
IF @i THEN ...
IF (@i <> 0) ...
```

You can test an EXTADDR or EXT32ADDR pointer for a nonzero value without specifying the constant zero. For example, if *j* is declared:

```
int.EXT j;
INT .EXT32 k;
```

Then these four statements are equivalent:

```
IF @j THEN ...
IF @k THEN ...
IF @j <> 0D THEN ...
IF @k <> 0D THEN ...
```

You can test an EXT64ADDR pointer for a nonzero value without specifying the constant zero. For example, if *m* is declared, then these two statements are equivalent:

```
IF @m THEN ...
IF @m <> 0F THEN ...
```


Constants

- [Character String \(page 57\)](#)
- [STRING Numeric \(page 58\)](#)
- [INT Numeric \(page 58\)](#)
- [INT\(32\) Numeric \(page 59\)](#)
- [FIXED Numeric \(page 61\)](#)
- [REAL and REAL\(64\) Numeric \(page 62\)](#)
- [Constant Lists \(page 63\)](#)
- [Constant List Alignment Specification \(page 64\)](#)

Character String

A character string constant consists of one or more ASCII characters stored in a contiguous group of bytes.



string

is a sequence of one or more ASCII characters enclosed in quotation mark delimiters. If a quotation mark is a character within the sequence of ASCII characters, use two quotation marks (in addition to the quotation mark delimiters). The compiler does not upshift lowercase characters.

Each character in a character string requires one byte of contiguous storage. The maximum length of a character string you can specify differs for initializations and for assignments.

Initializations

You can initialize simple variables or arrays of any data type with character strings.

When you initialize a simple variable, the character string can have the same number of bytes as the simple variable or fewer. This example declares an INT variable and initializes it with a character string:

```
INT chars := "AB";
```

When you initialize an array, the character string can have up to 127 characters and must fit on one line. If a character string is too long for one line, use a constant list (described [Constant Lists \(page 63\)](#)) to break the character string into smaller character strings.

Assignments

You can assign character strings to STRING, INT, and INT(32) variables, but not to FIXED, REAL, or REAL(64) variables.

In assignment statements, a character string can contain at most four characters, depending on the data type of the variable:

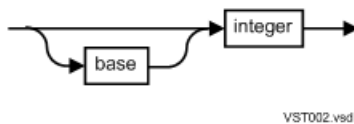
Number of Bytes in String	Data Types to Which String Can Be Assigned
1	STRING, INT
2	STRING, INT
3	INT(32)
4	INT(32)

Example 10 Assigning Character Strings to Variables

```
STRING s;  
INT i;  
s := "a";      ! OK  
s := "ab";     ! OK: same as s := "b"  
s := "abc";    ! ERROR: too big  
i := "a";      ! OK  
i := "ab";     ! OK  
I := "abc";    ! ERROR: too big
```

STRING Numeric

Representation	Unsigned 8-bit integer
Range	0 through 255



base

indicates a number base as follows:

Octal	%
Binary	%b
Hexadecimal	%h

If you omit the *base*, the default *base* is decimal.

integer

is one or more of the following digits:

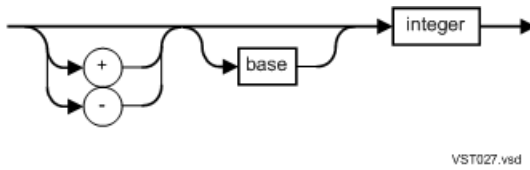
Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F (not case-sensitive)

Examples of STRING numeric constants:

Decimal	255
Octal	%12
Binary	%B101
Hexadecimal	%h2A

INT Numeric

Representation	Signed or unsigned 16-bit integer
Range (unsigned)	0 through 65,535
Range (signed)	-32,768 through 32,767



base

indicates a number base as follows:

Octal	%
Binary	%b
Hexadecimal	%h

The default *base* is decimal.

integer

is one or more of the following digits:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F (not case-sensitive)

Examples of INT numeric constants:

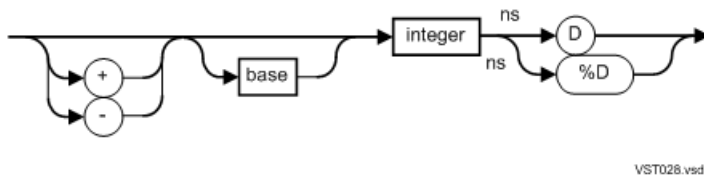
Decimal	3 -32045
Octal	%177 -%5
Binary	%B01010 %b1001111000010001
Hexadecimal	%H1A %h2f

The system stores signed integers in two's complement notation. It obtains the negative of a number by inverting each bit position in the number, and then adding 1.

2 is stored as	0000000000000010
-2 is stored as	1111111111111110

INT(32) Numeric

Representation	Signed or unsigned 32-bit integer
Range	-2,147,483,648 through 4,294,967,295



base

indicates a number base as follows:

Octal	%
Binary	%b
Hexadecimal	%h

The default *base* is decimal.

integer

is one or more of the following digits:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F (not case-sensitive)

D, %D

are suffixes that specify INT(32) constants:

Decimal	D
Octal	D
Binary	D
Hexadecimal	%D

Examples of INT(32) numeric constants:

Decimal	0D +14769D -327895066d
Octal	%1707254361d -%24700000221D
Binary	%B000100101100010001010001001d
Hexadecimal	%h096228d%d -%H99FF29%D

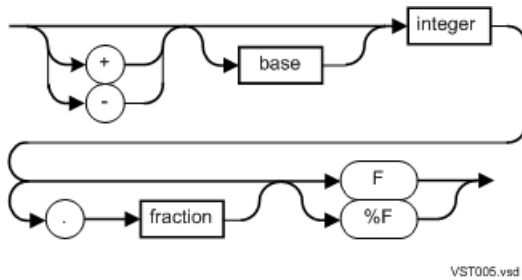
For readability, always specify the % in the %D hexadecimal suffix to prevent the suffix from being confused with the integer part of the constant. The following format, where a space replaces the % in the %D suffix, is allowed but not recommended:

-%H99FF29 D

The system stores signed integers in two's complement notation (see [INT Numeric \(page 58\)](#)).

FIXED Numeric

Representation	Signed 64-bit fixed-point number
Range	-9,223,372,036,854,775,808 through +9,223,372,036,854,775,807



base

indicates a number base as follows:

Octal	%
Binary	%B
Hexadecimal	%H

The default *base* is decimal.

integer

is one or more of the following digits:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F

fraction

is one or more decimal digits. *fraction* is legal only for decimal base.

F, *%F*

are suffixes that specify FIXED constants:

Decimal	F
Octal	F
Binary	F
Hexadecimal	%F

Examples of FIXED numeric constants:

Decimal	1200.09F 0.1234567F 239840984939873494F -10.09F
Octal	%765235512F

Binary	%B1010111010101101010110F
Hexadecimal	%H298756%F

For readability, always specify the % in the %F hexadecimal suffix to prevent the suffix from being confused with the integer part of the constant. The following format, where a space replaces the % in the %F suffix, is allowed but not recommended:

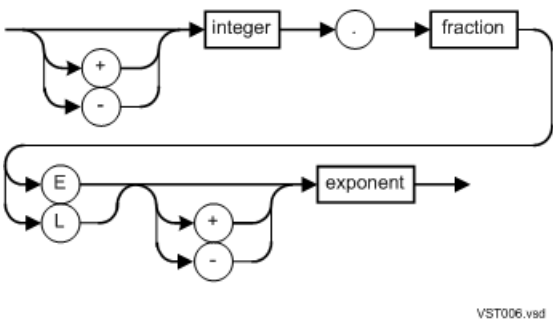
-%H99FF29 F

The system stores a FIXED number in binary notation. When the system stores a FIXED number, it scales the constant as dictated by the declaration or expression. Scaling means the system multiplies or divides the constant by powers of 10 to move the decimal.

For information about scaling of FIXED values in expressions, see [Chapter 5 \(page 69\)](#). For information about scaling of FIXED values in declarations, see [Chapter 7 \(page 103\)](#).

REAL and REAL(64) Numeric

Representation	Signed 32-bit REAL or 64-bit REAL(64) floating-point number
Range	$\pm 8.636168555094446 * 10^{-78}$ through $\pm 1.15792089237316189 * 10^{+77}$
Precision	REAL—to approximately 6.5 significant decimal digits REAL(64)—to approximately 16.5 significant decimal digits



integer

is one or more decimal digits that compose the integer part.

fraction

is one or more decimal digits that compose the fractional part.

E

specifies the floating-point constant REAL.

L

specifies the floating-point constant REAL(64).

exponent

is one or two decimal digits that compose the exponential part.

Examples of REAL and REAL(64) numeric constants, showing the integer part, the fractional part, the E or L suffix, and the exponent part:

Decimal Value	REAL	REAL(64)
0	0.0E0	0.0L0
2	2.0e0 0.2E1	2.0L0 0.2L1

Decimal Value	REAL	REAL(64)
	20.0E-1	20.0L-1
-17.2	-17.2E0 -1720.0E-2	-17.2L0 -1720.0L-2

The system stores the number in binary scientific notation in the form:

$$x * 2^y$$

x is a value of at least 1 but less than 2. Because the integer part of x is always 1, only the fractional part of x is stored.

The exponent can be in the range -256 through 255 (%377). The system adds 256 (%400) to the exponent before storing it as y . Thus, the value stored as y is in the range 0 through 511 (%777), and the exponent is y minus 256.

If the value of the number to be represented is zero, the sign is 0, the fraction is 0, and the exponent is 0.

The system stores the parts of a floating-point constant as follows:

Data Type	Sign Bit	Fraction	Exponent
REAL	<0>	<1:22>	<23:31>
REAL(64)	<0>	<1:54>	<55:63>

Examples of storage formats:

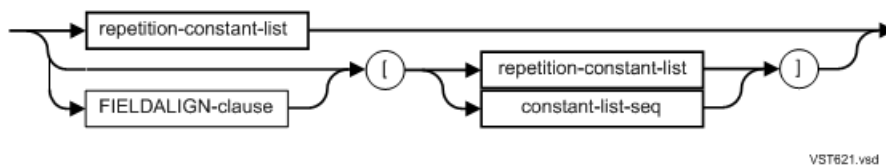
- For the following REAL constant, the sign bit is 0, the fraction bits are 0, and the exponent bits contain %400 + 2, or %402:
 $4 = 1.0 * 22$ stored as %0000000 %000402
- For the following REAL constant, the sign bit is 1, the fraction bits contain %.2 (decimal .25 is 2/8), and the exponent bits contain %400 + 3, or %403:
 $-10 = -(1.25 * 23)$ stored as %120000 %000403
- For the following REAL(64) constant, the sign bit is 0, the fraction bits contain the octal representation of .33333..., and the exponent bits contain %400 - 2, or %376:
 $1/3 = .33333... * 2^{-2}$ stored as %025252 %125252 %125252 %125376

Constant Lists

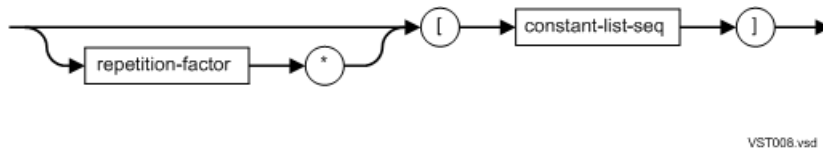
A constant list is a list of one or more constants. You can use constant lists in:

- initializations of array declarations that are not contained in structures
- group comparison expressions
- move statements

You cannot use constant lists in assignment statements



repetition-constant-list

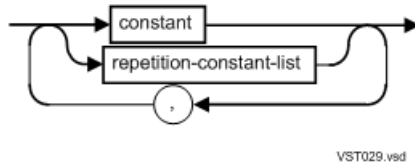


repetition-factor

is an INT constant that specifies the number of times *constant-list-seq* occurs.

constant-list-seq

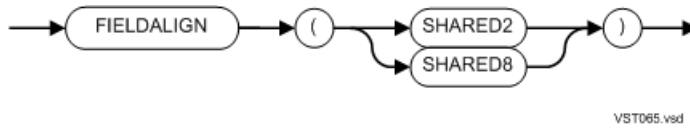
is a list of one or more constants, each stored on an element boundary:



constant

is a character string, a number, or a LITERAL specified as a single operand. The range and syntax for specifying constants depends on the data type, as described for each data type on preceding pages.

FIELDALIGN-clause



specifies how you want the compiler to align the base of the structure and fields in the structure. The offsets of fields in a structure are aligned relative to the base of the structure. For more information about constant list alignment, see [Constant List Alignment Specification \(page 64\)](#).

SHARED2

specifies that the base of the structure and each field in the structure must begin at an even byte address except STRING fields, which can begin at any byte address, and UNSIGNED fields.

SHARED8

specifies that the offset of each field in the structure from the base of the structure must be begin at an address that is an integral multiple of the width of the field.

Constant List Alignment Specification

A constant list alignment specification controls the alignment of elements of constant lists whose element type is not STRING. Such a constant list can have an alignment of SHARED2 or SHARED8. Nested constant lists cannot have an alignment specification; they inherit the alignment of the containing constant list. SHARED2 causes alignment identical to TAL. SHARED8 additionally requires that 4-byte and 8-byte scalars are aligned to their size. You must insert filler constants to ensure proper alignment of 4-byte and 8-byte aligned items. A SHARED8 constant list containing an item that is misaligned is an error.

An optional alignment specification gives the alignment of a constant list. It occurs immediately before the opening bracket of the constant list. There is no default constant list alignment. The alignment specification is required if SHARED2 and SHARED8 would give different results.

A1	':=' [1,2,3,4];	! No alignment specification ! required
A1	':=' FIELDALIGN(SHARED2) [1,2D,4];	! Alignment specification required ! to specify 2-byte alignment for ! 2D

Examples of constant lists:

1. In each of the following pairs, the list on the left is equivalent to the list on the right:

["A", "BCD" , "...", "Z"]	["ABCD...Z"]
10 * [0];	[0,0,0,0,0,0,0,0,0,0]
[3 * [2 * [1], 2 * [0]]]	[1,1,0,0,1,1,0,0,1,1,0,0]
10 * [" "]	[" "]

2. The following is an example of the FIELDALIGN clause:

```
STRING i [0:3] := FIELDALIGN(SHARED2) [0,1,2,3];
```

3. The following example shows how you can break a constant string that is too long for one line into smaller constant strings specified as a constant list. The system stores one character to a byte:

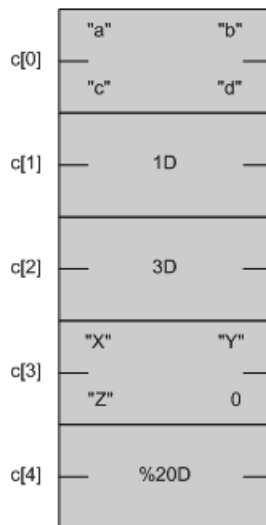
```
STRING a[0:99] := ["These three constant strings will ",
                  "appear as if they were one constant ",
                  "string continued on multiple lines."];
```

4. The following example initializes a STRING array with a repetition constant list:

```
STRING b[0:79] := 80 * [ " " ];
```

5. The following example initializes an INT(32) array with a mixed constant list containing values of the same data type. The diagram shows how the compiler allocates storage for the variable and the constant list that initializes the variable:

```
INT(32) c[0:4];
["abcd", 1D, 3D, "XYZ",
%20D];
!Mixed constant list
```



VST331.vsd

4 Data Alignment

In native mode, a data item is aligned if its address is a multiple of its size. For example, a 4-byte data item is aligned if its address is a multiple of four. An address that is not aligned is called misaligned. In native mode, a compiler requires data to be aligned unless otherwise indicated. Unexpected misalignment causes the program to run slowly, but usually with the expected results.

The only time a nonprivileged program running in TNS/R native mode could have data alignment problems is when calling the atomic routines whose names begin with “\$ATOMIC_”. Those routines operate correctly only when given aligned operand addresses. This section explains some ways to diagnose bad calls at run time.

TNS-compiled programs must follow more stringent alignment rules, which apply to all data. Those rules are explained in:

Product	T Number	Document
Accelerator	T9276	<i>Accelerator Manual Data Alignment Addendum</i>
TNS C	T9255	<i>C/C++ Programmer's Guide</i>
TNS C++	T9541	<i>C/C++ Programmer's Guide</i>
TNS c89	T8629	<i>C/C++ Programmer's Guide</i>
TNS COBOL	T9257	<i>COBOL Manual for TNS and TNS/R Programs</i>
TAL	T9250	<i>TAL Programmer's Guide Data Alignment Addendum</i>

Topics:

- [Misalignment Tracing Facility \(page 66\)](#)
- [Misalignment Handling \(page 67\)](#)

Misalignment Tracing Facility

The misalignment tracing facility is enabled or disabled on a system-wide basis (that is, for all processors in the node). By default, it is enabled (set to ON). It can be disabled (set to OFF) only by the persons who configure the system, by means of the Subsystem Control Facility (SCF) attribute MISALIGNLOG. Instructions are in the *SCF Reference Manual for the Kernel Subsystem*.

NOTE: HP recommends that the MISALIGNLOG attribute be left ON (its default setting) so that a process that is subject to rounding of misaligned addresses generates log entries, facilitating diagnosis and repair of the code. Only if the volume of misalignment events degrades performance should this attribute be turned OFF.

When a misaligned address causes an exception that RVUs prior to G06.17 would have rounded down, the tracing facility traces the exception.

NOTE: The tracing facility does not count and trace every misaligned address, only those that cause round-down exceptions. Other accesses that use misaligned addresses without rounding them down do not cause exceptions and are not counted or traced. Also, only a periodic sample of the counted exceptions are traced by means of their own EMS event messages.

While a process runs, the tracing facility:

- Counts the number of misaligned-address exceptions that the process causes (the exception count)
- Records the program address and code-file name of the instruction that causes the first misaligned-address exception

Because a process can run for a long time, the tracing facility samples the process (that is, checks its exception data) periodically (approximately once an hour). If the process recorded an exception since the previous sample, the tracing facility records an entry in the EMS log. If the process ends and an exception has occurred since the last sample, the operating system produces a final Event Management Service (EMS) event.

The EMS event includes:

- The process's exception count
- Details about one misaligned-address exception, including the program address, data address, and relevant code-file names

Sampling is short and infrequent enough to avoid flooding the EMS log, even for a continuous process with many misaligned-address exceptions. One sample logs a maximum of 100 events, and at most one event is logged for any process.

If misaligned-address exceptions occur in different periods of a process, the operating system produces multiple EMS events for the same process, and these EMS events might have different program addresses.

For more information about EMS events or the EMS log, see the *EMS Manual*.

Misalignment Handling

Misalignment handling is determined by the following SCF attributes, which are set system-wide (that is, for all processors in the node) by the persons who configure the system:

- MISALIGNLOG
- TNSMISALIGN (applies only to programs running in TNS mode or TNS accelerated mode, and therefore, does not apply to pTAL programs)
- NATIVEATOMICMISALIGN

MISALIGNLOG enables or disables the tracing facility (see [Misalignment Tracing Facility \(page 66\)](#)).

NATIVEATOMICMISALIGN applies to atomic routines in programs running in TNS/R native mode; that is, the pTAL and TAL routines whose names begin with "\$ATOMIC_".

For normal, nonatomic access in TNS/R native mode, the system uses the operand's full address (never rounded down) to complete the operation.

For normal, nonatomic access in TNS/R native mode, the system uses the operand's full address (never rounded down) to complete the operation.

[Table 22 \(page 67\)](#) lists and describes the possible settings for NATIVEATOMICMISALIGN. Each setting represents a different misalignment handling method. For more information about NATIVEATOMICMISALIGN, see the *SCF Reference Manual for the Kernel Subsystem*.

Table 22 TNS/R Native Atomic Misalignment Handling Methods

Method	Description
ROUND (default)	After rounding down a misaligned address, the system proceeds to access the address atomically, as in G06.16 and earlier RVUs.
FAIL	Instead of rounding down a misaligned address, the system considers the call to have failed. This failure generates a SIGILL signal (signal #4). By default, this signal causes process termination, but the program can specify other behavior (for example, entering the debugger or calling a specified signal-handler procedure). The signal cannot be ignored. For information about signal handling, see the explanation of the <code>sigaction()</code> function in the <i>Open System Services System Calls Reference Manual</i> .

The method that you choose does not apply to every misaligned address, only to those that would have been rounded down in earlier RVUs.

NOTE: ROUND misalignment handling is intended as a temporary solution, not as a substitute for changing your atomic calls to ensure that they have only aligned addresses. ROUND misalignment handling cannot be migrated to past and future NonStop OS platforms.

5 Expressions

An expression is a sequence of operands and operators that, when evaluated, produces a single value. Operands in an expression include variables, constants, and routine identifiers. Operators in an expression perform arithmetic or conditional operations on the operands. pTAL supports the following types of expressions:

Expression	Description	Examples
Arithmetic expression	An expression, consisting of operands and arithmetic operators, that produces a single numeric value.	<code>398 + num / 8410 LOR 12</code>
Address expression	An expression containing relational and or add and subtract arithmetic operators. You can use arithmetic expressions to compute addresses, compare addresses, or compare them to a constant.	<code>IF @p + 0 > @q THEN ...</code> <code>IF @p <> 0 THEN ...;</code> <code>IF @p - @q > 5 THEN ...;</code>
Constant expression	An arithmetic expression that contains only constants, LITERALS, and DEFINES as operands.	<code>398 + 46 / 84</code>
Conditional expression	An expression establishing the relationship between values and resulting in a true or false value. A conditional expression consists of relational conditions and conditional operators.	<code>a < c a OR b</code>

Expressions can appear in:

- LITERAL declarations
- Variable initialization and assignments
- Array and structure bounds
- Indexes to variables
- Conditional statements
- Parameters to procedures or subprocedures

An expression can be:

- A single operand, such as the number 5
- A unary plus or minus (+ or -) operator applied to a single operand, such as -5
- A binary operator applied to two operands, such as 5 * 8
- A complex sequence such as:
`((alpha + beta) / chi) * (delta - 145.9)) / zeta`

Topics:

- [Data Types of Expressions \(page 70\)](#)
- [Operator Precedence \(page 70\)](#)
- [Arithmetic Expressions \(page 72\)](#)
- [Signed Arithmetic Operators \(page 73\)](#)
- [Unsigned Arithmetic Operators \(page 75\)](#)

- [Comparing Addresses \(page 77\)](#)
- [Constant Expressions \(page 81\)](#)
- [Conditional Expressions \(page 81\)](#)
- [Special Expressions \(page 85\)](#)
- [Bit Operations \(page 92\)](#)

Data Types of Expressions

The result of an expression can be any data type or address type except `STRING` or `UNSIGNED`. The compiler determines the data type of the result from the data type of the operands in the expression. All operands in an expression must have the same data type, with the following exceptions:

- An `INT` expression can include `STRING`, `INT`, and `UNSIGNED(1-16)` operands. The system treats `STRING` and `UNSIGNED(1-16)` operands as if they were 16-bit values. That is, the system:
 - Puts a `STRING` operand in the right byte of a 16-bit word and sets the left byte to 0, with no sign extension.
 - Puts an `UNSIGNED(1-16)` operand in the right bits of a 16-bit word and sets the unused left bits to 0, with no sign extension. For example, for an `UNSIGNED(2)` operand, the system fills the 14 leftmost bits of the word with zeros.
- An `INT(32)` expression can include `INT(32)` and `UNSIGNED(17-31)` operands. The system treats `UNSIGNED(17-31)` operands as if they were 32-bit values. The system places an `UNSIGNED(17-31)` operand in the right bits of a doubleword and sets the unused left bits to 0, with no sign extension. For example, for an `UNSIGNED(29)` operand, the system fills the three leftmost bits of the doubleword with zeros.

In all other cases, if the data types do not match, use the type transfer functions described in [Chapter 15 \(page 274\)](#).

Operator Precedence

Operators in expressions can be arithmetic (signed, unsigned, or logical) or conditional (relational, signed or unsigned). Within an expression, the compiler evaluates the operators in the order of precedence. Within each level of precedence, the compiler evaluates the operators from left to right.

Table 23 Precedence of Operators

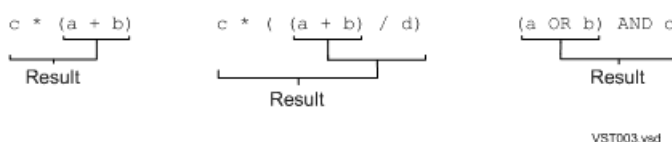
Operator	Operation	Precedence
<<	Signed left bit shift	0 (highest)
>>	Signed right bit shift	
'<<'	Unsigned left bit shift	
'>>'	Unsigned right bit shift	
[<i>n</i>]	Indexing	1
@	Address of identifier	
+	Unary plus	
-	Unary minus	2
<...>	Bit extraction	
*	Signed multiplication	3

Table 23 Precedence of Operators *(continued)*

Operator	Operation	Precedence
/	Signed division	
'*'	Unsigned multiplication	
'/'	Unsigned division	
'\'	Unsigned remainder	
+	Signed addition	4
-	Signed subtraction	
'+'	Unsigned addition	
'-'	Unsigned subtraction	
LOR	Bitwise logical OR	
LAND	Bitwise logical AND	
XOR	Bitwise exclusive OR	
<	Signed less than	5
=	Signed equal to	
>	Signed greater than	
<=	Signed less than or equal to	
>=	Signed greater than or equal to	
<>	Signed not equal to	
'<'	Unsigned less than	
'='	Unsigned equal to	
'>'	Unsigned greater than	
'<='	Unsigned less than or equal to	
'>='	Unsigned greater than or equal to	
'<>'	Unsigned not equal to	
NOT	Negation	6
AND	Conjunction	7
OR	Disjunction	8
:=	Assignment	9 (lowest)
<...> :=	Bit deposit	

You can use parentheses to override the precedence of operators. You can nest the parenthesized operations. The compiler evaluates nested parenthesized operations outward starting with the innermost level.

Figure 1 Parentheses' Effect on Operator Precedence



Arithmetic Expressions

An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type.



$+$, $-$

are unary plus and minus operators. The default is unary plus.

operand

is one of the elements in [Table 24 \(page 72\)](#).

arithmetic-operator

is one of the following:

Signed arithmetic operator	$+$, $-$, $*$, $/$
Unsigned arithmetic operator	$'+$ ', $'-'$, $'*'$, $'/'$, $'\backslash'$
Logical operator	LOR, LAND, XOR

Table 24 Operands in Arithmetic Expressions

Element	Description	Example
Variable	The identifier of a simple variable, array element, pointer, structure data item, or equivalenced variable, with or without @ or an index	var[10]
Constant	A character string or numeric constant	103375
LITERAL	The identifier of a named constant	file_size
Function invocation	The invocation of a procedure that returns a value	\$LEN (x)
expression	Any expression	x := y
(expression)	Any expression, enclosed in parentheses	(x := y)
Code space item	The identifier of a procedure, subprocedure, or label prefixed with @ or a read-only array optionally prefixed with @, with or without an index	@label_a

Table 25 Arithmetic Expressions

Syntax	Example
<i>operand</i>	var-1
<i>- operand</i>	-var-1
<i>+ operand arithmetic-operator operand</i>	+var-1 * 2
<i>operand arithmetic-operator operand</i>	var-1 / var-2
<i>operand arithmetic-operator operand</i>	var-1 / (-var-2)

Table 25 Arithmetic Expressions *(continued)*

Syntax	Example
<i>expression operand expression</i>	<code>2 * 3 + var / 2</code>
<i>expression operand expression</i>	<code>2 * var * 4</code>

A condition code cannot appear inside an arithmetic expression; for example, the following is not valid in pTAL:

```
a := <; !Illegal
```

Signed Arithmetic Operators

Table 26 Signed Arithmetic Operators

Operator	Operation	Operand Type*	Example
+	Unary plus	Any data type	+5
-	Unary minus	Any data type	-5
+	Binary signed addition	Any data type	alpha + beta
-	Binary signed subtraction	Any data type	alpha - beta
*	Binary signed multiplication	Any data type	alpha * beta
/	Binary signed division	Any data type	alpha / beta

* The data type of the operands must match, except as noted in [Data Types of Expressions \(page 70\)](#).

In [Table 27 \(page 73\)](#), the order of the data types is interchangeable.

Table 27 Signed Arithmetic Operand and Result Types

Operand Type	Operand Type	Result Type	Example
STRING	STRING	INT	<code>byte1 + byte2</code>
INT	INT	INT	<code>word1 - word2</code>
INT(32)	INT(32)	INT(32)	<code>dbl1 * dbl2</code>
REAL	REAL	REAL	<code>real1 + real2</code>
REAL(64)	REAL(64)	REAL(64)	<code>quad1 + quad2</code>
FIXED	FIXED	FIXED	<code>fixed1 * fixed2</code>
INT	STRING	INT	<code>word1 / byte1</code>
INT	UNSIGNED(1-16)	INT	<code>word + unsign12</code>
INT(32)	UNSIGNED(17-31)	INT(32)	<code>double + unsign20</code>
UNSIGNED(1-16)	UNSIGNED(1-16)	INT	<code>unsign6 + unsign9</code>
UNSIGNED(17-31)	UNSIGNED(17-31)	INT(32)	<code>unsign26 + unsign31</code>

The compiler treats a STRING or UNSIGNED(1-16) operand as an INT operand. If bit <0> contains 0, the operand is positive; if bit <0> contains 1, the operand is negative. For more information, see [Data Types of Expressions \(page 70\)](#).

The compiler treats an UNSIGNED(17-31) operand as a positive INT(32) operand.

Signed arithmetic operators affect the hardware indicators as described in [Chapter 13 \(page 234\)](#).

Topics:

- [Scaling of FIXED Operands \(page 74\)](#)
- [Using FIXED\(*\) Variables \(page 74\)](#)

Scaling of FIXED Operands

When you declare a FIXED variable, you can specify an implied fixed-point (*fpoint*) setting (see [Specifying Data Types \(page 47\)](#)).

When FIXED operands in an arithmetic expression have different fixed-points, the system “scales” them, depending on the operation:

Operation	Scaling
Addition or subtraction	The system adjusts the smaller fixed-point to match the larger fixed-point. The result inherits the larger fixed-point. For example, the system adjusts the smaller fixed-point in <code>3.005F + 6.01F</code> to <code>6.010F</code> , and the result is <code>9.015F</code> .
Multiplication	The fixed-point of the result is the sum of the fixed-points of the two operands. For example, <code>3.091F * 2.56F</code> results in the <code>FIXED(5)</code> value <code>7.91296F</code> .
Division	The fixed-point of the result is the fixed-point of the dividend minus the fixed-point of the divisor (some precision is lost). For example, <code>4.05F / 2.10F</code> results in the <code>FIXED</code> value <code>1</code> .

To retain precision when you divide operands that have nonzero fixed-points, use the routine [\\$SCALE \(page 487\)](#).

Using FIXED(*) Variables

pTAL does not scale data items that it stores into `FIXED(*)` items.

The following procedure has one local variable whose data type is `FIXED(*)`:

```
PROC p;  
BEGIN  
    FIXED(*) f;  
    f := 1234F;  
END;
```

The data type of a `FIXED(*)` variable is the same as a `FIXED` variable when it is used in an expression:

```
FIXED(*) f1 := 123F;  
FIXED(2) f2;  
f2 := f1;                ! f2 is assigned 123.00
```

pTAL does not scale data when it is stored into a `FIXED(*)` variable:

```
FIXED(2) f1 := 1.23F;  
FIXED(*) f2;  
FIXED    f3;  
f2 := f1;                ! f2 is assigned 123  
f3 := f1;                ! f3 is assigned 1
```

The following example further illustrates this:

```
FIXED(*) f1;  
FIXED(3) f2 := 1.234F;  
f1 := f2;                ! f1 = 1234  
f2 := f1;                ! f2 = 1234.000  
f1 := f2;                ! f1 = 1234000  
f2 := f1;                ! f2 = 1234000.000  
f1 := f2;                ! f1 = 1234000000  
f2 := f1;                ! f2 = 1234000000.000
```

Unsigned Arithmetic Operators

Typically, you use binary unsigned arithmetic on operands with values in the range 0 through 65,535. For example, you can use unsigned arithmetic with pointers that contain standard addresses.

Table 28 Unsigned Arithmetic Operators

Operator	Operation	Operand Type	Example
'+'	Unsigned addition	STRING, INT, or UNSIGNED(1-16)	alpha '+' beta
'-'	Unsigned subtraction	STRING, INT, or UNSIGNED(1-16)	alpha '-' beta
'*'	Unsigned multiplication	STRING, INT, or UNSIGNED(1-16)	alpha '*' beta
'/'	Unsigned division	INT(32) or UNSIGNED (17-31) dividend and STRING, INT, or UNSIGNED(1-16) divisor	alpha '/' beta
'\''	Unsigned remainder*	INT(32) or UNSIGNED (17-31) dividend and STRING, INT, or UNSIGNED(1-16) divisor	alpha '\ ' beta

* If the quotient exceeds 16 bits, an overflow condition occurs and the results will have unpredictable values. For example, the operation 200000D '\ ' 2 causes an overflow because the quotient exceeds 16 bits.

In [Table 29 \(page 75\)](#), the order of the operand types in each combination is interchangeable except in the last case.

Table 29 Unsigned Arithmetic Operand and Result Types

Operator	Operand Type	Operand Type	Result Type	Example
'+' '-'	STRING	STRING	INT	byte1 '-' byte2
	INT	INT	INT	word1 '+' word2
	INT	STRING	INT	byte1 '-' word1
	INT	UNSIGNED (1-16)	INT	word1 '+' uns8
	STRING	UNSIGNED (1-16)	INT	byte1 '-' uns5
	UNSIGNED(1-16)	UNSIGNED(1-16)	INT	uns1 '+' uns7
'*'	STRING	STRING	INT(32)	byte1 '*' byte2
	INT	INT	INT(32)	word1 '*' word2
	STRING	INT	INT(32)	byte1 '*' word1
	INT	UNSIGNED (1-16)	INT(32)	word1 '*' uns9
	STRING	UNSIGNED (1-16)	INT(32)	uns1 '*' uns7
	UNSIGNED(1-16)	UNSIGNED(1-16)	INT(32)	uns1 '*' uns7
'/' '\'	UNSIGNED(17-31) or INT(32) dividend	STRING, INT, or UNSIGNED(1-16) divisor	INT	dbwd '\ ' word1

Topics:

- [Bitwise Logical Operators \(page 76\)](#)
- [Using Bitwise Logical Operators and INT\(32\) Operands \(page 76\)](#)

Bitwise Logical Operators

Use bitwise logical operators (LOR, LAND, and XOR) to perform bit-by-bit operations on STRING, INT, UNSIGNED(1-16) operands. Use INT(32) operands to return INT(32) results. 16-bit operands produce a 16-bit result. 32-bit operands produce a 32-bit result. Bitwise logical operators are not defined for 64-bit operands.

Table 30 Bitwise Logical Operators

Operator	Operation	Operand Type	Bit Operations	Example
LOR	Bitwise logical OR	STRING, INT, or UNSIGNED(1-16)	$1 \text{ LOR } 1 = 1$ $1 \text{ LOR } 0 = 1$ $0 \text{ LOR } 0 = 0$	$10 \text{ LOR } 12 = 14$ $10 \quad 1 \ 0 \ 1 \ 0$ $12 \quad 1 \ 1 \ 0 \ 0$ <hr/> $14 \quad 1 \ 1 \ 1 \ 0$
LAND	Bitwise logical AND	STRING, INT, or UNSIGNED(1-16)	$1 \text{ LAND } 1 = 1$ $1 \text{ LAND } 0 = 0$ $0 \text{ LAND } 0 = 0$	$10 \text{ LAND } 12 = 8$ $10 \quad 1 \ 0 \ 1 \ 0$ $12 \quad 1 \ 1 \ 0 \ 0$ <hr/> $8 \quad 1 \ 0 \ 0 \ 0$
XOR	Bitwise exclusive OR	STRING, INT, or UNSIGNED(1-16)	$1 \text{ XOR } 1 = 0$ $1 \text{ XOR } 0 = 1$ $0 \text{ XOR } 0 = 0$	$10 \text{ XOR } 12 = 6$ $10 \quad 1 \ 0 \ 1 \ 0$ $12 \quad 1 \ 1 \ 0 \ 0$ <hr/> $6 \quad 0 \ 1 \ 1 \ 0$

The Bit Operations column in [Table 30 \(page 76\)](#) shows the bit-by-bit operations that occur on 16-bit values. Each 1-bit operand pair results in a 1-bit result. The bit operands are commutative.

Using Bitwise Logical Operators and INT(32) Operands

You can use INT(32) operands with:

- Logical operators (LOR, LAND, and XOR)
The following example swaps the values stored in `i` and `j`:

```
INT(32) i;
INT(32) j;
i := i XOR j;
j := i XOR j;
i := i XOR j;
```
- Unsigned relational operators ('<', '<=', '=', '<>', '>=', and '>').
The INT(32) operands are treated as nonnegative values in the range 0 to 232-1.
- Unsigned addition and subtraction operators ('+' and '-')
The INT(32) operands are treated as nonnegative values in the range 0 to 232-1.
Unsigned and signed addition and subtraction are the same except that [\\$OVERFLOW \(page 335\)](#) returns false after an unsigned operation.
- Unsigned multiplication operator ('*')
The INT(32) operands are treated as nonnegative values in the range 0 to 232-1. The unsigned product of two INT(32) values is an FIXED value. [\\$OVERFLOW \(page 335\)](#) returns false after an unsigned multiplication operator.
- Unsigned division and remainder operators ('/' and '\')
You can use an FIXED dividend and INT(32) divisor with the unsigned-division and remainder operators. The FIXED dividend is treated as a nonnegative value in the range 0 to 264-1. The INT(32) divisor is treated as a nonnegative value in the range 0 to 232-1.

The quotient or remainder of an FIXED dividend and an INT(32) divisor is an INT(32) quotient in the range 0 to 232-1.

`$OVERFLOW` (page 335) returns false after an unsigned division or remainder operator unless either of the following is true:

- The divisor is 0
- The quotient is greater than 216-1 for an INT quotient, 232-1 for an INT(32)

Comparing Addresses

pTAL rules for comparing address types are more restrictive than the rules for comparing nonaddress types.

Table 31 Valid Address-Type Comparisons

	Extended Addresses	Non-Extended Addresses
Operators	Unsigned relational operators: none Signed relational operators: <, =, >, <=, <>, >=	Unsigned relational operators: '<', '=', '>', '<=', '<>', '>=' Signed relational operators: =, <>
Abbreviated forms	Testing address type as true or false: IF @p THEN IF NOT @p THEN...	Testing address type as true or false: IF @p THEN IF NOT @p THEN...

Topics:

- [Extended Addresses \(page 77\)](#)
- [Nonextended Addresses \(page 78\)](#)

Extended Addresses

The following rules apply when you compare extended addresses (EXTADDRs, EXT32ADDRs, and EXT64ADDRs):

- Use signed relational operators to compare extended addresses. Unsigned operators are not valid.
- If you compare an EXTADDR or EXT32ADDR address to a constant, the constant must be a 32-bit integer.
- If you compare an EXT64ADDR address to a constant, the constant must be of type FIXED.
- If you compare two different sized extended addresses, the smaller address is implicitly cast to the larger address and then compared.

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, [“64-bit Addressing Functionality” \(page 531\)](#).

Example 11 Extended Addresses

```
EXTADDR e;
EXT32ADDR e32;
EXT64ADDR e64;
INT .EXT i;
INT .EXT32 j;
INT .EXT64 k;

IF e < @i THEN ...      ! OK: e and @i are both EXTADDR
IF e < @j THEN ...      ! OK, e and @j are both extended addresses
IF e < @k THEN ...      ! OK, e and @k are both extended addresses
IF @i >= 0D THEN ...     ! OK: @i is EXTADDR, 0D is 32 bits
IF @j >= 0D THEN ...     ! OK: @j is EXT32ADDR, 0D is 32 bits
IF @k >= 0F THEN ...     ! OK: @K is EXT64ADDR, 0F is 64 bits
IF e = 0D THEN ...       ! OK
IF e <> 0D THEN ...       ! OK
IF e THEN ...            ! OK
IF NOT e THEN ...        ! OK
IF e32 = 0D THEN ...     ! OK
IF e32 <> 0D THEN ...     ! OK
IF e32 THEN ...          ! OK
IF NOT e32 THEN ...      ! OK
IF e64 = 0F THEN ...     ! OK
IF e64 <> 0F THEN ...     ! OK
IF e64 THEN ...          ! OK
IF NOT e64 THEN ...      ! OK
IF e > i THEN ...        ! ERROR: e is EXTADDR, i is INT
IF e32 > i THEN ...      ! ERROR: e32 is EXT32ADDR, i is INT
IF e64 > i THEN ...      ! ERROR: e64 is EXT64ADDR, i is INT
IF e '<' @i THEN ...      ! ERROR: Unsigned operators are
                        !         not valid with EXTADDRs
IF e32 '<>' 0D THEN ...    ! ERROR: Unsigned operators are
                        !         not valid with EXT32ADDRs
IF e64 '>' 0F THEN ...    ! ERROR: Unsigned operators are
                        !         not valid with EXT64ADDRs
```

NOTE: The address types, EXT32ADDR and EXT64ADDR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Nonextended Addresses

The following rules apply when you compare nonextended addresses:

- Use unsigned relational operators ('<', '=', '>', '<=', '<>', '>='), a signed equality operator (=), or a signed inequality operator (<>) to compare nonextended addresses. The signed and unsigned equality operators produce the same results. Similarly, the signed and unsigned inequality operators produce the same results.
- Do not compare nonextended addresses using signed operators that test for greater than or less than (<, <=, >, >=).

Valid comparisons:

```
INT .p, .q;
IF @p = 0 THEN ...
IF @p <> 0 THEN ...
IF @p = @q THEN ...
```

Both operands must have the same address type:

```
STRING .s;
BADDR .b;
```

```

IF @s = b THEN ... ! OK: @s is BADDR, b is BADDR
IF @s = @b THEN ... ! ERROR: @s is BADDR, @b is WADDR

```

- If one operand of a relational operator is a nonextended address and the other is a constant, the constant must be 16 bits in length:

```

INT .p;
IF @p = 100 THEN ... ! OK
IF @p = 100D THEN ... ! ERR

```

Table 32 Valid Address Expressions

Template	Result Type	Examples
<i>atype</i> [k];	<i>atype</i> *	INT .EXT p; @p := @p[2];
<i>atype</i> '+' INT	<i>atype</i> *	INT .p; @p := @p '+' 2; @p := @p '-' 4;
INT '+' <i>atype</i>	<i>atype</i> *	INT .p; @p := 2 '+' @p;
EXTADDR '±' INT(32)	EXTADDR	INT .EXT p; @p := @p + 4D; @p := @p - 4D;
INT(32) '+' EXTADDR	EXTADDR	INT .EXT p; @p := 4D '+' @p;
EXT32ADDR ± INT(32)	EXT32ADDR**	INT .EXT32 p; @p := @p + 4D; @p := @p - 4D;
INT(32) + EXT32ADDR	EXT32ADDR**	INT .EXT32 p; @p := 4D + @p;
EXT64ADDR ± FIXED	EXT64ADDR**	INT .EXT64 p; @p := @p + 8F; @p := @p - 8F;
FIXED + EXT64ADDR	EXT64ADDR**	INT .EXT64 p; @p := 8F + @p;
<i>atype</i> '-' <i>atype</i>	INT	INT .b, .bp, i; i := @bp '-' @b; The result of subtracting two byte-oriented (BADDR, CBADDR, SGBADDR, SGXBADDR) addresses is the number of bytes between them. The result of subtracting two word-oriented (WADDR, CWADDR, SGWADDR, SGXWADDR) addresses is the number of 16-bit words between them.
EXTADDR - EXTADDR	INT(32)**	INT .EXT b, bp, i32; i32 := @bp - @b; EXT32ADDR - EXT32ADDR INT(32) INT.EXT32 b, bp; INT(32) i32; i32 := @bp - @b; EXTADDR - EXT32ADDR INT(32) INT.EXT b; INT.EXT32 bp; INT(32) i32; i32 := @bp - @b; EXT32ADDR - EXTADDR INT(32) INT.EXT32 b; INT.EXT bp;

Table 32 Valid Address Expressions *(continued)*

Template	Result Type	Examples
		<pre>INT(32) i32; i32 := @bp - @b;</pre> <pre>EXT64ADDR - EXT64ADDR FIXED(0) INT .EXT64 b bp; FIXED i64; i64 := @pb - b;</pre>
<i>atype relational atype</i>	INT	<pre>BADDR b1, b2; INT i; IF b1 '<' b2 THEN ...; i := b1 '<' b2;</pre> <p><i>relational</i> must be an unsigned relational operation ('<', '=', '>', '<=', '<>', '>=') or signed equal or not equal (=, <>).</p>
<i>Extended address type relational extended address type</i>	INT	<pre>EXTADDR ea1, ea2; EXT32ADDR e32a1, e32a2; EXT64ADDR e64a1, e64a2; INT i; IF ea1 < ea2 THEN ...; i := b1 < b2; IF e32a1 < ea32a2 THEN ...; i := e32a1 < ea32a2; IF ea64a1 < ea64a2 THEN ...; i := e64a1 < ea64a2; IF ea1 < e32a1 THEN ...; IF ea1 < e32a1 THEN ...; IF ea1 < e64a1 THEN ...; IF e64a1 < ea1 THEN ...; IF e64a1 < e32a1 THEN ...; IF e32a1 < e64a1 THEN ...;</pre> <p>If the sizes of the extended addresses differ, the smaller address is implicitly sign-extended to the size of the larger address before the comparison is performed.</p>
<i>atype relational CONSTANT</i>	INT	<pre>BADDR b1; INT i; IF b1 '>' 100 THEN ...; i := b1 '<>' nil;</pre> <p><i>relational</i> must be an unsigned relational operation ('<', '=', '>', '<=', '<>', '>=') or signed equal or not equal (=, <>).</p>
<i>Extended address relational CONSTANT</i>	INT	<pre>EXTADDR ea; EXT32ADDR e32a; EXT64ADDR e64a; INT i; IF ea < 0D THEN ...; IF e32a >= 0D THEN ... IF e64a <> 0F THEN ... i := b1 < 65535D; i := e32a > 100D; i := e64a = 10F;</pre> <p><i>relational</i> must be a signed relational operation (<, =, >, <=, <>, >=).</p> <p>CONSTANT relational extended address INT.</p> <pre>EXTADDR ea; EXT32ADDR e32a; EXT64ADDR e64a; INT I; IF 0D > ea THEN ...</pre>

Table 32 Valid Address Expressions *(continued)*

Template	Result Type	Examples
		IF 20d < e32a THEN ... IF 0D <> e64a THEN ...
* <i>atype</i> represents any address type except PROCADDR or EXTADDR.		
** 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).		

Constant Expressions

A constant expression is an arithmetic expression that contains only constants, LITERALS, or DEFINES as operands.

You can use a constant expression anywhere a single constant is allowed.

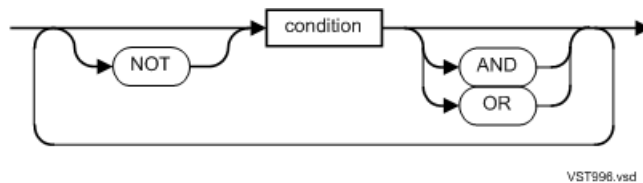
Example 12 Constant Expressions

```
255
8 * 5 + 45 / 2
```

For more information, see [Chapter 6](#) (page 97).

Conditional Expressions

A conditional expression is a sequence of conditions and relational operators that establishes the relationship between values. You can use conditional expressions to direct program flow.



NOT

is an operator that produces a true state if *condition* has the value false:

Value of a	Value of NOT a
True	False
False	True

condition

is an expression whose value is either true or false.

AND

is an operator that produces a true state if both its operands have the value true:

Value of a	Value of b	Value of a AND b
True	True	True
True	False	False
False	True	False
False	False	False

OR

is an operator that produces a true state if at least one of its operands has the value true:

Value of a	Value of b	Value of a AND b
True	True	True
True	False	False
False	True	False
False	False	False

Table 33 Conditional Expressions

Syntax	Example
<i>condition</i>	<i>a</i>
<i>NOT condition</i>	<i>NOT a</i>
<i>condition OR condition</i>	<i>a OR b</i>
<i>condition AND condition</i>	<i>a AND b</i>
<i>condition AND NOT condition</i>	<i>a AND NOT b OR c</i>

Table 34 Conditions in Conditional Expressions

Element	Description	Example
Relational expression	Two conditions connected by a relational operator. The result type is INT; a -1 if true or a 0 if false. The example is true if A equals B.	<i>If a = b THEN ...</i>
Group comparison expression	Unsigned comparison of a group of contiguous elements with another. The result type is INT; a -1 if true or a 0 if false. The example compares 20 words of two INT arrays.	<i>IF a = b FOR 20 WORDS THEN ...</i>
(conditional expression)	A conditional expression enclosed in parentheses. The result type is INT; a -1 if true or a 0 if false. The example is true if both B and C are false. The system evaluates the parenthesized condition first, then applies the NOT operator.	<i>IF NOT (b OR c) THEN ...</i>
Arithmetic expression	An arithmetic, assignment, CASE, or IF expression that has an INT or INT(32) result*. The expression is treated as true if its value is not 0 and false if its value is 0. The example is true if the value of X is not 0.	<i>IF x THEN ...</i>
Relational operator	A signed or unsigned relational operator that tests a condition code. Condition code settings are CCL (negative), CCE (0), or CCG (positive). The example is true if the condition code setting is CCL.	<i>IF < THEN ...</i>

* If an arithmetic expression has a result other than INT, use a signed relational expression.

Topics:

- [NOT, OR, and AND Operators \(page 82\)](#)
- [Relational Operators \(page 83\)](#)

NOT, OR, and AND Operators

You use the operators NOT, OR, and AND to set the state of a single value or the relationship between two values.

Table 35 Results of NOT, OR, and AND Operators

Operator	Operation	Operand Type	Result	Example
NOT	Negation; tests condition for false state	STRING, INT, or UNSIGNED(1-16)	True/False	NOT a
OR	Disjunction; produces true state if either adjacent condition is true	STRING, INT, or UNSIGNED(1-16)	True/False	a OR b
AND	Conjunction; produces true state if both adjacent conditions are true	STRING, INT, or UNSIGNED(1-16)	True/False	a AND b

Topics:

- [Evaluating NOT, OR, and AND Operations \(page 83\)](#)
- [NOT, OR, and AND Operators and Condition Codes \(page 83\)](#)

Evaluating NOT, OR, and AND Operations

NOT, OR, and AND operations are evaluated by means of “short-circuit expression evaluation;” that is:

- Conditions connected by the OR operator are evaluated from left to right only until a true condition occurs.
- Conditions connected by the AND operator are evaluated from left to right until a false condition occurs. The next condition is evaluated only if the preceding condition is true.

In [Example 13 \(page 83\)](#), function `f` will not be called because `a <> 0` is false.

Example 13 Short-Circuit Expression Evaluation

```
a := 0;
IF a <> 0 AND f(x) THEN ... ;
```

NOT, OR, and AND Operators and Condition Codes

If the root operator in the conditional expression of an IF statement is a relational operator (<, =, >, <=, <>, >=, '<', '=', '>', '<=', '<>', '>='), pTAL sets the condition code according to the result of the comparison.

Relational operators that test the condition code (for example, IF < THEN...) do not set the condition code.

NOT, OR, and AND operators set the condition code indicator as described in [Chapter 13 \(page 234\)](#).

Relational Operators

- [Signed Relational Operators \(page 83\)](#)
- [Unsigned Relational Operators \(page 84\)](#)

Signed Relational Operators

Signed relational operators perform signed comparison of two operands and return a true or false state.

Table 36 Signed Relational Operators

Operator	Meaning	Operand Type*	Result
<	Signed less than	Any data type	True/False
=	Signed equal to	Any data type	True/False
>	Signed greater than	Any data type	True/False
<=	Signed less than or equal to	Any data type	True/False
>=	Signed greater than or equal to	Any data type	True/False
<>	Signed not equal to	Any data type	True/False

* The data type of the operands must match except as noted in [Data Types of Expressions \(page 70\)](#). Only the operators = and <> can be used when comparing operands of procedure pointer types (PROC_PTR, PROC32_PTR, and PROC64_PTR). For more information, see [“Procedure Pointers” \(page 263\)](#) and [“Syntax Summary” \(page 432\)](#). Comparison of procedure pointer types are only allowed if the signatures of the types (return and parameter types) match.

Unsigned Relational Operators

Unsigned relational operators perform unsigned comparison of two operands and return a true or false state.

Table 37 Unsigned Relational Operators

Operator	Operation	Operand Type*	Result
'<'	Unsigned less than	STRING, INT, INT(32), UNSIGNED (1-16)	True/False
'='	Unsigned equal to	STRING, INT, UNSIGNED (1-16)	True/False
'>'	Unsigned greater than	STRING, INT, INT(32), UNSIGNED (1-16)	True/False
'<='	Unsigned less than or equal to	STRING, INT, INT(32), UNSIGNED (1-16)	True/False
'>='	Unsigned greater than or equal to	STRING, INT, INT(32), UNSIGNED (1-16)	True/False
'<>'	Unsigned not equal to	STRING, INT, INT(32), UNSIGNED (1-16)	True/False

*Unsigned relational operators cannot be used with operands of the extended address (EXTADDR, EXT32ADDR, and EXT64ADDR) and the procedure pointer (PROC_PTR, PROC32_PTR, and PROC64_PTR) types.

NOTE: The extended address (EXTADDR, EXT32ADDR, and EXT64ADDR) and the procedure pointer (PROC_PTR, PROC32_PTR, and PROC64_PTR) types are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see [Appendix E, “64-bit Addressing Functionality” \(page 531\)](#).

A condition code reflects the value of the most recently evaluated root operator.

Used with no operands, signed and unsigned operators are equivalent. The result returned by such a relational operator is:

Relational Operator	Result Returned
< or '<'	True if CCL
> or '>'	True if CCG
= or '='	True if CCE
<> or '<>'	True if not CCE

Relational Operator	Result Returned
<= or '<='	True if CCL or CCE
>= or '>='	True if CCE or CCG

Examples of unsigned operators are as follows:

```

IF VAR '>' 10 THEN
...
ELSE IF < THEN    -- Unsigned less than
... ;
IF VAR '<' 5 THEN
...
ELSE IF '>' THEN  -- Unsigned greater than
... ;

```

Special Expressions

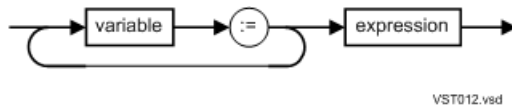
Special expressions allow you to perform specialized arithmetic or conditional operations.

Table 38 Special Expressions

Special Expression	Expression Type	Description
Assignment	Arithmetic	Assigns the value of an expression to a variable
CASE	Arithmetic	Selects one of several expressions
IF	Arithmetic	Selects one of two expressions
Group Comparison	Conditional	Performs unsigned comparison of two sets of data

Assignment

The assignment expression assigns the value of an expression to a variable.



variable

is the identifier of a variable in which to store the result of *expression* (*variable* can have an optional bit-deposit field).

expression

is an expression of the same data type as *variable*. The result of *expression* becomes the result of the assignment expression. *expression* is either:

- An arithmetic expression
- A conditional expression (excluding a relational operator with no operands), the result of which has data type INT.

Examples of assignment expressions:

1. This example decrements a. As long as a- 1 is not 0, the condition is true and the THEN clause is executed:

```
IF (a := a - 1) THEN ... ;
```

2. This example shows the assignment form used as an index. It decrements a and accesses the next array element:

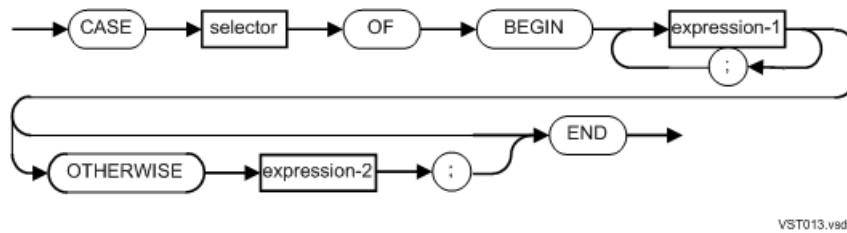
```
IF array[a := a - 1] <> 0 THEN ... ;
```

3. This example mixes the assignment form with a relational form. It assigns the value of *b* to *a*, then checks for equality with 0:

```
IF (a := b) = 0 THEN ... ;
```

CASE

The CASE expression selects one of several expressions.



selector

is an arithmetic expression [INT or INT(32)] that selects the expression to evaluate.

expression-1

is an expression of any data type. If you specify more than one *expression-1*, their data types must be compatible.

expression-2

is an expression whose data type is compatible with the data type of *expression-1*. It is evaluated if *selector* does not select an *expression-1*. If you omit the OTHERWISE clause and an out-of-range case occurs, results are unpredictable.

All expressions in the CASE statement must have compatible data types:

- Every data type is compatible with itself.
- String and unsigned (1-16) data types are compatible with INT
- Unsigned (17-32) data types are compatible with INT(32)
- All fixed-point data types are compatible with each other; however, if *expression-1* and *expression-2* are differently scaled fixed-point expressions, the data type of the IF expression is the data type of the fixed-point expression whose scale factor is largest. The smaller operand is implicitly scaled to match the type of the IF expression. For example, these return statements are equivalent:

INT i;	INT i;
FIXED(2) f2;	FIXED(2) f2;
FIXED(3) f3;	FIXED(3) f3;
FIXED(3) PROC p;	FIXED(3) PROC p;
BEGIN	BEGIN
RETURN IF i THEN f2	RETURN IF i THEN \$SCALE(f2,1);
ELSE f3;	ELSE f3;
END	END

The compiler numbers the instances of *expression-1* consecutively, starting with 0. If *selector* matches the compiler-assigned number of an *expression-1*, that *expression-1* is evaluated and becomes the result of the CASE expression. If *selector* does not match a compiler-assigned number, *expression-2* is evaluated.

You can nest CASE expressions. CASE expressions resemble unlabeled CASE statements except that CASE expressions select expressions rather than statements.

[Example 14 \(page 87\)](#) selects an expression based on the value of *a* and assigns it to *x*:

Example 14 CASE Expression

```
INT x, a, b, c, d;  
!Code to initialize variables  
x := CASE a OF  
  BEGIN  
    b;           ! If a is 0, assign value of b to X.  
    c;           ! If a is 1, assign value of c to X.  
    d;           ! If a is 2, assign value of d to X.  
    OTHERWISE -1; ! If a is any other value,  
  END;           ! assign -1 to x.
```

IF

The IF expression selects one of two expressions, usually for assignment to a variable.



condition

is either:

- A conditional expression
- An INT arithmetic expression. If the result of the arithmetic expression is not 0, the *condition* is true. If the result is 0, *condition* is false.

expression-1, expression-2

are expressions of any data type, but their data types must be compatible:

- Every data type is compatible with itself.
- String and unsigned (1-16) data types are compatible with INT
- Unsigned (17-32) data types are compatible with INT(32)
- All fixed-point data types are compatible with each other; however, if *expression-1* and *expression-2* are differently scaled fixed-point expressions, the data type of the CASE expression is the data type of the fixed-point expression whose scale factor is largest. For example, these return statements are equivalent:

INT i; FIXED(-2) f2; FIXED(-3) f3; FIXED(-4) f4; FIXED(-2) proc p; BEGIN RETURN CASE i OF BEGIN f3; f2; OTHERWISE f4 END END	INT i; FIXED(-2) f2; FIXED(-3) f3; FIXED(-4) f4; FIXED(-2) PROC p; BEGIN RETURN CASE i OF BEGIN \$SCALE(f3,1); f2; OTHERWISE \$SCALE(f4,2) END END
--	--

If *condition* is true, the result of the *expression-1* becomes the result of the overall IF expression.

If *condition* is false, the result of *expression-2* becomes the result of the overall IF expression.

You can nest IF expressions within an IF expression or within other expressions. The IF expression resembles the [IF \(page 217\)](#) except that the IF expression:

- Requires the ELSE clause
- Contains expressions, not statements

Examples of IF expressions:

1. This example assigns an arithmetic expression to `var` based on the condition `length > 0`:

```
var := IF length > 0 THEN 10 ELSE 20;
```

2. This example nests an IF expression (in parentheses) within another expression:

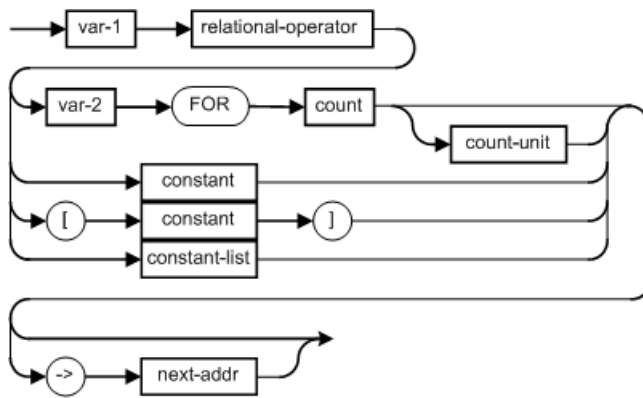
```
var * index + (IF index > limit THEN var * 2 ELSE var * 3)
```

3. This example nests an IF expression within another IF expression:

```
var := IF length < 0 THEN -1
      ELSE IF length = 0 THEN 0
      ELSE 1;
```

Group Comparison

The group comparison expression compares a variable with a variable or constant. With PVU T9248AAD, you can compare any variable up to the current maximum allowed size for any object of 127.5 megabytes.



VST015.vsd

var-1

is the identifier of a variable, with or without an index, that you want to compare to *var-2*, *constant*, or *constant-list*. *var-1* can be a simple variable, array, simple pointer, structure, structure data item, or structure pointer, but not a read-only array.

relational-operator

is one of the following operators:

Signed relational operator	<, =, >, <=, >=, <>
Unsigned relational operator	'<', '=', '>', '<=', '>=', '<>'

All comparisons are unsigned whether you use a signed or unsigned operator.

var-2

is the identifier of a variable, with or without an index, to which *var-1* is compared. *var-2* can be a simple variable, array, read-only array, simple pointer, structure, structure item, or structure pointer.

count

is a unsigned INT arithmetic expression that defines the number of units in *var-2* to compare. When *count-unit* is not present, the units compared are:

var-2	Data Type	Units Compared
Simple variable, array, simple pointer (including those declared in structures)	STRING	Bytes
	INT	Words
	INT(32) or REAL	Doublewords
	FIXED or REAL(64)	Quadruplewords
Structure	Not applicable	Words
Substructure	Not applicable	Bytes
Structure pointer	STRING	Bytes
	INT	Words

count-unit

is BYTES, WORDS, or ELEMENTS. *count-unit* changes the meaning of *count* to the following:

BYTES	Compares <i>count</i> bytes; however, if <i>var-1</i> and <i>var-2</i> both have word addresses, BYTES implicitly generates a word comparison for $(count + 1)/2$ words.
WORDS	Compares <i>count</i> words.
ELEMENTS	Compares <i>count</i> elements. The elements compared depend on the nature of <i>var-2</i> and its data type as follows:

var-2	Data Type	Units Compared
Simple variable, array, simple pointer (including those declared in structures)	STRING	Bytes
	INT	Words
	INT(32) or REAL	Doublewords
	FIXED or REAL(64)	Quadruplewords
Structure (For structure pointers, STRING and INT have meaning only in group comparison expressions and move statements.)	Not applicable	Structure occurrences
Substructure	Not applicable	Substructure occurrences
Simple variable, array, simple pointer (including those declared in structures)	STRING	Bytes
	INT	Words
	INT(32) or REAL	Doublewords
	FIXED or REAL(64)	Quadruplewords

If *count-unit* is specified and is not BYTES, WORDS, or ELEMENTS, the compiler issues an error. If you specify BYTES, WORDS, or ELEMENTS, the term cannot also appear as an identifier in a LITERAL or DEFINE declaration in the global declarations or in any procedure or subprocedure in which the group comparison expression appears.

constant

is a number, a character string, or a LITERAL to which *var-1* is compared.

If you enclose a simple numeric constant in brackets ([]) and if the destination has a byte address or is a STRING structure pointer, the system compares a single byte regardless of the size of *constant*. If you do not enclose *constant* in brackets or if the destination has a word address or is an INT structure pointer, the system compares a word, doubleword, or quadrupleword as appropriate for the size of *constant*.

constant-list

is a list of one or more constants, which are concatenated and compared to *var-1*. Specify *constant-list* in the form shown in [Chapter 3 \(page 46\)](#).

next-addr

is a variable to contain the address of the first byte or word in *var-1* that does not match the corresponding byte or word in *var-2*. The compiler returns a 16-bit or 32-bit address as described in the following subsection.

pTAL programs access all data using byte addresses. pTAL uses the low-order bit of addresses; therefore, when you use an odd-byte address to access a 16-bit word that you have declared with .EXT, you access the data beginning at the odd-byte address.

You can use group comparisons for:

- [Changing the Data Type of the Data \(page 90\)](#)
- [Testing Group Comparisons \(page 91\)](#)

Changing the Data Type of the Data

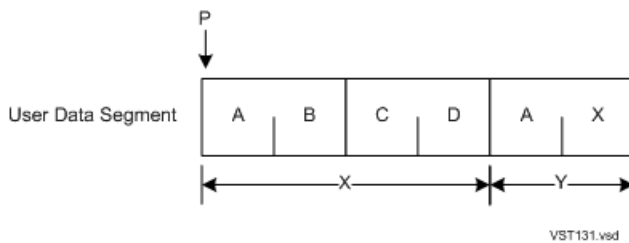
You can compare two strings using a group comparison expression, and save the address where the comparison stopped in a variable or pointer.

[Figure 2 \(page 90\)](#) and [Figure 3 \(page 91\)](#) show that changing the data type of a variable from INT to STRING can affect whether the address stored in the result pointer, *p*, is an even-byte or odd-byte address.

In [Figure 2 \(page 90\)](#), the IF statement compares *x* to *y* on a word-by-word basis. Because the 16 bits in *x* are not equal to the 16 bits in *y*, the conditional expression is false, and *p* points to the beginning of string *x*.

Figure 2 Ending Address After Comparing INT Strings

```
PROC p;  
BEGIN  
  INT x[0:1] := ["AB", "CD"]  
  INT y      := "AX";  
  INT .p;  
  INT q;  
  IF x = y FOR 1 WORDS -> @p THEN ... ;  
  q := p; ! Assign "AB" to q  
END;
```

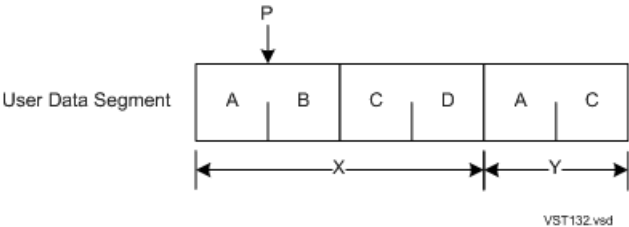


[Figure 3 \(page 91\)](#) is the same as [Figure 2 \(page 90\)](#) except that in [Figure 3 \(page 91\)](#), *y* is a 2-byte STRING array. The IF statement, therefore, compares *x* to *y* on a byte-by-byte basis. Because the first (upper) bytes of *x* and *y* are equal, the comparison continues to the second byte.

Because the second byte of *x* is "B", but the second byte of *y* is "C", the conditional expression is false. In [Figure 3 \(page 91\)](#), therefore, the IF statement stores in *p* the address of the second (lower) byte of *x*.

Figure 3 Ending Address After Comparing Strings of Data Type STRING and INT

```
PROC p;  
BEGIN  
  INT    x[0:1] := ["AB", "CD"];  
  STRING y[0:1] := ["A", "C"];  
  STRING p;  
  INT    q;  
  IF x = y FOR 1 WORDS -> @p THEN ... ;  
  q := p; ! Assign "BC" to q  
END;
```



Testing Group Comparisons

If you use a group comparison in an IF statement, you can test the condition code after the group comparison is evaluated by setting by using the following relational operators (with no operands) in a conditional expression:

Operator	Meaning
<	CCL if <i>var-1</i> '<' <i>var-2</i>
=	CCE if <i>var-1</i> = <i>var-2</i>
>	CCG if <i>var-1</i> '>' <i>var-2</i>

See [Example 15 \(page 92\)](#).

The compiler does a standard comparison and returns a 16-bit *next-addr* if:

- Both *var-1* and *var-2* have standard byte addresses
- Both *var-1* and *var-2* have standard word addresses

The compiler does an extended comparison (which is slightly less efficient) and returns a 32-bit *next-addr* if:

- Either *var-1* or *var-2* has a standard byte address and the other has a standard word address
- Either *var-1* or *var-2* has an extended address

Variables (including structure data items) are byte addressed or word addressed as follows:

Byte addressed	STRING simple variables STRING arrays Variables to which STRING simple pointers point Variables to which STRING structure pointers point Substructures
Word addressed	INT, INT(32), FIXED, REAL, or REAL(64) simple variables INT, INT(32), FIXED, REAL, or REAL(64) arrays Variables to which INT, INT(32), FIXED, REAL, or REAL(64) simple pointers point Variables to which INT structure pointers point Structures

After an element comparison, *next-addr* might point into the middle of an element, rather than to the beginning of the element, because *next-addr* always refers to the first byte or 16-bit word (as appropriate) that differs.

[Example 15 \(page 92\)](#) compares two arrays and then tests the condition code setting to see if the value of the element in *d_array* that stopped the comparison is less than the value of the corresponding element in *s_array*.

Example 15 Array Comparison

```
INT d_array[0:9];
INT s_array[0:9];
! Code to assign values to arrays
IF d_array = s_array FOR 10 ELEMENTS -> @pointer THEN
    BEGIN
        ! They matched
        ! Do something
    END
ELSE
    IF < THEN ... ;           ! Pointer points to element of d_array
    ! Do something else       that is less than the corresponding
                             ! element of s_array
```

When you compare array elements (as in [Example 15 \(page 92\)](#)), the **ELEMENTS** keyword is optional but provides clearer source code.

To compare structure or substructure occurrences, you must specify the **ELEMENTS** keyword in the group comparison expression, as in [Example 16 \(page 92\)](#).

Example 16 Structure Comparison

```
STRUCT struct_one [0:9];
BEGIN
    INT a[0:2];
    INT b[0:7];
    STRING c;
END;
STRUCT struct_two (struct_one) [0:9];
! Code here to assign values to structures
IF struct_one = struct_two FOR 10 ELEMENTS THEN ... ;
```

[Example 17 \(page 92\)](#) contrasts a comparison to a bracketed (single-byte) constant with a comparison to an unbracketed (element) constant.

Example 17 Constant Comparison

```
STRING var[0:1];
...
IF var = [0] THEN ... ; ! Compare var[0] to one byte
IF var = 0 THEN ... ;   ! Compare var[0:1] to two bytes or
                        ! one 16-bit word
```

Bit Operations

You can access individual bits or groups of bits in a **STRING** or **INT** variable.

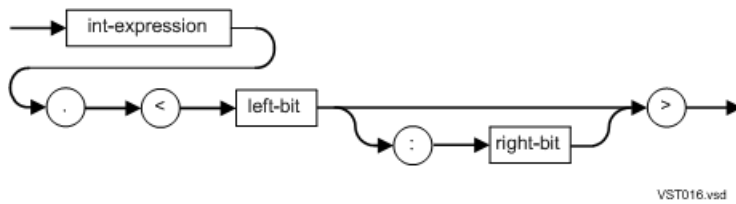
Table 39 Bit Operations

Bit Operation	Description
Extraction	Accesses a bit-extraction field in an INT expression without altering the expression
Deposit	Assigns a bit value to a bit-deposit field in a variable
Shift	Shifts a bit-shift field in an INT or INT(32) expression to the left or to the right by a specified number of bits

Topics:

- [Bit Extractions \(page 93\)](#)
- [Bit Shifts \(page 94\)](#)

Bit Extractions



int-expression

is an INT(32) expression.

left-bit

is an INT constant in the range 0 through 15 that specifies the bit number of either:

- The leftmost bit of the bit-extraction field
- The only bit (if *right-bit* is the same value as *left-bit* or is omitted)

If *int-expression* is a STRING value, *left-bit* must be in the range 8 through 15. (In a string value, bit 8 is the leftmost bit and bit 15 is the rightmost bit.)

right-bit

is an INT constant that specifies the rightmost bit of the bit field. If *int-expression* is a STRING value, *right-bit* must be in the range 8 through 15. *right-bit* must be equal to or greater than *left-bit*. To access a single bit, omit *right-bit* or specify the same value as *left-bit*.

You can perform bit extractions and deposits on 16-bit and 32-bit items. pTAL reports an error, however, if you attempt to reference bits outside of the bits declared in the variable's declaration.

Example 18 Bit Extraction

```
INT      i;
UNSIGNED(4) j;
STRING   k;
INT(32)   l;
i := j.<7:11>; ! ERROR: You can reference only bits 12
              ! through 15 of j
i := k.<0:7>;  ! ERROR: You can reference only bits 8
              ! through 15 of k
i := l.<8:31>; ! OK: You can reference any of bits 0
              ! through 31 of l
```

Example 19 Bit Extraction From an Array

```
STRING right_byte;
INT array[0:7];
right_byte := array[5].<8:15>;
```

Example 20 (page 94) assigns bits 4 through 7 of the sum of two numbers to RESU<. The parentheses cause the numbers to be added before the bits are extracted.

Example 20 Modifying Bits Before Extracting Them

```
INT result;
INT num1 := 51;
INT num2 := 28;
result := (num1 + num2).<4:7>;
```

Example 21 (page 94) checks bit 15 for a nonzero value.

Example 21 Checking a Bit for a Nonzero Value

```
STRING var;
IF var.<15> THEN ... ;
```

Bit Shifts

A bit shift operation shifts a bit field a specified number of positions to the left or to the right within a variable without altering the variable. RISC and Itanium architectures do not include a signed left-shift operation, so pTAL compiles a signed left shift (for example, `i << 8`) as an unsigned left shift (`i '<<' 8`).



VST017.vsd

int-expression

is an INT arithmetic expression. *int-expression* can contain STRING, INT, or UNSIGNED(1-16) operands. The bit shift occurs within a word.

dbl-expression

is an INT(32) arithmetic expression. *dbl-expression* can contain INT(32) or UNSIGNED(17-31) operands. The bit shift occurs within a doubleword.

shift-operator

is one of the operators described in Table 23 (page 70).

positions

is an INT expression that specifies the number of bit positions to shift the bit field. A value greater than 31 gives undefined results.

The shift count must be less than the number of bits in the shifted value; therefore, you can shift an INT value up to 15 bits left or right, and an INT(32) value up to 31 bits left or right.

The compiler reports an error if the shift count is a constant and its value is greater than the number of bits in the value to shift.

If the shift amount is a dynamic expression and is greater than the maximum allowed (one bit less than the number of bits being shifted), the result depends on the CHECKSHIFTCOUNT compiler directive, as follows:

- If CHECKSHIFTCOUNT is enabled and a dynamic shift count is equal to or greater than the number of bits in the value being shifted, the system aborts your program with an instruction trap.
- If CHECKSHIFTCOUNT is disabled (you specify NOCHECKSHIFTCOUNT), and a dynamic shift count is equal to or greater than the number of bits in the value being shifted, program operation is undefined.

The compiler implements arithmetic left shifts as unsigned left shifts (and warns you when it does this).

Most programs do not need to perform signed left shifts. If your program does require an arithmetic left shift, use the functions in [Example 22 \(page 95\)](#) to perform signed left-shift operations.

Example 22 Arithmetic Left Shift

```
INT PROC ashift16(a, count);
  INT a, count;
BEGIN
  STRUCT s = a;
  BEGIN
    UNSIGNED(1)  sign_bit;
    UNSIGNED(15) rest;
  END;
  s.rest := s.rest '<<' count;
  RETURN a;
END;
INT(32) PROC ashift32(a, count);
  INT(32) a;
  INT count;
BEGIN
  STRUCT s = a;
  BEGIN
    UNSIGNED(1)  sign_bit;
    UNSIGNED(31) rest;
  END;
  s.rest := s.rest '<<' count;
  RETURN a;
END;
```

Table 40 Bit-Shift Operators

Operator	Routine	Result
'<<'	Unsigned left shift through bit 0	Zeros fill vacated bits from the right
'>>'	Unsigned right shift	Zeros fill vacated bits from the left.
>>	Signed right shift	Sign bit (bit 0) unchanged; sign bit fills vacated bits from the left

Bit-shift operations include:

Operation	User Action
Multiplication by powers of 2	For each power of 2, shift the field one bit to the left. (Some data might be lost.)
Unsigned division by powers of 2	For each power of 2, shift the field one bit to the right (Some data might be lost.)
Word-to-byte address conversion	Shift the word address one bit to the left by using an unsigned shift operator.

Bit shift examples:

1. This unsigned left shift shows how zeros fill the vacated bits from the right:

```
Initial value = 0 010 111 010 101 000
'<<' 2 = 1 011 101 010 100 000
```

2. This unsigned right shift shows how zeros fill the vacated bits from the left:

```
Initial value = 1 111 111 010 101 000
'>>' 2 = 0 011 111 110 101 010
```

3. This signed right shift shows how the sign bit fills the vacated bits from the left:

```
Initial value = 1 111 010 101 000 000
>> 3 = 1 111 111 010 101 000
```

4. These examples show multiplication and division by powers of two:

```
a := b << 1; ! Multiply by 2
a := b << 2; ! Multiply by 4
a := b >> 3; ! Divide by 8
a := b >> 4; ! Divide by 16
```

5. This unsigned bit shift converts the word address of an INT array to a byte address, which allows byte access to the INT array:

```
INT a[0:5]; ! INT array>
STRING .p := @a[0] '<<' 1; ! Initialize STRING simple
! pointer with byte address
p[3] := 0; ! Access fourth byte of A
```

6. This example shifts the right-byte value into the left byte of the same word and sets the right byte to a zero:

```
INT b; ! INT variable
b := b '<<' 8; ! Shift right-byte value into left byte
```


6 LITERALS and DEFINES

A LITERAL declaration associates identifiers with constant values. A DEFINE declaration associates identifiers (and parameters if any) with text.

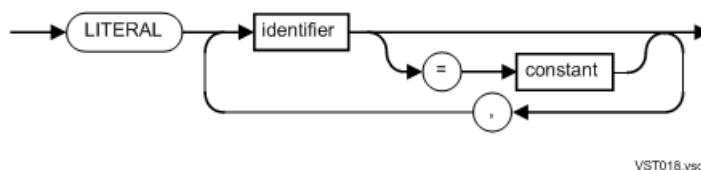
You can declare LITERALS and DEFINES once in a program, and then refer to them by identifier many times throughout the program. They allow you to efficiently make significant changes in the source code. You only need to change the declaration, not every reference to it in the program.

Topics:

- Declaring Literals (page 97)
- Declaring DEFINES (page 98)
- Calling DEFINES (page 100)
- How the Compiler Processes DEFINES (page 100)
- Passing Actual Parameters to DEFINES (page 100)

Declaring Literals

A LITERAL declaration specifies one or more identifiers and associates each with a constant value. Each identifier in a LITERAL declaration is known as a LITERAL.



identifier

is the LITERAL identifier. Literal identifiers make the source code more readable. For example, identifiers such as BUFFER_LENGTH and TABLE_SIZE are more meaningful than their respective constant values of 80 and 128.

constant

is one of the following:

- A character string of 1 to 4 characters.
- Any of the following numeric constant expressions whose value is not the address of a global variable (global variables are relocatable during linking):
 - `FIXED(n)`
 - `INT`
 - `INT(32)`
 - `REAL`
 - `REAL(64)`
 - `UNSIGNED(n)`

If you omit any constants, the compiler supplies the omitted numeric constants. The compiler uses unsigned arithmetic to compute the constants it supplies:

- If you omit the first constant in the declaration, the compiler supplies a zero.
- If you omit a constant that follows an INT constant, the compiler supplies an INT constant that is one greater than the preceding constant. If you omit a constant that follows a constant of any data type except INT, an error message results.

You access a LITERAL constant by using its identifier in declarations and statements. The compiler does not allocate storage for LITERAL constants. It substitutes the constant at each occurrence of the identifier.

Example 23 Literal Declarations

All constants specified:

```
LITERAL true      = -1,
      false      = 0,
      buffer_length = 80,
      table_size  = 128,
      table_base  = %1000,
      entry_size  = 4,
      timeout     = %100000D,
      CR          = %15,
      LF          = %12;
```

All constants supplied by compiler:

```
LITERAL a, ! Compiler assigns 0
      b, ! Compiler assigns 1
      c; ! Compiler assigns 2
```

Two constants specified, six supplied by compiler:

```
LITERAL d, ! Compiler assigns 0
      e, ! Compiler assigns 1
      f, ! Compiler assigns 2
      g = 0,
      h, ! Compiler assigns 1
      i = 17,
      j, ! Compiler assigns 18
      k; ! Compiler assigns 19
```

LITERAL identifier in array declaration:

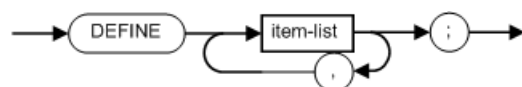
```
LITERAL length = 50; ! Length of array
INT buffer[0:length - 1]; ! Array declaration
```

LITERAL identifiers in subsequent LITERAL declarations:

```
LITERAL number_of_file_extents = 16;
LITERAL file_extents_size_in_pages = 32;
LITERAL file_size_in_bytes = (number_of_file_extents '*'
      file_extents_size_in_pages) * 2048D ! bytes per page !;
```

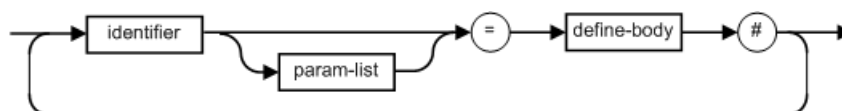
Declaring DEFINES

A DEFINE declaration associates an identifier (and optional parameters) with text.



VST019.vsd

item-list

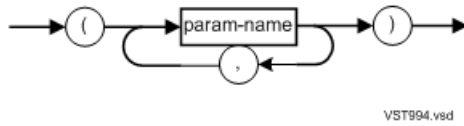


VST995.vsd

identifier

is the identifier of the DEFINE.

param-list



param-name is the identifier of a formal parameter. You can specify up to 31 formal parameters. An actual parameter can be up to 500 bytes. A formal parameter cannot be a pTAL reserved word.

define-body

specifies all characters between the = and # delimiters. *define-body* can span multiple source lines. Enclose character strings in quotation marks ("). To use # as part of the *define-body* rather than as a delimiter, enclose the # in quotation marks or embed the # in a character string.

DEFINE declaration requirements:

- If a DEFINE and a formal parameter have the same identifier, the formal parameter has priority during expansion.
- A DEFINE must not reference itself.
- A DEFINE declaration must not appear within a DEFINE body; that is, do not nest a DEFINE within a DEFINE.
- To ensure proper grouping and order of evaluation of expressions in the DEFINE body, use parentheses around each DEFINE parameter used in an expression.
- Within the DEFINE body, place any compound statements within a BEGIN-END block.
- Directives appearing within a DEFINE body are evaluated immediately; they are not part of the DEFINE itself.
- Expanded DEFINES must produce correct pTAL constructs. To list the expanded DEFINES in the compiler listing, specify the DEFEXPAND directive before the DEFINE declarations.

Example 24 DEFINE Declarations

Parentheses direct the DEFINE body evaluation:

```
DEFINE value = ( (45 + 22) * 8 / 2 ) #;
```

Incrementing and decrementing utilities included:

```
DEFINE increment (x) = x := x + 1 #;
DEFINE decrement (y) = y := y - 1 #;
```

Loads numbers into specified bit positions:

```
DEFINE word_val (a, b) = ((a) '<<' 12) LOR (b) #;
```

When a STRUCT item and a DEFINE have the same name, the compiler issues a warning when the STRUCT item is referenced. In [Example 25 \(page 99\)](#), DEFINE myname accesses the structure item1 named in the DEFINE body. The compiler issues a warning because 2 is assigned to mystruct.yrname, not to mystruct.myname.

Example 25 STRUCT and DEFINE Macro Items With the Same Name

```
PROC myproc MAIN;
BEGIN
  DEFINE myname = item1#,
            yrname = item2#;
  STRUCT mystruct;
  BEGIN
```

```

    INT item1;
    INT item2;
    INT yrname;           ! Structure item has same
END;                     ! identifier as a DEFINE
mystruct.myname := 1;    ! OK: 1 is assigned to mystruct.item1
mystruct.yrname := 2;    ! Compiler issues warning;
                        ! 2 is assigned to mystruct.yrname,
                        ! not to mystruct.item2

! More code
END;

```

Calling DEFINES

You call a DEFINE by using its identifier in a statement. The invocation can span multiple lines.

If you call a DEFINE within an expression, make sure the expression evaluates as you intend. For instance, if you want the DEFINE body to be evaluated before it becomes part of the expression, enclose the DEFINE body in parentheses.

Example 26 Parenthesized and Nonparenthesized DEFINE Bodies

```

DEFINE expr = (5 + 2) #;
j := expr * 4;           ! Expands to: (5 + 2) * 4;
                        ! assigns 28 to j

DEFINE expr = 5 + 2 #;
j := expr * 4;           ! Expands to: 5 + 2 * 4;
                        ! assigns 13 to j

```

DEFINE identifiers are not called when specified:

- Within a comment
- Within a character string constant
- On the left side of a declaration

For example, the following declaration can call a DEFINE named *y* but not a DEFINE named *x*:

```
INT x := y;
```

How the Compiler Processes DEFINES

The compiler does not allocate storage for DEFINE declarations. When the compiler encounters a statement using a DEFINE identifier, the compiler expands the DEFINE declaration as follows:

- It replaces the DEFINE identifier with the DEFINE body, replaces formal parameters with actual parameters, and compiles the resulting declaration.
- It expands quoted character strings intact.
- It expands actual parameters after instantiation. Depending on the order of evaluation, the expansion can change the scope of a DEFINE declaration.
- Emits machine instructions at the appropriate processing interval.

If the DEFEXPAND directive is active, the compiler lists each expanded DEFINE declaration in the compiler listing following the invocation of the DEFINE. The expanded listing includes:

- The DEFINE body, excluding comments
- Parameters to the DEFINE declaration

Passing Actual Parameters to DEFINES

If the DEFINE declaration has formal parameters, supply the actual parameters when you use the DEFINE identifier in a statement.

The number of actual parameters can be less than the number of formal parameters. If actual parameters are missing, the corresponding formal parameters expand to empty text. For each missing actual parameter, you can use a placeholder comma, as in [Example 27 \(page 101\)](#).

Example 27 Fewer Actual Parameters Than Formal Parameters

```
INT PROC d (a, b, c) EXTENSIBLE; EXTERNAL;  
DEFINE something (a, b, c) = d (a, b, c) #;  
nothing := something ( , , c); ! Placeholder commas
```

If a DEFINE has formal parameters and you pass no actual parameters to the DEFINE, you must specify an empty actual parameter list. You can include commas between the list delimiters, but need not, as in [Example 28 \(page 101\)](#).

Example 28 No Actual Parameters

```
DEFINE something (a, b, c) = anything and everything #;  
nothing := something ( ); ! Empty parameter list
```

If the number of actual parameters exceeds the number of formal parameters, as in [Example 29 \(page 101\)](#), the compiler issues an error.

Example 29 More Actual Parameters Than Formal Parameters

```
DEFINE something (a, b, c) = anything and everything #;  
nothing := something (a, b, c, d); ! Too many parameters
```

If an actual parameter in a DEFINE invocation requires commas, enclose each comma in apostrophes ('). An example is an actual parameter that is a parameter list, as in [Example 30 \(page 101\)](#).

Example 30 Commas in an Actual Parameter

```
DEFINE varproc (procl, param) = CALL procl (param) #;  
varproc (myproc, i ', ' j ', ' k); ! Expands to:
```

An actual parameter in a DEFINE invocation can include parentheses, as in [Example 31 \(page 101\)](#).

Example 31 Parentheses in an Actual Parameter

```
DEFINE varproc (procl, param) = CALL procl (param) #;  
varproc (myproc, (i + j) * k); ! Expands to:  
! CALL myproc ((i+j)*k);
```

[Example 32 \(page 102\)](#) shows a DEFINE declaration that has one formal parameter and an assignment statement that uses the DEFINE identifier, passing a parameter of 3.

Example 32 Assignment Statement Using DEFINE Macro Identifier

```
DEFINE cube (x) = ( x * x * x ) #;
INT result;
result := cube (3) '>>' 1;
! Expands to: (3 * 3 * 3) '>>' 1 = 27 '>>' 1 = 13
```

Example 33 Incrementing and Decrementing Utilities

```
DEFINE increment (x) = x := x + 1 #;
DEFINE decrement (y) = y := y - 1 #;
INT index := 0;
increment(index); ! Expands to: index := index + 1;
```

Example 34 Filling an Array With Zeros

```
DEFINE zero_array (array, length) =
BEGIN
    array[0] := 0;
    array[1] := array FOR length - 1;
END #;
LITERAL len = 50;
INT buffer[0:len - 1];
zero_array (buffer, len); ! Fill buffer with zeros
```

[Example 35 \(page 102\)](#) displays a message, checks the condition code, and assigns an error if one occurs.

Example 35 Checking a Condition Code

```
INT error;
INT file;
INT .buffer[0:50];
INT count_written;
INT i;
DEFINE emit (filenum, text, bytes, count, err) =
BEGIN
    CALL WRITE (filenum, text, bytes, count);
    IF < THEN
        BEGIN
            CALL FILEINFO (filenum, err);
            ! Process errors if any
        END;
    END #;
! Lots of code
IF i = 1 THEN
    emit (file, buffer, 80, count_written, error);
```

7 Simple Variables

A simple variable is a single-element data item of a specified data type that is not an array, a structure, or a pointer. After you declare a simple variable, you can use its identifier in statements to access or change the data contained in the variable. You must declare variables before you use them.

This section defines the syntax for declaring simple variables. The declaration determines:

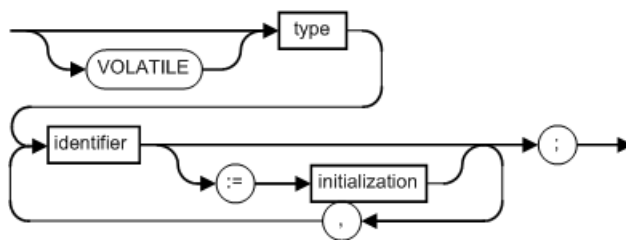
- The kind of values the simple variable can represent
- The amount of storage the compiler allocates for the variable
- The operations you can perform on the variable
- The byte or word addressing mode of the variable
- The direct or indirect addressing mode of the variable
- How the compiler allocates storage for simple variables
- How you access the variables

Topics:

- [Declaring Simple Variables \(page 103\)](#)
- [Specifying Simple Variable Address Types \(page 105\)](#)
- [Initializing Simple Variables With Numbers \(page 105\)](#)
- [Initializing Simple Variables With Character Strings \(page 105\)](#)
- [Examples \(page 105\)](#)

Declaring Simple Variables

The simple variable declaration associates an identifier with a single-element data item and optionally initializes it.



VST622.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a **VOLATILE** data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, **VOLATILE** also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is one of the following data types:

- **BADDR**
- **CBADDR**
- **CWADDR**

- EXTADDR
- EXT32ADDR
- EXT64ADDR
- INT
- INT(32)
- FIXED
- FIXED (*fpoint*)
- PROCADDR
- PROC32ADDR
- PROC64ADDR
- REAL
- REAL
- REAL(64)
- SGBADDR
- SGWADDR
- SGXBADDR
- SGXWADDR
- STRING
- UNSIGNED (*width*)
- WADDR

NOTE: The data types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561 H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

fpoint

For the FIXED data type, *fpoint* is the implied fixed-point setting. *fpoint* is an integer in the range -19 through 19. If you omit *fpoint*, the default *fpoint* is 0 (no decimal places).

A positive *fpoint* specifies the number of decimals to the right of the decimal point. A negative *fpoint* specifies a number of integers to the left of the decimal point.

fpoint can also be an asterisk (*).

width

For INT, REAL, and UNSIGNED data types, the value in parentheses is a constant expression specifying the width, in bits of the variable. The constant expression can include LITERALs and DEFINEs. The result of the constant expression must be one of the following values:

Data Type Prefix	width in bits
INT	16, 32, or 64
REAL	32 or 64
UNSIGNED (simple variable, parameter, or function result)	a value in the range 1 through 31
UNSIGNED (array)	1, 2, 4, or 8

identifier

is the identifier of the simple variable, specified in the form described in Section 2, Language Elements.

initialization

is an expression that represents the value to store in *identifier*. The default number base is decimal. The kind of expression you can specify depends on the scope of the simple variable:

- For a global simple variable, use a constant expression.
- For a local or sublocal simple variable, use any arithmetic expression including variables.

You can initialize simple variables of any data type except UNSIGNED. For more information about initializing a simple variable, see the following subsections.

Specifying Simple Variable Address Types

The address type of a simple variable is the same as the address type of a pointer to data of the same object data type as the simple variable, as in the following examples:

```
INT .j;    ! A pointer: address type is WADDR
INT i;     ! A simple variable: address type is WADDR
```

Initializing Simple Variables With Numbers

When you initialize with a number, it must match the data type specified for the simple variable. The data type determines what kind of values the simple variable can store:

- STRING, INT, and INT(32) simple variables can contain integer constants in binary, decimal, hexadecimal, or octal base.
- REAL and REAL(64) simple variables can contain signed floating-point numbers.
- FIXED simple variables can contain signed 64-bit fixed-point numbers in binary, decimal, hexadecimal, or octal base. For decimal numbers, you can also specify a fractional part, preceded by a decimal point. If a FIXED number has a different decimal setting than the specified *fpoint*, the system scales the number to match the *fpoint*. If the number is scaled down, some precision is lost.

[Chapter 3 \(page 46\)](#) describes the syntax for specifying numeric constants in each number base by data type.

Initializing Simple Variables With Character Strings

STRING, INT, and UNSIGNED simple variables can be initialized with character strings. The character string can contain the same number of bytes as the simple variable or fewer. Unspecified bytes are zero bytes. Each character in a character string requires one byte of storage.

Examples

- [Example 36 \(page 106\)](#)
- [Example 37 \(page 106\)](#)
- [Example 38 \(page 107\)](#)
- [Example 39 \(page 107\)](#)
- [Example 40 \(page 107\)](#)
- [Example 41 \(page 107\)](#)

Example 36 Declaring Simple Variables Without Initializing Them

```
STRING b;  
INT(32) dblwd1;  
REAL(64) long;  
UNSIGNED(5) flavor;  
BADDR ba;  
WADDR wa;  
EXTADDR ea;
```

Example 37 Declaring and Initializing Simple Variables

```
STRING y := "A";           ! Character string  
STRING z := 255;           ! Byte value  
INT a := "AB";            ! Character string  
INT b := 5 * 2;           ! Expression  
INT c := %B110;           ! Word value  
INT(32) dblwd2 := %B1011101D; ! Doubleword value  
INT(32) dblwd3 := $DBL(%177775); ! Built-in routine  
REAL flt1 := 365335.6E-3;  ! Doubleword value  
REAL(64) flt2 := 2718.2818284590452L-3; ! Quadrupleword value  
WADDR w;  
INT t;  
STRING s;  
INT ro_wd = 'p' := 3;  
STRING ro_b = 'p' := "A";  
BADDR ba := @s;  
WADDR wa := @t;  
CWADDR cwa := @ro_wd;  
CBADDR cba := @ro_b;  
SGWADDR sgwa := 0;  
SGBADDR sgnq := 1;  
EXTADDR ea := $DBL (1);  
EXT32ADDR e32a := 10D;  
EXT64ADDR e64a := 10F;
```

Example 38 Effect of fpoint on FIXED Simple Variables

```
FIXED(-3) f := 642987F;    ! Stored as 642; accessed as 642000
FIXED(3)  g := 0.642F;     ! Stored as 642, accessed as 0.642
FIXED(2)  h := 1.234F;     ! Stored as 123; accessed as 1.23
```

Example 39 Initializing Simple Variables With Constants and Variables

```
INT global := 34;           ! Only constants allowed
                                ! in global initialization
PROC mymain MAIN;
  BEGIN
    INT local := global + 10; !Any expression allowed
    INT local2 := global * local; ! in local or sublocal
    FIXED local3 := $FIX(local2); ! initialization
    !Lots of code
  END;                       ! End of mymain procedure
```

Example 40 Declaring Simple VOLATILE Variables

```
VOLATILE INT      i;
VOLATILE UNSIGNED(3) mask;
VOLATILE STRING   gs;
```

Example 41 Procedure Addresses and Procedure Pointers

```
PROCADDR pa;
PROC32ADDR p32a;
PROC64ADDR p64a;
PROCPTR p (j); INT j; END PROCPTR;
PROC32PTR p32 (k); INT k; END PROC32PTR;
INT PROC64PTR p64 (l, m); INT(32) l; INT(64) m; END PROC64PTR;
STRUCT abc;
BEGIN
  PROCPTR z (i); INT i; END PROCPTR;
END;
```

NOTE: The address and pointer types, EXT32ADDR, EXT64ADDR, PROC32ADDR, PROC64ADDR, PROC32PTR, and PROC64PTR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

8 Arrays

An array is a one-dimensional set of elements of the same data type. Each array is stored as a collective group of elements. You use arrays to store constants, especially character strings. After you declare an array, you can use its identifier to access the array elements individually or as a group.

You can declare:

- Arrays
- Read-only arrays
- Address arrays

The declaration includes initializing the array as well as allocating storage for the array. In addition the declaration determines:

- The kind of values the array can represent
- The operations you can perform on the array
- The byte or word addressing mode of the array

This section defines the syntax for declaring:

- Arrays
- Read-only arrays
- Address arrays

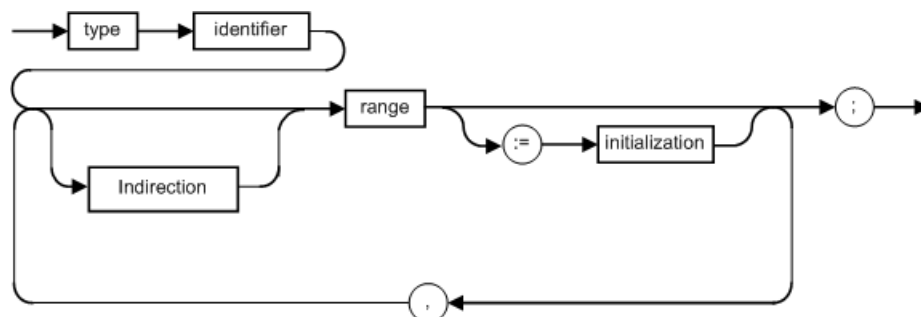
[Chapter 9 \(page 114\)](#) describes the syntax for declaring arrays within structures and how to declare structures that simulate arrays of arrays, or arrays of structures (including multidimensional arrays).

Topics:

- [Declaring Arrays \(page 108\)](#)
- [Declaring Read-Only Arrays \(page 111\)](#)
- [Using Constant Lists in Array Declarations \(page 113\)](#)

Declaring Arrays

An array declaration associates an identifier with a set of elements of the same data type. The data type of an array can be one of the pTAL address types.



type

is one of the following:

- BADDR
- CBADDR

- CWADDR
- EXTADDR
- EXT32ADDR
- EXT64ADDR
- FIXED (*fpoint*)
- INT
- INT(32)
- FIXED
- PROCADDR
- PROC32ADDR
- PROC64ADDR
- REAL
- REAL
- REAL(64)
- SGBADDR
- SGWADDR
- SGXBADDR
- SGXWADDR
- STRING
- UNSIGNED (*width*)
- WADDR

NOTE: The data types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, [“64-bit Addressing Functionality” \(page 531\)](#).

The data type determines:

- The kind of values that are appropriate for the array
- The storage unit the compiler allocates for each array element as follows:

width

is a constant expression specifying the width, in bits, of the variable.

fpoint

is the implied fixed point of the FIXED variable.

identifier

is the array name.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

range



VST993.vsd

lower-bound

is an INT or INT(32) constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated.

upper-bound

is an INT or INT(32) constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated.

For arrays declared outside of structures, *upper-bound* must be equal to or larger than *lower-bound*.

Here are some examples of bounds:

```
STRING a_array [0:2];
INT b_array [0:19];
UNSIGNED(1) flags [0:15];
```

initialization

is a constant or a constant list of values to assign to the array elements, beginning with the lower-bound element. (Constant lists are described in [Chapter 3 \(page 46\)](#).) If you specify fewer initialization values than the number of elements, the values of uninitialized elements are undefined. You cannot initialize extended indirect local arrays or UNSIGNED arrays.

Specify initialization values that are appropriate for the data type of the array. For example, if the decimal setting of an initialization value differs from the *fpoint* of a FIXED array, the system scales the initialization value to match the *fpoint*. If the initialization value is scaled down, some precision is lost.

Examples:

- [Example 42 \(page 110\)](#)
- [Example 43 \(page 110\)](#)
- [Example 44 \(page 110\)](#)
- [Example 45 \(page 111\)](#)
- [Example 46 \(page 111\)](#)
- [Example 47 \(page 111\)](#)
- [Example 48 \(page 112\)](#)

Example 42 Declaring Arrays With Various Bounds

```
FIXED      .array_a[0:3];    ! Four-element array
INT        .array_b[0:49];   ! Fifty-element array
UNSIGNED(1) flags[0:15];     ! Array of 16 one-bit elements
```

Example 43 Declaring Arrays and Initializing Them With Constants

```
INT a_array[0:3] := -1;      ! Store -1 in element [0];
                                ! values in elements [1:3] are undefined
INT b_array[0:1] := "abcd"; ! Store one character per byte
```

Example 44 Declaring Arrays and Initializing Them With Constant Lists

```
INT c_array[0:5] := [1,2,3,4,5,6]; ! Constant list
STRING buffer[0:102] := [ "A constant list can consist ",
                          "of several character string constants, ",
                          "one to a line, separated by commas." ];
INT(32) mixed[0:3] := ["abcd", 1D, %B0101011D, %20D]; ! Mixed constant list
LITERAL len = 80;                                     ! Length of array
```

```

STRING buffer[0:len - 1] := len * [" "];    ! Repetition factor
FIXED f[0:35] := 3*[2*[1F,2F], 4*[3F,4F]]; ! Repetition factors
LITERAL cr = %15,
         lf = %12;
STRING err_msg[0:9] := [cr, lf, "ERROR", cr, lf, 0]; ! Constant list

```

Example 45 Initializing Arrays

```

INT(32) a[0:1] := [5D, 7D];    ! Initialize global array
PROC my_procedure;
BEGIN
  STRING b[0:1] := ["A","B"]; ! Initialize local standard array
  FIXED EXT c[0:3];           ! Cannot initialize local
                              ! extended indirect array

  SUBPROC my_subproc;
  BEGIN
    INT d[0:2] := ["Hello!"]; ! Initialize sublocal array
    ! Lots of code
  END;
END;

```

Example 46 Array With Positive fpoint

```

FIXED(2) x[0:1] := [ 0.64F, 2.348F ];
! Stored as 64 and 234; accessed as 0.64 and 2.34

```

Example 47 Array With Negative fpoint

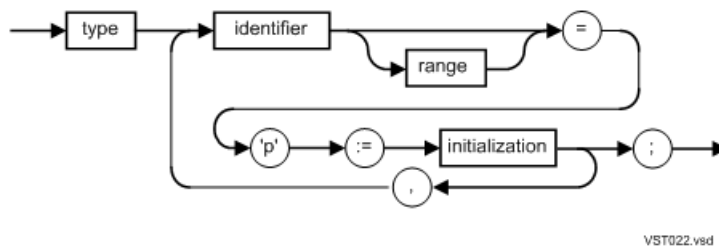
```

FIXED(-3) y[0:1] := [ 642913F, 1234F ];
! Stored as 642 and 1; accessed as 642000 and 1000

```

Declaring Read-Only Arrays

A read-only array declaration allocates storage for a nonmodifiable array in a user code segment. Read-only arrays are sometimes referred to as p-relative arrays, because they are addressed using the program counter (the p register).



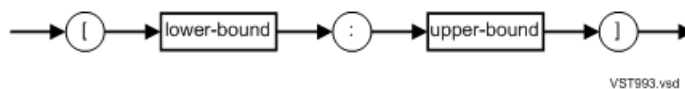
type

is any data type described in [Declaring Arrays \(page 108\)](#) except UNSIGNED. The data type of a read-only array cannot be an address type.

identifier

is the identifier of the read-only array.

range



lower-bound

is an INT or INT(32) constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated. The default value is 0.

upper-bound

is an INT or INT(32) constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated. The default value is the number of elements initialized minus one.

'P'

specifies a read-only array.

initialization

is a constant list to assign to the array elements. You must initialize read-only arrays when you declare them. (Constant lists are described in [Chapter 3 \(page 46\)](#).)

Specify initialization values that are appropriate for the data type of the array. For example, if the decimal setting of an initialization value differs from the *fpoint* of a FIXED array, the system scales the initialization value to match the *fpoint*. If the initialization value is scaled down, some precision is lost.

The compiler reports a warning if a read-only array declaration specifies an indirection symbol (see [Table 14 \(page 41\)](#)).

Example 48 Read-Only Array Declaration With Indirection Symbol

```
PROC p;
BEGIN
  INT      i = 'P' := [2,3,4];    ! OK
  INT      .j = 'P' := [5,6,7];    ! Compiler reports a warning
  STRING   k = 'P' := ["abc"];    ! OK
  STRING   .l = 'P' := ["ccc"];    ! Compiler reports a warning
END;
```

You must initialize read-only arrays.

UNSIGNED read-only arrays are not allowed, because they cannot be initialized.

If you declare a read-only array in a RESIDENT procedure, the array is also resident in main memory.

The linker links each global read-only array into any code segment containing a procedure that references the array.

You can access read-only arrays as you access any other array, except that:

- You cannot modify a read-only array; that is, you cannot specify a read-only array on the left side of an assignment or move operator.
- You cannot specify a read-only array on the left side of a group comparison expression.
- In a SCAN or RSCAN statement, you cannot use *next-addr* to read the last character of a string. You can use *next-addr* to compute the length of the string.

Example 49 Declaring and Initializing a Read-Only Array

```
STRING prompt = 'P' := ["Enter Character: ", 0];  
INT error = 'P' := ["INCORRECT INPUT"];
```

Using Constant Lists in Array Declarations

pTAL requirements for array declarations are:

- If an array declaration includes an initialization string, the size of a constant list must be less than or equal to the size of the array. If the constant list is larger than the array, an error occurs.
- If the alignment of the elements of the initialization string under SHARED2 rules and SHARED8 rules is different, you must specify a FIELDALIGN clause in the initialization string.

The number of bits in a constant list that you assign to a read-write array must be the same as the number of bits in the array. The number of bits in a constant list that you assign to a read-only array must be the less than or equal to the number of bits in the array.

Topics:

- [Read-Only Arrays \(page 113\)](#)
- [Nonstring Arrays \(page 113\)](#)

Read-Only Arrays

The number of bits in the initialization string must equal the number of bits in the read-only array.

If the read-only-array declaration does not specify array bounds, the number of bits in the initialization string must be an integral multiple of the number of bits in the array's base type.

Example 50 Declaring Read-Only Arrays With Constant Lists

```
INT p      = 'P' := "abcd";      ! OK  
INT q[0:3] = 'P' := "abcdabcd"; ! OK  
STRING r    = 'P' := "abcd";    ! OK  
STRING s[0:3] = 'P' := "abcd";  ! OK  
STRING t     = 'P' := [1,2,3];   ! OK  
STRING u[0:3] = 'P' := [1,2,3];  ! ERROR: Initialization size  
! (24 bits) must equal the  
! array's size (32 bits)  
  
STRING v     = 'P' := [1,2,3,4]; ! OK  
STRING w[0:3] = 'P' := [1,2,3,4]; ! OK  
INT x        = 'P' := "abc";     ! ERROR: Initialization size  
! must be an integral  
! multiple of array's base  
! type size
```

Nonstring Arrays

You can specify an initialization string when you declare an array:

```
INT .a[0:3] := [0,1,2,3];
```

The length of the initialization string must be less than or equal to the length of the array.

Example 51 Declaring Nonstring Arrays With Constant Lists

```
INT .a[0:3] := [0,1,2];      ! OK: Init string is shorter than array  
INT .a[0:3] := [0,1,2,3];    ! OK: Init string is right length  
INT .a[0:3] := [0,1,2,3,4];  ! ERROR: Init string is too long  
INT .a[0:3] := [%H1234567812345678%F]; !OK: Init string is right length
```

9 Structures

A structure is a collectively stored set of data items that you can access individually or as a group. Structures contain structure items (fields) such as simple variables, arrays, simple pointers, structure pointers, and nested structures (called substructures). The structure items can be of different data types.

Structures usually contain related data items such as the fields of a file record. For example, in an inventory control application, a structure might contain an item number, the unit price, and the quantity on hand.

A structure declaration associates an identifier with one of the kinds of structures listed in [Table 41](#) (page 114).

Table 41 Kinds of Structures

Structure	Description
Definition	Describes a structure layout and allocates storage for it
Template	Describes a structure layout but allocates no storage for it
Referral	Allocates storage for a structure whose layout is the same as the layout of a previously declared structure

The TNS/E instructions `set jmp ()` and `long jmp ()` require data to be aligned on 16-byte boundaries. To ensure that this data is aligned on 16-byte boundaries, you must declare it in a template structure using `STRUCTALIGN (MAXALIGN)`.

Topics:

- [Structure Layout \(page 115\)](#)
- [Overview of Field Alignment \(page 117\)](#)
- [Field and Base Alignment \(page 119\)](#)
- [Array Alignment in Structures \(page 122\)](#)
- [Structure Alignment \(page 123\)](#)
- [Substructure Alignment \(page 124\)](#)
- [Alignment Considerations for Substructures \(page 126\)](#)
- [FIELDALIGN Clause \(page 127\)](#)
- [FIELDALIGN Compiler Directive \(page 127\)](#)
- [SHARED2 Parameter \(page 128\)](#)
- [SHARED8 Parameter \(page 129\)](#)
- [Reference Alignment With Structure Pointers \(page 134\)](#)
- [STRUCTALIGN \(MAXALIGN\) Attribute \(page 137\)](#)
- [VOLATILE Attribute \(page 138\)](#)
- [Declaring Definition Structures \(page 138\)](#)
- [Declaring Template Structures \(page 139\)](#)
- [Declaring Referral Structures \(page 141\)](#)
- [Declaring Simple Variables in Structures \(page 142\)](#)
- [Declaring Arrays in Structures \(page 143\)](#)
- [Declaring Substructures \(page 144\)](#)

- [Declaring Filler \(page 147\)](#)
- [Declaring Simple Pointers in Structures \(page 148\)](#)
- [Declaring Structure Pointers in Structures \(page 151\)](#)
- [Declaring Redefinitions \(page 153\)](#)
- [Simple Variable \(page 153\)](#)
- [Array \(page 154\)](#)
- [Definition Substructure \(page 155\)](#)
- [Referral Substructure \(page 157\)](#)
- [Simple Pointer \(page 158\)](#)
- [Structure Pointer \(page 159\)](#)

Equivalenced structures are discussed in [Chapter 11 \(page 177\)](#).

Structure Layout

The structure layout (or body) is a BEGIN-END block that contains declarations of structure items.

Table 42 Structure Items

Structure Item	Description
Simple variable	A single-element variable
Array	A variable that contains multiple elements of the same data type
Substructure	A structure nested within a structure (to a maximum of 64 levels)
Filler byte	A place-holding byte
Filler bit	A place-holding bit
Simple pointer	A variable that contains a memory address, usually of a simple variable or array, which you can access with this simple pointer
Structure pointer	A variable that contains the memory address of a structure, which you can access with this structure pointer
Redefinition	A new identifier and sometimes a new description for a substructure, simple variable, array, or pointer declared in the same structure

You can nest substructures within structures (that is, you can declare a substructure within a substructure within a substructure, and so on) as deeply as the pTAL stack allows (approximately 60 levels). The structure and each substructure has a BEGIN-END level depending on the level of nesting.

The syntax for declaring each structure item is described after the syntax for declaring structures. The following rules apply to all structure items:

- You can declare the same identifier in different structures and substructures, but you cannot repeat an identifier at the same BEGIN-END level.
- You cannot initialize a structure item when you declare it. After you have declared it, however, you can assign a value to it by using an assignment statement or move statement.

- You can control how the compiler aligns a structure in memory and the fields of a structure within a structure by using the `FIELDALIGN` clause or `FIELDALIGN` compiler directive. Definition structure and template structure declarations can optionally include a `FIELDALIGN` clause. You cannot specify a `FIELDALIGN` clause on a referral structure declaration.
- If you declare a structure pointer and assign the address of a structure to it or use a reference parameter to address structure data you can specify a `REALIGNED` clause to ensure that the structure is well-aligned.

Topics:

- [Overview of Structure Alignment \(page 116\)](#)
- [Structures Aligned at Odd-Byte Boundaries \(page 117\)](#)

Overview of Structure Alignment

The memory alignment of the fields of a structure is important to pTAL. A field that is aligned for fastest access is said to be well-aligned. A field that is not aligned for fastest access is said to be misaligned.

A structure is well-aligned if the address of the base of the structure in memory is a multiple of its base alignment; otherwise, the structure is misaligned. If a structure is misaligned, some or all of its fields will also be misaligned.

The layout of structures and the alignment options you specify affect the object code generated by pTAL. If you specify that the fields of a structure are not well-aligned (by specifying the `FIELDALIGN` clause with the `SHARED2` parameter) pTAL generates conservative code for each reference. Conservative code might require more instructions to reference structure fields than references to well aligned fields.

Each structure declaration specifies whether pTAL generates fast code or conservative code when your program references a field of the structure.

Fast code takes full advantage of the RISC and Itanium architectures and produces the best performance, provided that the field being referenced is well-aligned. If the field is misaligned, an exception occurs. Access to the misaligned field is handled by a millicode exception handler that completes the access but at a significant performance cost.

Conservative code is somewhat slower than fast code but does not cause exceptions when it accesses misaligned data.

pTAL ensures that definition structures and referral structures are well-aligned; however, if you declare a structure pointer and assign the address of a structure to it or use a reference parameter to address structure data, the compiler cannot ensure that the structure is well-aligned; therefore, when you declare a structure pointer, you can specify what assumptions you want pTAL to make when it generates code to access your data. You can specify a `REALIGNED` clause (see [REALIGNED Clause \(page 134\)](#)).

The overall guidelines for alignment for a native process are:

- Accessing data in memory takes the least amount of time if the data is well-aligned and either the compiler has allocated the data or you reference the data with a pointer that specifies `REALIGNED(8)`. A data item is well-aligned if its byte address is an integral multiple of its length. For example, an `INT` is well-aligned if it begins at an even-byte address, an `INT(32)` at an address that is a multiple of four, and so forth.
- Accessing data is somewhat slower if the data is not well-aligned and you reference the data using a pointer that specifies `REALIGNED(2)`.
- Accessing data is significantly slower if the data is not well-aligned, but pTAL generates code that functions as if the data is well-aligned. In this case, your program traps to the millicode exception handler, which completes the data access and returns to your program.

To comply with these guidelines, some structures require that you explicitly add filler to ensure that:

- Each field begins at an address that is a multiple of its length.
- The total length of a structure is a multiple of the widest field in the structure.

Structures Aligned at Odd-Byte Boundaries

If you attempt to access data at an odd-byte address, the results are undefined, whether the data is a simple variable or a field of a structure. Your program might or might not trap.

Overview of Field Alignment

This subsection gives you an overview of the `FIELDALIGN` clause and the `FIELDALIGN` compiler directive, and the field alignment parameters `SHARED2`, `SHARED8`, `PLATFORM`, and `AUTO`.

The `FIELDALIGN` clause specifies the alignment of a structure and of all substructures that do not specify a `FIELDALIGN` clause. For details, see [FIELDALIGN Clause \(page 127\)](#).

The `FIELDALIGN` compiler directive specifies the default alignment for all structures. It includes the `SHARED2`, `SHARED8`, `PLATFORM`, and `AUTO` parameters as well as a `NODEFAULT` parameter. For more information, see [Chapter 17 \(page 367\)](#).

When you declare a definition or template structure, you specify (either explicitly using a `FIELDALIGN` clause or implicitly according to the current setting of the `FIELDALIGN` compiler directive) how you want the compiler to allocate memory for the structure. The field alignment for each such structure is specified by one of the following parameters to a `FIELDALIGN` clause or directive:

You use `SHARED2` and `SHARED8` field alignment for structure data used by processes running in either pTAL or TAL. You can share data by interprocess communication or by accessing it on a shared storage medium such as disk or tape.

Your program might use library routines that require that structure data be in a `SHARED2` or `SHARED8` format. If you use library routines that include structures that specify `SHARED2` or `SHARED8`, you might need to declare your structures with the same field alignment as the structures in the library.

If more than one program uses the same source file, you might want to include a `FIELDALIGN` clause on every structure declaration in the source file. This ensures that the field alignment of every structure is consistent across all programs that compile the source file.

If you do not specify a `FIELDALIGN` clause, each structure will use the current setting of the `FIELDALIGN` compiler directive, which might be different for different compilations.

Topics:

- [SHARED2 \(page 117\)](#)
- [SHARED8 \(page 118\)](#)
- [PLATFORM \(page 118\)](#)
- [AUTO \(page 118\)](#)
- [Differences Between PLATFORM and AUTO \(page 119\)](#)

SHARED2

`FIELDALIGN(SHARED2)` directs the compiler to allocate the structure's fields. Specify `FIELDALIGN(SHARED2)` when:

- Your process is limited by the available stack space in TAL programs.
- You want the structure to hold data (for example, interprocess messages, memory, or files) that is shared by processes or applications running on a combination of TAL-compiled processes and RISC and Itanium architectures.

For more information, see [FIELDALIGN Clause \(page 127\)](#) and [SHARED2 Parameter \(page 128\)](#).

SHARED8

FIELDALIGN(SHARED8) directs the compiler to allocate the structure's fields for optimal performance in pTAL. Specify FIELDALIGN(SHARED8) when:

- You want optimal performance in pTAL.
 - The fields you reference in the structure are well-aligned.
 - All processes that share the data can use SHARED8 alignment.
 - You want the structure to hold data (for example, interprocess messages, memory, or files) that is shared by processes or applications that are composed of both pTAL and TAL code.
- In TAL, access to SHARED8 components is as efficient as access to SHARED2 components, but SHARED8 components usually require more space than SHARED2 components.

For more information, see [FIELDALIGN Clause \(page 127\)](#) and [SHARED8 Parameter \(page 129\)](#).

PLATFORM

FIELDALIGN(PLATFORM) directs the compiler to allocate the structure's fields according to a layout that is consistent across different programming languages running on a given architecture. (PLATFORM field alignment is not consistent across different architectures.) The data might be shared among modules written in different programming languages, in one of these ways:

- Running within a single process
- Running in separate processes, all of which are either pTAL, TAL, or C/C++, but not a combination of these

pTAL allocates the fields of a PLATFORM structure according to the rules used by the native mode HP C compiler for PLATFORM layouts; that is:

- Each field begins at an address that is an integral multiple of the length of the field. That is, pTAL allocates 1-byte, 2-byte, 4-byte, and 8-byte fields at addresses that are integral multiples of one, two, four, and eight, respectively.
- UNSIGNED fields are not necessarily aligned to byte boundaries. They can share 1-byte, 2-byte, and 4-byte containers with other items. An UNSIGNED field, however, cannot span an address that is an integral multiple of four. If an UNSIGNED item would span a 4-byte address boundary, the compiler allocates the UNSIGNED field beginning at the next 4-byte boundary.
- The alignment of a structure or substructure is the alignment of its widest field, unless the structure or substructure contains an UNSIGNED field, in which case, the alignment of the structure or substructure is at least four.
- The compiler adds bytes, as needed, to the end of a PLATFORM structure or substructure such that the length of the structure or substructure is an integral multiple of its widest field.

AUTO

FIELDALIGN(AUTO) directs the compiler to align structure fields for optimal access on the architecture on which the object file will be run. Specify AUTO only for structures whose data exists solely within a process. Use PLATFORM to share data across processes.

Use AUTO field alignment for a structure that you use only locally—that is, only within a process—not between processes that run on different architectures. (AUTO field alignment is not consistent across different architectures and compilers.)

A structure's layout can be different in pTAL, TAL, and C/C++ if the structure describes data that is used only within a process and only for the duration of the process. In this case, you can specify AUTO as the FIELDALIGN parameter.

Specify FIELDALIGN(AUTO) for structures that are not used to exchange data between processes.

Do not assume that fields of an AUTO structure are contiguous in memory. The compilers insert filler where required for optimal alignment.

Pointer fields and nonpointer fields in structures declared with AUTO field alignment can be any address type or data type, respectively.

TAL, pTAL, and C lay out AUTO structures differently.

Differences Between PLATFORM and AUTO

PLATFORM structures and substructures can contain UNSIGNED and STRING items within a 2-byte word. In AUTO structures and substructures, STRING items and UNSIGNED items are not allocated within a 2-byte word.

PLATFORM structures and substructures can contain an odd number of bytes. AUTO (and SHARED8) structures must contain an even number of bytes. pTAL adds an extra byte at the end of AUTO structures if, without the byte, the structure would contain an odd number of bytes.

The length of PLATFORM structures or substructures that contains an UNSIGNED item must be an integral multiple of four bytes. pTAL adds extra bytes, as needed, to the end of such structures and substructures.

Field and Base Alignment

The field alignment of a structure specifies the offsets at which fields of the structure must begin relative to the base of the structure. A scalar field is well-aligned when its byte offset is an integral multiple of its width. A substructure is well-aligned when the offset of its base, relative to its encompassing structure, is an integral multiple of its widest field.

Use the FIELDALIGN clause to specify how you want pTAL to align the fields in the structure. For more information, see [FIELDALIGN Clause \(page 127\)](#).

Topics:

- [Base Alignment \(page 119\)](#)
- [Structure Alignment Examples \(page 120\)](#)

Base Alignment

The base alignment of a structure is the alignment of the widest field in the structure. The base alignment determines where the structure can be located in memory and be well-aligned. A structure is well-aligned when the memory address at which it is located is an integral multiple of its base alignment.

Table 43 Base Alignment and Field Alignment Relationships

Width of Widest Field in Structure	FIELDALIGN(SHARED2)	FIELDALIGN(SHARED8)
1	1 or 2*	1 or 2*
2	2	2
4	2	4
8	2	8

*Definition (inline) substructures have a base alignment of one. All other structures—definition structures, referral structures, and referral substructures—have a base alignment of two.

A structure is well-aligned if the address of the base of the structure in memory is a multiple of its base alignment; otherwise, the structure is misaligned. If a structure is misaligned, some or all of its fields will also be misaligned.

Structure Alignment Examples

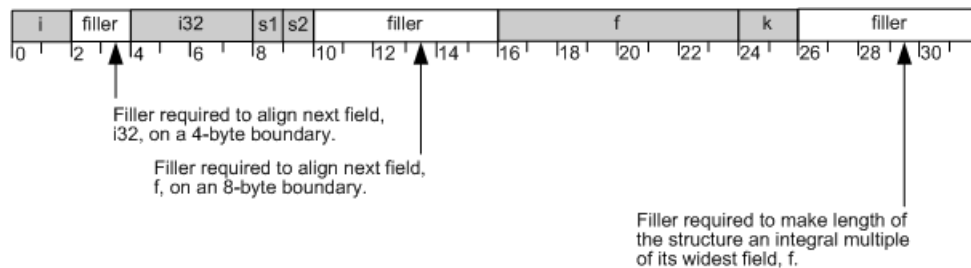
The following examples illustrate how your structure data layout is affected by structure alignment. Only SHARED8 structures are shown because SHARED2 structures are not well-aligned. pTAL always generates conservative code for references to fields of a SHARED2 structure that are more than 16 bits long.

Figure 4 (page 120) shows a structure, `s1`, that specifies `FIELDALIGN(SHARED8)`. Because the widest field in the structure, `f`, is a `FIXED` field, the base alignment of `s1` is 8. To be well-aligned, `s1` must be allocated at a memory address that is an integral multiple of eight. Filler is added as follows:

- Before `i32` so that `i32` begins at an offset that is a multiple of four relative to the beginning of the structure.
- Before `f` so that `f` begins at an offset that is a multiple of eight relative to the beginning of the structure.
- At the end of the structure so that the total length of the structure is an integral multiple of the widest field in the structure.

Figure 4 Alignment of SHARED8 Structure With Base Alignment of 8

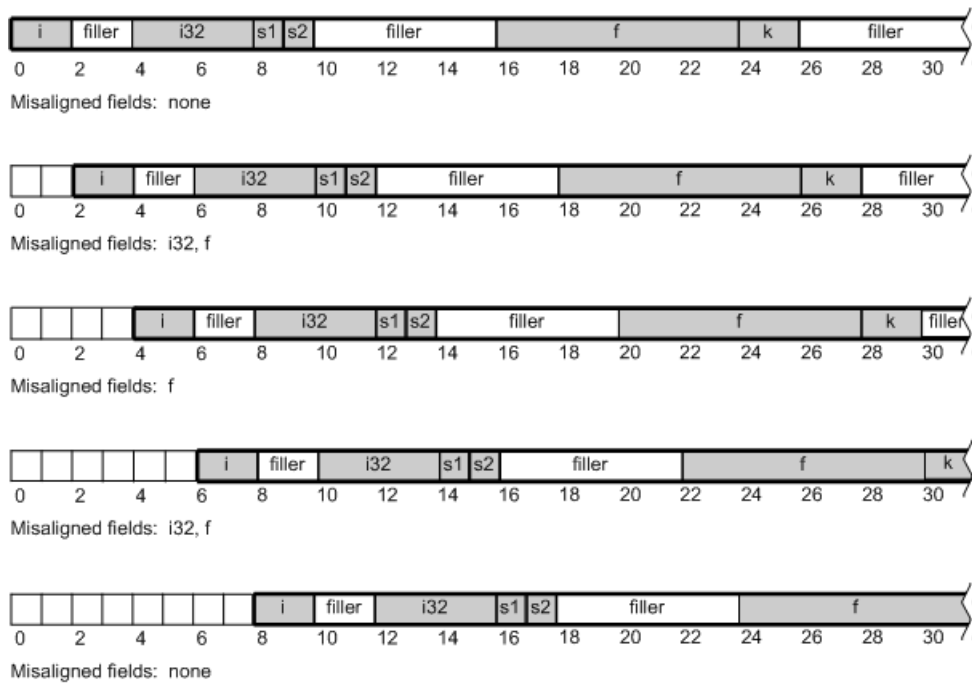
```
STRUCT s FIELDALIGN(SHARED8); ! Base alignment of s1 is 8
BEGIN
  INT    i;      ! Begins at offset 0
  FILLER 2;      ! 2 bytes of filler required
  INT(32) i32;   ! Begins at offset 4
  STRING s1;     ! Begins at offset 8
  STRING s2;     ! Begins at offset 9
  FILLER 6;      ! 6 bytes of filler required
  FIXED  f;      ! Begins at offset 16
  INT    k;      ! Begins at offset 24
  FILLER 6;      ! Must pad to multiple of widest field, f
END;             ! Total length of s1: 32 bytes
```



VST007.vsd

Figure 5 (page 121) shows which fields of `s1` are misaligned if the base of the structure in memory is not at an integral multiple of its base alignment. Only structures whose base is at an even-byte address are shown. Accessing structures whose base is at an odd-byte offset produces undefined results. For more information, see [Overview of Structure Alignment \(page 116\)](#).

Figure 5 Well-Aligned and Misaligned SHARED8 Structures With Base Alignment of 8

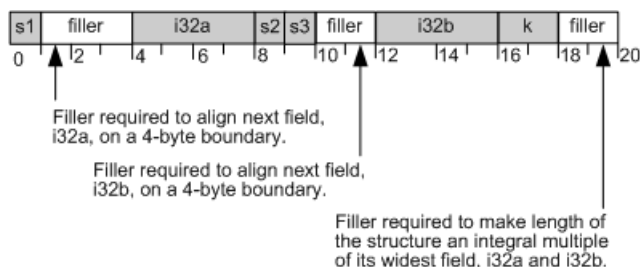


VST009.vsd

Figure 6 (page 121) shows a structure that is declared `FIELDALIGN(SHARED8)`. The widest fields in `s2`, `i32a` and `i32b`, are each four bytes; therefore, although the field alignment of `s2` is `SHARED8`, the base alignment of `s2` is four, not eight. `s2` is well-aligned in memory if the base of the structure begins at any address that is a multiple of four.

Figure 6 Alignment of a SHARED8 Structure With Base Alignment of 4

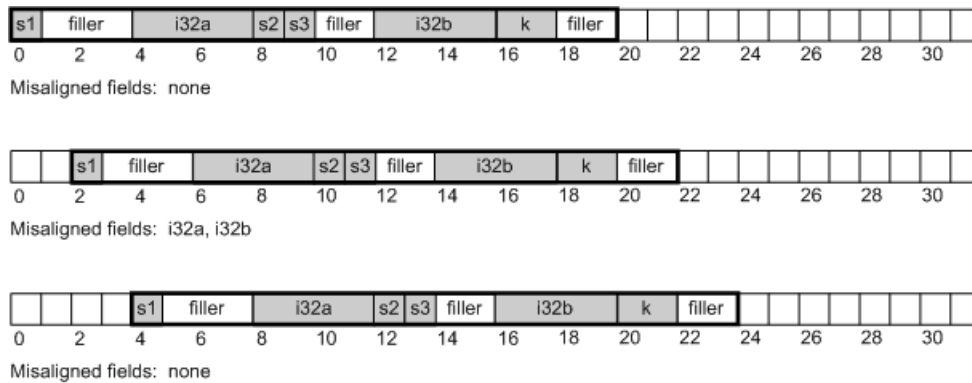
```
STRUCT s2 FIELDALIGN(SHARED8); ! Base alignment is 4
BEGIN
    STRING s1      ! Begins at offset 0
    FILLER 3;      ! 3 bytes of filler required
    INT(32) i32a;   ! Begins at offset 4
    STRING s2      ! Begins at offset 8
    STRING s3      ! Begins at offset 9
    FILLER 2;      ! 2 bytes of filler required
    INT(32) i32b;   ! Begins at offset 12
    INT k;         ! Begins at offset 16
    FILLER 2;      ! Must pad to multiple of base alignment
END;              ! Total length of s2: 20 bytes
```



VST997.vsd

Figure 7 (page 122) shows which fields are misaligned if `s2` is allocated at an address other than a 4-byte address.

Figure 7 Well-Aligned and Misaligned SHARED8 Structures With Base Alignment of 4



VST114.vsd

Array Alignment in Structures

When you declare an array in a structure, the alignment of the beginning of the array is the alignment of the base type of the array. Thus, for example, the field alignment of an array of INTs is the same as the field alignment of a single INT, which is 2. Declaring an array in a structure is the same as explicitly declaring individual fields, each with the same data type as the array's base type.

In [Example 52 \(page 122\)](#), the layouts and base alignments of `s1` and `s2` are identical:

Example 52 Arrays Within Structures

```
STRUCT s1 FIELDALIGN(SHARED8);
BEGIN
  INT i;
  INT a[0:2];
  STRUCT s1[0:1];
  BEGIN
    INT(32)w;
    INT y;
    INT x;
  END;
END;

STRUCT s2 FIELDALIGN(SHARED8);
BEGIN
  INT i;
  INT a;
  INT b;
  INT c;
  STRUCT sla;
  BEGIN
    INT(32)w;
    INT y;
    INT x;
  END;
  STRUCT slb;
  BEGIN
    INT(32)w;
    INT y;
    INT x;
  END;
END;
```

An array of structures or substructures is the same as an array of a pTAL data type. The width of the widest field of an element of such an array, combined with the FIELDALIGN parameter you specify, determines the required alignment of the structure or substructure and of its fields.

Example 53 SHARED2: 2-Byte Alignment

```
STRUCT s1[0:9] FIELDALIGN(SHARED2);  ! s1 is SHARED2
BEGIN                                ! Alignment is 2
    INT      i;
    FILLER 2;
    INT(32)   j;
END;
```

Example 54 SHARED8: 4-Byte Alignment

```
STRUCT s2[0:9] FIELDALIGN(SHARED8);  ! s2 is SHARED8
BEGIN                                ! Alignment is 4
    INT      i;
    FILLER 2;
    INT(32)   j;
END;
```

Example 55 8-Byte Alignment and 4-Byte Alignment

```
STRUCT s3[0:9] FIELDALIGN(SHARED8);  ! s3 is SHARED8
BEGIN                                ! Alignment is 8
    STRUCT s4[0:4];                  ! s4 is SHARED8
    BEGIN                            ! Alignment is 4
        INT      i;
        FILLER 2;
        INT(32)   j;
    END;
    REAL(64)      k;
END;
```

Structure Alignment

A structure's alignment is the alignment of the widest field declared in the structure and is always less than or equal to the alignment specified in a FIELDALIGN clause or FIELDALIGN compiler directive. For alignment values of structure fields, see [Table 43 \(page 119\)](#). The alignment of a field that is a substructure is the alignment of the widest field contained in the substructure.

If the alignment of the widest field in a SHARED8 structure is 2, the structure must begin at a 2-byte address, and the structure's base alignment is 2. If the alignment of the widest field in the structure is four bytes, for example an INT(32), the structure must begin at a 4-byte address. If the alignment of the widest field in the structure is eight bytes, for example an FIXED field, the structure must begin at an 8-byte address.

Example 56 SHARED8 Structures With Different Base Alignments

```
STRUCT s1 FIELDALIGN(SHARED8);  ! Base alignment of structure is
BEGIN                            ! 2 because of INT c
    STRING a;
    STRING b;
    INT     c;                  ! c is field with widest
END;                             ! alignment: 2
STRUCT s2 FIELDALIGN(SHARED8);  ! Base alignment of structure is
BEGIN                            ! 4 because of INT(32) c
    STRING a;
    STRING b;
    FILLER 2;
```

```

    INT(32) c;                ! c is field with widest
END;                        ! alignment: 4
STRUCT s3 FIELDALIGN(SHARED8); ! Base alignment of structure is
BEGIN                      ! 8 because of FIXED c
    STRING a;
    STRING b;
    FILLER 6;
    FIXED c;                ! c is field with widest
END;                        ! alignment: 8

```

Substructure Alignment

The rules for field alignment of substructures are the same as the rules for structures. You can specify the field alignment of a substructure explicitly using a `FIELDALIGN` clause or implicitly by allowing the field alignment of the substructure to default to the field alignment of the containing structure or substructure. In either case, the alignment of fields must conform to the rules described previously, under “Using Field Alignment.” For `SHARED8` structures, you must ensure that every field begins at an appropriate address and that the end of the structure includes filler, if necessary, so that the total length of the substructure is an integral multiple of its widest field.

The following rules apply to substructures:

- A definition substructure that does not specify a `FIELDALIGN` clause inherits the field alignment of its containing structure or substructure.
- The base alignment of a substructure is the alignment of the widest field of the substructure.
- Begin the base of a substructure at an offset that is an integral multiple of the substructure’s alignment, relative to the start of its containing structure or substructure. If the substructure is a definition substructure and both the structure and substructure have `SHARED8` field alignment, the substructure must be well aligned.

Example 57 Well-Aligned Structure With Well-Aligned Substructure

```

STRUCT s FIELDALIGN(SHARED8);
BEGIN
    INT i;
    FILLER 2;                ! ss is 4-byte aligned. Use FILLER 2 to
                            ! force ss to a 4-byte address
    STRUCT ss;              ! Specified alignment of ss is SHARED8,
BEGIN                      ! inherited from s
    INT(32) m;
    INT n;
    FILLER 2;              ! Alignment of substructure ss is 4
END;                      ! FILLER 2 makes total length of ss 8
    INT j;
    STRING t[0:2];
    FILLER 3;              ! Alignment of structure s is 4: declare
END;                      ! FILLER 3 to make length of s an integral
                            ! multiple of its widest field

```

For further information about substructures, see [Alignment Considerations for Substructures \(page 126\)](#).

Example 58 SHARED8 Structures With SHARED2 Substructures

```
STRUCT t_s2(*) FIELDALIGN(SHARED2);    ! Base alignment of t_s2
BEGIN                                ! is 2
    INT(32) j;
END;
STRUCT t_s8(*) FIELDALIGN(SHARED8);    ! Base alignment of t_s8
BEGIN                                ! is 4
    INT(32) j;
END;
```

Example 59 SHARED2 Structures With SHARED8 Substructure

```
STRUCT s1 FIELDALIGN(SHARED2);
BEGIN
    INT i;
    STRUCT s2(t_s8);                ! s2 has SHARED8 alignment
END;                                ! Base alignment of s2 is 2
INT .p2(t_s8) REFALIGNED(2);        ! Reference alignment is 2
INT .p3(t_s8);                      ! Reference alignment defaults
                                    ! to 8

PROC p;
BEGIN
    INT i;
    @p2 := @s1.s2 '>>' 1;
    @p3 := @s1.s2 '>>' 1;
    i := p2.j;
    i := p3.j;
    ...
END;
```

In [Example 59 \(page 125\)](#):

- Because `s1` specifies SHARED2 field alignment, pTAL generates conservative code that ensures that an exception does not occur when you reference `s1.s2.j`.
- `p2` refers to `t_s8`, a SHARED8 substructure. `p2` specifies a reference alignment of 2, which ensures that pTAL generates conservative code that will not cause exceptions for misaligned memory references.
- `p3` does not have a REFALIGNED clause. Its reference alignment, therefore, defaults to the field alignment of its referent, which is `t_s8`, which has SHARED8 field alignment. pTAL generates fast code for each reference to `p3.j`.

In the formal parameter specification for a structure pointer, declare reference alignment 2 unless you are certain that all pointers passed to the parameter reference SHARED8 structures that you know are well-aligned. If you are not certain that all references are well-aligned, use the same approach as that shown earlier to ensure that references to structures passed as actual parameters do not cause a trap.

When you design routines that return addresses to their callers, return addresses that are well-aligned whenever possible.

Example 60 SHARED8 Structure With SHARED2 Substructure

```
STRUCT s3 FIELDALIGN(SHARED8);
BEGIN                                ! Base alignment of s3 is 4
  INT i;
  STRUCT s4(t_s2); ! s2 has SHARED8 alignment
END;                                ! Base alignment of s2 is 4
INT .p4(t_s2); ! Uses default alignment: 2
```

The compiler always generates conservative code. In [Example 60 \(page 126\)](#), references to `s3.s4.j` do not cause traps because, although `s3` is SHARED8, the offset of `s3.s4.j` is not a multiple of 4. For each reference pTAL determines whether the referenced field is well-aligned. References to fields in `s4` using the pointer `p4`—for example, `p4.j`—do not cause traps because the field alignment of `s4` is SHARED2 and the compiler generates conservative code for such references.

Example 61 Combining SHARED2 and SHARED8 Structures

```
PROC p;
BEGIN
  STRUCT s1 FIELDALIGN(SHARED8); ! OK
  BEGIN
    FIXED i;
  END;
  STRUCT s2 FIELDALIGN(SHARED2); ! OK
  BEGIN
    INT(32) i; ! OK
    STRUCT sub (s1); ! WARNING: SHARED8
  END; ! substructure in SHARED2
  ! structure can cause
  ! significant loss of
  ! performance

  STRUCT s3 (s1) = s2;
  STRUCT s4 FIELDALIGN(SHARED8);
  BEGIN
    INT i;
    STRUCT sub1 (s2); ! OK: SHARED2
    STRUCT sub2 (s1) = sub1; ! WARNING: SHARED8 substructure
    FILLER 2; ! redefines SHARED2 substructure
  END; ! can cause significant loss of
END; ! performance
```

Alignment Considerations for Substructures

When you declare a substructure, you must be aware of how the base alignment of the substructure and its containing structure affect references to the fields of the structure and substructure.

Table 44 Field Alignment of Substructures

Substructure Field Alignment	Structure Field Alignment			
	AUTO	PLATFORM	SHARED8	SHARED2
AUTO	AUTO	PLATFORM	Invalid	Invalid
PLATFORM	invalid	PLATFORM	invalid	invalid
SHARED8	SHARED8	SHARED8	SHARED8	SHARED8
SHARED2	SHARED2	SHARED2	SHARED2	SHARED2
Default	AUTO	PLATFORM	SHARED8	SHARED2

If a SHARED8 substructure is contained in a SHARED2 structure (or in an AUTO structure), fields in the SHARED8 substructure will be well-aligned with respect to the base of the SHARED8

substructure but might not be well-aligned with respect to the base of the SHARED2 structure. Performance will be somewhat degraded when fields in the substructure are referenced.

If a SHARED2 substructure is contained in a SHARED8 structure (or in an AUTO structure), fields in the SHARED2 substructure will be well-aligned with respect to the base of the SHARED2 substructure but might not be well-aligned with respect to the base of the SHARED8 structure. Performance will be significantly degraded when fields in the substructure are not well-aligned for SHARED8 access. Each such reference will cause a trap to the millicode exception handler to resolve the reference. Your program will behave correctly but will be significantly slower than it would without the trap.

Example 62 AUTO Field Alignment in Structure (Error)

```
STRUCT s FIELDALIGN(SHARED8) ;
BEGIN
  STRUCT s1 FIELDALIGN(AUTO);    ! ERROR: Substructure cannot be
  BEGIN                          ! FIELDALIGN(AUTO)
    ...
  END;
END;
```

The compiler pads SHARED2 structures and substructures with an extra byte if the end of the last field in the structure or substructure ends at an odd-byte address, unless the structure has 1-byte alignment—that is, all fields in the structure or substructure are STRINGS or UNSIGNED(1-8).

STRING fields in structures can begin at any byte offset.

FIELDALIGN Clause

You use a FIELDALIGN clause in a structure declaration to specify how you want pTAL to align the fields in the structure. Fields can be aligned for:

Access	FIELDALIGN Clause
Exclusive, optimized for best resource utilization on each architecture	FIELDALIGN(AUTO)
Shared between pTAL and TAL programs	FIELDALIGN(SHARED2)
Shared by program modules written in different programming languages and running on the same architecture	FIELDALIGN(PLATFORM)
Shared between TNS, TNS/R, and TNS/E architecture with optimal performance on TNS/R and TNS/E architecture	FIELDALIGN(SHARED8)

FIELDALIGN Compiler Directive

As with the FIELDALIGN clause, the parameters to the FIELDALIGN compiler directive include SHARED2, SHARED8, PLATFORM, and AUTO. In addition, you can specify NODEFAULT as the parameter to the FIELDALIGN compiler directive.

You can specify only one FIELDALIGN directive within a compilation, and it must precede all data, block, and procedure declarations. Only comments, blank lines, and other directives can precede a FIELDALIGN directive.

The default value of the FIELDALIGN directive is AUTO.

If you specify the FIELDALIGN (NODEFAULT) compiler directive, pTAL requires you to specify a FIELDALIGN clause on every structure declaration. You might use the FIELDALIGN (NODEFAULT) directive to ensure that you do not inadvertently omit a FIELDALIGN clause on any structure.

If you do not specify a FIELDALIGN (NODEFAULT) directive, pTAL does not require you to specify a FIELDALIGN clause on each structure declaration.

SHARED2 Parameter

Since the SHARED2 parameter is included with both the FIELDALIGN clause and the FIELDALIGN compiler directive, the following information relates to both usages:

- In a SHARED2 structure, all fields must begin at an even-byte address except STRING fields, which can begin at any byte address, and UNSIGNED fields, which can begin at any bit address except as follows:
 - An UNSIGNED(1-16) field cannot cross an even-byte address boundary.
 - An UNSIGNED(17-31) field can cross only one even-byte address boundary.
 - An UNSIGNED field that is not preceded by an UNSIGNED field must begin at an even-byte address.

- The address type of pointers in a SHARED2 structure must be EXTADDR, EXT32ADDR, EXT64ADDR, PROC32ADDR, PROC64ADDR, SGBADDR, or SGWADDR; for example:

```
STRUCT s FIELDALIGN(SHARED2);
BEGIN
    INT .EXT ea;          ! OK: EXTADDR pointer
    INT .EXT32 e32a;      ! OK: EXT32ADDR pointer
    INT .EXT64 e64a;      ! OK: EXT64ADDR pointer
    INT .SG j;            ! OK: SGWADDR pointer
    STRING .s;            ! ERROR: BADDR pointer is not valid
END;
```

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

- If the data type of a field in a SHARED2 structure is an address type, the type must be EXTADDR, EXT32ADDR, EXT64ADDR, PROC32ADDR, PROC64ADDR, SGBADDR, or SGWADDR; for example:

```
STRUCT s FIELDALIGN(SHARED2);
BEGIN
    EXTADDR ea;          ! OK
    EXT32ADDR e32a;      ! OK
    EXT64ADDR e64a;      ! OK
    PROCADDR pa;         ! ERROR: not allowed in SHARED2 struct.
    PROC32ADDR p32a;     ! OK
    WADDR w;             ! ERROR: not allowed in a SHARED2 struct;
END;
```

- If you include a FIELDALIGN(SHARED2) compiler directive, include a REFALIGNED(2) compiler directive as well. The default for the REFALIGNED compiler directive is 8. With field alignment SHARED2, pTAL can allocate a 32-bit or 64-bit field at any even-byte address. pTAL generates optimal code for data references that use a pointer whose reference alignment is 8. If the pointer is used to reference 32-bit or 64-bit data that is not well-aligned, each reference to the data will be slow. By default, pTAL generates conservative code when you reference data using a pointer that specifies REFALIGNED(2). The REFALIGNED(2) directive ensures that pTAL generates conservative code for pointers that do not specify a REFALIGNED clause.

Example 63 FIELDALIGN(SHARED2) and REFALIGNED(2) Directives

```
?FIELDALIGN(SHARED2)
?REFALIGNED(2)
INT(32).ptr;           ! Global pointer
PROC p;
BEGIN
  @ptr := @str.F32;     ! str.F32 might or might not be aligned
                      ! at a 32-bit address. REFALIGNED
  ... := ptr + 3D;      ! directive ensures that pTAL
                      ! generates conservative code for
END;                    ! references to ptr.
```

Example 64 Byte Offsets (Decimal) of Fields of a SHARED2 Structure

```
STRUCT s1 FIELDALIGN(SHARED2);
BEGIN
  INT      i;           ! i begins at byte offset: 0
  INT(32)  j;           ! j begins at byte offset: 2
  STRING   s1;          ! s1 begins at byte offset: 6
  UNSIGNED(3) u1;        ! u1 begins at byte offset: 8
  UNSIGNED(2) u2;        ! u2 begins at byte offset: 8 + 3 bits
  STRING   s2;          ! s2 begins at byte offset: 10
  FIXED    f;           ! f begins at byte offset: 12
  INT      k;           ! k begins at byte offset: 20
END;
```

SHARED8 Parameter

Since the SHARED8 parameter is included with both the FIELDALIGN clause and the FIELDALIGN compiler directive, the following information relates to both usages:

- The structure must begin at an address that is an integral multiple of the width of the widest field in the structure. Thus:
 - A 1-byte field (STRING) can begin at any byte address.
 - The byte offset of a 2-byte field [INT or UNSIGNED(1-16)] must be an even number, except that contiguous UNSIGNED fields can be packed.
 - The byte offset of a 4-byte field [INT(32), REAL, UNSIGNED(17-31)] must be an integral multiple of four, except that contiguous UNSIGNED fields can be packed.
 - The byte offset of an 8-byte field [FIXED or REAL(64)] must be an integral multiple of eight.
 - The byte offset of a substructure field must be an integral multiple of the widest field in the substructure.
 - The byte offset of an array must be an integral multiple of an element of the array—that is, one of the previous items in this list.
- In a SHARED8 structure or substructure, you must explicitly declare filler items as needed to ensure that fields are aligned according to the preceding rules.

Table 45 Variable Alignment

Data Type	Alignment	Notes
STRING	1	
INT	2	
UNSIGNED(1-16)	2	Multiple UNSIGNED fields can be packed in a word or doubleword.
.SG pointers	2	.SG pointers are 16 bits in both pTAL and TAL.*

Table 45 Variable Alignment *(continued)*

Data Type	Alignment	Notes
.SGX pointers	4	Allowed in structures only with AUTO field alignment.**
Other 16-bit pointers	4	Allowed in structures only with AUTO field alignment.**
32-bit Pointer	4	
64-bit Pointer***	8	
INT(32)	4	
REAL	4	
UNSIGNED(17-31)	4	Multiple UNSIGNED fields can be packed into a doubleword.
FIXED	8	
REAL(64)	8	
<p>* In pTAL, the alignment for all address types is 4, except SGBADDR, SGWADDR, EXT64ADDR, and PROC64ADDR addresses for which the alignment is 2, 2, 8, and 8 respectively. In TAL, the alignment of all address types is 2.</p> <p>** The alignment of an array is the alignment of its element type.</p> <p>*** 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, "64-bit Addressing Functionality" (page 531).</p>		

- For compatibility with TAL, pTAL requires you to explicitly declare filler items to optimally align a SHARED8 structure's fields for RISC and Itanium architecture. pTAL does not add filler automatically to SHARED8 structures and reports a syntax error if you do not declare filler where a structure requires it. The compiler listing shows where each structure requires filler. You must add filler:
 - Before a field if the field's offset from the beginning of the structure is not an integral multiple of the field's width (see [Table 43 \(page 119\)](#))
 - If the total length of the structure or substructure would not be an integral multiple of the structure or substructure's widest field
 - If an UNSIGNED(1-16) field would otherwise cross an even byte address
 - If an UNSIGNED(17-31) field would otherwise cross a four byte address
- The address type of pointers in a SHARED8 structure must be EXTADDR, SGBADDR, or SGWADDR.
- If the data type of a field in a SHARED8 structure is an address type, the type must be EXTADDR, EXT32ADDR, EXT64ADDR, PROC32ADDR, PROC64ADDR, SGBADDR, or SGWADDR, as shown in the following example:

```

STRUCT s FIELDALIGN(SHARED8);
BEGIN
  EXTADDR      x;      ! OK: EXTADDR field
  EXT32ADDR    y;      ! OK: EXT32ADDR field
  EXT64ADDR    z;      ! OK: EXT64ADDR field
  PROC32ADDR   a;      ! OK: PROC32ADDR field
  FILLER 4;
  PROC64ADDR   b;      ! OK: PROC64ADDR field
  INT .EXT     ea;      ! OK: EXTADDR pointer
  INT .EXT32   e32a;    ! OK: EXT32ADDR pointer
  INT .EXT64   e64a;    ! OK: EXT64ADDR pointer
  INT.SG       j;      ! OK: SGWADDR pointer

```

```

        STRING      s;      ! ERROR: BADDR pointer is not valid
    END;

```

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Topics:

- [Alignment of Fields \(page 131\)](#)
- [Optimizing Structure Layouts \(page 131\)](#)
- [Structure Length \(page 132\)](#)
- [Alignment of UNSIGNED\(17-31\) Fields \(page 133\)](#)

Alignment of Fields

If a field in a SHARED8 structure is not well-aligned, you must explicitly declare filler to force the field to be well-aligned.

Example 65 Filler Forcing Alignment in a SHARED8 Structure

```

STRUCT s FIELDALIGN(SHARED8);
BEGIN
    INT          a;          ! 0 a uses 2 bytes
    FILLER 2;          ! 2 Force b to a 4-byte offset
    INT(32)      b;          ! 4 b uses 4 bytes
    STRING      c [0:2];    ! 8 c uses 3 bytes
    FILLER 5;          ! 11 Force d to an 8-byte offset
    FIXED       d;          ! 16 d uses 8 bytes
    INT(32)      e[0:1];    ! 24 e uses 8 bytes
    INT         f;          ! 32 f uses 2 bytes
    UNSIGNED(5) g;          ! 34 g uses 5 bits
    BIT_FILLER 11;        ! Force h to a 4-byte offset
    UNSIGNED(17) h;        ! 36 h uses 2 bytes plus 1 bit
    UNSIGNED(15) i;        ! 38 i uses 15 bits
END;                  ! Total structure length: 40 bytes

```

The first filler item (FILLER 2) forces *b* to begin at a 4-byte address. The second filler item (FILLER 5) forces *d* to begin at an 8-byte address. The third filler item (BIT_FILLER 11) forces *h* to begin at a 4-byte address.

Optimizing Structure Layouts

You do not need to declare filler items in a SHARED2 structure to align its fields. If filler is needed—for example to align bit fields, string fields, or fields that follow bit fields and string fields—the compiler inserts the needed filler.

Example 66 Structure With SHARED2 Field Alignment

```
STRUCT s1 FIELDALIGN(SHARED2);
BEGIN
    INT    i;    ! i begins at offset: 0
    INT(32) j;    ! j begins at offset: 2
    STRING s1;    ! s1 begins at offset: 6
    STRING s2;    ! s1 begins at offset: 7
    FIXED  f;    ! f begins at offset: 8
    INT    k;    ! k begins at offset: 16
END;           ! Total length of s1: 18 bytes
```

Structures that specify SHARED8 field alignment, however, require you to explicitly declare filler items to force fields to be well-aligned, as previously described. You might be able to reduce the size of a structure if you can arrange its fields to minimize the number of filler items required.

In [Example 67 \(page 132\)](#), the structure `s2` has the same fields as `s1` in [Example 66 \(page 132\)](#), but `s2` has SHARED8 field alignment and includes filler items where required. Offsets are shown in bytes. `s2` is 32 bytes.

Example 67 Structure With SHARED8 Field Alignment

```
STRUCT s2 FIELDALIGN(SHARED8);
BEGIN
    INT    i;    ! i begins at offset 0
    FILLER 2;    ! 2 bytes of filler
    INT(32) j;    ! j begins at offset 4
    STRING s1;    ! s1 begins at offset 8
    STRING s2;    ! s2 begins at offset 9
    FILLER 6;    ! 6 bytes of filler
    FIXED  f;    ! f begins at offset 16
    INT    k;    ! k begins at offset 24
    FILLER 6;    ! Pad to a multiple of the widest field (f)
END;           ! Total length of s2: 32 bytes
```

`s2` has SHARED8 field alignment, and uses 14 more bytes than `s1`. If the order of the fields within the structure is not important; however, you can rearrange the fields so that the structure contains fewer bytes, as shown in [Example 68 \(page 132\)](#).

Example 68 Optimized Structure With SHARED8 Field Alignment

```
STRUCT s3 FIELDALIGN(SHARED8);
BEGIN
    INT    i;    ! i begins at offset 0
    STRING s1;    ! s1 begins at offset 2
    STRING s2;    ! s2 begins at offset 3
    INT(32) j;    ! j begins at offset 4
    FIXED  f;    ! f begins at offset 8
    INT    k;    ! k begins at offset 16
    FILLER 6;    ! Pad to a multiple of the widest field (f)
END;           ! Total length of s3: 24 bytes
```

By rearranging the order of the fields, `s3` requires 24 bytes, rather than the 32 bytes required by `s2`, even though the information in `s3` and `s2` is the same. `s3` uses only six more bytes than `s1`.

Structure Length

The total number of bytes in a SHARED8 structure must be an integral multiple of the widest field in the structure. If needed, you must explicitly declare filler at the end of a SHARED8 structure to ensure this condition.

Example 69 Structures That Need Filler

```
STRUCT s1 FIELDALIGN(SHARED8);
BEGIN
    FIXED    i;    ! Structure's widest field is 8 bytes
    INT(32) j;    ! j is 4 bytes FILLER 4
    FILLER 4;    ! Pad with 4 bytes
END;
STRUCT s2 FIELDALIGN(SHARED8);
BEGIN
    INT(32) i;    ! Structure's widest field is 4 bytes
    INT      j;    ! j is 2 bytes
    FILLER 2;    ! Pad with 2 bytes
END;
```

UNSIGNED(1-16) fields cannot cross an even-byte address.

Example 70 Structure Field Crossing an Even-Byte Address (Error)

```
STRUCT s FIELDALIGN(SHARED8);
BEGIN
    UNSIGNED(10) u1;
    UNSIGNED(16) u2;    ! Invalid field -- crosses even-byte address
END;
```

In [Example 71 \(page 133\)](#), `u1` starts at the beginning of the structure. `u2`, therefore, would begin at a 10-bit offset from the beginning of `s`. Because `u2` is 16 bits, the last ten bits of `u2` would be allocated in a second word, which would cause `u2` to cross an even-byte address; therefore, you must explicitly declare filler to force `u2` to begin at the next even-byte offset from the beginning of `s`.

Example 71 Structure That Needs Filler

```
STRUCT s FIELDALIGN(SHARED8);
BEGIN
    UNSIGNED(10) u1;
    BIT_FILLER 6;    ! Forces u2 to begin at next even-byte address
    UNSIGNED(16) u2;
END;
```

Alignment of UNSIGNED(17-31) Fields

In a SHARED8 structure, UNSIGNED(17-31) fields cannot cross a 4-byte address. Because an UNSIGNED(17-31) field is longer than 16 bits, its base alignment is 4 bytes.

In [Example 72 \(page 133\)](#), `i` starts at the beginning of the structure. `u`, therefore, begins at an even-byte offset from the beginning of `s`. Because `u` is 28 bits, the last 12 bits of `u` would be allocated in the next word, which would cause `u` to cross a 4-byte address.

Example 72 SHARED8 Structure With Misaligned UNSIGNED Fields

```
STRUCT s FIELDALIGN(SHARED8);
BEGIN
    INT i;
    UNSIGNED(28) u;    ! Invalid field
END;
```

You must explicitly declare filler to force `u` to begin at the next 4-byte offset from the beginning of `s`.

Example 73 SHARED8 Structure With Correctly Aligned UNSIGNED Fields

```
STRUCT s FIELDALIGN(SHARED8);
BEGIN
  INT i;
  FILLER 2;           ! Forces u to begin at a 4-byte address
  UNSIGNED(28) u;
  BIT_FILLER 4;       ! Makes length of s an integral multiple of
END;                  ! 4 bytes
```

Reference Alignment With Structure Pointers

When you declare a structure pointer, you can specify a `REFALIGNED` clause as part of the declaration. (For the syntax of a structure pointer, see [Chapter 11 \(page 177\)](#).) You can use a `REFALIGNED` clause to override the base alignment of an instance of a structure, even though the field alignment for the structure does not change. For example, if you specify a `REFALIGNED(2)` clause on a structure pointer, pTAL generates conservative code each time you use the pointer to reference fields of the structure.

A `REFALIGNED` clause specifies the base alignment of the structures that the structure pointer will reference. The distinction between `FIELDALIGN` and `REFALIGNED` is required because structures referenced by a structure pointer can be located anywhere in memory, and might not be well-aligned. A structure might not be well-aligned if it is located in a dynamic memory area such as a heap, or was read from a file as part of a larger record.

The alignment of a structure pointer is the alignment specified in a `REFALIGNED` clause if present, or if not present, by the field alignment of the structure it references.

The `REFALIGNED` compiler directive does not affect the reference alignment of structure pointers. It is used only for pointers to nonstructure data.

You can specify the `REFALIGNED` clause on any pointer field. In [Example 74 \(page 134\)](#), field `d` is a 32-bit pointer in pTAL and is valid only if the field alignment of structure `s` is `AUTO` or `PLATFORM`.

Example 74 REFALIGNED Clause With Structure Pointers

```
STRUCT s;
BEGIN
  INT .d REFALIGNED(2);      ! Standard pointer with REFALIGNED
                             ! clause
  INT .EXT e REFALIGNED(8);  ! An extended pointer
END;
```

The same syntax and semantics of fields and pointer fields declared with a `REFALIGNED` clause is the same as that of variables and pointers declared with a `REFALIGNED` clause, respectively.

Topics:

- [REFALIGNED Clause \(page 134\)](#)
- [Default Reference Alignment \(page 135\)](#)
- [REFALIGNED\(2\) \(page 135\)](#)
- [REFALIGNED\(8\) \(page 136\)](#)
- [Code Generation for Structure References \(page 137\)](#)

REFALIGNED Clause

In a `SHARED2` or `SHARED8` structure, you can include only pointers whose address type is `SGBADDR`, `SGWADDR`, `EXTADDR`, `EXT32ADDR`, `EXT64ADDR`, `PROC32ADDR`, or `PROC64ADDR`. Pointers whose address type is any other type are 16 bits in TAL, but 32 bits in pTAL.

Similarly, if the data type of a nonpointer field in a SHARED2 and SHARED8 structure is an address type, its type must be SGBADDR, SGWADDR, EXTADDR, EXT32ADDR, EXT64ADDR, PROC32ADDR, or PROC64ADDR.

NOTE: The address types, EXT32ADDR, EXT64ADDR, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Example 75 REALIGNED Clause

```
STRUCT s FIELDALIGN(SHARED2);
BEGIN
    INT i; ! OK: i is a simple variable
    INT .j; ! ERROR: j's address type is WADDR
    BADDR b; ! ERROR: b's data type is BADDR
END;
```

Default Reference Alignment

If you do not specify a REALIGNED clause in a structure pointer declaration, the reference alignment for the pointer is the alignment of the structure that the pointer references in its declaration. In [Example 76 \(page 135\)](#), none of the pointers p1, p2, or p3 specifies an alignment. Their alignment, therefore, is the field alignment of the structures s1, s2, and s3 that they reference.

Example 76 Default Reference Alignment

```
STRUCT s1 FIELDALIGN(SHARED2);
BEGIN
    INT i;
    INT(32) j;
END;
STRUCT s2 FIELDALIGN(SHARED8);
BEGIN
    INT i;
    FILLER 2;
    INT(32) j;
END;
STRUCT s3 FIELDALIGN(AUTO);
BEGIN
    INT i;
    INT(32) j;
END;
INT .p1(s1); ! Reference alignment is 2
INT .p2(s2); ! Reference alignment is 8
INT .p3(s3); ! Reference alignment is 8
```

REALIGNED(2)

When a structure pointer specifies REALIGNED(2), the base of the structure might or might not be well-aligned for RISC and Itanium access. When you reference the pointer in an expression, pTAL generates conservative code that might not be as optimal as the code it generates when you specify REALIGNED(8).

When you use a structure pointer in an executable statement, the field to which the pointer refers might not be well-aligned. For example, if you are accessing a structure whose address was passed as a parameter to a procedure, you might not know whether the field is well-aligned. Although the fields of the structure are well-aligned from the base of the structure, the base of the structure might not be well-aligned in memory.

Similarly, if you reference a field in a structure that is stored at an arbitrary address on a heap, you might not know in advance whether the fields in the structure are well aligned.

To ensure good performance, use `REFALIGNED(2)` to access the field, even if it happens to be well-aligned. Always use `REFALIGNED(2)` unless you are certain that nearly all fields referenced by the pointer are well-aligned.

Example 77 `REFALIGNED(2)`

```
WADDR a_str;
STRUCT s_templ(*) FIELDALIGN(SHARED8);
BEGIN
  INT i;
  FILLER 2;
  INT(32) j;
END;
STRUCT s(s_templ);
PROC p(struct_addr, p1);
  WADDR struct_addr;
  INT .p1(s_templ);
  ! Use template for structure definition
BEGIN
  INT .p2(s);
  ! Reference compiler-allocated structure with
  ! SHARED8 alignment
  INT .p3(s_templ) REFALIGNED(2) = p2; ! Equivalence p3 to p2
  INT .p4(s_templ) REFALIGNED(2) := struct_addr;
  ! Use template but use address passed as parameter
  INT .p5(s_templ) REFALIGNED(2) := a_str;
  ! Use template but address stored in globals
  @p2 := @s;
  ! Ensure p2 is well-aligned
  a := p1.i;
  ! Might incur significant overhead if p1.i is not
  ! well-aligned. See REFALIGNED(8) (page 136)
  a := p2.i;
  ! Optimal code: p2 references s which is known to
  ! be well-aligned
  a := p3.i;
  ! Suboptimal access
  a := p4.i;
  ! Suboptimal access
  a := p5.i;
  ! Suboptimal access
END;
```

The field alignment of `s_templ` is `SHARED8`. Pointers `p1`, `p3`, `p4`, and `p5` use `s_templ` to define the layout of the structures they reference. `p2` uses the global definition structure `s` to define its layout.

The field alignment of `s` and `s_templ` is `SHARED8`. Because the declaration of `p1` does not specify a `REFALIGNED` clause, the statement `a := p1.i` might cause performance degradation. See [REFALIGNED\(8\) \(page 136\)](#). The pointers `p3`, `p4`, and `p5` specify `REFALIGNED(2)`. Compared to `p1`, references to `p3`, `p4`, and `p5` will have somewhat degraded performance when the fields they reference are well-aligned. When the fields they reference are not well-aligned, references to `p1` will have significantly degraded performance compared `p3`, `p4`, or `p5`.

REFALIGNED(8)

When the reference alignment specified for a structure pointer is 8, the code generated by pTAL for each reference to the pointer assumes that the base of the structure and the fields in the structure are well-aligned in memory. If the field alignment of a structure is `SHARED8`—or is declared `AUTO` and the program is compiled by pTAL to run on RISC and Itanium architecture—and the base of the structure is well-aligned, references to the pointer will execute with optimal performance in both pTAL and TAL.

If a structure pointer specifies `REFALIGNED(8)` or inherits its reference alignment from a `SHARED8` structure, but the base of the structure is not well-aligned, your program might run significantly slower than you anticipate. You will observe significantly degraded performance if your `REFALIGNED(8)` pointer references a structure field that is not, in fact, well-aligned. Each such reference in your program will cause a trap to the millicode exception handler, which accesses the field your program is referencing and then returns to your program. Your program's behavior is not affected by having to access the field from the exception handler except that its performance for each such trap is significantly degraded.

pTAL generates conservative code for references to a pointer that specifies `REFALIGNED(8)` if it detects that a trap would occur if it generated optimal code.

Example 78 REFALIGNED(8)

```
STRUCT t1 (*) FIELDALIGN(SHARED2);
BEGIN
    INT(32) i;
END;
STRUCT t2 (*) FIELDALIGN(SHARED8);
BEGIN
    STRUCT s (t1);
    INT(32) i;
END;
INT .EXT p1 (t1) REFALIGNED (8) := extended-address;
INT .EXT p2 (t2) REFALIGNED (2) := extended-address;
INT .EXT p3 (t2)                := extended-address;
INT(32) i32;
i32 := p1.i;
i32 := p2.i;
i32 := p3.s.i;
```

For the assignment `i32 := p1.i`, pTAL generates fast code to access the field described by `t1` because the declaration of pointer `p1` specifies `REFALIGNED(8)`. If the field is not well-aligned, your program will run significantly slower because each reference to elements of `p1` will trap to the millicode exception handler to resolve each memory access.

For the assignment `i32 := p2.i`, pTAL generates conservative code to access the field described by `t2` because the field might not be well-aligned. The compiler might generate extra instructions to access the field.

For the assignment `i32 := p3.s.i`, pTAL generates fast code to access the field because the declaration of `p3` does not include a `REFALIGNED` clause. The reference alignment therefore defaults to the field alignment of `t2`, which is `SHARED8`. Even though the layout of is based on `t2` (which, in turn, incorporates `t1`, which is `SHARED2`), the reference alignment of `p3` is 8 because `t2` is `SHARED8`. The access uses optimal code because, even though substructure `s` has `SHARED2` alignment, its containing structure has `SHARED8` alignment, and pTAL can determine that the offset of `p3.s.i` is well-aligned.

Code Generation for Structure References

When pTAL generates code for references to the fields of structures and substructures, it generates two kinds of code. These are referred to as:

- Fast code
- Conservative code

pTAL generates fast code if you reference fields in a structure compiled with `FIELDALIGN(SHARED8)`. It generates conservative code if you reference fields in a structure compiled with `FIELDALIGN(SHARED2)`.

STRUCTALIGN (MAXALIGN) Attribute

NOTE: Use this clause only with the EpTAL compiler. The pTAL compiler reports a syntax error.

The `STRUCTALIGN (MAXALIGN)` attribute applies only to template structures. If a template structure has this attribute:

- Each definition structure that uses the template structure is aligned on a 16-byte boundary.
- If this template is used within a `SHARED8` or `PLATFORM` structure, the enclosing structures are also aligned on 16-byte boundaries.
- If this template is used within a `SHARED8` structure, the EpTAL compiler warns you that this structure is not compatible with the same `SHARED8` structure on the TNS/R architecture.

Do not use STRUCT(MAXALIGN) within a SHARED2 structure.

VOLATILE Attribute

The VOLATILE attribute specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

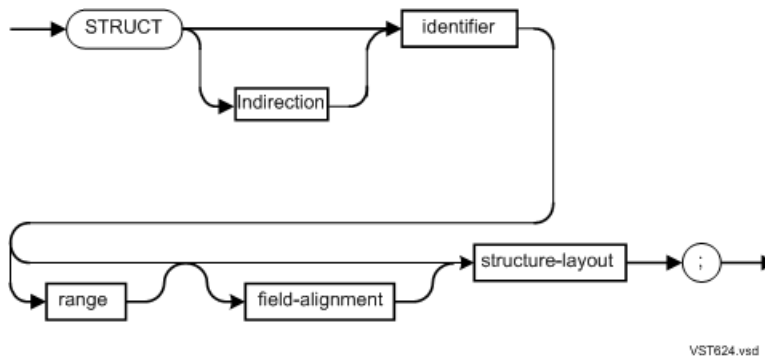
You can specify the VOLATILE attribute on any field except a substructure.

The syntax and semantics of VOLATILE fields and VOLATILE pointer fields is the same as those of VOLATILE variables and pointers, respectively.

Example 79 VOLATILE Attribute

```
STRUCT s;  
BEGIN  
    VOLATILE INT a;           ! A simple VOLATILE field  
    VOLATILE INT .EXT b;      ! A VOLATILE extended pointer  
    VOLATILE INT .c REFALIGNED(2); ! A VOLATILE standard pointer  
END;                          ! with a REFALIGNED clause
```

Declaring Definition Structures



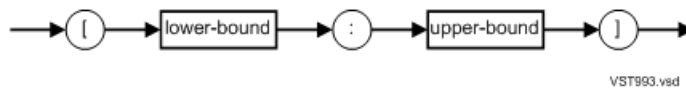
., Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the new referral structure.

range



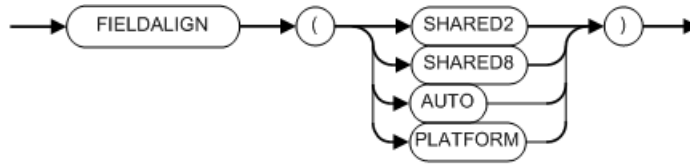
lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the first structure occurrence you want to allocate. Each occurrence is one copy of the structure.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the last structure occurrence you want to allocate. For a single-occurrence structure, omit both bounds or specify the same value for both bounds.

field-alignment



VST992.vsd

FIELDALIGN

specifies how you want the compiler to align the base of the structure and fields in the structure. The offsets of fields in a structure are aligned relative to the base of the structure.

If a definition substructure does not specify a FIELDALIGN clause, the contained substructure's field alignment is the field alignment of its encompassing structure or substructure.

If you do not specify a FIELDALIGN clause on a structure declaration, pTAL uses the current value of the FIELDALIGN compiler directive. The default value of the FIELDALIGN directive is AUTO.

If you specify a FIELDALIGN (NODEFAULT) compiler directive, you must specify a FIELDALIGN clause on every definition structure and template structure.

SHARED2

specifies that the base of the structure and each field in the structure must begin at an even-byte address except STRING fields.

SHARED8

specifies that the offset of each field in the structure from the base of the structure must begin at an address that is an integral multiple of the width of the field.

AUTO

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the AUTO attribute has in the native mode HP C compiler).

PLATFORM

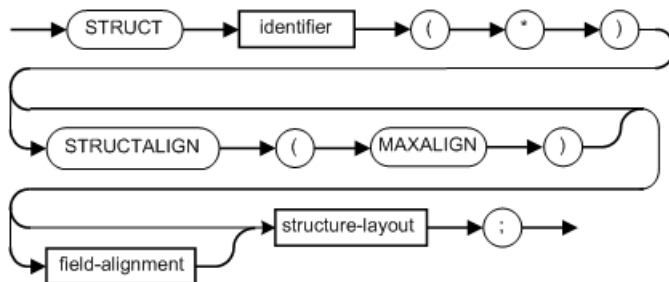
specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture.

structure-layout

is the identifier of a previously declared structure or structure pointer that provides the structure layout for this structure.

Declaring Template Structures

A template structure declaration describes a structure layout but allocates no space for it. You use the template layout in subsequent structure, substructure, or structure pointer declarations.



VST625.vsd

identifier

is the identifier of the template structure.

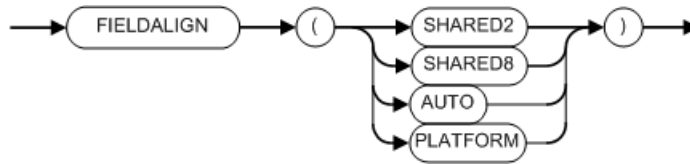
(*)

is the symbol for a template structure.

STRUCTALIGN (MAXALIGN)

causes each definition structure that uses this template to be aligned on a 16-byte boundary (for more information, see [STRUCTALIGN \(MAXALIGN\) Attribute \(page 137\)](#)).

field-alignment



FIELDALIGN

specifies how you want the compiler to align the base of the structure and fields in the structure. The offsets of fields in a structure are aligned relative to the base of the structure.

If a definition substructure does not specify a FIELDALIGN clause, the contained substructure's field alignment is the field alignment of its encompassing structure or substructure.

If you do not specify a FIELDALIGN clause on a structure declaration, pTAL uses the current value of the FIELDALIGN compiler directive. The default value of the FIELDALIGN directive is AUTO.

If you specify a FIELDALIGN (NODEFAULT) compiler directive, you must specify a FIELDALIGN clause on every definition structure and template structure.

SHARED2

specifies that the base of the structure and each field in the structure, except STRING fields, must begin at an even-byte address.

SHARED8

specifies that the offset of each field in the structure from the base of the structure must begin at an address that is an integral multiple of the width of the field.

AUTO

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the AUTO attribute has in the native mode HP C compiler).

PLATFORM

specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture.

A template structure has meaning only when you refer to it in the subsequent declaration of a referral structure, referral substructure, or structure pointer. The subsequent declaration allocates space for a structure whose layout is the same as the template layout.

The declaration in [Example 80 \(page 141\)](#) associates an identifier with a template structure layout but allocates no space for it.

Example 80 Template Structure Declaration

```
STRUCT inventory (*);    ! Template structure
BEGIN                  ! Structure layout
    INT item;
    FIXED(2) price;
    INT quantity;
END;
```

In [Example 81](#) (page 141):

- a and b are template structures. The compiler does not allocate space for them.
- a1 and b1 are definition structures, defined using the layouts of template structures a and b, respectively. The compiler allocates space for a1 and b1.
- STRUCTALIGN(MAXALIGN) in template structure a affects the alignment of definition structures a1 and b1 and causes a warning (see the comments in the code).

Example 81 Template Structure With STRUCTALIGN(MAXALIGN)

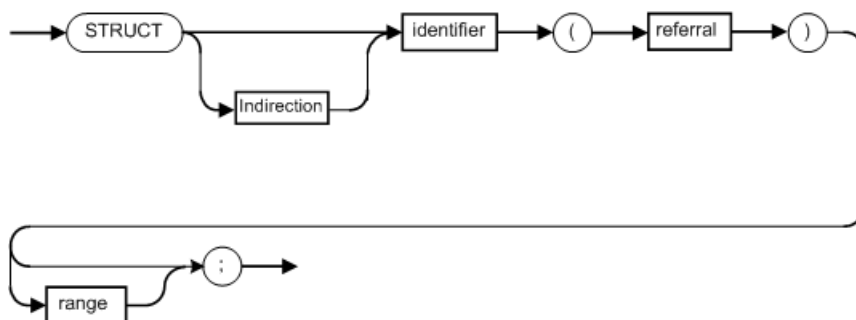
```
STRUCT A (*) STRUCTALIGN (MAXALIGN) FIELDALIGN (SHARED8);
BEGIN
    INT      I;      ! Located at byte-offset 0 as defined by SHARED8
    FILLER   2;
    INT(32) J;      ! Located at byte-offset 3 as defined by SHARED8
END;

STRUCT A1 (A); ! Base of A1 is guaranteed to be aligned on a
               ! 16-byte boundary
STRUCT B (*) FIELDALIGN (SHARED8);
BEGIN
    INT      K;
    FILLER   14;
    STRUCT A2 (A); ! Base of A2 is guaranteed to be aligned on
                   ! 16-byte boundary. Compiler issues warning
                   ! here because A is declared with STRUCTALIGN
                   ! (MAXALIGN) and B is a SHARED8 structure.
END;

STRUCT B1 (B); ! Base of B1 is guaranteed to be aligned on
               ! 16-byte boundary because the largest
               ! alignment of the components of B1 (A2) is
               ! 16 bytes.
```

Declaring Referral Structures

A referral structure declaration allocates storage for a structure whose layout is the same as the layout of a previously declared structure or structure pointer.



V5T025.vsd

Indirection

`., .EXT, .EXT32, .EXT64, .SG, and .SGX` are indirection symbols (see [Table 14 \(page 41\)](#)).

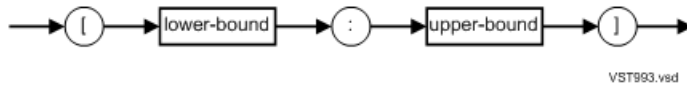
identifier

is the identifier of the new referral structure.

referral

is the identifier of a previously declared structure or structure pointer that provides the structure layout for this structure.

range



lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the first structure occurrence you want to allocate. Each occurrence is one copy of the structure.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the last structure occurrence you want to allocate. For a single-occurrence structure, omit both bounds or specify the same value for both bounds.

The compiler allocates storage for the referral structure based on the following characteristics:

- The addressing mode and number of occurrences specified in the new declaration
- The layout of the previous declaration

Structures declared in subprocedures must be directly addressed.

Structures always start on a word boundary.

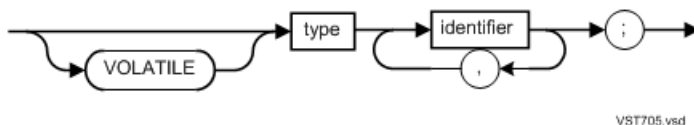
[Example 82 \(page 142\)](#) declares a template structure and a referral structure that references the template structure. The referral structure imposes its addressing mode and number of occurrences on the layout of the template structure.

Example 82 Referral Structure That References a Template Structure

```
STRUCT record (*);           ! Declare template structure
BEGIN
  STRING name[0:19];
  STRING addr[0:29];
  INT acct;
END;
STRUCT .customer (record) [1:50]; ! Declare referral structure
```

Declaring Simple Variables in Structures

The simple variable declaration associates a name with a single-element data item. When you declare a simple variable inside a structure, the form is:



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is any data type described in [Chapter 3 \(page 46\)](#).

identifier

is the identifier of the simple variable.

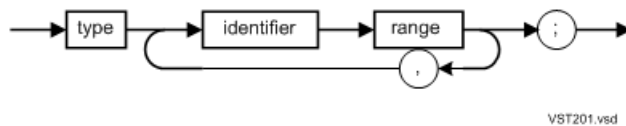
You cannot initialize a simple variable when you declare it inside a structure. You can subsequently assign a value to the simple variable by using an assignment statement.

Example 83 Simple Variables Within a Structure

```
STRUCT .inventory[0:49]; ! Declare definition structure
BEGIN
    INT      item;          ! Declare three simple variables
    FIXED(2) price;         ! within structure layout
    INT      quantity;
END;
```

Declaring Arrays in Structures

An array declaration associates an identifier with a collectively stored set of elements of the same data type. When you declare an array inside a structure, the form is:



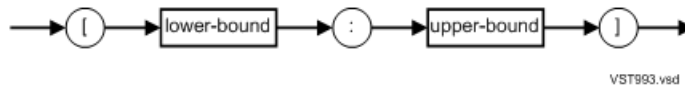
type

is any data type described in [Chapter 3 \(page 46\)](#).

identifier

is the identifier of the array.

range



lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth array element) of the first array element you want allocated. Both lower and upper bounds are required.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth array element) of the last array element you want allocated. Both lower and upper bounds are required.

When you declare arrays inside a structure, the following guidelines apply:

- You cannot initialize arrays declared in structures. You can assign values to such arrays only by using assignment statements.
- You cannot declare indirect arrays or read-only arrays in structures.
- You can specify array bounds of $[n : n-1]$ in structures (for example, $[6:5]$).

Such an array is called a zero-length array. It is often used to initialize a structure, as in [Example 85 \(page 144\)](#). This method of initialization allows you to name something with the same address as the next “thing” in the list without allocating data for it, similar to a union or equivalence.

Example 84 Arrays Within a Structure

```
STRUCT record;           ! Declare definition structure
BEGIN
  STRING name[0:19];      ! Declare arrays within the structure
  STRING addr[0:29];      ! layout
  INT acct;
END;
```

Example 85 Using a Zero-Length Array to Initialize a Structure

```
STRUCT s;
BEGIN
  STRING a[0:-1];         ! @a[0] is the same as @b
  INT b;
  STRUCT t;
  BEGIN
    ...
  END;
  ...
END;
s.a[0] := 0;
s.a[1] := s.a[0] for $LEN(s); ! Very efficient
...
```

Declaring Substructures

A substructure is a structure embedded within another structure or substructure. You can declare substructures that have the following characteristics:

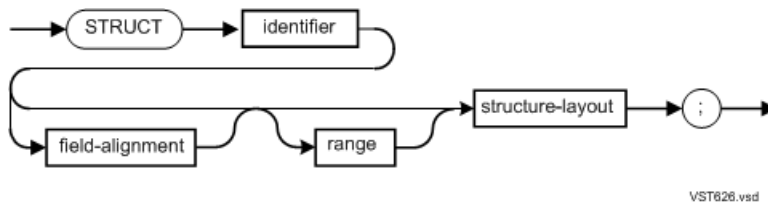
- Substructures must be directly addressed.
- Substructures have byte addresses, not word addresses.
- Substructures can be nested to a maximum of 64 levels.
- Substructures can have bounds of $[n : n-1]$ (for example, $[6:5]$).

Topics:

- [Definition Substructures \(page 144\)](#)
- [Referral Substructures \(page 146\)](#)

Definition Substructures

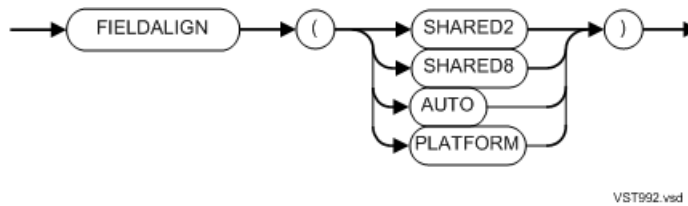
A definition substructure describes a layout and allocates storage for it.



identifier

is the identifier of the definition substructure.

field-alignment



FIELDALIGN

specifies how you want the compiler to align the base of the structure and fields in the structure. The offsets of fields in a structure are aligned relative to the base of the structure.

If a definition substructure does not specify a FIELDALIGN clause, the contained substructure's field alignment is the field alignment of its encompassing structure or substructure.

If you do not specify a FIELDALIGN clause on a structure declaration, pTAL uses the current value of the FIELDALIGN compiler directive. The default value of the FIELDALIGN directive is AUTO.

If you specify a FIELDALIGN (NODEFAULT) compiler directive, you must specify a FIELDALIGN clause on every definition structure and template structure.

SHARED2

specifies that the base of the structure and each field in the structure must begin at an even-byte address except STRING fields.

SHARED8

specifies that the offset of each field in the structure from the base of the structure must be begin at an address that is an integral multiple of the width of the field.

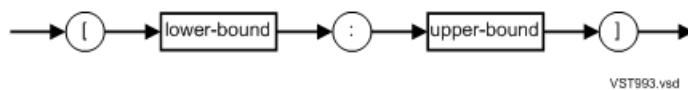
AUTO

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the AUTO attribute has in the native mode HP C compiler).

PLATFORM

specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture.

range



lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated. For a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

structure-layout

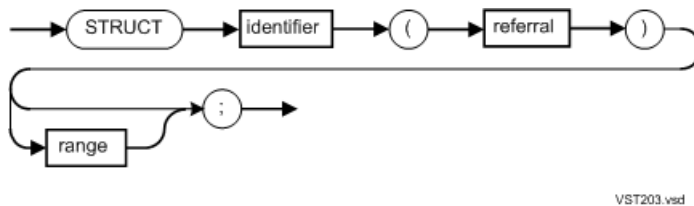
is the same BEGIN-END block as for structures. It can contain declarations for simple variables, arrays, substructures, filler bits, filler bytes, redefinitions, simple pointers, and structure pointers. The size of one substructure occurrence is the size of the layout, either in odd or even bytes. The total layout for one occurrence of the encompassing structure must not exceed 32,767 bytes.

Example 86 Declaring Definition Substructures

```
STRUCT .warehouse[0:1];      ! Two warehouses
BEGIN
  STRUCT inventory [0:49];    ! Definition substructure
  BEGIN                      ! 50 items in each warehouse
    INT item_number;
    FIXED(2) price;
    INT on_hand;
  END;
END;
```

Referral Substructures

A referral substructure allocates storage for a substructure whose layout is the same as the layout of a previously declared structure or structure pointer.



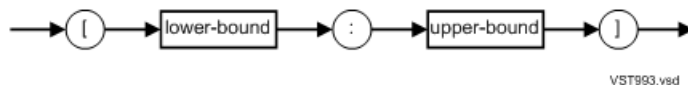
identifier

is the identifier of the referral substructure.

referral

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (except the encompassing structure) or structure pointer. If the previous structure has an odd-byte size, the compiler rounds the size of the new substructure up so that it has an even-byte size.

range



lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure.

upper-bound

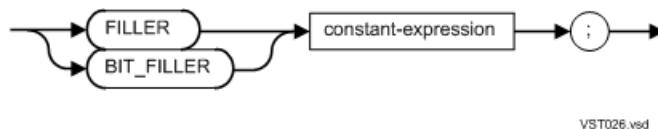
is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth occurrence) of the last substructure occurrence you want allocated. For a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

Example 87 Declaring a Referral Substructure

```
STRUCT temp(*);           ! Template structure --
BEGIN                     ! no space allocated
    STRING a[0:2];
    INT    b;
    STRING c;
END;
STRUCT .ind_struct;       ! Definition structure --
BEGIN                     ! space allocated
    INT    header[0:1];
    STRING abyte;
    STRUCT abc (temp) [0:1]; ! Declare referral substructure
END;                      ! Size of ind_struct.abc[0] is
                          ! 8 bytes
```

Declaring Filler

A filler declaration allocates a byte or bit place holder in a structure.



FILLER

allocates the specified number of byte place holders.

BIT_FILLER

allocates the specified number of bit place holders.

constant-expression

is a positive integer constant value that specifies a number of filler units in one of the following ranges:

FILLER	0 through 32,767 bytes
BIT_FILLER	0 through 255 bits

You can declare filler bits and filler bytes, but you cannot access such filler locations.

If the structure layout must match a structure layout defined in another program, your structure declaration need only include data items used by your program and can use filler bits or bytes for the unused space.

The compiler allocates space for each byte or bit you specify in a filler declaration. If the alignment of the next data item requires additional pad bytes or bits, the compiler allocates those also.

Example 88 Filler Byte Declarations

```
LITERAL last = 11;      ! Last occurrence
STRUCT .x[1:last];
BEGIN
  STRING byte[0:2];
  FILLER 1;              ! Document word-alignment pad byte
  INT word1;
  INT word2;
  INT(32) integer32;
  FILLER 30;            ! Place holder for unused space
END;
```

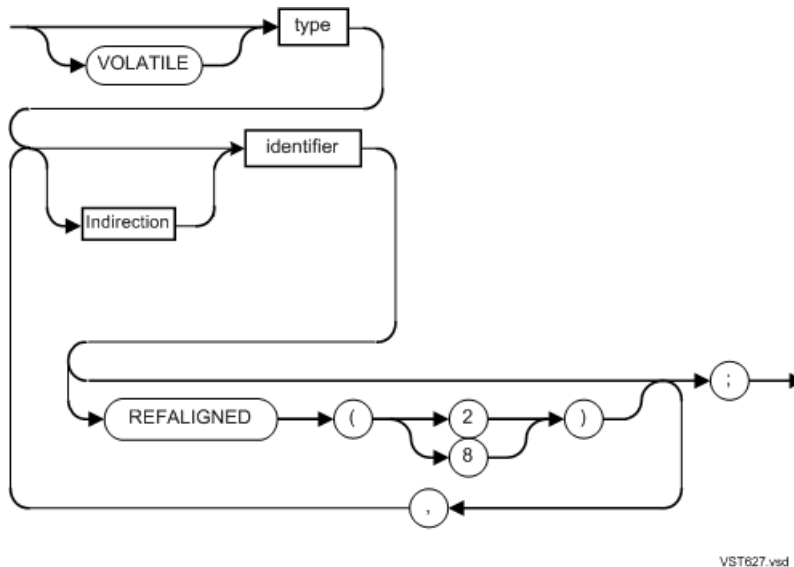
See also the filler byte example in [Definition Substructure \(page 155\)](#).

Example 89 Filler Bit Declaration

```
STRUCT .flags;
BEGIN
  UNSIGNED(1) flag1;
  UNSIGNED(1) flag2;
  UNSIGNED(2) state; ! State = 0, 1, 2, or 3
  BIT_FILLER 12;     ! Place holder for unused space
END;
```

Declaring Simple Pointers in Structures

A simple pointer is a variable that contains the memory address of a simple variable or an array. When you declare a simple pointer inside a structure, the form is:



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time—a byte, word, doubleword, or quadrupleword.

Indirection

`., .EXT, .EXT32, .EXT64, .SG, and .SGX` are indirection symbols (see [Table 14 \(page 41\)](#)).

NOTE: Indirection symbols, `.EXT32` and `.EXT64` are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

identifier

is the identifier of the simple pointer.

`REFALIGNED`

specifies the base alignment of the structures that the structure pointer will reference.

`2`

references a structure that might not be well-aligned.

`8`

indicates that the base of the structure and the fields in the structure are well aligned in memory

Example 90 Simple Pointers Within a Structure

```
STRUCT my_struct;  
BEGIN  
    FIXED      .      std_pointer;    ! Standard simple pointer  
    STRING     .EXT    ext_pointer;    ! Extended 32-bit simple pointer  
    STRING     .EXT32  ext32_pointer   ! Extended 32-bit simple pointer  
    INT        .EXT64  ext64_pointer   ! Extended 64-bit simple pointer  
END;
```

Topics:

- [Using Simple Pointers \(page 149\)](#)
- [Assigning Addresses to Pointers in Structures \(page 150\)](#)

Using Simple Pointers

The data type determines the size of data a simple pointer can access at a time.

Table 46 Data Accessed by Simple Pointers

Data Type	Accessed Data
STRING	Byte
INT	Word
INT(32)	Doubleword
REAL	Doubleword
REAL(64)	Quadrupleword
FIXED	Quadrupleword

The addressing mode and data type determine the kind of address the simple pointer can contain.

Table 47 Addresses in Simple Pointers

Addressing Mode	Data Type	Kind of Address
Standard	STRING	16-bit byte address
Standard	Any except STRING	16-bit word address
Extended	STRING	32-bit or 64-bit* byte address, normally in the automatic extended data segment

Table 47 Addresses in Simple Pointers *(continued)*

Addressing Mode	Data Type	Kind of Address
Extended	Any except STRING	32-bit or 64-bit* even-byte address, normally in the automatic extended data segment (if you specify an odd-byte address, results are undefined)

* 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Assigning Addresses to Pointers in Structures

You can assign to pointers the kinds of addresses listed in [Table 44 \(page 126\)](#) and [Table 45 \(page 129\)](#). To assign an address to a pointer within a structure, specify the fully qualified pointer identifier in an assignment statement. Prefix the structure identifier with @. For example, the assignment statement to assign an address to `ptr_x` declared in `substruct_a` in `struct_b` is:

```
@struct_b.substruct_a.ptr_x := arith_expression;
```

In the preceding example, @ applies to `ptr_x`, the most qualified item. On the left side of the assignment operator, @ changes the address contained in the pointer, not the value of the item to which the pointer points.

You can also prefix @ to a variable on the right side of the assignment operator. If the variable is a pointer, @ returns the address contained in the pointer. If the variable is not a pointer, @ returns the address of the variable itself.

Example 91 Assigning Addresses to Pointers in Structures

```
INT .array[0:99];
STRUCT .st;
BEGIN
  INT .std_ptr;
  INT .EXT ext_ptr;
  INT .EXT32 ext32_ptr;
  INT .EXT64 ext64_ptr;
END;
PROC e MAIN;
BEGIN
  @st.std_ptr := @array[0];
  @st.ext_ptr := $XADR(array[0]);
  @st.ext_ptr := $XADR32(array[0]);
  @st.ext32_ptr := @array[0];
  @st.ext32_ptr := $XADR(array[0]);
  @st.ext32_ptr := $XADR32(array[0]);
  @st.ext64_ptr := @array[0];
  @st.ext64_ptr := $XADR(array[0]);
  @st.ext64_ptr := $XADR32(array[0]);
  @st.ext64_ptr := $XADR64(array[0]);
END;
```

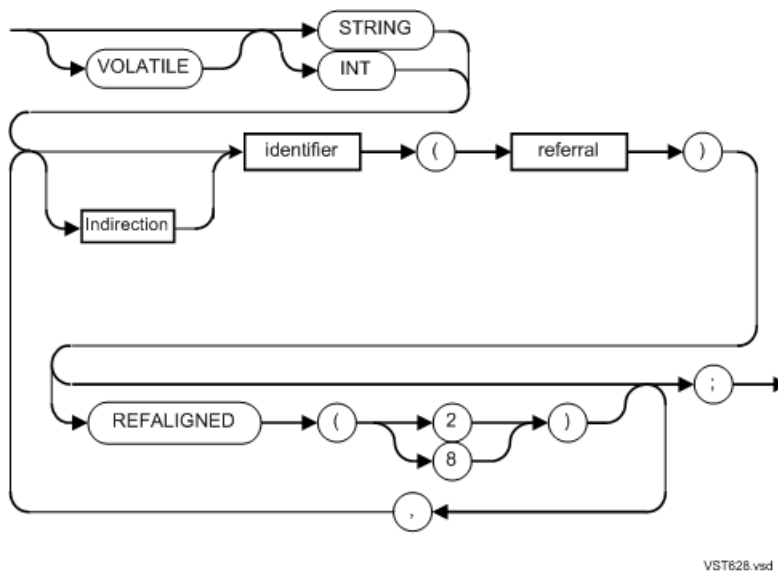
[Example 92 \(page 151\)](#) assigns the address of a structure to structure pointers declared in another structure.

Example 92 Assigning Addresses to Pointers in Structures

```
STRUCT .s1;
BEGIN
  INT var1;
  INT var2;
END;
STRUCT .s2;
BEGIN
  INT .std_ptr (s1);
  INT .EXT ext_ptr (s1);
  INT .EXT32 ext32_ptr (s1);
  INT .EXT64 ext64_ptr (s1);
END;
PROC g MAIN;
BEGIN
  @s2.std_ptr := @s1;
  @s2.ext_ptr := $XADR(s1);
  @s2.ext32_ptr := $WADDR_TO_EXTADDR(@s1);
  @s2.ext32_ptr := $XADR32(s1);
  @s2.ext64_ptr := $WADDR_TO_EXTADDR(@s1);
  @s2.ext64_ptr := $XADR32(s1);
  @s2.ext64_ptr := $XADR64(s1);
END;
```

Declaring Structure Pointers in Structures

A structure pointer is a variable that contains the address of a structure. When you declare a structure pointer inside a structure, the form is:



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

STRING

is the STRING attribute.

INT

is the INT attribute

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

NOTE: Indirection symbols, .EXT32 and .EXT64 are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

identifier

is the identifier of the structure pointer.

referral

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (including the encompassing structure) or structure pointer.

REFALIGNED

specifies the base alignment of the structures that the structure pointer will reference.

2

references a structure that might not be well-aligned.

8

indicates that the base of the structure and the fields in the structure are well aligned in memory

The addressing mode and STRING or INT attribute determine the kind of addresses a structure pointer can contain, as described in [Table 48 \(page 152\)](#).

Table 48 Addresses in Structure Pointers

Addressing Mode	STRING or INT Attribute	Kind of Address
Standard	STRING ¹	16-bit byte address of a substructure, STRING simple variable, or STRING array declared in a structure
Standard	INT ²	16-bit word address of any structure data item
Extended	STRING ¹	32-bit or 64-bit ³ byte address of any structure item located in any segment, normally the automatic extended data segment
Extended	INT ²	32-bit or 64-bit ³ byte address of any structure item located in any segment, normally the automatic extended data segment

¹ If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is BYTES.

² If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is WORDS.

³ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Example 93 Declaring a Structure Pointer Within a Structure

```
STRUCT struct_a;  
BEGIN  
    INT a;  
    INT b;  
END;  
STRUCT struct_b;  
BEGIN  
    INT .EXT struct_pointer (struct_a);  
    STRING a;  
END;
```

Declaring Redefinitions

A redefinition declares a new identifier and sometimes a new description for a previous item in the same structure.

The following rules apply to all redefinitions in structures:

- The new item must be of the same length or shorter than the previous item.
- The new item and the previous item must be at the same BEGIN-END level of a structure.

Additional rules are given in subsections that describe each kind of redefinition in the following topics:

- [Simple Variable \(page 153\)](#)
- [Array \(page 154\)](#)
- [Definition Substructure \(page 155\)](#)
- [Referral Substructure \(page 157\)](#)
- [Simple Pointer \(page 158\)](#)
- [Structure Pointer \(page 159\)](#)

For information about redefinitions outside structures, see [Chapter 11 \(page 177\)](#).

Simple Variable

A simple variable redefinition associates a new simple variable with a previous item at the same BEGIN-END level of a structure.



VST706.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is any data type except UNSIGNED.

identifier

is the identifier of the new simple variable.

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. You cannot specify an index with this identifier.

In a redefinition, the new item and the previous (nonpointer) item both must have a byte address or both must have a word address. If the previous item is a pointer, the data it points to must be word addressed or byte addressed to match the new item.

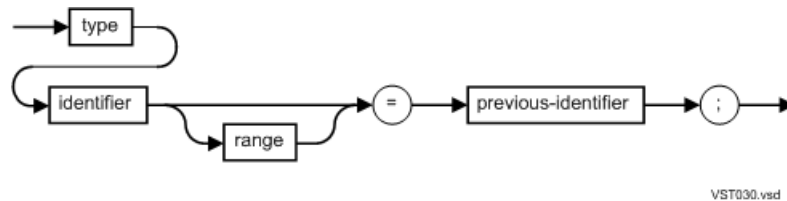
[Example 94 \(page 154\)](#) redefines the left byte of `int_var` as `string_var`.

Example 94 Simple Variable Redefinition

```
STRUCT .mystruct;  
BEGIN  
    INT int_var;  
    STRING string_var = int_var; ! Redefinition  
END;
```

Array

An array redefinition associates a new array with a previous item at the same BEGIN-END level of a structure.



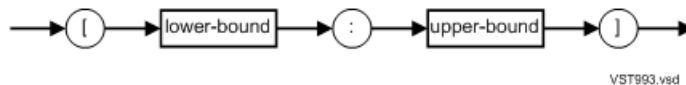
type

is any data type except UNSIGNED.

identifier

is the identifier of the new array.

range



lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated.

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. You cannot specify an index with this identifier.

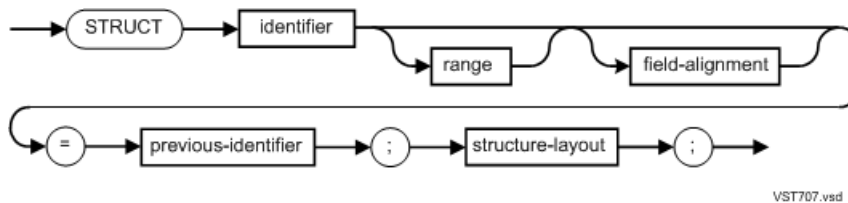
In a redefinition, the new item and the previous (nonpointer) item both must have a byte address or both must have a word address. If the previous item is a pointer, the data it points to must be word addressed or byte addressed to match the new item.

Example 95 Array Redefinition

```
STRUCT .s;  
BEGIN  
    INT    a[0:3];  
    INT(32) b[0:1] = a;    ! Redefine INT array as INT(32) array  
END;
```

Definition Substructure

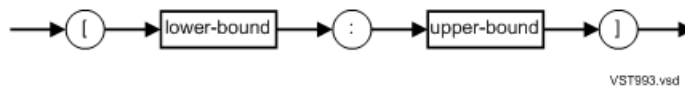
A definition substructure redefinition associates a new definition substructure with a previous item at the same BEGIN-END level of a structure.



identifier

is the identifier of the new substructure.

range



lower-bound

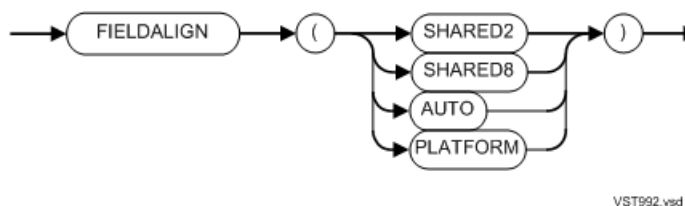
is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated.

To declare a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

field-alignment



FIELDALIGN

specifies how you want the compiler to align the base of the structure and fields in the structure. The offsets of fields in a structure are aligned relative to the base of the structure.

If a definition substructure does not specify a `FIELDALIGN` clause, the contained substructure's field alignment is the field alignment of its encompassing structure or substructure.

If you do not specify a `FIELDALIGN` clause on a structure declaration, pTAL uses the current value of the `FIELDALIGN` compiler directive. The default value of the `FIELDALIGN` directive is `AUTO`.

If you specify a `FIELDALIGN (NODEFAULT)` compiler directive, you must specify a `FIELDALIGN` clause on every definition structure and template structure.

`SHARED2`

specifies that the base of the structure and each field in the structure must begin at an even byte address except `STRING` fields.

`SHARED8`

specifies that the offset of each field in the structure from the base of the structure must be begin at an address that is an integral multiple of the width of the field.

`AUTO`

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the `AUTO` attribute has in the native mode HP C compiler).

`PLATFORM`

specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture.

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

structure-layout

is the same `BEGIN-END` block as for structures. It can contain declarations for simple variables, arrays, substructures, filler bits, filler bytes, redefinitions, simple pointers, and structure pointers. The size of one substructure occurrence is the size of the layout, either in odd or even bytes. The total layout for one occurrence of the encompassing structure must not exceed 32,767 bytes.

If the previous item is a substructure and you omit the bounds or if either bound is 0, the new substructure and the previous substructure occupy the same space and have the same offset from the beginning of the structure.

Example 96 Definition Substructure Redefinition

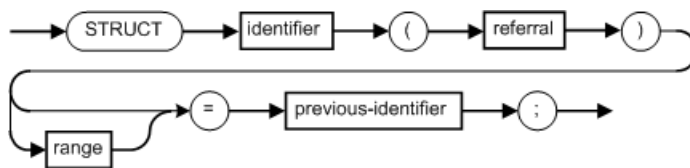
```
STRUCT a;  
BEGIN  
  STRING x;  
  STRUCT b;      ! b starts on odd byte  
  BEGIN  
    STRING y;  
  END;  
  STRUCT c = b;  ! Redefine b as c, also on odd byte  
  BEGIN  
    STRING z;  
  END;  
END;
```

Example 97 Definition Substructure Redefinition

```
STRUCT mystruct;  
BEGIN  
  STRUCT mysub1;  
  BEGIN  
    INT int_var;  
  END;  
  STRUCT mysub2 = mysub1;  ! Redefine mysub1 as mysub2  
  BEGIN  
    STRING string_var;  
  END;  
END;
```

Referral Substructure

A referral substructure redefinition associates a new referral substructure with a previous item at the same BEGIN-END level of a structure.



VST208.vsd

identifier

is the identifier of the new substructure.

referral

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (except the encompassing structure) or structure pointer. If the previous structure has an odd-byte size, the compiler rounds the size of the new substructure up so it has an even-byte size.

range



VST993.vsd

lower-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure.

upper-bound

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated.

To declare a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

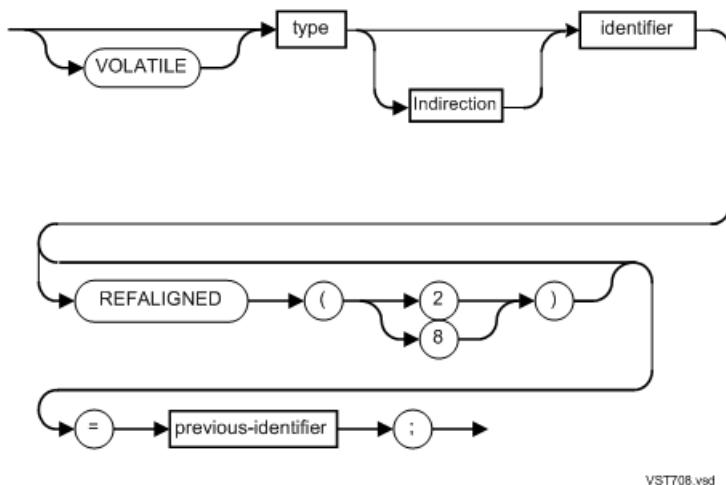
If the previous item is a substructure and you omit the bounds or if either bound is 0, the new substructure and the previous substructure occupy the same space and have the same offset from the beginning of the structure.

Example 98 Referral Substructure Redefinition

```
STRUCT temp(*);                                ! Template structure
BEGIN
  STRING a[0:2];
  INT    b;
  STRING c;
END;
STRUCT .ind_struct;                            ! Definition structure
BEGIN
  INT    header[0:1];
  STRING abyte;
  STRUCT abc (temp) [0:1];
  STRUCT xyz (temp) [0:1] = abc; ! Redefine abc as xyz
END;
```

Simple Pointer

A simple pointer redefinition associates a new simple pointer with a previous item at the same BEGIN-END level of a structure.



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time—a byte, word, doubleword, or quadrupleword.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the new simple pointer.

REFALIGNED

specifies the base alignment of the structures that the structure pointer will reference.

2

references a structure that might not be well-aligned.

8

indicates that the base of the structure and the fields in the structure are well aligned in memory

previous-identifier

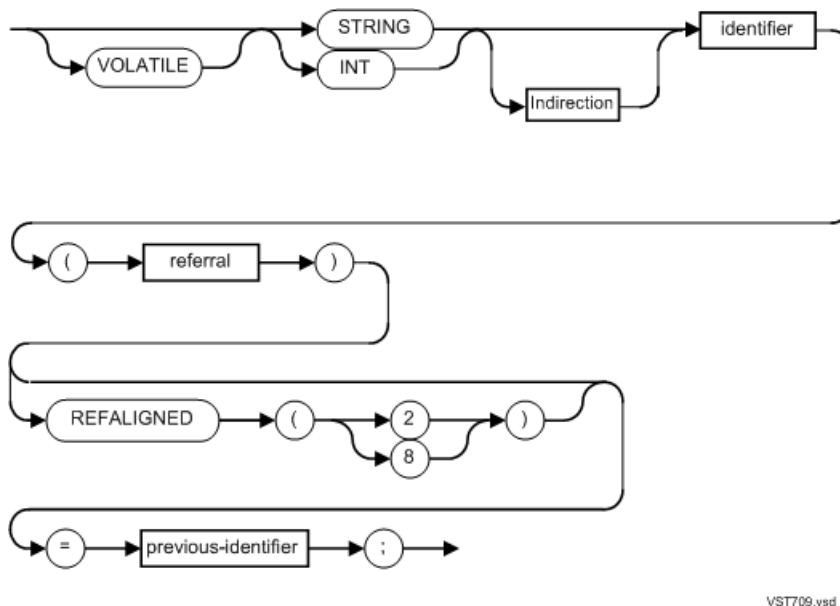
is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

Example 99 Simple Pointer Redefinition

```
STRUCT my_struct;  
BEGIN  
  STRING var[0:5];           ! Simple variable  
  STRING .EXT ext_pointer = var; ! Redefine var as simple  
                               ! pointer, ext_pointer  
END;
```

Structure Pointer

A structure pointer redefinition associates a new structure pointer with a previous item at the same BEGIN-END level of a structure.



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE

also causes that precise order of memory references to be preserved, again, when code is optimized.

STRING

is the STRING attribute.

INT

is the INT attribute.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the new structure pointer.

referral

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (including the encompassing structure) or structure pointer.

REFALIGNED

specifies the base alignment of the structures that the structure pointer will reference.

2

references a structure that might not be well-aligned.

8

indicates that the base of the structure and the fields in the structure are well aligned in memory

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

The addressing mode and STRING or INT attribute determine the kind of addresses a structure pointer can contain, as described in [Table 47 \(page 149\)](#).

Example 100 Structure Pointer Redefinition

```
STRUCT record;
BEGIN
    FIXED data;
    INT std_link_addr;
    INT .std_link (record) = std_link_addr;      ! Redefinition
    INT(32) ext_link_addr;
    INT .EXT ext_link (record) = ext_link_addr;  ! Redefinition
END;
```

10 Pointers

This section describes the syntax for declaring and initializing pointers you manage yourself. You can declare the following kinds of pointers:

- Simple pointer—a variable into which you store a memory address, usually of a simple variable or array, which you can access with this simple pointer.
- Structure pointer—a variable into which you store the memory address of a structure which you can access with this structure pointer.

The compiler allocates 32 bits for all pointers except .SG. In expressions involving addresses, however, the compiler treats all operands as if they were word addresses except extended addresses and addresses of strings. The pointer's object data type determines the pointer's address type and identifies the addressing type and location of data that your pointers will reference. For information about working with addresses, see [Chapter 5 \(page 69\)](#).

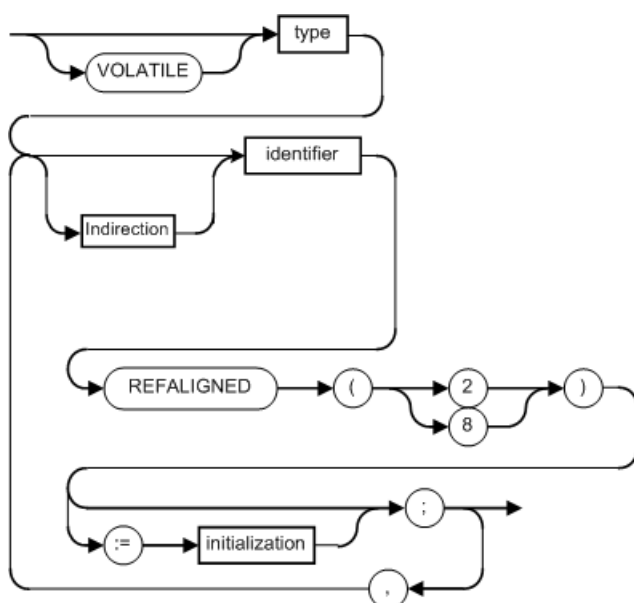
Some portions of this section describe how you reference data in system globals. System globals can be accessed only by programs running as privileged procedures.

Topics:

- [Overview of Pointer Declaration \(page 161\)](#)
- [Declaring VOLATILE Pointers \(page 163\)](#)
- [Address Types \(page 164\)](#)
- [Declaring Simple Pointers \(page 170\)](#)
- [Initializing Simple Pointers \(page 172\)](#)
- [REAL and REAL\(64\) Numeric \(page 62\)](#)
- [Initializing Structure Pointers \(page 174\)](#)
- [Declaring System Global Pointers \(page 176\)](#)

Overview of Pointer Declaration

This subsection gives you the general pointer syntax and explains the syntax elements.



VST676.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

Is one of the following data types depending on whether the pointer is a simple pointer or a structure pointer:

- BADDR
- CBADDR
- CWADDR
- EXTADDR
- EXT32ADDR
- EXT64ADDR
- FIXED [*fpoint*]
- INT
- INT (16)
- INT (32)
- INT (64)
- PROCADDR
- PROC32ADDR
- PROC64ADDR
- REAL
- REAL (32)
- REAL (64)
- SGBADDR
- SGWADDR
- SGXBADDR
- SGXWADDR
- STRING
- UNSIGNED (*width*)
- WADDR

fpoint

is the implied fixed point of the FIXED variable. *fpoint* can also be an asterisk (*) as in:

```
FIXED(*) .f;
```

width

is a constant expression specifying the width, in bits, of the variable.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the pointer.

REFALIGNED

Specifies the alignment of the variables or structures that *identifier* references.

2

specifies that the variables and structures *identifier* references are aligned as they would be aligned in TAL (and might not be well aligned in pTAL).

8

specifies that the variables and structures are well aligned for use in pTAL.

For nonstructure pointers, the default for REFALIGNED is the value you specify in the [REFALIGNED \(page 410\)](#).

initialization

is an expression representing a memory address. For more information about operations on addresses, see [Chapter 5 \(page 69\)](#).

Declaring VOLATILE Pointers

Declare pointers VOLATILE if they can be accessed asynchronously by other processes such as another process in your application or an I/O driver.

Topics:

- [Simple \(page 163\)](#)
- [Structure \(page 164\)](#)

Simple

When you declare a VOLATILE simple pointer, the value of the pointer and the data referenced by the pointer are treated as VOLATILE and are maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

Example 101 Declaring VOLATILE Simple Pointers

```
INT i;
INT a;
INT b[0:9];
VOLATILE INT .p1 := @a;
VOLATILE INT .p2 := @b;

i := p1;      ! pTAL treats pointer p1 and data p1 references,
              ! a, as volatile.  Program reads value of pointer
              ! p1 each time statement executes

i := a;      ! pTAL does not treat direct reference to a as
              ! volatile even though p1 still points to it,
              ! because a is not declared volatile

i := p2[a];   ! For each reference to p2[a], program:
              ! * Reads value of pointer p2 from memory
              ! * Adds value of a, which can be kept in a
              !   register because a is not volatile
              ! * Reads from memory the value referenced by
              !   p2[a]

i := p2[p1];  ! Data referenced by p2[p1] is the same as p2[a]
              ! in the preceding example, but both p1 and p2
```

```
! are volatile. Program reads from memory p1,  
! p2, a, and element of array b referenced by p1
```

Structure

When you declare a VOLATILE structure pointer, the compiler generates code that maintains the value of the pointer in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

You must specify the VOLATILE attribute on each field that you want to be volatile.

Example 102 Declaring VOLATILE Structure Pointers

```
INT i;  
STRUCT s;  
BEGIN  
    INT m;           ! Field m is never treated as volatile  
    VOLATILE INT n;  ! Field n is always treated as volatile  
END;  
  
VOLATILE INT .s1(s) := @s;  
            INT .s2(s) := @s;  
  
i := s1.m;        ! Value of pointer s1 is read from memory on  
                  ! every reference, but value of field s1.m  
                  ! might be maintained in a register  
  
i := s1.n;        ! Value of pointer s1 is read from memory on  
                  ! every reference, as is value of s1.n  
                  ! because field n specifies VOLATILE  
  
i := s2.n;        ! Value of pointer s2 might or might not be  
                  ! from memory, but having read the pointer,  
                  ! the field s2.n is always read from memory
```

Address Types

pTAL address types control the addresses you store into pointers. A 32-bit address can reference data anywhere in memory with optimal performance. The hardware does not require programs to specify an addressing type or memory storage area.

The compiler determines the address type of a pointer from the pointer declaration. You cannot explicitly declare a pointer's address type.

Address types are used primarily to describe the addresses that you assign to a pointer, not the data your program is processing.

Only operations that are meaningful for addresses are valid on address types.

A pointer is associated with two data types:

Data Type	Description
Object data type	Data type of the objects that the pointer can reference
Address data type	Data type of the addresses that you can store in the pointer

Table 49 Address Types

Data Type	Address Type	Target Data	Pointer Size	Example
BADDR	Byte	16-bit address to 1-byte-aligned data	32	STRING .s;
WADDR	Word	16-bit address to 2-byte-aligned data	32	INT .i;
CBADDR	Byte	16-bit address to 1-byte-aligned, read-only data	32	STRING s = 'P' := "A";
CWADDR	Word	16-bit address to 2-byte-aligned, read-only data	32	INT i = 'P' := 123;
SGBADDR	Byte	16-bit address to 1-byte-aligned, 'SG'-relative data	16	STRING .SG s;
SGWADDR	Word	16-bit address to 2-byte-aligned, 'SG'-relative data	16	INT .SG i;
SGXBADDR	Byte	32-bit address to 1-byte-aligned, 'SG'-relative data	32	STRING .SGX s;
SGXWADDR	Word	32-bit address to 2-byte-aligned, 'SG'-relative data	32	INT .SGX i;
EXTADDR	Byte	32-bit address to data	32	INT .EXT x;
EXT32ADDR*	Byte	32-bit address to data	32	INT .EXT32 x;
EXT64ADDR*	Byte	64-bit address to data	32	INT .EXT64 x;
PROCADDR	N.A.	Address denoting PROC Code	32	PROCPTR p; BEGIN END PROCPTR;
PROC32ADDR*	N.A.	32-bit address denoting PROC code	32	PROC32PTR p; BEGIN END PROCPTR;
PROC64ADDR*	N.A.	64-bit address denoting PROC code	64	PROC64PTR p; BEGIN END PROCPTR;

* These data types and indirection symbols, .EXT32 and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, ["64-bit Addressing Functionality"](#) (page 531).

You cannot explicitly declare or change the address type of a pointer. pTAL determines the address type based on the pointer declaration.

Every identifier you declare in a pTAL program has an object data type and an address type. [Table 50 \(page 166\)](#) lists the address type for all pTAL constructs except simple variables. The address type of a simple variable is the same as the address type of a pointer to data of the same object data type as the simple variable.

Example 103 Determining Address Types

```
INT .j; ! Pointer: address type is WADDR
INT i; ! Simple variable: address type is WADDR
```

Example 104

```
STRING .EXT s;
EXTADDR STRING INT .EXT i;
EXTADDR INT INT(32) .EXT j;
EXTDDR INT(32)REAL .EXT r;
EXTADDR REAL REAL(64) .EXT s;
EXTADDR REAL(64)FIXED .EXT f;
EXTADDR FIXED UNSIGNED(n) .EXT u;
EXTADDR UNSIGNED STRUCT .EXT t;
EXTADDR none SUBSTRUCT .EXT v;
EXTADDR none address_typeaddr-type .EXT a;
EXTADDR 2
```

Table 50 Object Data Types and Their Addresses

Declaration		Address Type	Object Data Type
STRING	.EXT s;	EXTADDR	STRING
INT	.EXT i;	EXTADDR	INT
INT(32)	.EXT j;	EXTADDR	INT(32)
REAL	.EXT r;	EXTADDR	REAL
REAL(64)	.EXT s;	EXTADDR	REAL(64)
FIXED	.EXT f;	EXTADDR	FIXED
STRUCT	.EXT t;	EXTADDR	none
SUBSTRUCT	.EXT v;	EXTADDR	none
<i>addr-type</i> ¹	.EXT a;	EXTADDR	<i>address_type</i> ²
STRING	.EXT32 s; ³	EXT32ADDR ³	STRING
INT	.EXT32 i; ³	EXT32ADDR ³	INT
INT(32)	.EXT32 j; ³	EXT32ADDR ³	INT(32)
REAL	.EXT32 r; ³	EXT32ADDR ³	REAL
REAL(64)	.EXT32 s; ³	EXT32ADDR ³	REAL(64)
FIXED	.EXT32 f; ³	EXT32ADDR ³	FIXED
STRUCT	.EXT32 t; ³	EXT32ADDR ³	none
SUBSTRUCT	.EXT32 v; ³	EXT32ADDR ³	none
<i>addr-type</i> ¹	.EXT32 a; ³	EXT32ADDR ³	<i>address_type</i> ²
STRING	.EXT64 s; ³	EXT64ADDR ³	STRING
INT	.EXT64 i; ³	EXT64ADDR ³	INT
INT(32)	.EXT64 j; ³	EXT64ADDR ³	INT(32)
REAL	.EXT64 r; ³	EXT64ADDR ³	REAL
REAL(64)	.EXT64 s; ³	EXT64ADDR ³	REAL(64)
FIXED	.EXT64 f; ³	EXT64ADDR ³	FIXED

Table 50 Object Data Types and Their Addresses *(continued)*

Declaration		Address Type	Object Data Type
STRUCT	.EXT64 t; ³	EXT64ADDR ³	none
SUBSTRUCT	.EXT64 v; ³	EXT64ADDR ³	none
<i>addr-type</i> ¹	.EXT64 a; ³	EXT64ADDR ³	<i>address_type</i> ²
PROCPTR p(); BEGIN END PROCPTR;	PROCADDR	PROCPTR	
PROC32PTR p(); BEGIN END PROC32PTR;	PROC32ADDR ³	PROC32PTR ³	
PROC64PTR p(); BEGIN END PROC64PTR;	PROC64ADDR ³	PROC64PTR ³	

¹ *addr-type* is any of the twelve address types: WADDR, BADDR, SGWADDR, SGBADDR, CWADDR, CBADDR, EXTADDR, EXT32ADDR, EXT64ADDR, PROCADDR, SGXWADDR, and SGXBADDR.

² *address_type* is the same address type as specified in the declaration.

³ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Topics:

- [BADDR and WADDR](#) (page 167)
- [SGBADDR, SGWADDR, SGXBADDR, and SGXWADDR \(System Globals\)](#) (page 167)
- [PROCADDR, PROC32ADDR, and PROC64ADDR \(Procedures, Procedure Pointers, and Procedure Entry Points\)](#) (page 168)
- [Subprocedures, Subprocedure Entry Points, Labels, and Read-Only Arrays \(CBADDR and CWADDR Address Types\)](#) (page 169)
- [EXTADDR, EXT32ADDR, and EXT64ADDR \(Extended Addresses\)](#) (page 169)

BADDR and WADDR

The address type of pointers is WADDR, except for STRING pointers, for which the address type is BADDR.

Example 105 BADDR and WADDR

```

INT    a;          ! Variable:      address type is WADDR
INT    .b;         ! Pointer:      address type is WADDR
STRING .c;         ! Pointer:      address type is BADDR
INT(32) d[0:9];    ! Direct array: address type is WADDR
INT    .e[0:9];    ! Indirect array: address type is WADDR

```

SGBADDR, SGWADDR, SGXBADDR, and SGXWADDR (System Globals)

The address type of a pointer to system global data is one of SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR. You can declare pointers to system global data by using either .SG notation or .SGX notation. In either case, the pointer is not in system globals, but the data is. pTAL allocates 16 bits for pointers you declare with .SG and 32 bits pointers declared with .SGX.

Example 106 SGBADDR, SGWADDR, SGXBADDR, and SGXWADDR

```
STRING .SG s; ! s is 16 bits
INT .SG i; ! i is 16 bits
STRING .SGX t; ! t is 32 bits
INT .SGX j; ! j is 32 bits
```

PROCADDR, PROC32ADDR, and PROC64ADDR (Procedures, Procedure Pointers, and Procedure Entry Points)

The address type of procedures, procedure pointers (PROCPTRs), and procedure entry points is PROCADDR.

Example 107 PROCADDR, PROC32ADDR, and PROC64ADDR

```
PROCADDR pa;
PROC32ADDR p32a;
PROC64ADDR p64a;
PROCPTR q( j ); INT j; END PROCPTR; ! @q is type PROCADDR
PROC32PTR r( j ); INT j; END PROC32PTR; ! @r is type PROC32ADDR
PROC64PTR s( j ); INT j; END PROC64PTR; ! @s is type PROC64ADDR
PROC64PTR t( j ); INT(32) j; END PROC64PTR; ! @t is type PROC64ADDR
PROC p( j ); ! @p is type PROCADDR
  INT j;
BEGIN
  ENTRY pl; ! @pl is type PROCADDR
pl:
  pa := @q;
  pa := @r;
  pa := @s; ! ERROR: can't implicitly convert from larger procedure
            ! address type to smaller procedure address type
  pa := @p;
  pa := @pl;

  p32a := @q;
  p32a := @r;
  p32a := @s; ! ERROR: can't implicitly convert larger procedure
              ! address type to smaller procedure address type
  p32a := @p;
  p32a := @pl;

  p64a := @q;
  p64a := @r;

  p64a := @s; ! OK
  p64a := @p;
  p64a := @pl;

  pa := p32a;
  p32a := pa;

  p64a := pa;
  p64a := p32a;

  pa := p64a; ! ERROR: can't implicitly convert from larger procedure
              ! address type to smaller procedure address type

  P32a := p64a; ! ERROR: can't implicitly convert from larger procedure
               ! address type to smaller procedure address type

  pa := $PROCADDR (p32a);
  pa := $PROCADDR (p64a); ! OK
  p32a := $PROCADDR (pa);
  p32a := $PROCADDR (p64a); ! OK
```



```

p64a := $PROCADDR (pa);
p64a := $PROCADDR (p32a);
END;

```

NOTE: PROC32ADDR, PROC64ADDR, PROC32PTR, and PROC64PTR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Subprocedures, Subprocedure Entry Points, Labels, and Read-Only Arrays (CBADDR and CWADDR Address Types)

The address type of a pointer to code in a user code segment—that is, a read-only array—is CWADDR if the pointer is type INT and is CBADDR if the pointer is type STRING. The address type of subprocedures, subprocedure entry points, and all labels—in both procedures and subprocedures—is CWADDR.

Example 108 CBADDR and CWADDR

```

INT    sa = 'P' := [1,2,3,4]; ! Address type of sa is CWADDR
STRING sb = 'P' := ["ABCD"];  ! Address type of sb is CBADDR
PROC p;
BEGIN
  LABEL lab1;
  SUBPROC subp1;
  BEGIN
    CWADDR cw;
    CBADDR cb;
    ENTRY ent1;
    ent1:
    lab2:
    cw := @subp1; ! Address type of @subp1 is CWADDR
    cw := @lab1;  ! Address type of @lab1  is CWADDR
    cw := @lab2;  ! Address type of @lab2  is CWADDR
    cw := @ent1;  ! Address type of @ent1  is CWADDR
    cw := @sa;    ! Address type of @sa    is CWADDR
    cb := @sb;    ! Address type of @sb    is CBADDR
  END;
  lab1:
END;

```

EXTADDR, EXT32ADDR, and EXT64ADDR (Extended Addresses)

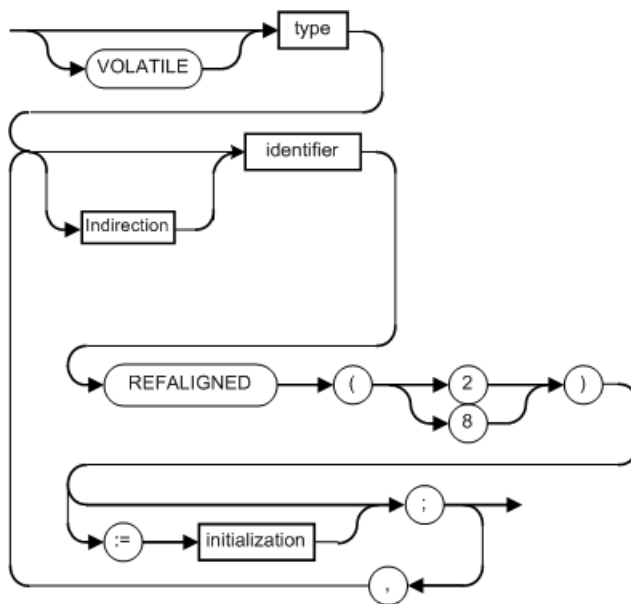
An EXTADDR is 32 bits. You can store the address of any of your processes' 32-bit addressable data in an EXTADDR pointer. An EXT64ADDR is 64 bits. You can store the address of any of your processes' data in an EXT64ADDR pointer.

Example 109 EXTADDR, EXT32ADDR, and EXT64ADDR Declarations

```
INT      .EXT i;  
STRING  .EXT s;  
INT      .EXT g = 'SG' + 0;  
REAL    .EXT r;  
INT      .EXT32 i32;  
STRING  .EXT32 s32;  
INT      .EXT32 g32 = 'SG' + 0;  
REAL    .EXT32 r32;  
INT      .EXT64 i64;  
STRING  .EXT64 s64;  
INT      .EXT64 g64 = 'SG' + 0;  
REAL    .EXT64 r64;
```

Declaring Simple Pointers

A simple pointer declaration associates an identifier with a memory location that contains the user-initialized address of a simple variable or array.



VST676.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is one of the following data types:

- BADDR
- CBADDR
- CWADDR
- EXTADDR
- EXT32ADDR
- EXT64ADDR

- `FIXED [(fpoint)]`
- `INT`
- `INT (16)`
- `INT (32)`
- `INT (64)`
- `PROCADDR`
- `PROC32ADDR`
- `PROC64ADDR`
- `REAL`
- `REAL (32)`
- `REAL`
- `REAL (32)`
- `REAL (64)`
- `SGBADDR`
- `SGWADDR`
- `SGXBADDR`
- `SGXWADDR`
- `STRING`
- `WADDR`

fpoint

The implied fixed point of the `FIXED` variable. *fpoint* can also be an asterisk (*) as in:

```
FIXED(*) .f;
```

Indirection

`., .EXT, .EXT32, .EXT64, .SG, and .SGX` are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the pointer.

REFALIGNED

For simple pointers, the default for **REFALIGNED** is the value you specify in the [REFALIGNED \(page 510\)](#).

2

specifies that the variables and structures that identifier references are aligned as they would be aligned in TAL (and might not be well-aligned in pTAL).

8

specifies that the variables and structures are well-aligned for use in pTAL (and in TAL, that they have more space).

For nonstructure pointers, the default for **REFALIGNED** is the value you specify in the [REFALIGNED \(page 510\)](#).

initialization

An expression representing a memory address. For more information about operations on addresses, see [Chapter 5 \(page 69\)](#).

The data type determines the size of data a simple pointer can access at a time.

The addressing mode and data type of the simple pointer determines the kind of address the pointer can contain.

For information about data types and addresses, see [Table 49 \(page 165\)](#) and [Table 50 \(page 166\)](#).

Furthermore, the kind of expression you can specify for the address depends on the level at which you declare the pointer:

- At the global level, use a constant expression.
- At the local or sublocal level, you can use any arithmetic expression.

Initializing Simple Pointers

You can initialize global standard pointers by using constant expressions such as:

Expression	Meaning
<code>@identifier</code>	Accesses address of variable
<code>@identifier '<<' 1</code>	If <code>@identifier</code> is a WADDR address, '<<' converts it to a BADDR address. If <code>@identifier</code> is a SGWADDR address, '<<' converts it to a SGBADDR address.
<code>@identifier '>>' 1</code>	If <code>@identifier</code> is a BADDR address, '>>' converts it to a WADDR address. If <code>@identifier</code> is a SGBADDR address, '>>' converts it to a SGWADDR address.
<code>@identifier [index]</code>	Accesses address of variable indicated by <i>index</i>
Built-in routine	Any that return a constant value, such as \$OFFSET

Expressions other than those in the preceding list can perform valid type conversions, but the compiler recognizes only those in the preceding list and might diagnose others as errors.

You can apply the @ operator to these global variables:

Variable	@identifier?
Direct array	Yes
Standard indirect array	Yes
Extended indirect array	No
Direct structure	Yes
Standard indirect structure	Yes
Extended indirect structure	No
Simple pointer	No
Structure pointer	No

Simple pointers receive their initial values when you compile the source code. Local or sublocal simple pointers receive their initial values at each activation of the encompassing procedure or subprocedure.

Example 110 Declaring But Not Initializing a Simple Pointer

```
INT(32) .ptr;
```

Example 111 Declaring and Initializing a Simple Pointer

```
STRING .bytes[0:3];           ! Indirect array
STRING .s_ptr := @bytes[3];    ! Simple pointer initialized with
                                ! address of bytes[3]
```

Example 112 Declaring and Initializing a STRING Simple Pointer

```
INT .a[0:39];                 ! INT array
STRING .ptr := @a[0] '<<' 1;   ! STRING simple pointer
                                ! initialized with byte address
                                ! of a[0]
```

Example 113 Declaring and Initializing Simple Pointers

```
INT a[0:1] := [%100000, %110000]; ! Array
INT .int_ptr1 := a[0];             ! Simple pointer
                                ! initialized with %100000
INT .int_ptr2 := a[1];             ! Simple pointer
                                ! initialized with %110000
```

Example 114 Declaring and Initializing a Simple Pointer, Using \$XADR

```
INT a[0:1];                    ! 16-bit word-addressed array
STRING .EXT s := $XADR (a[0]); ! Extended simple pointer
                                ! initialized with
                                ! 32-bit byte address of a[0]
```

Example 115 Declaring and Initializing an Extended Simple Pointer

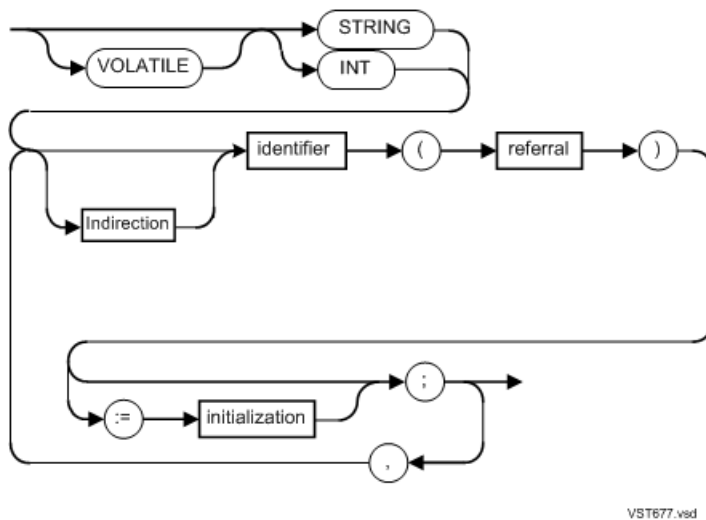
```
INT .EXT32 x32 [-100:100];      ! Array
INT .EXT32 x32_ptr := @x[-1];   ! Extended simple pointer initialized
                                ! 32-bit byte address if x32[-1];

INT .EXT64 x64 [-100:100];      ! Array
INT .EXT64 x64_ptr := @x[-1];   ! Extended simple pointer initialized
                                ! 64-bit byte address if x32[-1];
```

NOTE: The “Indirection Symbols” (page 41) .EXT32 and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Declaring Structure Pointers

The structure pointer declaration associates a previously declared structure with the memory location to which the structure pointer points. You access data in the associated structure by referencing the qualified structure pointer identifier.



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

STRING

is the STRING attribute.

INT

is the INT attribute, as INT, INT(32), or FIXED.

., .EXT, .EXT32, .EXT64, .SG, .SGX

are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the pointer.

referral

is the identifier of a previously-declared structure, structure template, or structure pointer.

Specify *referral* only for pointers to structures.

initialization

is an expression representing a memory address. For more information about operations on addresses, see [Chapter 5 \(page 69\)](#).

Initializing Structure Pointers

The addressing mode and data type of the simple pointer determines the kind of address the pointer can contain.

For information about data types and addresses, see [Table 49 \(page 165\)](#) and [Table 50 \(page 166\)](#).

Furthermore, the kind of expression you can specify for the address depends on the level at which you declare the pointer:

- Use a constant expression at the global level. See also [Initializing Simple Pointers \(page 172\)](#).
- At the local or sublocal level, you can use any arithmetic expression.

If the expression is the address of a structure with an index, the structure pointer points to a particular occurrence of the structure. If the expression is the address of an array, with or without an index, you impose the structure on top of the array.

Global structure pointers receive their initial values when you compile the source code. Local and sublocal structure pointers receive their initial values each time the procedure or subprocedure is activated.

Example 116 Declaring and Initializing a Structure Pointer, Using \$OFFSET

```
STRUCT t (*);    ! Template structure
BEGIN
  INT k;
END;
STRUCT .st;      ! Definition structure
BEGIN
  INT j;
  STRUCT ss (t);
END;
INT .ip := @st '+' $OFFSET (st.j) '>>' 1;    ! Simple pointer
INT .stp (t) := @st '+' $OFFSET (st.ss) '>>' 1;
! INT structure pointer
STRING .sstp (t) := @st '<<' 1 '+' $OFFSET (st.ss);
! STRING structure pointer
```

A standard STRING structure pointer can access only these structure items:

- Substructure
- STRING simple variable
- STRING array

The last declaration in [Example 116 \(page 175\)](#) shows a STRING structure pointer initialized with the converted byte address of a substructure.

[Example 117 \(page 175\)](#) shows another way to access a STRING item in a structure. You can convert the word address of the structure to a byte address when you initialize the STRING structure pointer and then access the STRING item in a statement.

Example 117 Declaring and Initializing a STRING Structure Pointer

```
STRUCT .astruct[0:1];
BEGIN
  STRING s1;
  STRING s2;
  STRING s3;
END;
STRING .ptr (astruct) := ! STRING ptr initialized with converted
  @astruct[1] '<<' 1;    ! byte address of astruct[1].
ptr.s2 := %4;           ! Access STRING structure item
```

Example 118 Declaring and Initializing a Local Structure Pointer

```
PROC my_proc MAIN;
BEGIN
  STRUCT my_struct[0:2];
  BEGIN
    INT array[0:7];
  END;
  INT .struct_ptr (my_struct) := @my_struct[1];
  ! Structure pointer contains address of my_struct[1]
END;
```

Example 119 Declaring and Initializing a Local STRING Structure Pointer

```
STRUCT name_def (*);
BEGIN
```

```

    STRING first[0:3];
    STRING last[0:3];
END;
STRUCT .record;
BEGIN
    STRUCT name (name_def); ! Declare substructure
    INT age;
END;
STRING .my_name (name_def) := @record.name; ! Structure pointer
                                           ! contains address
                                           ! of substructure

my_name ':= ' ["Sue Law"];

```

Example 120 Declaring and Initializing a Local STRING Structure Pointer

```

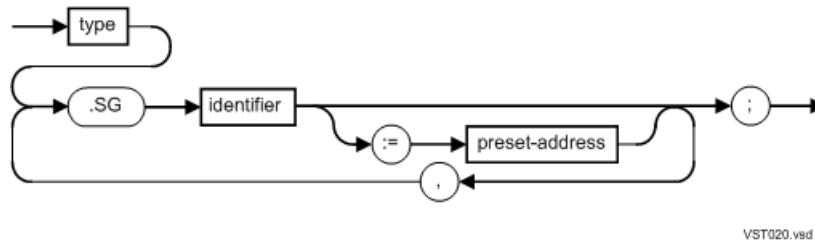
BEGIN
    INT array[0:7];
    STRUCT a_struct (*);
    BEGIN
        INT var;
        INT buffer1[0:3];
        STRING buffer2[0:4];
    END;
    INT .struct_ptr (a_struct) := @array; ! Structure pointer
                                           ! contains address of
                                           ! array
END;

```

Declaring System Global Pointers

NOTE: Only procedures that operate in privileged mode can access system global data.

The system global pointer declaration associates an identifier with a memory location at which you store the address of a variable located in the system global data area.



type

is any data type except UNSIGNED; specifies the data type of the value to which the pointer points.

.SG

is an indirection symbol (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of the pointer.

preset-address

is the address of a variable in the system global data area. The address is determined by you or the system during system generation.

Example 121 System Global Pointer Declaration

```

INT .SG newname;

```


11 Equivalenced Variables

Equivalencing lets you declare more than one identifier and description for a location in a storage area. Equivalenced variables that represent the same location can have different data types and byte-addressing and word-addressing attributes. For example, you can refer to an INT(32) variable as two separate words or four separate bytes.

You can equivalence any variable in the first column of [Table 51 \(page 177\)](#) to any variable in the second column.

Table 51 Equivalenced Variables

Equivalenced (New) Variable	Previous Variable
Simple variable	Simple variable
Simple pointer	Simple pointer
Structure	Structure
Structure pointer	Structure pointer
	Array
	Equivalenced variable

You can use an equivalenced variable in an expression anywhere an operand is valid.

Table 52 Equivalenced Variable Terminology

Term	Definition
Equivalenced variable	The identifier that appears on the left side of an equivalenced declaration; for example: <pre>INT previous; INT equivalenced = previous;</pre>
Previous variable	The identifier that appears on the right side of the equivalenced declaration. The previous variable can, itself, be an equivalenced variable; for example: <pre>INT base_previous; INT equivalenced1 = base_previous; INT equivalenced2 = equivalenced1;</pre>
Direct equivalent declaration	The equivalenced variable is a simple variable, direct array, direct structure, standard pointer (including a standard structure pointer), or extended pointer (including an extended structure pointer). Direct items can be equivalenced only to other direct items (with two exceptions).
Indirect equivalent declaration	The equivalenced variable is a standard indirect array or standard indirect structure. Standard indirect items can be equivalenced only to other standard indirect items.
Extended equivalent declaration	The equivalenced variable is an extended indirect array or extended indirect structure. Extended indirect items can be equivalenced only to other extended indirect items.
Standard pointer equivalent declaration	The equivalenced variable is a pointer to data.
Extended pointer equivalent declaration	The equivalenced variable is a pointer to data in an EXTADDR.

Topics:

- [Declaring Equivalenced Variables \(page 178\)](#)
- [Memory Allocation \(page 179\)](#)
- [Declaring Nonstructure Equivalenced Variables \(page 180\)](#)
- [Equivalencing Procedure Addresses \(PROCADDR, PROC32ADDR, and PROC64ADDR\) and Pointer Variables \(PROCPTR, PROC32PTR, and PROC64PTR\) \(page 187\)](#)

- [Declaring Equivalenced Definition Structures \(page 188\)](#)
- [System Global Equivalenced Variable Declarations \(page 193\)](#)

Declaring Equivalenced Variables

Table 53 Valid Equivalenced Variable Declarations

Equivalenced Variable Category	Equivalenced Variable	Variable Example	Previous Variable Category
Direct	Simple Variable	<code>INT i;</code>	Direct or Pointer
	Direct array	<code>INT i[0:3];</code>	
	Direct structure	<code>STRUCT s; BEGIN INT i; END;</code>	
Indirect	Indirect array	<code>INT .a[0:3];</code>	Indirect
	Indirect structure	<code>STRUCT s; BEGIN INT i; END;</code>	
Extended	Extended array	<code>INT .EXT a[0:3];</code>	EXTADDR
		<code>INT .EXT32 b[0:3];¹</code>	EXT32ADDR ¹
		<code>INT .EXT64 c[0:3];¹</code>	EXT64ADDR ¹
	Extended structure	<code>STRUCT .EXT s; BEGIN INT i; END;</code>	EXTADDR
		<code>STRUCT .EXT32 t; BEGIN INT I; END;</code>	EXT32ADDR ¹
		<code>STRUCT .EXT64 u; BEGIN INT I; END;</code>	EXT64ADDR ¹
Standard Pointer	Standard pointer	<code>INT .p;</code>	A pointer, simple variable, indirect array, or indirect structure
	Standard structure pointer	<code>INT .s(t);</code>	
Extended Pointer	Extended pointer	<code>INT .EXT e;</code>	Direct or Extended with the same address type (EXTADDR)
		<code>INT .EXT32 f;¹</code>	EXT32ADDR ¹
		<code>INT .EXT64 g;¹</code>	EXT64ADDR ¹
	Extended structure pointer	<code>INT .EXT s(t);</code>	
		<code>INT .EXT32 u(t);¹</code>	
		<code>INT .EXT64 v(t);¹</code>	

¹ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

You can index a variable that participates in an equivalenced declaration either as the equivalenced variable or as the previous variable even if none of the variables in the equivalenced group specify array bounds.

Example 122 Declaring Equivalenced Variables

```
INT a;          ! a is a simple variable, and cannot be indexed
INT b;          ! "Previous variable" for the next decl
INT c = b;      ! b and c can be indexed
INT d = c;      ! c and d can be indexed
! Variables b, c, and d can be indexed because each appears in
! an equivalenced declaration:
c[2] := b[1];   ! OK: b and c appear in equivalenced declaration
d[2] := c[1];   ! OK: c and d appear in equivalenced declaration
d[2] := a;      ! OK: d appears in equivalenced declaration
               ! and can be indexed.
a[1] := d;      ! ERROR: a cannot be indexed
```

Memory Allocation

pTAL does not allocate memory for equivalenced variables. In the following example, pTAL allocates memory only for `base_previous`:

```
INT base_previous;
INT equivalenced1 = base_previous;
INT equivalenced2 = equivalenced1;
```

An equivalenced variable is an alias for memory allocated for a previously declared variable. The equivalenced declaration can specify different attributes; for example, a different data type than those of the previous variable. In the following example, pTAL allocates 32 bits for `i`. The equivalenced declaration for `j` references the memory allocated for `i`, but specifies that the bits be treated as a REAL number:

```
INT(32) i;
REAL    j = i;
```

If an equivalenced variable is a standard or extended pointer and the previous variable is the implicit pointer of an indirect array or indirect structure, the equivalenced variable is a read-only pointer. You can use the value of the pointer in an expression, but you cannot store an address or other value into the pointer because doing so would be the same as storing an address into the implicit pointer of the array or structure. You can, however, use a pointer to read or write the data to which the pointer points.

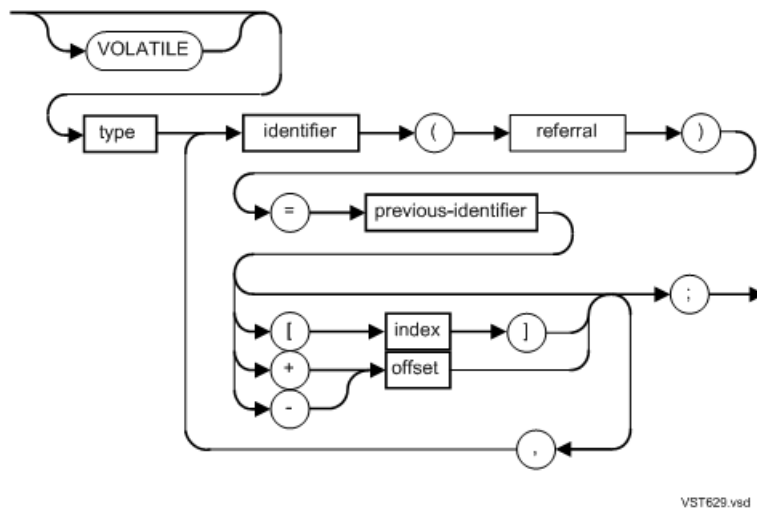
Example 123 Equivalenced Pointers

```
INT .a[0:3];
INT .p = a;
WADDR w;
PROC p1;
BEGIN
    a[1] := a[1] + 1; ! Increment second word of a
    p[1] := p[1] + 1; ! Increment second word of p (= a)
    w := @a;         ! Assign address of first word of a to w
    w := @p;         ! Assign address of first word of p (= a)
                   ! to w
END;
PROC p2;
BEGIN
    @p := w; ! ERROR: Cannot assign to an equivalenced pointer
             ! when it is equivalenced to an indirect or
```

```
! standard indirect variable
END;
```

Declaring Nonstructure Equivalenced Variables

Nonstructure equivalenced declarations include simple variables, pointers, and arrays.



VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a *VOLATILE* data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, *VOLATILE* also causes that precise order of memory references to be preserved, again, when code is optimized.

type

If *referral* is present, must be *STRING* or *INT*; otherwise, *type* is any data type except *UNSIGNED*.

identifier

is the identifier of the equivalenced variable to be made equivalent to *previous-identifier*.

referral

is the identifier of a previously declared structure, structure layout, or structure pointer.

previous-identifier

the identifier of a previously-declared variable, direct array element, pointer, structure, structure pointer, or equivalenced variable.

index

is an *INT* constant that specifies an element offset from *previous-identifier* to which the equivalenced pointer or variable refers. Specify *index* only with direct variables. *index* must end on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-identifier*, where the equivalenced variable is placed. For example, if *a* and *b* are declared:

```
INT a[0:9];
INT b = a+5
```

then *b* is placed at *a[5]*.

offset

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the *offset* is from the location of the pointer, not from the location of the data pointed to.

The following are valid equivalenced declarations:

```
INT a;
INT b = a;
INT(32) c[0:3];
INT d[0:7] = c;
```

Topics:

- [Memory Usage for Nonstructured Equivalenced Variables \(page 181\)](#)
- [Equivalenced Arrays \(page 181\)](#)
- [Indirect Arrays \(page 182\)](#)
- [Equivalenced Simple Variables \(page 182\)](#)
- [Equivalenced Simple Pointers \(page 183\)](#)

Memory Usage for Nonstructured Equivalenced Variables

The memory referenced by an equivalenced variable including all fields of an equivalenced structure and all elements of an equivalenced array must be contained entirely within the memory allocated for the previous variable. You can index the previous variable, but the memory referenced after applying the index must be contained within the memory allocated for the previous variable.

An equivalenced variable, including all elements of an equivalenced array or equivalenced structure, must be the same size or smaller than the lowest-level previous variable, even if an intermediate previous variable is not as the equivalenced variable you are declaring:

```
INT      h;
FIXED    i;

INT      j = i;  ! OK: j is smaller than I
INT(32)  k = j;  ! OK: k is 32 bits, i is 64 bits
FIXED    l = h;  ! ERROR: l > h
```

The number of bits in an equivalenced variable (including all elements of an array or structure) must be less than or equal to the number of bits in the previous variable. Equivalenced variables for which the previous variable is itself an equivalenced variable, must be contained entirely within the memory allocated for the previous variable for which the compiler allocates memory.

Example 124 Memory Usage for Nonstructured Equivalenced Variables

```
FIXED    i;                ! i is 64 bits
INT(32)  j[0:1] = i;       ! OK: j is 64 bits and coincident with i
INT      k[0:1] = i;       ! OK: k is 32 bits and contained within i
INT      m[0:3] = k;       ! OK: Although m is 64 bits and k is
                           ! 32 bits, pTAL requires only that
                           ! m be contained within i, not k.

INT      x[0:15];
FIXED    y = x[10];        ! ERROR: y does not fit entirely within x
```

Equivalenced Arrays

Use the *lower-bnd1* and *upper-bnd2* parameters as shown in the nonstructure declaration syntax.

Indirect Arrays

Figure 8 (page 182) shows how pTAL implements indirect arrays. The compiler allocates storage for the four elements of the array *a*, but not for a pointer to *a*. References to *a* access the data directly not indirectly through a pointer.

Figure 8 Indirect Array

pTAL Indirect Array

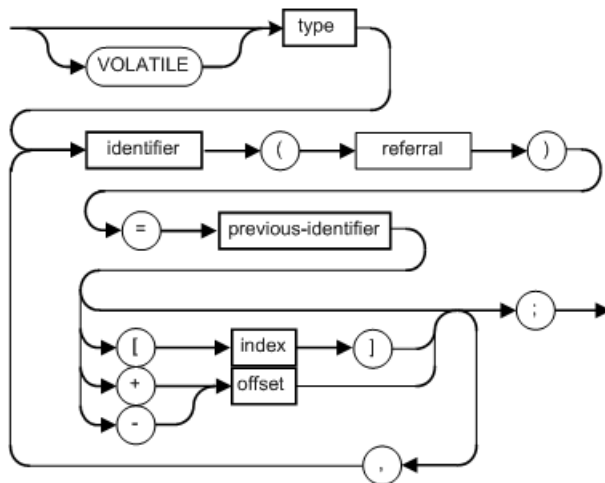
```
INT .A[0:3] := [10,20,30,40];  
! Object data type: INT  
! Address type: WADDR
```

A: 1000	10
1001	20
1002	30
1003	40

VST121.vsd

Equivalenced Simple Variables

An equivalenced simple variable declaration associates a new simple variable with a previously declared variable.



VST004.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

If *referral* is present, *type* must be STRING or INT; otherwise, *type* is any data type except UNSIGNED.

identifier

is the identifier of the simple equivalenced variable to be made equivalent to *previous-identifier*.

previous-identifier

is the identifier of a previously declared simple variable.

index

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT(32) a[0:9];
INT b = a+6
```

then b is placed in the first six bits of a.

offset

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size.

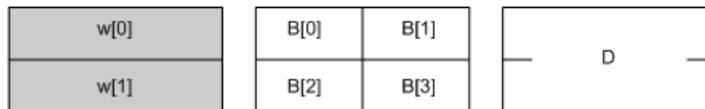
The location represented by *index* must begin on a word boundary.

Equivalencing a simple variable to an indirect array or structure is not recommended. If you do so, the simple variable is made equivalent to the location of the implicit pointer, not the location of the data pointed to.

In [Figure 9 \(page 183\)](#), a STRING variable and an INT(32) variable are equivalenced to an INT array.

Figure 9 Equivalenced Simple Variables

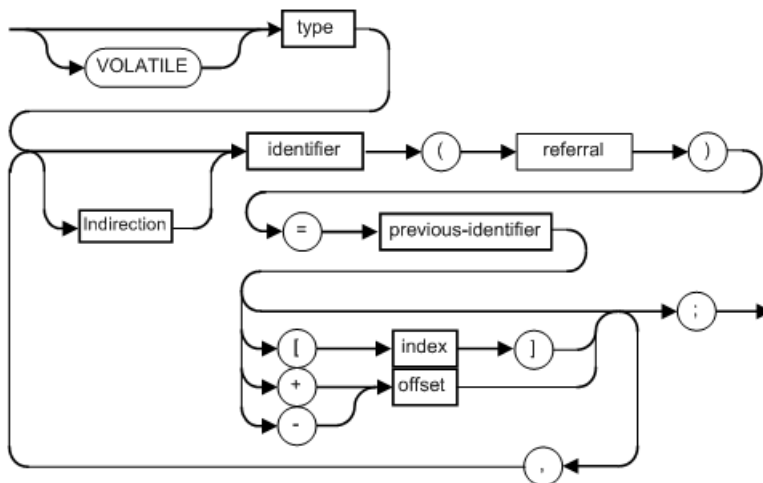
```
INT w[0:1];
STRING b = w[0];
INT(32) d = b;
```



VST302.vsd

Equivalenced Simple Pointers

An equivalenced simple pointer declaration associates a new simple pointer with a previously declared variable.



VST630.vsd

VOLATILE

specifies that the value of this variable must be maintained in memory, not in a register. Each reference to a VOLATILE data item causes the data item to be read or written to memory even when code is optimized. Based on the order of reads and writes in the source code, VOLATILE also causes that precise order of memory references to be preserved, again, when code is optimized.

type

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time (byte, word, doubleword, or quadrupleword).

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

NOTE: Indirection symbols, .EXT32 and .EXT64 are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

identifier

is the identifier of a simple pointer to be made equivalent to *previous-identifier*.

previous-identifier

is the identifier of a previously-declared variable, direct array element, pointer, structure, structure pointer, or equivalenced variable.

index

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT(32) a[0:9];  
INT b = a+6
```

then b is placed in the first six bits of a.

offset

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

Topics:

- [Using Equivalenced Simple Pointers \(page 184\)](#)
- [REALIGNED Clause for Equivalenced Simple Pointers \(page 187\)](#)

Using Equivalenced Simple Pointers

If the previous variable is a pointer, an indirect array, or an indirect structure, the previous pointer and the new pointer must both contain either:

- A standard byte address
- A standard word address
- An extended address

Otherwise, the pointers will point to different locations, even if they both contain the same value. That is, a standard STRING or extended pointer normally points to a byte address, and a standard pointer of any other data type normally points to a word address.

You can equivalence standard pointers to indirect arrays and indirect structures, but you can only read the value of the pointer. You cannot store an address into the pointer. You can, however, read or write the data to which the pointer points.

You can equivalence extended pointers to extended arrays and extended structures, but you can only read the value of the pointer. You cannot store an address into the pointer. You can, however, read or write the data to which the pointer points.

You can equivalence a standard pointer to an indirect array or indirect structure but you cannot equivalence an indirect array or indirect structure to a standard pointer. A pointer equivalenced to an indirect item is a read-only pointer—you can read the address in the pointer, but you cannot store an address into the pointer.

Example 125 Read-Only Pointer

```
INT .a[0:3];      ! Indirect array
INT .b;           ! Standard pointer
INT .c = a;       ! OK: Equivalence a pointer to an indirect
                  ! array (c is read-only)
INT .d[0:3] = b;  ! ERROR: Cannot equivalence an indirect item
                  ! to a pointer
@c := @c + 1;     ! ERROR: Cannot modify a pointer that is
                  ! equivalenced to an indirect item
```

When you declare indirect and extended pointers in equivalenced declarations:

- The address type of a STRING standard pointer is BADDR. The address type of all other standard pointers is WADDR.
- The address type of extended pointers is always EXTADDR, regardless of the data type of the objects to which the pointer will refer.

Figure 10 (page 185) shows two examples of the data types associated with pointers. Figure 10 (page 185) shows the object data type and address type. Use Table 54 (page 185) to determine valid equivalenced declarations.

Figure 10 The Object and Address Types of a Pointer

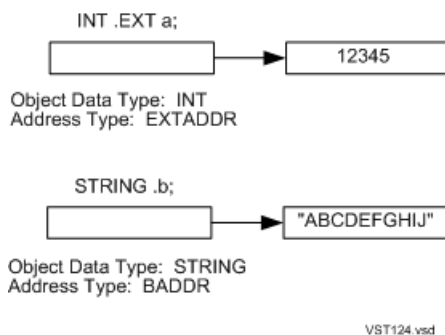


Table 54 Data Types for Equivalenced Variables

Example	Object Data Type	Address Type
INT a;	INT	WADDR
INT .b;	INT	WADDR
INT .EXT c;	INT	EXTADDR
INT .EXT32 e; ¹	INT	EXT32ADDR ¹
INT .EXT64 f; ¹	INT	EXT64ADDR ¹
BADDR a;	BADDR	WADDR

Table 54 Data Types for Equivalenced Variables *(continued)*

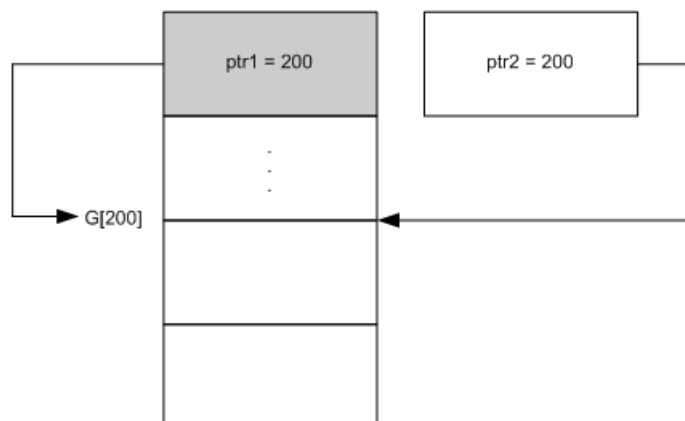
Example	Object Data Type	Address Type
BADDR .b;	BADDR	WADDR
BADDR .EXT c;	BADDR	EXTADDR
BADDR EXT32 d; ¹	BADDR	EXT32ADDR ¹
BADDR EXT64 .e; ¹	BADDR	EXT64ADDR ¹
EXTADDR a;	EXTADDR	WADDR
EXTADDR .b;	EXTADDR	WADDR
EXTADDR .EXT c;	EXTADDR	EXTADDR
EXTADDR .EXT32 d; ¹	EXTADDR	EXT32ADDR ¹
EXTADDR .EXT64 e; ¹	EXTADDR	EXT64ADDR ¹
EXT32ADDR .EXT f;	EXT32ADDR ¹	EXTADDR
EXT32ADDR .EXT32 g; ¹	EXT32ADDR ¹	EXT32ADDR ¹
EXT32ADDR .EXT64 h; ¹	EXT32ADDR ¹	EXT64ADDR ¹
EXT64ADDR .EXT i;	EXT64ADDR	EXTADDR
EXT64ADDR .EXT32 j; ¹	EXT64ADDR ¹	EXT32ADDR ¹
EXT64ADDR .EXT64 k; ¹	EXT64ADDR ¹	EXT64ADDR ¹
STRING a;	STRING	BADDR
STRING .b;	STRING	BADDR
STRING .EXT c;	STRING	EXTADDR
STRING .EXT32 d; ¹	STRING	EXT32ADDR ¹
STRING .EXT64 e; ¹	STRING	EXT64ADDR ¹

¹ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

The code in [Figure 11 \(page 186\)](#) declares an INT(32) simple pointer equivalent to an INT simple pointer. Both contain a word address.

Figure 11 Equivalenced Simple Pointer Declaration

```
INT      .ptr1 := 200;
INT(32) .ptr2 := ptr1;
```



VST307.vsd

pTAL does not verify that the lengths of the objects to which an equivalenced pointer refers are equal. pTAL accepts the declaration in [Example 126 \(page 187\)](#) because the address types of both pointers are WADDR.

Example 126 Equivalenced Objects of Unequal Length

```
INT    .a;          ! a is a pointer to an INT
FIXED  .b = a;      ! OK: a and b are pointers; pTAL does not require
                  ! that the data referenced by b be contained
                  ! inside the data referenced by a
```

REFALIGNED Clause for Equivalenced Simple Pointers

The REFALIGNED clause assigns a REFALIGNED attribute (2 or 8) to a simple equivalenced pointer when you declare the pointer. Equivalenced pointers do not inherit the reference alignment of the previous variable.

Example 127 REFALIGNED Clause for Equivalenced Simple Pointers

```
?REFALIGNED(8)      ! Default reference alignment is 8
INT .p REFALIGNED(2); ! Reference alignment of p is 2
INT .q REFALIGNED(8) = p; ! Reference alignment of q is 8
INT .r REFALIGNED(2) = p; ! Reference alignment of r is 2
INT .s               = p; ! Reference alignment of s is 8
INT .t;              ! Reference alignment of t is 8
```

Equivalencing Procedure Addresses (PROCADDR, PROC32ADDR, and PROC64ADDR) and Pointer Variables (PROCPTR, PROC32PTR, and PROC64PTR)

You can equivalence Pointer Variables to Procedure Addresses and other Pointer Variables, and you can equivalence Procedure Addresses to Pointer Variables and other Procedure Addresses.

NOTE: The procedure address and pointer types, PROC32ADDR, PROC64ADDR, PROC32PTR, and PROC64PTR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, [“64-bit Addressing Functionality” \(page 531\)](#).

Example 128 Equivalencing Procedure Addresses and Pointer Variables

```

PROCPTR pp;                ! pp is a 32-bit procedure pointer
END PROCPTR;

PROC32PTR p32p;            ! p32p is a 32-bit procedure pointer
END PROCPTR;

PROC64PTR p64p;            ! p64p is a 64-bit procedure pointer
END PROCPTR;

PROCADDR pae = pp;         ! pa is a procedure address equivalenced to a
                           ! procedure pointer
PROC32ADDR p32ae = p32p;   ! pa is a procedure address equivalenced to a
                           ! procedure pointer
PROC64ADDR p64ae = p64p;   ! p64a is a procedure address equivalenced to a
                           ! procedure pointer

PROCADDR pa;
PROC32ADDR p32a;
PROC64ADDR p64a;

PROCPTR ppe;               ! ppe is a procedure pointer equivalenced to a
END PROCPTR = pa;          ! procedure pointer
PROC32PTR p32pe;           ! p32pe is a procedure pointer equivalenced to a
END PROCPTR = p32a;        ! procedure pointer
PROC64PTR p64pe;           ! p64pe is a procedure pointer equivalenced to a
END PROCPTR = p64a;        ! procedure pointer

PROCPTR ppl;               ! pp is a procedure pointer equivalenced to a
END PROCPTR = pp;          ! procedure pointer

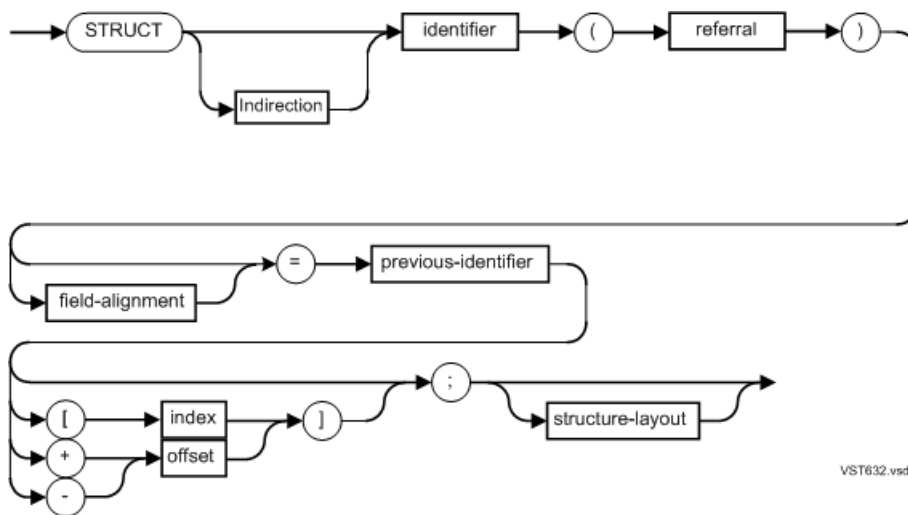
INT PROCPTR i;             ! OK, however it is not recommended to
END PROCPTR = p64p;        ! equivalence procedure pointers with different
                           ! signatures or with different sizes

PROCADDR pal = p64a;       ! OK, however it is not recommended to
                           ! equivalence procedure pointers or addresses
                           ! to procedure pointers or addresses with
                           ! different sizes

```

Declaring Equivalenced Definition Structures

An equivalenced definition structure declaration associates a new structure with a previously declared variable.



Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)\)](#)).

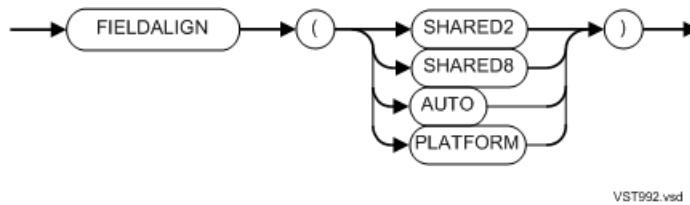
structure

is the identifier that the declaration creates.

referral

is the identifier of a previously declared structure, structure layout, or structure pointer.

field-alignment



FIELDALIGN

specifies the memory alignment for the base of the structure and for fields within the structure. For details about the FIELDALIGN clause, see [Chapter 9 \(page 114\)](#).

SHARED2

specifies that the base of the structure and each field in the structure must begin at an even byte address except STRING fields.

SHARED8

specifies that the offset of each field in the structure from the base of the structure must be begin at an address that is an integral multiple of the width of the field.

AUTO

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the AUTO attribute has in the native mode HP C compiler).

PLATFORM

specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture.

previous-identifier

is the name of a previously declared simple variable, direct array element, structure, structure layout, structure pointer, or equivalenced variable.

index

is an INT constant that specifies the offset of the element in *previous-ident* to which the equivalenced pointer or variable refers. Specify index only with direct variables. index must end on a word boundary.

+ -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT(32) a[0:9];  
INT b = a+6
```

then b is placed in bytes 12 and 13 of a.

offset

is an INT constant that specifies a word offset. Specify offset only with indirect variables. The offset is from the location of the pointer, not from the location of the data pointed to.

structure-layout

is a BEGIN-END block that contains declarations. For more information about the structure layout, see [Chapter 9 \(page 114\)](#).

You must specify either *referral* or *structure-layout* but not both in an equivalenced structure declaration.

You can specify a `FIELDALIGN` clause only if you specify *structure-layout*. You cannot specify a `FIELDALIGN` clause for a *referral* structure.

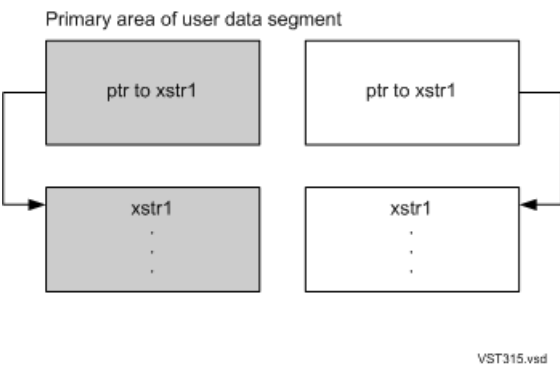
Example 129 Declaring Equivalenced Structures

```
STRUCT a;  
BEGIN  
    INT i;  
    INT j;  
END;  
STRUCT b = a;  
BEGIN  
    INT(32) z;  
END;  
STRUCT c[0:3];  
BEGIN  
    INT i;  
    INT j;  
END;  
STRUCT d[0:3] = c;  
BEGIN  
    INT(32) z;  
END;
```

The code in [Figure 12 \(page 190\)](#) declares an extended indirect definition structure equivalent to a previously declared extended indirect structure.

Figure 12 Equivalenced Definition Structure for CISC Architecture

```
STRUCT .EXT xstr1;  
BEGIN  
    STRING old_name[0:20];  
    STRING old_addr[0:50];  
END;  
STRUCT .EXT xstr2;  
BEGIN  
    STRING new_name[0:30];  
    STRING new_addr[0:40];  
END;
```



If the new structure is to occupy the same location as the previous variable, their addressing modes must match. You can declare a direct or indirect structure equivalent to the following previous variables:

New Structure	Previous Variable
Direct structure	Simple variable

New Structure	Previous Variable
	Direct structure Direct array
Standard indirect structure	Standard indirect structure Standard indirect array Standard structure pointer
Extended indirect structure	Extended indirect structure Extended indirect array Extended structure pointer

If the previous variable is a structure pointer, the new structure is really a pointer.

Topics:

- [Structure Variants \(page 191\)](#)
- [Memory Usage for Structured Equivalenced Variables \(page 192\)](#)
- [FIELDALIGN Clause \(page 193\)](#)

Structure Variants

You use substructures to declare variant records in structures. pTAL does not detect addresses that are redefined by equivalenced variant structures.

Example 130 Structure Variants

```

STRUCT s FIELDALIGN(AUTO);
BEGIN
  STRUCT v1;
  BEGIN
    INT .p;          ! .p is 4 bytes
    INT  q;
  END;
  STRUCT v2 = v1;    ! v2 is equivalenced to v1
  BEGIN              ! v2 is 4 bytes
    INT .EXT e;
  END;
END;

```

When you compile [Example 130 \(page 191\)](#), the compiler allocates 8 bytes, the length of v1. Although v1 and v2 are different lengths and their fields have different data types, the compiler does not report an error or a warning. You must ensure that the variants are meaningful for your algorithms.

The structure in [Example 131 \(page 192\)](#) contains the same variants as the structure in [Example 130 \(page 191\)](#), but the variants are in reverse order.

Example 131 Structure Variants

```
STRUCT s FIELDALIGN(AUTO);
BEGIN
  STRUCT v1;
  BEGIN
    INT .EXT e; ! e is 4 bytes
  END;
  STRUCT v2 FIELDALIGN(SHARED8) = v1;
  BEGIN
    INT .p; ! p is 4 bytes
    INT q; ! Compiler reports a warning
  END;
END;
```

In [Example 131 \(page 192\)](#), `v1` is 4 bytes, but `v2` is 8 bytes. The compiler reports a warning. Data that your program stores into `s.v2.q` overwrites the data in the memory locations that follow `s.v2` is 8 bytes to maintain the alignment of variables in memory. For more information about lengths of pTAL structures, see [Chapter 9 \(page 114\)](#).

Memory Usage for Structured Equivalenced Variables

The memory referenced in an equivalenced declaration must fit within the memory allocated for the previous variable. When you determine the length of a structure, you must account for filler that pTAL adds to the structure. In [Example 132 \(page 192\)](#), the equivalenced declaration is not valid because `b` is 4 bytes, but `a` is only 3 bytes. pTAL adds an extra byte at the end of `b` so that its total length is an integral multiple of its longest component, `i`.

Example 132 Memory Usage for Structured Equivalenced Variables (Incorrect)

```
STRUCT a FIELDALIGN(SHARED2); ! Structure a is 3 bytes
BEGIN
  STRING i;
  STRING j;
  STRING k;
END;
STRUCT b FIELDALIGN(AUTO) = a; ! Structure b is 4 bytes
BEGIN
  INT i;
  STRING j; ! pTAL adds a byte after field j
END;
```

If you declare `b` and then declare `a`, pTAL does not report an error because `a` fits within the four bytes already allocated for `b`, as in [Example 133 \(page 193\)](#).

Example 133 Memory Usage for Structured Equivalenced Variables (Correct)

```
STRUCT b FIELDALIGN(AUTO);
BEGIN
  INT i;
  STRING j; ! pTAL adds a byte after the declaration of j
END;
STRUCT a FIELDALIGN(SHARED2) = b;
BEGIN
  STRING i;
  STRING j;
  STRING k;
END;
```

FIELDALIGN Clause

The FIELDALIGN clause specifies the alignment of the fields of a structure and the alignment of the structure itself in memory. You can use an equivalenced declaration to create two layouts for the same area, one optimized for TAL programs on TNS architecture and the other optimized for pTAL programs on TNS/R or TNS/E architecture. Declare the pTAL structure first.

Example 134 FIELDALIGN Clause in Structured Equivalenced Variables

```
STRUCT a FIELDALIGN(SHARED8);
BEGIN
  INT i;
  INT j;
END;
STRUCT b FIELDALIGN(SHARED2) = a;
BEGIN
  INT i;
  INT j;
END;
```

In [Example 134 \(page 193\)](#), structures a and b declare the same fields, but a specifies FIELDALIGN(SHARED8), the optimal alignment for pTAL, whereas b specifies FIELDALIGN(SHARED2), the alignment for TAL. pTAL generates fast code for references to a.i, but conservative code for references to b.i.

For more information about using the FIELDALIGN clause, see [Chapter 9 \(page 114\)](#).

System Global Equivalenced Variable Declarations

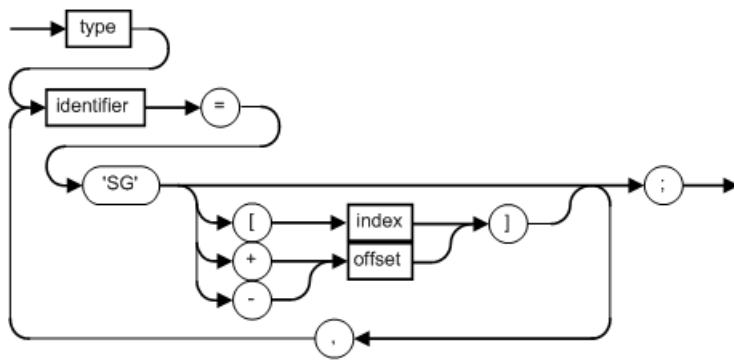
NOTE: Only procedures that operate in privileged mode can access system global data.

System global equivalencing associates a global, local, or sublocal identifier with a location that is relative to the base address. You can declare the following equivalenced variables for either system global ('SG') or extended system global ('SGX') addresses:

- [Equivalenced Simple Variable \(page 193\)](#)
- [Equivalenced Definition Structure \(page 194\)](#)
- [Equivalenced Referral Structure \(page 195\)](#)
- [Equivalenced Simple Pointer \(page 196\)](#)
- [Equivalenced Structure Pointer \(page 197\)](#)

Equivalenced Simple Variable

An equivalenced simple variable declaration associates a simple variable with a location that is relative to the 'SG' base address.



VST068.vsd

type

is any data type except UNSIGNED; specifies the data type of *identifier*.

identifier

is the identifier of a simple variable to be made equivalent to 'SG'.

'SG'

a symbol that denotes a 16-bit system global address.

index

is an INT constant that specifies the offset of the element in *previous-ident* to which the equivalenced pointer or variable refers. Specify index only with direct variables. index must end on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT a[0:9];
```

```
INT b = a+5
```

then b is placed at a[5].

offset

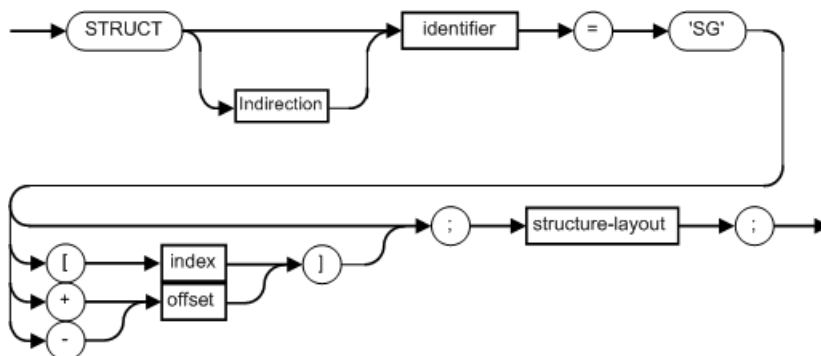
an equivalent INT values in the range 0 through 63.

Example 135 Equivalenced Simple Variable Declaration

```
INT item = 'SG' + 15;
```

Equivalenced Definition Structure

An equivalenced definition structure declaration associates a definition structure with a location relative to a system global ('SG') or extended system global ('SGX') base address.



VST710.vsd

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of a definition structure to be made equivalent to 'SG'.

'SG'

denotes a 16-bit system global address.

index

is an INT constant that specifies the offset of the element in *previous-indent* to which the equivalenced pointer or variable refers. Specify index only with direct variables. index must end on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-indent*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT a[0:9];
```

```
INT b = a+5
```

then b is placed at a[5].

offset

an equivalent INT values in the range 0 through 63.

structure-layout

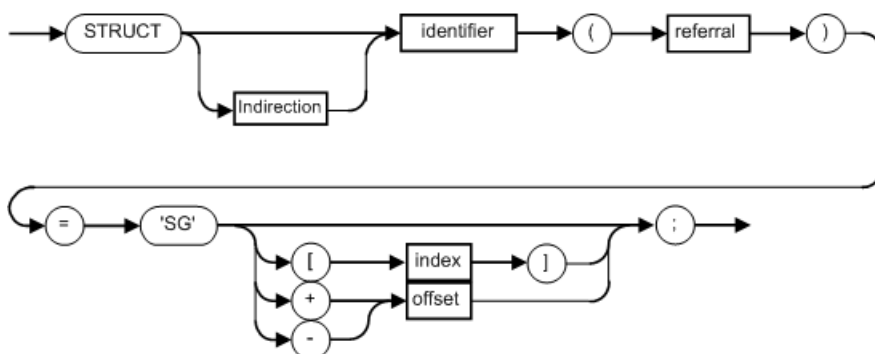
a BEGIN-END block that contains declarations for structure items.

Example 136 Equivalenced Definition Structure Declaration

```
STRUCT def_struct = 'SG'[10];  
BEGIN  
  STRING out;  
  FIXED up;  
  REAL in;  
END;
```

Equivalenced Referral Structure

The equivalenced referral structure declaration associates a referral structure with a location relative to the base address of the system global ('SG') or the extended system global ('SGX') data area.



VST703.vsd

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of a referral structure to be made equivalent to 'SG'.

referral

is the identifier of a previously declared structure or structure pointer that is to provide the layout for this structure.

'SG'

denotes a 16-bit system global address.

index

is an INT constant that specifies the offset of the element in *previous-ident* to which the equivalenced pointer or variable refers. Specify index only with direct variables. *index* must end on a word boundary.

+, -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT a[0:9];  
INT b = a+5
```

then b is placed at a[5].

offset

an equivalent INT values in the range 0 through 63.

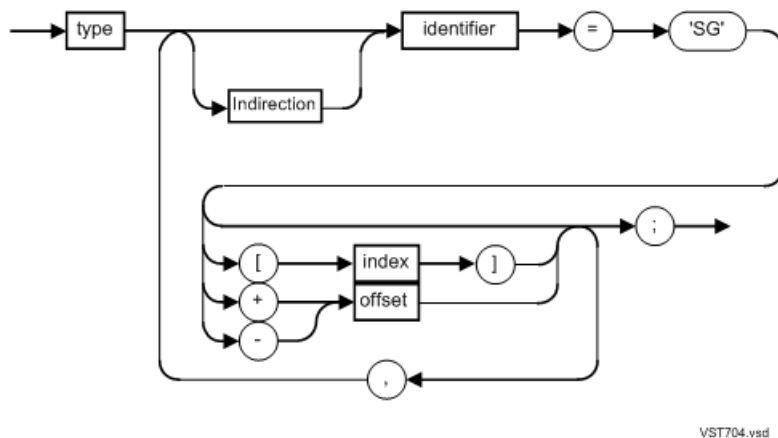
If you specify an indirection symbol (see [Table 14 \(page 41\)](#)), the structure behaves like a structure pointer. If you do not specify an indirection symbol, the structure has direct addressing.

Example 137 Equivalenced Referral Structure Declaration

```
STRUCT def_struct;  
BEGIN  
  STRING a[0:99];  
  REAL b[0:9];  
END;  
STRUCT ref_struct (def_struct) = 'SG'[30];
```

Equivalenced Simple Pointer

The equivalenced simple pointer declaration associates a simple pointer with a location relative to the base address of the system global ('SG') or the extended system global ('SGX') data area.



type

is any data type except UNSIGNED and specifies the data type of the value to which the pointer points.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

identifier

is the identifier of a simple pointer to be made equivalent to 'SG'.

'SG'

denotes a 16-bit system global address.

index

is an INT constant that specifies the offset of the element in *previous-indent* to which the equivalenced pointer or variable refers. Specify *index* only with direct variables. *index* must end on a word (16-bit) boundary.

+, -

is the word or byte offset, relative to the base of *previous-indent*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT a[0:9];  
INT b = a+5
```

then b is placed at a[5].

offset

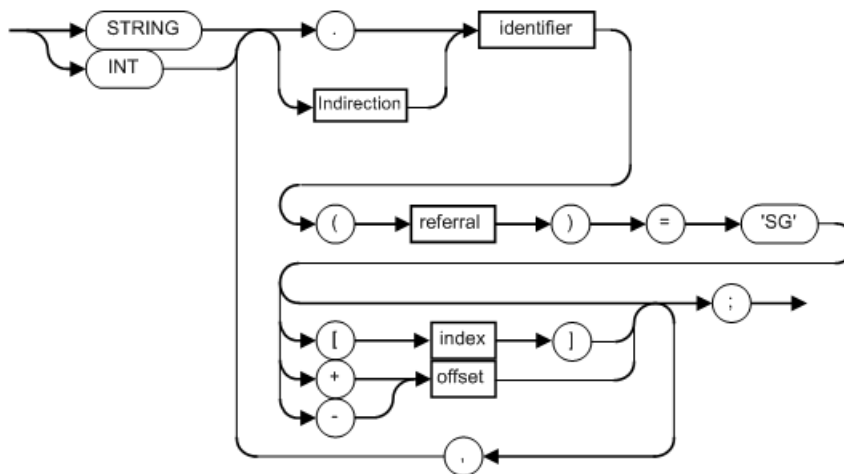
an equivalent INT values in the range 0 through 63.

Example 138 Equivalenced Simple Pointer Declaration

```
INT .ptr = 'SG' + 2;
```

Equivalenced Structure Pointer

The equivalenced structure pointer declaration associates a structure pointer with a location relative to the base address of the system global (.SG) or the extended system global (SGX) data area.



STRING

is the STRING attribute.

INT

is the INT attribute.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see Table 14 (page 41)).

identifier

is the identifier of a structure pointer to be made equivalent to 'SG'.

referral

is the identifier of a previously declared structure or structure pointer that is to provide the layout for *identifier*.

'SG'

denotes a 16-bit system global address.

index

is an INT constant that specifies the offset of the element in *previous-ident* to which the equivalenced pointer or variable refers. Specify index only with direct variables. index must end on a word (16-bit) boundary.

+, -

is the word or byte offset, relative to the base of *previous-ident*, where the equivalenced variable is placed. For example, if a and b are declared:

```
INT a[0:9];  
INT b = a+5
```

then b is placed at a[5].

offset

an equivalent INT values in the range 0 through 63.

on page 10-6 describes the kind of addresses a structure pointer can contain depending on the STRING or INT attribute and addressing symbol.

Example 139 Equivalenced Structure Pointer Declaration

```
STRUCT .some_struct;  
BEGIN  
    INT a;  
    INT b[0:5];  
END;  
INT .struct_ptr (some_struct) = 'SG' + 30;
```

12 Statements

Statements—also known as executable statements—perform operations in a program. They can modify the program’s data or control the program’s flow.

Table 55 Summary of Statements

Category	Statement	Operation
Program control	ASSERT	Conditionally calls a procedure
	CALL	Calls a procedure or subprocedure
	CASE	Selects a set of statements based on a selector value
	DO-UNTIL	A post-test loop that repeatedly executes a statement until a specified condition becomes true
	FOR	Executes a pretest loop n times
	GOTO	Unconditionally branches to a label within a procedure or subprocedure
	IF	Conditionally selects one of two possible statements
	RETURN	Returns from a procedure or a subprocedure to the caller; returns a value from a function; returns a condition code value
	WHILE	Executes a pretest loop while a condition is true
Data transfer	Assignment	Stores a value in a variable
	Bit-Deposit Assignment	Stores a value in a bit or in a group of sequential bits
Data scan	SCAN and RSCAN	Scan data for a test character, left-to-right and right-to-left, respectively
Data allocation	DROP	Removes either a label (from the symbol table) or a temporary variable that was created by a USE statement
	USE	Creates a temporary variable

In addition to the statements summarized in [Table 55 \(page 199\)](#), this section describes:

- [Using Semicolons in Statements \(page 199\)](#)
- [Compound Statements \(page 200\)](#)

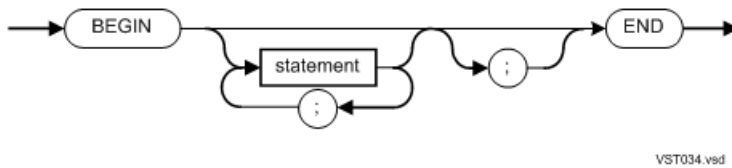
Using Semicolons in Statements

You use semicolons with statements as follows:

- A semicolon is required between successive statements.
- A semicolon is optional before an END keyword that terminates a compound statement.
- A semicolon must not immediately precede an ELSE or UNTIL keyword.
- A semicolon alone in place of a statement creates a null statement. The compiler generates no code for null statements. You can use a null statement wherever you can use a statement except immediately before an ELSE or UNTIL keyword.

Compound Statements

A compound statement is a BEGIN-END block that groups statements to form a single logical statement.



BEGIN

indicates the start of the compound statement.

statement

is a statement described in this section.

; (semicolon)

is a statement separator that is required between successive statements. A semicolon before an END that terminates a compound statement is optional and represents a null statement.

END

indicates the end of the compound statement.

You can use compound statements anywhere you can use a single statement. You can nest them to any level in statements such as IF, DO, FOR, WHILE, or CASE.

Example 140 Null Compound Statement

```
BEGIN
END;
```

Example 141 Compound Statement

```
BEGIN
  a := b + c;
  d := %B101;
  f := d - e;
END;
```

ASSERT

The ASSERT statement conditionally calls the procedure specified in the active directive [ASSERTION \(page 495\)](#).



assert-level

is an integer in the range 0 through 32,767.

If *assert-level* is greater than or equal to the *assertion-level* specified in the active ASSERTION directive and if *condition* is true, the program calls the procedure specified in the active ASSERTION directive.

condition

is a conditional expression (see [Conditional Expressions \(page 81\)](#)).

To use the ASSERT statement and the ASSERTION directive together for debugging or error-handling:

1. Put an ASSERTION directive in the source code, specifying an *assertion-level* and an error-handling procedure.
2. Put an ASSERT statement at each place where you want to execute the error-handling procedure when an error occurs. In each ASSERT statement, specify:
 - An *assert-level* that is greater than or equal to the *assertion-level*
 - A *condition* that will be true when an error occurs

During program execution, if an *assert-level* is greater than or equal to the active *assertion-level* and the associated *condition* is true, the program calls the error-handling procedure.

Example 142 (page 201) calls PROCESS_DEBUG_ whenever a carry or overflow condition occurs.

Example 142 ASSERTION Directive and ASSERT Statement

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS (PROCESS_DEBUG_)
?ASSERTION 5, PROCESS_DEBUG_ ! Activates all ASSERT conditions
SCAN array WHILE " " -> @pointer;
ASSERT 10 : LOCAL_CARRY_FLAG;
!Lots of code
ASSERT 10 : LOCAL_CARRY_FLAG;
!More code
ASSERT 20 : LOCAL_OVERFLOW_FLAG; ! $OVERFLOW routine tests for
                                ! arithmetic overflow
```

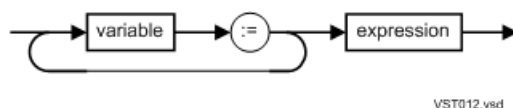
In Example 142 (page 201):

- If you change the *assertion-level* from 5 to 15, you nullify the two ASSERT statements that specify *assert-level* 10 and the LOCAL_CARRY_FLAG condition.
- If you change the *assertion-level* from 5 to 30, you nullify all the ASSERT statements.

If all ASSERT statements that cover a particular condition have the same *assert-level*, it is easier to nullify specific levels of ASSERT statements.

Assignment

The assignment statement assigns a value to a previously declared variable.



variable

is the identifier of a simple variable, array element, simple pointer, structure pointer, or structure data item, with or without a bit deposit field and/or index. To update a pointer's content, prefix the pointer identifier with @.

expression

is either:

- An arithmetic expression of the same data type as *variable*
- A conditional expression, which always has an INT result

expression can be a bit extraction value or an identifier prefixed with @ (the address of a variable). *expression* cannot be a constant list.

In general, the data types of *variable* and *expression* must match. To convert the data type of *expression* to match the data type of *variable*, use a type-transfer routine, described in Chapter 15 (page 274).

The following rules apply to assignment statements:

- The data type of the expression on the right side of an assignment statement must be compatible with the data type of the destination on the left side of the assignment statement.
- You cannot store a value into the implicit pointer of an indirect array or indirect structure.
- You cannot store a value into the implicit pointer of an equivalenced variable that references the data of an indirect array or indirect structure.
- Do not depend on whether the left side or the right side of an assignment statement is evaluated first.
- Address types must match.
- pTAL disallows all assignments of unlike data types except the following:

- STRING and UNSIGNED(1-16) variables are syntactically and semantically equivalent to INT variables. Thus, STRING and UNSIGNED(1-16) variables are valid anywhere an INT variable is valid.

An UNSIGNED(1-16) variable or one-byte STRING value that is used as an INT value is left-filled with binary zeros. Conversely, the high-order bits of an INT value are lost if the value is stored in an UNSIGNED variable that is less than 16 bits, or to a one-byte STRING variable, as shown in the following examples:

```
INT          i := 3;
STRING       s;
UNSIGNED(12) u1;
UNSIGNED(24) u2;
s := i + 1;    ! OK: Assignment of INT to STRING
i := s + %H20; ! OK: Assignment of STRING to INT
u1 := i + s;   ! OK: INT + STRING
u2 := i;       ! ERROR: INT and UNSIGNED(17-31) are not
               ! assignment-compatible
```

When an INT variable is assigned to a STRING variable, the upper 8 bits of the INT variable are not retained in any way in the STRING variable. Thus, the comparison of `i1` to `i2` in the final statement of the following code fails because `i2` still holds the full 16 bits that were assigned to it at the beginning of the code, but `i1` holds only the lower 8 bits. The upper 8 bits of `i1` are not transferred to `s` in the assignment statement `s := i1`.

```
INT i1 := %HFFFF,
i2;
STRING s;
i2 := i1;    ! Copy i1 to i2
s := i1;     ! Assign 16-bit INT to 8-bit STRING
i1 := s;     ! Assign 8-bit STRING to 16-bit INT
IF i1 = i2 THEN; ! i1 (%HFF) is not equal to i2 (%HFFFF)
```

- UNSIGNED(17-31) variables are syntactically and semantically equivalent to INT(32) variables. Thus, UNSIGNED(17-31) variables are valid anywhere INT(32) variables are valid:

```
INT(32)      i;
UNSIGNED(31) u;
INT          j;
i := u;      ! OK: No bits are lost in assignment
u := i;      ! WARNING: Most significant bit of i could
             ! be lost
u := j;      ! ERROR: INT and UNSIGNED(17-31) are not
             ! assignment-compatible
```

Topics:

- [Pointer Assignment \(page 203\)](#)
- [Assigning Numbers to FIXED Variables \(page 203\)](#)
- [Assigning Character Strings \(page 203\)](#)
- [Examples \(page 203\)](#)

Pointer Assignment

The result of applying an @ operator to a variable or pointer is an address whose data type is one of the pTAL address types.

In an assignment statement, one of the following must be true:

- Both operands are the same address type.
- Neither operand is an address type.

You can use type-conversion built-in routines to convert some address types to other address types. For more information:

Topic	See ...
Type-conversion built-in routines	Chapter 15 (page 274)
Converting addresses	Chapter 5 (page 69)
Pointers	Chapter 10 (page 161)

Assigning Numbers to FIXED Variables

When you assign a number to a FIXED variable, the system scales the value up or down to match the *fpoint* value. If the system scales the value down, you lose some precision depending on the amount of scaling; for example:

```
FIXED(2) a;  
a := 2.348F; ! System scales value to 2.34F
```

If the ROUND directive is active, the system scales the value as needed, then rounds the value away from zero as follows:

```
(IF value < 0 THEN value - 5 ELSE value + 5) / 10
```

For example, if you assign 2.348F to a FIXED(2) variable, the ROUND directive scales the value by one digit and then rounds it to 2.35F.

Assigning Character Strings

You can assign a character string to STRING, INT, or INT(32) variables.

If you assign a one-character string such as "A" to an INT simple variable, the system places the value in the right byte of a word and 0 in the left byte. To store a character in the left byte, assign the character and a space, as in:

```
"A "
```

If you assign a character string to a FIXED, REAL, or REAL(64) variable, the compiler issues a type incompatibility error.

Examples

Example 143 Assignment Statements

```
INT array[0:2];      ! Array  
INT .ptr;            ! Simple pointer  
REAL real_var;       ! REAL variable
```

```

FIXED fixed_var;           ! FIXED variable
array[2] := 255;          ! Assign a value to array[2]
@ptr := @array[1];        ! Assign address of array[1]
                           ! to ptr
ptr := array[2];          ! Assign value of array[2]
                           ! to array[1], to which ptr
                           ! points
real_var := 36.6E-3;      ! Assign a REAL value to a
                           ! REAL variable
fixed_var := $FIX (real_var); ! Convert a REAL value to FIXED
                           ! and assign to a FIXED variable

```

Assignment statements can assign character strings but not constant lists, so in [Example 144 \(page 204\)](#), the three assignment statements together store the same value as the one constant list in the declaration.

Example 144 Assignment Statements Equivalent to a Constant List

```

INT .b[0:2] := ["ABCDEF"]; ! Declare and initialize
                           ! with constant list
b[0] := ["AB"];
b[1] := ["CD"];
b[2] := ["EF"];

```

In [Example 145 \(page 204\)](#), the first assignment statement (which contains assignment expressions) is equivalent to the three separate assignments that follow it.

Example 145 Assignment Statement With Assignment Expressions

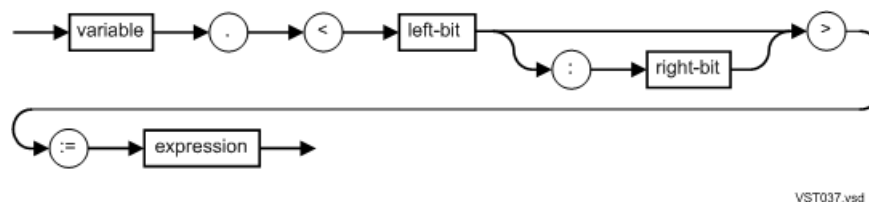
```

INT int1;
INT int2;
INT int3;
INT var := 16;
int1 := int2 := int3 := var; ! Assignment that contains
                             ! assignment expressions
int1 := var;                 ! Separate assignments
int2 := var;
int3 := var;

```

Bit-Deposit Assignment

The bit deposit form of the assignment statement lets you assign a value to an individual bit or to a group of sequential bits.



variable

is the identifier of a STRING or INT variable, but not an UNSIGNED(1-16) variable. *variable* can be the identifier of a simple variable, array element, or simple pointer (inside or outside a structure).

left-bit

is an INT constant that specifies the leftmost bit of the bit deposit field.

For STRING variables, specify a bit number in the range 8 through 15. Bit 8 is the leftmost bit in a STRING variable; bit 15 is the rightmost bit.

right-bit

is an INT constant specifying the rightmost bit of the bit deposit field. *right-bit* must be equal to or greater than *left-bit*.

For STRING variables, specify a bit number in the range 8 through 15. Bit 8 is the leftmost bit in a STRING variable; bit 15 is the rightmost bit.

expression

is an INT arithmetic or conditional expression, with or without a bit field specification.

The bit deposit field is on the left side of the assignment operator (:=). The bit deposit assignment changes only the bit deposit field. If the value on the right side has more bits than the bit deposit field, the system ignores the excess high-order bits when making the assignment.

Specify the variable/bit-field construct with no intervening spaces as shown:

```
myvar.<0:5>
```

Do not use bit deposit fields to pack data. Instead, declare an UNSIGNED variable and specify the appropriate number of bits in the bit field.

Examples:

1. The bit deposit assignment sets bits 3 through 7 of the word designated by *x*:

```
INT x;  
x.<3:7> := %B11111;
```

2. The bit deposit assignment replaces bits <10> and <11> with zeros:

```
INT old := -1;      ! old = %b1111111111111111  
old.<10:11> := 0;    ! old = %b1111111111001111
```

3. The bit deposit assignment sets bit <8>, the leftmost bit of *strng*, to 0:

```
STRING strng := -1; ! strng = %b11111111  
strng.<8> := 0;      ! strng = %b01111111
```

4. The value %577 is too large to fit in bits <7:12> of *var*. The system truncates %577 to %77 before performing the bit deposit:

```
INT var := %125252; ! var = %b1010101010101010  
var.<7:12> := %577;  ! %77 = %b0000000010111111  
                ! var = %b101010101111111010
```

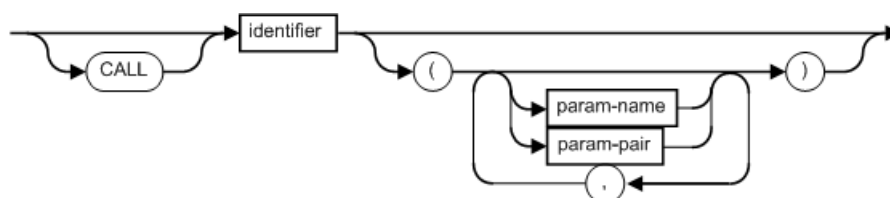
5. The bit deposit assignment replaces bits <7:8> of *new* with bits <8:9> of *old*:

```
INT new := -1;      ! new = %b1111111111111111  
INT old := 0;        ! old = %b0000000000000000  
new.<7:8> := old.<8:9>; ! new = %b1111111001111111
```

CALL

The CALL statement calls a procedure, subprocedure, or entry-point identifier and optionally passes parameters to it.

In pTAL, a procedure's formal and actual parameters either match if the data type of each formal parameter and its corresponding actual parameter match exactly or match if the data type of the actual parameter is converted according to the rules under [Converting Between Address Types](#) (page 52).



VST038.vsd

identifier

is the identifier of a previously declared procedure, subprocedure, or entry-point identifier.

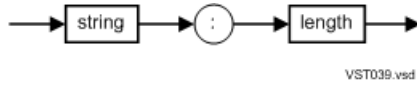
param-name

is a variable identifier or an expression that defines an actual parameter to pass to a formal parameter declared in *identifier*.

param-pair

is an actual parameter pair to pass to a formal parameter pair declared in *identifier*.

param-pair has the form:



string

is an expression of the type `STRING .` or `STRING .EXT`.

length

is an `INT` expression that specifies the length, in bytes, of *string*.

Use the `CALL` statement to call procedures and subprocedures (but usually not functions).

To call functions, you usually use their identifiers in expressions. If you call a function by using a `CALL` statement, the caller does not use the returned value of the function.

Actual parameters are value or reference parameters and are optional or required depending on the formal parameter specification in the called procedure or subprocedure declaration (described in [Chapter 14 \(page 246\)](#)). A value parameter passes the content of a location; a reference parameter passes the address of a location.

In a `CALL` statement to a `VARIABLE` procedure or subprocedure or to an `EXTENSIBLE` procedure, you can omit optional parameters in two ways:

- You can omit parameters or parameter pairs unconditionally. Use an empty comma for each omitted parameter or parameter pair up to the last specified parameter or parameter pair. If you omit all parameters, you can specify an empty parameter list (parentheses with no commas) or you can omit the parameter list altogether.
- You can omit parameters or parameter pairs conditionally. Use the `$OPTIONAL` built-in routine as described in [Chapter 15 \(page 274\)](#).

After the called procedure or subprocedure completes execution, control returns to the statement following the `CALL` statement that calls the procedure or subprocedure.

Example 146 CALL Statement

```
PROC p (a, b, c);  
  INT(32) a;  
  REAL b;  
  REAL(64) c;  
BEGIN  
END;  
PROC q;  
BEGIN  
  CALL p(1.0E0, ! ERROR: Cannot pass REAL to INT(32)  
          1D,    ! ERROR: Cannot pass INT(32) to REAL  
          1F);  ! ERROR: Cannot pass FIXED to REAL(64)  
END;
```

CASE

The CASE statement executes a choice of statements based on a selector value. Normally, you use labeled CASE statements. Labeled CASE statements are described first, followed by unlabeled CASE statements.

If a case index does not match any alternative, an instruction trap occurs.

Topics:

- [Empty CASE \(page 207\)](#)
- [Labeled CASE \(page 207\)](#)
- [Unlabeled CASE \(page 209\)](#)

Empty CASE

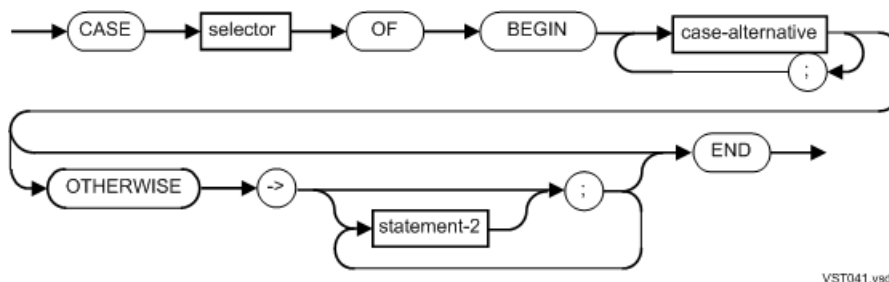
pTAL does not allow empty CASE statements. A CASE statement include at least one alternative, even if there are no statements specified for that alternative.

Example 147 Empty CASE Statement

```
CASE i OF  
BEGIN      ! In this unlabeled CASE statement,  
;          ! the semicolon creates an alternative  
END;
```

Labeled CASE

The labeled CASE statement executes a choice of statements when the value of the selector matches a case label associated with those statements.



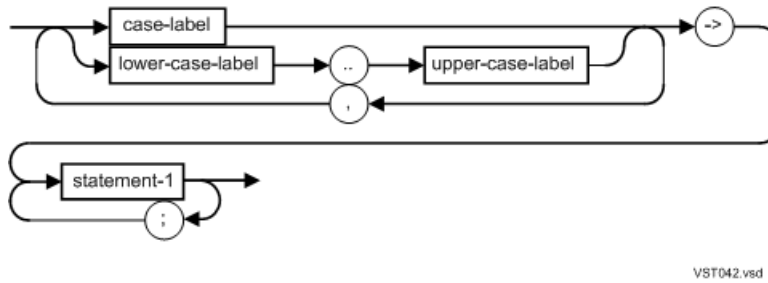
VST041.vsd

selector

is an INT or INT (32) value arithmetic expression that uniquely selects the *case-alternative* for the program to execute.

case-alternative

associates one or more *case-label* s or one or more ranges of *case-label* s with one or more *statement* s. The *statement* s of a *case-alternative* are executed if *selector* equals an associated *case-label*. Each *case-alternative* has the form:



case-label

a signed INT constant or LITERAL. Each *case-label* must be unique in the CASE statement.

lower-case-label

is the smallest value in an inclusive range of signed INT constants or LITERALS.

upper-case-label

is the largest value in an inclusive range of signed INT constants or LITERALS.

statement-1

is any statement described in this section.

OTHERWISE

specifies an optional sequence of statements to execute if *selector* does not select any *case-alternative*. If no OTHERWISE clause is present and *selector* does not match a *case-alternative*, a run-time error occurs. Always include an OTHERWISE clause, even if it contains no statements.

statement-2

is any statement described in this section.

A CASE statement must have at least one alternative.

If you omit the OTHERWISE clause and *selector* is out of range (negative or greater than *n*), then a divide-by-zero instruction trap occurs.

A CASE index matches an alternative identified by the keyword OTHERWISE if and only if the case index does not match any other alternative and OTHERWISE is an alternative.

Example 148 Labeled CASE Statement

```
LITERAL apple, orange, pear, peach, prune;
INT i;
i := peach;           ! Set index value
CASE i OF             ! Execute CASE
BEGIN
  apple    -> CALL p1;
  orange   -> CALL p2;
  prune    -> CALL p3;
  OTHERWISE -> CALL p4; ! Execute this alternative
END;
```

Example 149 Labeled CASE Statement

```
INT location;
LITERAL bay_area, los_angeles, hawaii, elsewhere;
PROC area_proc (area_code);
  INT area_code;           ! Declare selector as
```



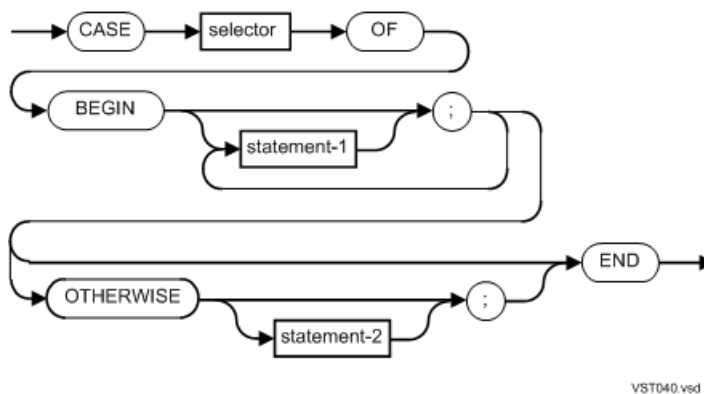
```

BEGIN                                ! formal parameter
CASE area_code OF                   ! Selector is area_code
BEGIN
    408, 415 ->
        location := bay_area;
    213, 818 ->
        location := los_angeles;
    808 ->
        location := hawaii;
    OTHERWISE ->
        location := elsewhere;
END;                                ! End CASE statement
END;                                ! End area_proc

```

Unlabeled CASE

The unlabeled CASE statement executes a choice of statements, based on an inclusive range of implicit selector values, from 0 through n , with one statement for each value.



selector

is an INT or INT (32) arithmetic expression that selects the statement to execute.

statement-1

is any statement described in this section. Include a *statement-1* for each value in the implicit *selector* range, from 0 through n . If a *selector* has no action, specify a null statement (semicolon with no statement). If you include more than one *statement-1* for a value, you must use a compound statement.

OTHERWISE

indicates the statement to execute for any case outside the range of *selector* values. If the OTHERWISE clause consists of a null statement, control passes to the statement following the unlabeled CASE statement.

statement-2

is any statement described in this section. Include a *statement-2* for each value in the implicit *selector* range, from 0 through n . If a *selector* has no action, specify a null statement (semicolon with no statement). If you include more than one *statement-2* for a value, you must use a compound statement.

The compiler numbers each *statement* in the BEGIN clause consecutively, starting with 0. If the *selector* matches the compiler-assigned number of a *statement*, that *statement* is executed. For example, if the *selector* is 0, the first *statement* executes; if the *selector* is 4, the fifth *statement* executes. Conversely, if the *selector* does not match a compiler-assigned number, the OTHERWISE *statement*, if any, executes.

The index of an unlabeled CASE statement and the selector of a labeled CASE statement can be INT(32) expressions.

Example 150 Unlabeled CASE Statement

```
INT(32) i;  
CASE i OF  
BEGIN  
    ...  
END;  
CASE i OF  
BEGIN  
    0-> ...  
    1-> ...  
END;
```

If you omit the OTHERWISE clause and *selector* is out of range (negative or greater than *n*), a divide-by-zero instruction trap occurs.

Example 151 Unlabeled CASE Statement

```
INT selector;  
INT var0;  
INT var1;  
CASE selector OF  
BEGIN  
    var0 := 0;           ! Executes if selector=0  
    var1 := 1;           ! Executes if selector=1  
    OTHERWISE  
        CALL error_handler; ! Executes if selector is not 0 or 1  
END;
```

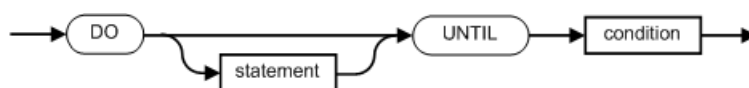
Example 152 (page 210) selectively moves one of several messages into an array.

Example 152 Unlabeled CASE Statement Assigning Text to Array

```
PROC msg_handler (selector);  
    INT selector;  
BEGIN  
    LITERAL len = 80;           ! Length of array  
    STRING .a_array[0:len - 1]; ! Destination array  
    CASE selector OF  
    BEGIN                       ! Move statements  
        !0! a_array := "Training Program";  
        !1! a_array := "End of Program";  
        !2! a_array := "Input Error";  
        !3! a_array := "Home Terminal Now Open";  
    OTHERWISE  
        a_array := "Bad Message Number";  
    END;                       ! End of CASE statement  
END;                           ! End of procedure
```

DO-UNTIL

The DO-UNTIL statement is a posttest loop that repeatedly executes a statement until a specified condition becomes true.



VST046.vsd

statement

is any statement described in this section.

condition

is either:

- A conditional expression
- An INT, INT(32), or FIXED arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

If *condition* is false, the DO-UNTIL statement continues to execute. If *condition* is true, the statement following the DO-UNTIL statement executes.

A DO-UNTIL statement always executes at least once (unlike the [WHILE \(page 232\)](#)).

In [Example 153 \(page 211\)](#), the DO-UNTIL statement loops through *array_a*, testing the content of each element until an alphabetic character occurs.

Example 153 DO-UNTIL Statement

```
index := -1;
DO index := index + 1 UNTIL $ALPHA (array_a[index]);
```

In [Example 154 \(page 211\)](#), the DO-UNTIL statement loops through *array_a*, assigning a 0 to each element until all the elements contain a 0.

Example 154 DO-UNTIL Statement

```
LITERAL limit = 9;
INT index := 0;
STRING .array_a[0:limit];y
DO
    BEGIN                                ! Compound statement to execute in
        array_a[index] := 0;             ! DO loop
        index := index + 1;
    END
UNTIL index > limit;                     ! Condition for ending loop
```

The conditional expression cannot reference hardware indicators (<, <=, =, <>, >, >=, '<', '<=', '=', '<>', '>', '>=', \$OVERFLOW, and \$CARRY). Only IF statements can test hardware indicators. For more information, see [Chapter 13 \(page 234\)](#).

To use a hardware indicator's value to control a DO-UNTIL loop, save the hardware indicator's value and either test the saved value (as in [Example 155 \(page 212\)](#)) or execute an explicit GOTO statement to exit the loop (as in [Example 156 \(page 212\)](#)). Hardware indicators cannot appear in the conditional expression of a DO-UNTIL statement.

Example 155 DO-UNTIL Statement With Hardware Indicator

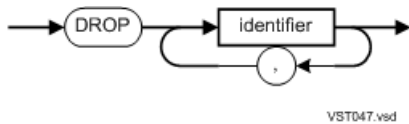
```
INT exit_loop;
...
exit_loop := FALSE;
DO
  BEGIN
    ...
    READ(...);
    IF <> THEN exit_loop := TRUE;
  END
UNTIL exit_loop;
```

Example 156 DO-UNTIL Statement With GOTO Statement

```
DO
  BEGIN
    ...
    READ(...);
    IF <> THEN GOTO out;
  END
UNTIL false;
out:
...
```

DROP

The DROP statement removes either a label (from the symbol table) or a temporary variable that was created by the statement [USE \(page 232\)](#).



identifier

is the identifier of either a label or a temporary variable.

Dropping Labels

You can drop a label only if you have already declared the label or used it to label a statement. Before you drop a label, be sure there are no further references to the label. If a GOTO statement refers to a dropped label, a run-time error occurs. After you drop a label, you can, however, use the identifier to label a statement preceding the GOTO statement that refers to the label.

Dropping Temporary Variables

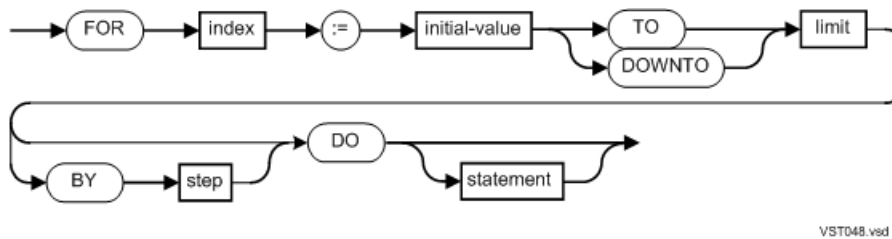
When you no longer need a temporary variable, drop (remove) it by using a DROP statement. After you drop a temporary variable, do not use its identifier without using a new USE statement to assign a value to the temporary variable.

If you do not drop all temporary variables, the compiler automatically drops them when the procedure or subprocedure completes execution.

If you reserve an temporary variable for a FOR loop, do not drop the temporary variable within the scope of the loop.

FOR

The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decrementing an index automatically.



index

is a value that increments or decrements automatically.

In [Standard \(page 214\)](#), *index* is the identifier of an INT or INT(32) simple variable, array element, simple pointer, or structure data item.

In [Optimized \(page 214\)](#), *index* is the identifier of an index register you have reserved with the [USE \(page 232\)](#).

initial-value

is an arithmetic expression (such as 0) that initializes *index*. If *index* is INT, *initial-value* is INT. If *index* is INT(32), *initial-value* is INT(32).

TO

increments *index* each time the loop executes until *index* is greater than or equal to *limit*, at which point the loop stops.

DOWNTO

decrements *index* each time the loop executes until *index* is less than or equal to *limit*, at which point the loop stops.

limit

is an arithmetic expression that terminates the loop. If *index* is INT, *initial-value* is INT. If *index* is INT(32), *initial-value* is INT(32).

step

is an arithmetic expression by which to increment or decrement *index* each time the loop executes. If *index* is INT, then *step* is INT; otherwise, *index* is INT(32). The default is 1.

statement

is any statement described in this section.

The FOR statement tests *index* at the beginning of each iteration of the loop. If *index* exceeds *limit* on the first test, the loop never executes.

Topics:

- [Nested \(page 213\)](#)
- [Standard \(page 214\)](#)
- [Optimized \(page 214\)](#)

Nested

You can nest FOR statements to any level.

The nested FOR statement in [Example 157 \(page 214\)](#) uses `multiples` as a two-dimensional array. It fills the first row with multiples of 1, the next row with multiples of 2, and so on.

Example 157 Nested FOR Statement

```
INT .multiples[0:10*10-1];
INT row;
INT column;
FOR row := 0 TO 9 DO
  FOR column := 0 TO 9 DO
    multiples [row * 10 + column] := column * (row + 1);
```

Standard

For *index*, standard FOR statements specify an INT or INT(32) variable. Standard FOR statements execute as follows:

- When the looping terminates, *index* is greater than *limit* if:
 - The *step* value is 1.
 - You use the TO keyword (not DOWNTO).
 - The *limit* value (not a GOTO statement) terminates the looping.
- *limit* and *step* are recomputed at the start of each iteration of the statement.

The standard FOR statement in [Example 158 \(page 214\)](#) uses the DOWNTO clause to reverse a string from "BAT" to "TAB".

Example 158 Standard FOR Statement

```
LITERAL len = 3;
LITERAL limit = len - 1;
STRING .normal_str[0:limit] := "BAT";
STRING .reversed_str[0:limit];
INT index;
FOR index := limit DOWNTO 0 DO
  reversed_str[limit - index] := normal_str[index];
```

Optimized

For *index*, an optimized FOR statement specifies a temporary variable created by the statement [USE \(page 232\)](#). Optimized FOR statements execute faster than standard FOR statements because limit is computed only once, at the start of the first iteration of the statement.

Example 159 Standard and Optimized FOR Statements

```
INT x;
INT y;
INT PROC f;
BEGIN
  x := x + 1;
  RETURN 10;
END;

INT PROC p1;                                ! p1 has standard FOR statement
BEGIN
  INT i;
  x := 0;
  FOR i := 1 to f() DO ... ;                ! f is called 10 times
                                           ! i=11 here
  RETURN x;                                ! p1 returns 10
END;

INT PROC q;
BEGIN
  x := y + 1;
```

```

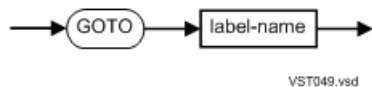
    RETURN 10;
END;

INT PROC p2;                                ! p2 has optimized FOR statement
BEGIN
    USE i;
    y := 0;
    FOR i := 1 to q() DO ... ;               ! q is called 1 time
                                           ! i=10 here
    RETURN x;                                ! p2 returns 1
END;

```

GOTO

The GOTO statement unconditionally transfers control to a labeled target statement.



label-name

is the label that precedes the target statement (see [Labels in Procedures \(page 273\)](#)).

A GOTO statement can be either local or nonlocal.

Topics:

- [Local \(page 215\)](#)
- [Nonlocal \(page 215\)](#) (not recommended)
- [GOTO and Target Statements With Different Trapping States \(page 216\)](#)

Local

If the GOTO statement and the target statement are in the same procedure or in the same subprocedure, the GOTO statement is local.

Example 160 Local GOTO Statement

```

PROC p
BEGIN
    LABEL calc_a;    ! Declare local label
    INT a;
    INT b := 5;
    calc_a :         ! Place label at local statement
    a := b * 2;
    ! Lots of code
    GOTO calc_a;     ! Local branch to local label
END;

```

Nonlocal

NOTE: Nonlocal GOTO statements are inefficient and not recommended.

If the GOTO statement is in a subprocedure and the target statement is in the enclosing procedure, the GOTO statement is nonlocal.

Example 161 Nonlocal GOTO Statement

```

int global_var;
proc p;
begin

```

```

int    i;
label L1;
int subproc s (x);
    int (x);
begin
    label L2:
    if x = 0 then goto L1;    ! Nonlocal goto
    if x > 10 goto L2;        ! Local goto
    return 1;
    L2: return x;
end;
i := s (global_var);
if i <> 1 then goto L1;    ! Local goto
! Processing occurs here
L1:
end;

```

GOTO and Target Statements With Different Trapping States

A GOTO statement, local or nonlocal, must have the same trapping state as its target statement.

Example 162 Local GOTO and Target Statements That Have Different Trapping States

```

proc p nooverflow_traps;
begin
    subproc s overflow_traps;
    begin
        goto L1;    ! Illegal trapping states differ
    end;
L1:
end;

```

If a GOTO statement and the target statement are in different BEGIN-END blocks:

- You must declare the target label in a LABEL declaration in the containing procedure.

NOTE: LABEL is an invalid data type for a formal parameter. You cannot pass a label as an actual parameter.

- Overflow trapping must be enabled in both blocks or disabled in both blocks. The respective overflow trapping states can be established by compiler directive, by procedure attribute, or by BEGIN-END block attribute.
- A GOTO statement in a BEGIN-END block that does not specify a block-level trapping attribute cannot branch to a label in a BEGIN-END block in which a block-level trapping attribute is specified.

The compiler uses attributes on BEGIN-END blocks to determine whether a GOTO within one BEGIN-END block can branch to a label in another BEGIN-END block.

For more information, see [Chapter 13 \(page 234\)](#).

Example 163 Nonlocal GOTO and Target Statements That Have Different Trapping States

```

PROC p OVERFLOW_TRAPS;
BEGIN
    INT i := 0;
    label_a:    ! Overflow traps are enabled at label_a
    i := i + 1;
    IF i < 10 THEN
        GOTO label_a    ! OK: Traps enabled here and at label_a
    ELSE
        BEGIN:ENABLE_OVERFLOW_TRAPS
            GOTO label_a;    ! OK: Branch from block with traps
        END
    END
END

```



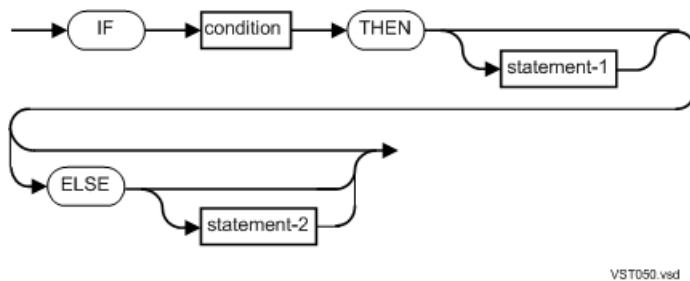
```

    IF i <> 1 THEN ! specified
        BEGIN
            label_b: ...
        END;
    END;
BEGIN:DISABLE_OVERFLOW_TRAPS
    GOTO label_b; ! ERROR: Cannot branch between blocks
END; ! that have different trapping states
BEGIN
    GOTO label_b; ! ERROR: Cannot branch from a BEGIN-END
END; ! block that does not specify a trapping
END; ! attribute to a BEGIN-END block that
      ! does

```

IF

The IF statement conditionally selects one of two statements to execute.



condition

is either:

- A conditional expression whose value has 16 bits
- An INT, INT(32), or FIXED arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

statement-1

specifies the statement to execute if *condition* is true. *statement-1* can be any statement described in this section. If you omit *statement-1*, no action occurs for the THEN clause.

statement-2

specifies the statement to execute if *condition* is false. *statement-2* can be any statement described in this section.

If the *condition* is true, *statement-1* executes. If the *condition* is false, *statement-2* executes. If no ELSE clause is present, the statement following the IF statement executes.

[Example 164 \(page 217\)](#) compares two arrays.

Example 164 IF Statement

```

INT .new_array[0:9];
INT .old_array[0:9];
INT item_ok;
IF new_array = old_array FOR 10 WORDS THEN
    item_ok := 1
ELSE
    item_ok := 0;

```

You can nest IF statements to any level.

Topics:

- [Testing Address Types \(page 218\)](#)
- [Testing Hardware Indicators \(page 218\)](#)

Testing Address Types

An IF statement can test the following as if they were Boolean values:

- Any 16-bit-compatible value:
 - INT
 - STRING
 - UNSIGNED(1-16)
- All address-typed variables except:
 - CBADDR
 - CWADDR
 - PROCADDR
 - PROC32ADDR
 - PROC64ADDR

NOTE: The procedure address types, PROC32ADDR, and PROC64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, [“64-bit Addressing Functionality” \(page 531\)](#).

Testing Hardware Indicators

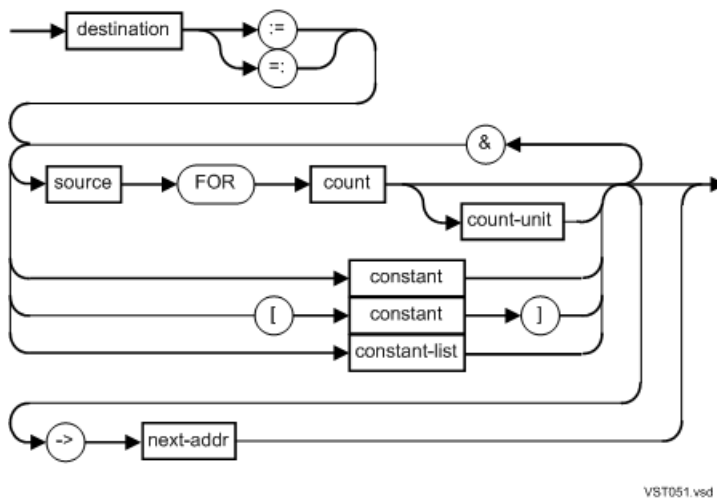
pTAL does not have the hardware indicators—the overflow bit, the carry bit, or the condition codes—that TAL has. Instead, the compiler emits code that supports the \$OVERFLOW, \$CARRY, and condition code test operators (<, >, =, <=, >=, <>, '<', '>', '=', '<=', '>=', '<>').

For more information, see [Chapter 13 \(page 234\)](#).

Move

A move statement copies a block of data from one location in memory to another. You specify the number of bytes, words, or elements to copy in the move statement. With PVU T9248AAD, you can move any variable up to the current maximum allowed size for any object on a TNS/R platform of 127.5 megabytes.

A value parameter cannot be the target of a move statement.



destination

the identifier, with or without an index, of the variable to which the copy operation begins. It can be a simple variable, array, simple pointer, structure, structure data item, or structure pointer, but not a read-only array.

`= ' := '`

specifies a left-to-right sequential move. It starts copying data from the leftmost item in *source*.

`= ' = '`

specifies a right-to-left sequential move. It starts copying data from the rightmost item in *source*.

source

the identifier, with or without an index, of the variable from which the copy operation begins. It can be a simple variable, array, read-only array, simple pointer, structure, structure data item, or structure pointer.

count

is an unsigned INT arithmetic expression that defines the number of units in *source* to copy. If you omit *count-unit*, the units copied (depending on the nature of the *source* variable) are:

Source Variable	Data Type	Units Copied
Simple variable, array, simple pointer (including structure item)	STRING	Bytes
	INT	Words
	INT(32) or REAL	Doublewords
	FIXED or REAL(64)	Quadruplewords
Structure	Not applicable	Words
Substructure	Not applicable	Bytes
Structure pointer	STRING	Bytes
	INT	Words

count-unit

is the value BYTES, WORDS, or ELEMENTS. *count-unit* changes the meaning of *count* from that described previously to the following:

BYTES	Copies <i>count</i> bytes. If both <i>source</i> and <i>destination</i> have word addresses, BYTES generates a word move for $(count + 1) / 2$ words.
WORDS	Copies <i>count</i> words. (WORDS is 16 bits)

ELEMENTS	Copies <i>count</i> elements as follows (depending on the nature of the <i>source</i> variable):		
	Source Variable	Data Type	Units Copied
	Simple variable	STRING	Bytes
	Array	INT	Words
	Simple pointer (including structure item)	INT(32) or REAL FIXED or REAL(64)	Doublewords Quadruplewords
	Structure	Not applicable	Structure occurrences
	Substructure	Not applicable	Substructure occurrences
	Structure Pointer (STRING and INT have meaning only in group comparison expressions and move statements.)	STRING INT	Structure occurrences

If *count-unit* is not BYTES, WORDS, or ELEMENTS, the compiler issues an error. If you specify BYTES, WORDS, or ELEMENTS for *count-unit*, that term cannot also appear as a LITERAL or DEFINE identifier in the global declarations or in any procedure or subprocedure in which the move statement appears.

constant

is a numeric constant, a character string constant, or a LITERAL to copy.

If you enclose *constant* in brackets ([]) and if *destination* has a byte address or is a STRING structure pointer, the system copies *constant* as a single byte regardless of the size of *constant*. If you do not enclose *constant* in brackets or if *destination* has a word address or is an INT structure pointer, the system copies a word, doubleword, or quadrupleword as appropriate for the size of *constant*.

constant-list

is a list of constants to copy. Specify *constant-list* in the form shown in [Chapter 3 \(page 46\)](#).

next-addr

is a variable to contain the location in *destination* that follows the last item copied. The compiler returns a 16-bit, 32-bit, or 64-bit address as described in “Usage Considerations” that follows.

&

is the concatenation operator. It lets you move more than one *source* or *constant-list*, each separated by the concatenation operator.

Usage Considerations

The following rules apply to using MOVE statements:

- A value parameter cannot be the target of a move statement.
- The compiler reports a warning if it can determine that there are more bytes in the source of the move than in the destination of the move (see [Destination Shorter Than Source \(page 222\)](#)).
- Built-in routines, \$FILL8, \$FILL16, and \$FILL32, fill an array with repetitions of the same 8-bit, 16-bit, or 32-bit data, respectively (see [\\$FILL8, \\$FILL16, and \\$FILL32 Statements \(page 222\)](#)).
- If the type of address specified as the *next-addr* is an extended address, it must be able to safely store the address value. For example,

Example 165

```
INT .EXT ea;
INT .EXT64 e64a;
INT .EXT64 n64a;
INT .EXT na;

e64a ':= ' ea FOR 10 words -> @n64a; ! OKAY
e64a ':= ' ea FOR 10 words -> @na;   ! Error: can't store an address
                                     ! of type EXT64ADDR in a variable of type EXTADDR
```

Example 166 (page 221) copies spaces into the first five elements of an array and then uses *next-addr* as *destination* to copy dashes into the next five elements.

Example 166 MOVE Statement Copying to an Array

```
LITERAL len = 10;           ! Length of array
LITERAL num = 5;            ! Number of elements
STRING .array[0:len - 1];   ! Destination array
STRING .next_addr;          ! Next address simple pointer
array[0] ':= ' num * [" "]; -> @next_addr;
! Do first copy and capture next-addr
next_addr ':= ' num * ["-"];
! Use next-addr as start of second copy
```

Example 167 (page 221) contrasts copying a bracketed constant with copying an unbracketed constant. A bracketed constant copies a single byte regardless of the size of the constant. An unbracketed constant copies words, doublewords, or quadruplewords depending on the size of the constant.

Example 167 MOVE Statement Copying Bracketed and Unbracketed Constants

```
STRING x[0:8];
x[0] ':= ' [0]; ! Copy one byte
x[0] ':= ' 0;   ! Copy two bytes
```

Example 168 MOVE Statement Copying From One Structure to Another

```
LITERAL copies = 3;           ! Number of occurrences
STRUCT .s[0:copies - 1];      ! Source structure
BEGIN
    INT a, b, c;
END;
STRUCT .d (s) [0:copies - 1]; ! Destination structure
PROC p;
BEGIN
    d ':= ' s FOR copies ELEMENTS; ! Word move of three
END;                             ! structure occurrences
```

Example 169 MOVE Statement Copying a Substructure

```
LITERAL copies = 3;           ! Number of occurrences
STRUCT .s;
BEGIN
    STRUCT s_sub[0:copies - 1]; ! Source substructure
    BEGIN
        INT a, b;
    END;
END;
STRUCT .d (s);                ! Destination substructure
                                ! is within structure d

PROC p;
```

```

BEGIN
  d.s_sub ':=' s.s_sub FOR copies ELEMENTS;  !Byte move of three
END;                                           ! substructure
                                              ! occurrences

```

Destination Shorter Than Source

The compiler reports a warning when it can detect that there are more bytes in the source of a move than in the destination of the move. For example, if the number of bytes to move is a constant or constant expression whose value is larger than the number of bytes in the destination. The compiler does not report a warning if the destination is:

- A global variable
- A reference parameter
- An array or an element of an array

If the number of bytes to move is a dynamic expression, the compiler reports a warning only if the number of bytes in the source is greater than the number of bytes in the destination. It cannot detect whether the number of bytes to move is too large.

Example 170 MOVE Statement With Destination Shorter Than Source

```

INT g;
INT(32) m;
PROC p( r );
  INT .r;

BEGIN
  FIXED f;
  INT n[0:9];
  INT i;
  g    ':=' f FOR 8 BYTES;  ! OK: g is global
  n    ':=' m FOR 8 BYTES;  ! OK: n is an array
  n[3] ':=' m FOR 8 BYTES;  ! OK: n is an array element
  r    ':=' m FOR 8 BYTES;  ! OK: r is a reference param
  i    ':=' m FOR 8 BYTES;  ! Warning
END;

```

\$FILL8, \$FILL16, and \$FILL32 Statements

pTAL provides the built-in routines \$FILL8, \$FILL16, and \$FILL32, which fill a data area with repetitions of the same 8-bit, 16-bit, or 32-bit value, respectively. This operation is sometimes referred to as a “smear.”

Example 171 FILL16 Statement

```
$FILL16(a, a_size, 0);
```

For more information, see \$FILL8, \$FILL16 and, \$FILL32 in [Chapter 15 \(page 274\)](#).

Variables (including structure data items) are byte addressed or word addressed as follows:

Byte addressed

- STRING simple variables
- STRING arrays
- Variables to which STRING simple pointers point
- Variables to which STRING structure pointers point
- Substructures

Word addressed

- INT, INT(32), FIXED, REAL, or REAL(64) simple variables
 - INT, INT(32), FIXED, REAL, or REAL(64) arrays
 - Variables to which INT, INT(32), FIXED, REAL, or REAL(64) simple pointers point
 - Variables to which INT structure pointers point
 - Structures
-

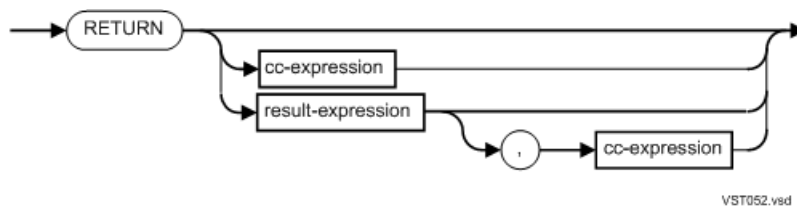
After a move, *next-addr* might point to the middle of an element, rather than to the beginning of the element. If *destination* is word addressed and *source* is byte addressed and you copy an odd number of bytes, *next-addr* will not point to an element boundary.

RETURN

A RETURN statement causes a procedure or function to return control to its caller. When you return from a function, the RETURN statement also specifies a value to return to the function's caller.

NOTE:

- In the discussion of the RETURN statement, the word “procedure” implies both procedures and subprocedures but not functions.
 - The EpTAL compiler issues a warning whenever a pTAL procedure returns both a *result-expression* and a *cc-expression* and has the procedure attribute RETURNSSC (see [Procedure Attributes \(page 248\)](#)). The reason for this warning is in [Appendix D \(page 528\)](#).
-

*cc-expression*

is an INT expression whose numeric value specifies the condition code value to return to the caller:

Value of <i>cc-expression</i>	The condition code is set to ...
Less than 0	Less than (<)
Equal to 0	Equal (=)
Greater than 0	Greater than (>)

Specify *cc-expression* in RETURN statements only in functions and procedures that specify the attribute RETURNSSC (see [Procedure Attributes \(page 248\)](#)).

result-expression

is an arithmetic or conditional expression that a function must return to the caller.

result-expression must be of the same return type as the data type specified in the function header. The data type of a conditional expression is always INT. Specify *result-expression* only when returning from a function.

If *result-expression* is any type except FIXED or REAL(64), a function can return both *result-expression* and *cc-expression*.

Topics:

- [Functions \(page 224\)](#)
- [Procedures and Subprocedures \(page 225\)](#)
- [Condition Codes \(page 225\)](#)

Functions

Every function must include at least one RETURN statement. The compiler does not verify that every path through a function's code includes a RETURN statement; therefore, a function can reach the end of its code without executing a RETURN statement. If this happens, the function returns zero.

Functions that return a condition code that is not based on the value returned by the function must specify explicitly the condition code value to return to the function's caller.

Example 172 RETURN Statements Nested in an IF Statement

```
INT PROC other (nuff, more);  ! Function with return type INT
    INT nuff;
    INT more;
BEGIN
    IF nuff < more THEN          ! IF statement
        RETURN nuff * more      ! Return a value
    ELSE
        RETURN 0;               ! Return a different value
END;
```

Example 173 RETURN Statement That Returns a Value and a Condition Code

```
INT PROC p (i);
    INT i;
BEGIN
    RETURN i, i - max_val;  ! Return a value and a condition code
END;
```

If you call a function, rather than calling it in an expression, you can test the returned condition code, as [Example 174 \(page 224\)](#) does.

Example 174 Testing a Condition Code

```
INT PROC p1 (i);
    INT i;
BEGIN
    RETURN i;
END;

INT PROC p2 (i);
    INT i;
BEGIN
    INT j := i + 1;
    RETURN i, j;
END;
```



```
CALL p1 (i);
IF < THEN ... ; ! Test return value
CALL p2 (i);
IF < THEN ... ; ! Test condition code
```

Procedures and Subprocedures

In procedures and subprocedures that are not functions, a RETURN statement is optional. A nonfunction procedure or subprocedure that returns a condition code value, however, must return to the caller by executing a RETURN statement that includes *cc-expression*.

In a procedure designated MAIN, a RETURN statement stops execution of the procedure and passes control to the operating system.

Procedures that return a condition code must specify explicitly the value of the condition code to return to the procedure's caller. In general, a procedure or subprocedure returns control to the caller when:

- A RETURN statement is executed.
- The called procedure or subprocedure reaches the end of its code.

Example 175 RETURN Statement in a Procedure

```
PROC something;
BEGIN
  INT a,
    b;
  ! Manipulate a and b
  IF a < b THEN
    RETURN;           ! Return to caller
  ! Lots more code
END;
```

The procedure in [Example 176 \(page 225\)](#) returns a condition code that indicates whether an add operation overflows.

Example 176 RETURN Statement in a Procedure That Returns a Condition Code

```
PROC p (s, x, y) RETURNSCC;
  INT .s, x, y;
BEGIN
  INT cc_result;
  INT i;
  i := x + y;
  IF $OVERFLOW THEN cc_result := 1
    ELSE cc_result := 0;
  s := i;
  RETURN cc_result; ! If overflow, condition code is >;
END;               ! otherwise, it is =
```

Condition Codes

A procedure (but not a function) returns a condition code only if the procedure declaration includes the RETURNSCC attribute. The compiler reports an error if a procedure attempts to test the condition code after calling a procedure that does not specify RETURNSCC.

Example 177 Procedure Without RETURNSCC

```
PROC p;
BEGIN
END;
PROC q;
BEGIN
    CALL p;
    IF < THEN ...    ! ERROR: p did not return a condition code
                    ! or a return value
END;
```

Example 178 (page 226) is similar to Example 177 (page 226), but is syntactically correct because p specifies RETURNSCC and returns a condition code value.

Example 178 Procedure With RETURNSCC

```
PROC p RETURNSCC;
BEGIN
    INT i;
    ...

    RETURN i;
END;

PROC q;
BEGIN
    CALL p;
    IF < THEN ...    ! OK: p returns a condition code
END;
```

Functions that do not specify RETURNSCC return a condition code that is based on the numeric value returned by the function, regardless of the nature of the expression in the RETURN statement.

Example 179 Function Without RETURNSCC

```
INT PROC p(i);
    INT i;
BEGIN
    RETURN IF i = 0 THEN -1          ! Returns a condition code
           ELSE IF i = 1 THEN 0      ! based on the value returned
           ELSE 1
END;
```

Functions can return a condition code that is independent of the value returned by the function, as follows:

- The function declaration must specify the RETURNSCC attribute.
- Each RETURN statement in the function must specify the value of the condition code.

Example 180 Function With RETURN SCC

```
INT i;
BEGIN
  INT cc_result;
  ...
  cc_result :=
  IF i < max_val THEN -1
  ELSE
    IF i = max_val THEN 0
    ELSE 1;
  RETURN i, cc_result; ! Return a function value and a
END;                  ! condition code that indicates
                      ! whether the function value is
                      ! less than, equal to, or
                      ! greater than some maximum
```

NOTE: The EpTAL compiler issues a warning whenever a pTAL procedure returns both a traditional function value and a condition code value. For details, see [Appendix D \(page 528\)](#).

A function or procedure that specifies RETURN SCC must include *cc-expression* on every RETURN statement. Conversely, specify *cc-expression* in RETURN statements only in functions and procedures that specify RETURN SCC.

You can test the condition code returned by a function, even if you call the function in a CALL statement.

Example 181 Condition Code Returned by Function Called by CALL Statement

```
INT PROC p1(i);
INT i;
BEGIN
  RETURN i;
END;
INT PROC p2(i) RETURN SCC;
INT i;
BEGIN
  INT j := i + 1;
  RETURN i, j;
END;
CALL p1(1);
IF < THEN ...
CALL p2(1);
IF < THEN ...
```

NOTE: The EpTAL compiler issues a warning whenever a pTAL procedure returns both a traditional function value and a condition code value. For details, see [Appendix D \(page 528\)](#).

Example 182 Condition Code Based on Numeric Value

```
INT PROC p(i);
INT i;
BEGIN
    ...
    RETURN i; ! Return i and set the condition code
END;         ! based on the numeric value of i
```

Example 183 Condition Code That Is Independent of the Function's Value

```
INT PROC p(i) RETURN SCC;
INT i;
BEGIN
    ...
    RETURN i, f(i); ! Return the value of i and set the
END;               ! condition code according to the
                  ! value returned by function f
```

NOTE: The EpTAL compiler issues a warning whenever a pTAL procedure returns both a traditional function value and a condition code value. For details, see [Appendix D \(page 528\)](#).

Example 184 Invalid Function That Attempts to Return an Explicit Condition Code

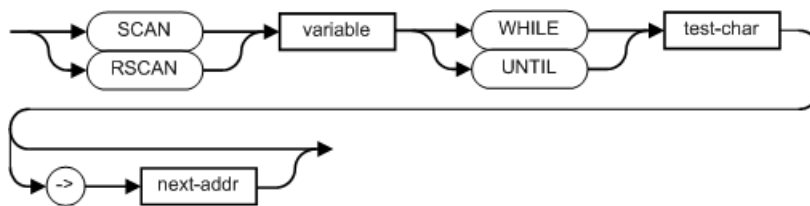
```
INT PROC p(i);
INT i;
BEGIN
    ...
    RETURN i, f(i); ! ERROR: Cannot specify an explicit
END;               ! condition code because procedure
                  ! header does not specify RETURN SCC
```

SCAN and RSCAN

The SCAN and RSCAN statements search a scan area for a test character from left to right or from right to left, respectively.

The scan variable in an RSCAN or SCAN statement can be a pointer declared with the .EXT, .EXT32, or .EXT64 indirection symbol.

NOTE: The “Indirection Symbols” (page 41), .EXT32 and .EXT64 are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).



VST053.vsd

SCAN

indicates a left-to-right search.

RSCAN

indicates a right-to-left search.

variable

is the identifier, with or without an index, of a variable at which to start the scan. The following restrictions apply:

- The variable can be a simple variable, array, read-only array, simple pointer, structure pointer, structure, or structure data item.
- The variable can be of any data type but UNSIGNED.
- The variable cannot have extended indirection.

WHILE

specifies that the scan continues until a character other than *test-char* occurs or until a 0 occurs. A scan stopped by a character other than *test-char* resets \$CARRY. A scan stopped by a 0 sets \$CARRY.

UNTIL

specifies that the scan continues either until *test-char* occurs or until a 0 occurs. A scan stopped by *test-char* resets the hardware carry bit. A scan stopped by a 0 sets the hardware carry bit.

test-char

is an INT arithmetic expression whose value is a maximum of eight significant bits (one byte). A larger value might cause execution errors.

next-addr

is a variable of address type BADDR, SGXBADDR, SGBADDR, EXTADDR, EXT32ADDR, or EXT64ADDR. If the source for the scan uses standard (non-extended) addressing, the next-address variable must have the type BADDR.

NOTE: The address types, EXT32ADDR and EXT64ADDR are available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Delimit the scan area with zeros; otherwise, a scan operation might continue to pass all valid data if either:

- A SCAN UNTIL operation does not find a zero or the test character.
- A SCAN WHILE operation does not find a zero or a character other than the test character.

To delimit the scan area, you can specify zeros as follows:

```
INT .buffer[-1:10] := [0," John James Jones ",0];
```

[Example 185 \(page 229\)](#) converts the word address of an INT array to a byte address. The assignment statement stores the resulting byte address in a STRING pointer. The SCAN statement then scans the bytes in the array until it finds a comma.

Example 185 SCAN UNTIL Statement

```
INT .words[-1:3] := [0,"Doe, J",0];
STRING .byte_ptr := @words[0] '<<' 1;    ! Initialize with byte
                                           ! address of words[0]
SCAN byte_ptr[0] UNTIL ",";              ! Scan bytes in words
```

Topics:

- [Determining What Stopped a Scan \(page 230\)](#)
- [Extended Pointers \(page 230\)](#)
- [Crossing Variable Boundaries \(page 230\)](#)
- [P-Relative Arrays \(page 231\)](#)

Determining What Stopped a Scan

To determine what stopped a scan, test `$CARRY` in an IF statement immediately after the SCAN or RSCAN statement.

Example 186 Determining What Stopped a Scan

```
IF $CARRY THEN ... ;           ! If test character not found
IF NOT $CARRY THEN ... ;       ! If test character found
```

If `$CARRY` is true after a SCAN UNTIL, the test character did not occur. If `$CARRY` is true after SCAN WHILE, a character other than the test character did not occur.

To determine the number of multibyte elements processed, divide (*next-addr* - ' byte address of *identifier*) by the number of bytes per element using unsigned arithmetic.

For more information about `$CARRY`, see [Chapter 13 \(page 234\)](#).

Extended Pointers

Example 187 Extended Pointers in SCAN and RSCAN Statements

```
STRING .EXT eas;
STRING .EXT eat;
EXTADDR ea;
STRING .EXT32 e32as;
STRING .EXT32 e32at;
EXT32ADDR e32a;
STRING .EXT64 e64as;
STRING .EXT64 e64at;
EXT64ADDR e64a;
SCAN eas until " " -> @eat;      ! OK
SCAN eas until " " -> ea;        ! OK
SCAN e32as until " " -> @e32at;  ! OK
SCAN e32as until " " -> e32a;    ! OK
SCAN e64as until " " -> @e64at;  ! OK
SCAN e64as until " " -> e64a;    ! OK
```

NOTE: EXT32ADDR, EXT64ADDR, .EXT32, and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Crossing Variable Boundaries

SCAN and RSCAN statements can access data contained within single named variables or arrays as long as the scan encounters either the target character specified in the SCAN or RSCAN statement or a 0 byte before it reaches the end of the named variable or array.

SCAN and RSCAN statements that depend on accessing data that precedes or follows the variable named in the SCAN or RSCAN statement do not work.

Topics:

- [Data Layout Considerations \(page 231\)](#)
- [Data Passed to Procedures in Reference Parameters \(page 231\)](#)

Example 188 Scanning Adjacent Fields Within a Structure

```
STRUCT s FIELDALIGN(SHARED2);
BEGIN
  STRING buffer[0:99];
  STRING stopper;
END;
BADDR end_addr;
...
s.stopper := 0;
SCAN s.buffer UNTIL char -> end_addr;
IF end_addr = @s.stopper THEN ! Target character was not found
  BEGIN
    ...
  END;
```

Data Passed to Procedures in Reference Parameters

The rules in of the preceding subsection about data layouts apply if the buffer scanned in a SCAN statement is a reference parameter.

P-Relative Arrays

The address type of pointers in a SCAN statement that scans a P-relative array must be CBADDR or CWADDR.

When the SCAN statement in [Example 189 \(page 231\)](#) completes, `t_start` points to the first character in the third message, and `t_end` points immediately after the last character in the third message. The object data type of `t_start` and `t_end` is STRING; therefore, their address type is BADDR.

Example 189 Scanning Data in a P-Relative Array

```
STRING s = 'P' := ! STRING P-relative array s
[ 1, "msg1.",
  2, "msg2.",
  3, "msg3.",
  0 ];
STRING .t_start, ! Start address of msg
      .t_end; ! End address of msg
INT c := 3, ! Value to scan for in s
   z := "."; ! Value to stop the scan
SCAN s UNTIL c -> @t_start; ! Scan s for c and store c's
                           ! address in t_start
@t_start := @t_start '+' 1; ! Skip c
SCAN s[@t_start '-' @s]
  UNTIL z -> @t_end; ! Find end of message
```

In [Example 190 \(page 231\)](#), the data type of `t_start` and of `t_end` is CBADDR. The object data type of `s` is STRING. Its address type is CBADDR, not BADDR; therefore, you can subtract `@s` from `t_start` because the data types of both are CBADDR.

Example 190 Scanning Data in a P-Relative Array

```
STRING s = 'P' := [ ! STRING P-relative array s
  1, "msg1.",
  2, "msg2.",
  3, "msg3.",
  0 ];
CBADDR t_start, ! Start address of msg
      t_end; ! End address of msg
```

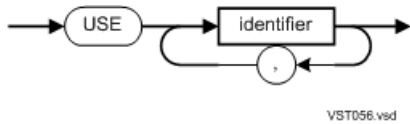
```

INT      c := 3,           ! Value to scan for in s
        z := -1;          ! Value to stop the scan?
SCAN s UNTIL c -> t_start; ! Scan s for c and store c's
                        ! address in t_start
t_start := t_start '+' 1;  ! Skip character in c
SCAN s[t_start '-' @s]
UNTIL z -> t_end;          ! Find end of message

```

USE

The USE statement creates a temporary variable.



identifier

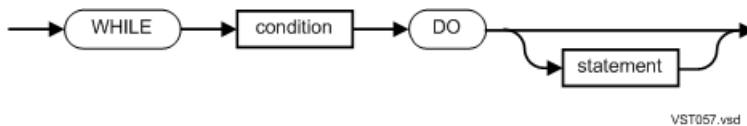
is the name of the temporary variable being created.

A temporary variable that the USE statement creates:

- Is equivalent to a variable declared INT
- Is usually kept in a register
- Exists until either:
 - You drop it with the statement [DROP \(page 212\)](#) (recommended)
 - The procedure that creates it ends

WHILE

The WHILE statement is a pretest loop that repeatedly executes a statement while a specified condition is true.



condition

is either:

- A conditional expression
- An INT, INT(32), or FIXED arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

statement

is any pTAL statement.

The WHILE statement tests the *condition* before each iteration of the loop. If the *condition* is false before the first iteration, the loop never executes.

Hardware indicators cannot appear in the conditional expression of a WHILE statement.

Example 191 WHILE Statement

```
LITERAL len = 100;
INT .array[0:len - 1];
INT item := 0;
WHILE item < len DO
    BEGIN
        array[item] := 0;
        item := item + 1;
    END;
! item equals len at this point
```

The WHILE statement in [Example 192 \(page 233\)](#) increments `index` until a nonalphabetic character occurs.

Example 192 WHILE Statement

```
LITERAL len = 255;
STRING .array[0:len - 1];
INT index := -1;
WHILE (index < len - 1) AND
    ($ALPHA(array[index := index + 1]))
DO ... ;
```

13 Hardware Indicators

Table 56 Hardware Indicators

Hardware Indicator	Representation	Meaning
Overflow bit	\$OVERFLOW	
Carry bit	\$CARRY	
Condition code	< or '<'	Less than
	> or '>'	Greater than
	= or '='	Equal
	<= or '<='	Less than or equal
	>= or '>='	Greater than or equal
	<> or '<>'	Not equal

In TNS architecture, a “hardware indicator” is one of three fields of the environment (ENV) register.

Topics:

- [Managing Overflow Traps \(page 234\)](#)
- [Hardware Indicators After Assignments \(page 236\)](#)
- [Hardware Indicators in Conditional Expressions \(page 239\)](#)
- [Nesting Condition Code Tests \(page 242\)](#)
- [Using Hardware Indicators Across Procedures \(page 244\)](#)

Managing Overflow Traps

pTAL provides a “static T flag” with which you specify whether traps are enabled or disabled at any point in your program. You manipulate the static T flag with:

Directive or Attribute	Scope	Comment
OVERFLOW_TRAPS	Compilation unit	OVERFLOW_TRAPS is the default
[NO]OVERFLOW_TRAPS Procedure Attribute	Procedure or subprocedure	Overrides the [NO]OVERFLOW_TRAPS directive
[EN DIS]ABLE_OVERFLOW_TRAPS Block Attribute	Block	Overrides both the [NO]OVERFLOW_TRAPS directive and the [NO]OVERFLOW_TRAPS procedure attribute

The directives and attributes active when a pTAL statement is compiled determine the overflow trapping state of the code that the compiler generates for that statement. A procedure does not inherit the trapping state of its caller.

[NO]OVERFLOW_TRAPS Procedure Attribute

The OVERFLOW_TRAPS and NOOVERFLOW_TRAPS procedure attributes specify the default overflow trapping behavior for a procedure or subprocedure (see [Procedure Attributes \(page 248\)](#)).

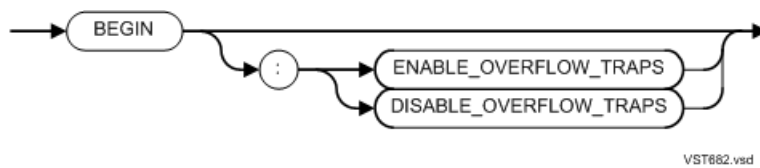
The OVERFLOW_TRAPS and NOOVERFLOW_TRAPS procedure attributes override the current setting of the directive [OVERFLOW_TRAPS \(page 508\)](#).

Example 193 OVERFLOW_TRAPS Compiler Directive and Procedure Attribute

```
?OVERFLOW_TRAPS           ! Enable traps
PROC x NOOVERFLOW_TRAPS;   ! Disable traps for x
BEGIN
    ...
END;
PROC y;                     ! Traps for y are still enabled
BEGIN
    ...
END;
PROC z NOOVERFLOW_TRAPS;   ! Disable traps for z
BEGIN
    SUBPROC s OVERFLOW_TRAPS; ! Enable traps for s
    BEGIN
        ...
    END;
    ...
    s;
    ...                     ! Traps for z are still disabled
END;                       ! upon return from s
```

[EN|DIS]ABLE_OVERFLOW_TRAPS Block Attribute

The ENABLE_OVERFLOW_TRAPS and DISABLE_OVERFLOW_TRAPS block attributes establish the trapping state of a block, regardless of the trapping state of the procedure's or subprocedure's caller or of the code surrounding the block.



Example 194 ENABLE_OVERFLOW_TRAPS and DISABLE_OVERFLOW_TRAPS Block Attributes

```
PROC p;
BEGIN: ENABLE_OVERFLOW_TRAPS    ! Enable traps for block
    ...
END;
PROC q;
BEGIN
    SUBPROC q1;
    BEGIN: DISABLE_OVERFLOW_TRAPS ! Disable traps for block
        ...
    END;
    ...
END;
PROC r;
BEGIN: ENABLE_OVERFLOW_TRAPS
    CALL p;                ! Call p with traps enabled
END;
PROC s;
BEGIN: DISABLE_OVERFLOW_TRAPS
    CALL p;                ! Call p with traps disabled
END;
```

Hardware Indicators After Assignments

Topics:

- [\\$OVERFLOW](#) (page 236)
- [\\$CARRY](#) (page 236)
- [Condition Codes](#) (page 237)

\$OVERFLOW

After every assignment statement, the compiler generates code that tests for overflow if either:

- Overflow traps are enabled
- All of the following conditions are true:
 - Overflow traps are disabled.
 - The root operator is one of the following:
 - Negation (unary -), +, -, *, /, '/'
 - \$DBL of an INT, FIXED, REAL, or REAL(64) value
 - \$DBLR of an INT, FIXED, REAL, or REAL(64) value
 - \$FLTR of a REAL(64) value
 - \$FIX of a REAL or REAL(64) value
 - \$FIXD
 - \$FIXI
 - \$FIXL
 - \$FIXR of a REAL or REAL(64) value
 - \$INT of a FIXED, REAL, or REAL(64) value
 - \$INTR of a FIXED, REAL, or REAL(64) value
 - \$FIXEDTOASCII
 - \$SCALE for which $1 \leq \text{exponent} \leq 4$
 - The next statement is an IF statement that tests \$OVERFLOW.

\$CARRY

You can test \$CARRY if the root operator is one of the following:

- Signed integer addition or subtraction

```
INT i, j, k;
i := j + k;      ! $CARRY can be tested after this statement
i := j - k;      ! $CARRY can be tested after this statement
```
- Unsigned integer addition or subtraction

```
INT i, j, k;
I := j '+' k;    ! $CARRY can be tested after this statement
I := j '-' k;    ! $CARRY can be tested after this statement
```
- Unary minus

```
INT i;
i := -i;         ! $CARRY can be tested after this statement
```

Condition Codes

When the condition code is accessible following an assignment statement, the numeric value of the evaluated expression on the right side of the assignment statement determines the value of the condition code.

Topics:

- [When Condition Codes Are Accessible \(page 237\)](#)
- [Typed Integer Constants \(page 44\)](#)

When Condition Codes Are Accessible

The condition code is accessible after an assignment statement only if:

- The right side of the assignment statement is not:
 - A 1-byte item:

```
STRING s;  
INT i;  
i := s;    ! Condition code is not accessible
```
 - A call to a built-in routine (for a list of these, see [Table 58 \(page 276\)](#)):

```
I := $ABS(i);    ! Condition code is not accessible
```
 - An expression whose value is an address type (for example, WADDR or EXTADDR):

```
INT i;  
INT .p;  
@p := @i;    ! Right side is a WADDR value;  
              ! condition code is not accessible
```
 - An expression whose value is a floating-point data type [REAL or REAL(64)]:

```
REAL r := 1.0E0;  
r := r + 1.0E0;    ! Right side is a floating-point number;  
                   ! condition is code not accessible
```
 - A constant or constant expression:

```
INT a;  
a := 2 << 3;    ! Right side is a constant expression;  
                ! condition is code not accessible
```
- None of the exceptions in [When Condition Codes Are Not Accessible](#) apply

When Condition Codes Are Not Accessible

The following exceptions override the conditions in [When Condition Codes Are Accessible \(page 237\)](#):

- If the last operation on the right side of an assignment statement is a function call, these rules apply:
 - If the function specifies the RETURNSCC attribute, you can test the condition code following the assignment statement, independent of the data type of the value returned by the function.
 - The numeric value returned by the function always determines the value of the condition code.
 - If the right side of an assignment statement is an INT function, the condition code is determined by the INT value returned by the function. The value returned is always an INT, even if the expression in the function's RETURN statement is a byte value. The byte value is not sign-extended.

```
INT PROC p;  
BEGIN
```

```

    RETURN "A";    ! P is an INT function but the
END;              ! expression in the RETURN statement
                  ! yields a single byte

BEGIN
PROC p1;
    INT i;
    i := p;        ! Condition code is accessible because
END;              ! p returns an INT value

```

The left side of the assignment statement must be one of:

- A local or sublocal simple variable:

```

PROC p;
BEGIN
    INT i;          ! Declare a local simple variable
    i := i + 1;     ! Condition code accessible
END;

```

- The address cell of a local pointer:

```

INT i;
INT .ptr;          ! Declare a local pointer
@ptr := f(i);      ! Condition code is accessible if
                  ! f specifies RETURNSCC

```

- A value parameter:

```

PROC p (param);
    INT param;      ! Value parameter
BEGIN
    INT i := 0;
    param := i;     ! Condition code is accessible
END;

```

- The left side of the assignment statement cannot be:

- A STRING variable:

```

STRING s;
s := "a";          ! Condition code is not accessible

```

- An UNSIGNED(*n*) variable:

```

UNSIGNED(12) u;
u := %HFFF;        ! Condition code is not accessible

```

- A global variable:

```

INT g;
PROC p;
BEGIN
    INT i := 0;
    g := i;         ! Condition code is not accessible
END;

```

- A pointer:

```

INT .p;
INT i := 0;
p := i;            ! Condition code is not accessible

```

- A variable containing indexing, field selection, or bit selection:

```

STRUCT s;
BEGIN
    INT f;
END;

INT a[0:9];
INT i;
s.f := i;          ! Field selection: condition code is

```

```

                                ! not accessible
a[9] := i;                      ! Index: condition code is not accessible
i.<3:5> := a;                    ! Bit Selection: condition code is
                                ! not accessible

```

Example 195 Assignments After Which You Can Test Condition Codes

```

INT m;

PROC p(x, y);
  INT    x;
  INT    .y;
BEGIN
  INT    a[0:9];
  INT    i;
  INT .EXT k;
  INT(32) j;
  STRING str;
  REAL   r;
  EXTADDR SUBPROC f(x) RETURNSCC;
  INT x;
BEGIN
  RETURN %200000D, x;
END;

STRUCT s;
BEGIN
  INT s1[0:4];
  INT s2;

END;

i := k;          ! OK: Left and right sides are simple
i := i + 1;      ! OK: Left and right sides are simple
x := x + 1;      ! OK: Left side is value parameter,
                ! right side is INT
@k := f(0);      ! OK: Left side is pointer cell,
                ! right side is RETURNSCC function
y := i + 1;      ! ERROR: Left side is reference parameter
a[i] := a[i+1];  ! ERROR: Left side has index
s.s1[0] := i;    ! ERROR: Left side has field and index
s.s2 := i;       ! ERROR: Left side has field reference
i.<0:8> := i;     ! ERROR: Left side has bit selection
i := str;        ! ERROR: Right side is 1-byte item
i := $ABS(i);    ! ERROR: Right side is call to
                ! built-in routine
@k := @k + 1D;   ! ERROR: Right side is address type
r := r + 1.0E0; ! ERROR: Right side is floating-point type
END;

```

Hardware Indicators in Conditional Expressions

Hardware indicators can appear in conditional expressions in these statements:

- [DO-UNTIL \(page 210\)](#)

The last statement in the DO-UNTIL statement must set the hardware indicator. The last statement can be nested in a BEGIN...END statement. See [Example 196 \(page 240\)](#).

- [IF \(page 217\)](#)

These are valid references to hardware indicators:

```

IF $OVERFLOW THEN ...
IF $CARRY THEN ...
IF < THEN ...

```

- **WHILE (page 232)**

Both the statement preceding the WHILE statement and the last statement in the WHILE statement must set the condition code indicator. See [Example 197 \(page 240\)](#).

Example 196 Hardware Indicators in DO-UNTIL Statements

```

proc p returnscc;
begin
    ...
end;

proc q;
begin
    ...
end;

do
    call p ()          ! Sets condition code indicator
until = ;             ! OK

do
    begin
        ...
        call p ();
    end
until = ;             ! OK

do
    begin
        ...
        call q ();
    end
until > ;             ! ERROR: last statement in do-until statement
                    ! does not set condition indicator

int i := 0;
...
do
    begin
        ...
        i := i + 1;
    end
until $overflow;      ! ERROR: $overflow and $carry not allowed
                    ! in do-until statement

```

Example 197 Hardware Indicators in WHILE Statements

```

int proc p;
begin
    ...
end;

proc q;
begin
    ...
end;

call p ();            ! Sets condition code indicator
while >= do           ! OK

call p ();            ! Sets condition code indicator

```



```

call p ();          ! Sets condition code indicator

while > do          ! OK
  begin
    ...
    call p ();      ! Sets condition code indicator
  end;

call q ();          ! Doesn't set the condition code indicator
while > do          ! ERROR: statement preceding WHILE
  begin            ! and last statement of WHILE
    ...            ! must both set condition code indicator
    call p ();      ! Sets condition code indicator
  end;

call p ();          ! Sets the condition code indicator
while >= do         ! ERROR: statement preceding WHILE
  begin            ! and last statement of WHILE
    ...            ! must both set condition code indicator
    call q ();      ! Doesn't set condition code indicator
  end;

int i;
...
i := i + 1;

while not $overflow do ! ERROR: not a condition code indicator
begin
  ...
  i := i + 1;
end;

```

You cannot:

- Reference a hardware indicator in an expression other than in the conditional expression of an IF statement

```

INT i;
i := IF < THEN -i ELSE i; !ERROR: invalid in IF expression

```

- Assign the value of a hardware indicator to a variable in an assignment statement

```

INT i;
i := >; ! ERROR: invalid in assignment statement

```

- Pass a hardware indicator as an actual parameter to a procedure

```

INT i;
CALL p( < ); ! ERROR: invalid as parameter

```

An IF statement that tests a hardware indicator must either:

- Immediately follow the statement that establishes the value of the hardware indicator

```

INT a;
a := a - 1;
IF < THEN ... ! OK: hardware indicator tested immediately
a := a + 1;
IF $CARRY THEN ... ! OK: hardware indicator tested immediately
CALL WRITEREAD(...);
IF <> THEN ... ! OK: hardware indicator tested immediately
a := a - 1;
BEGIN
  IF < THEN ... ! ERROR: intervening BEGIN is invalid
  ...

```

```

    a := a + 1;
END;
IF $CARRY THEN ... ! ERROR: intervening END is invalid
CALL WRITEREAD(...);
firstchar := str_buff;
IF <= THEN... ! ERROR: intervening assignment
! statement is invalid
CALL WRITEREAD(...);
IF < THEN ... ! ERROR: previous statement does not
! set condition code

```

- Be part of a nest of IF statements as described in [Nesting Condition Code Tests \(page 242\)](#)

The hardware indicator in the conditional expression of an IF statement must be the first operand in the expression.

```

IF $CARRY THEN ... ! OK: hardware indicator is
! first operand
IF <= OR a >= 99 THEN ... ! OK: hardware indicator is
! first operand
IF I <= 999 AND > THEN ... ! ERROR: condition code must be
! first operand
IF a = b OR $CARRY THEN ... ! ERROR: $CARRY must be
! first operand
IF a = b OR $OVERFLOW THEN ... ! ERROR: $OVERFLOW must be
! first operand

```

The first statement in an IF statement's THEN clause or ELSE clause (or both) can, in turn, be an IF statement that tests the condition code established by the conditional expression of the containing IF statement. In this case, the root operator in the containing IF statement's conditional expression must be either:

- A relational operator

```

I := i + 1;
IF i >= 0 THEN ! OK: >= is a relational operator
    IF > THEN ...

```

- An expression that consists only of a condition code

```

I := i + 1;
IF >= THEN ! OK: >= is a condition code
    IF > THEN...

```

An IF statement that tests a hardware indicator cannot be labeled.

Nesting Condition Code Tests

You can test for more than one value of a condition code by nesting IF statements; for example:

```

I := i + 1;
IF < THEN
    ...
ELSE IF = THEN
    ...
    ELSE ! Must be >
    ...
INT PROC p;

BEGIN
    CALL READX( ... );
    IF < THEN RETURN -1
    ELSE IF > THEN RETURN 1
        ELSE RETURN 0;
END;

```

The following rules apply to nested IF statements:

- Neither \$OVERFLOW nor \$CARRY can appear in the conditional expression of any IF statement in a nest of IF statements.

```
I := i + 1;
IF > THEN
    IF $OVERFLOW THEN ... ! ERROR: cannot test $OVERFLOW in
                           ! nest of IF statements
i := i + 1;
IF $CARRY THEN           ! ERROR: cannot test $CARRY in
    IF > THEN ...       ! nest of IF statements
```

You cannot test \$OVERFLOW or \$CARRY to determine if an overflow or carry occurred while evaluating an IF statement's conditional expression.

```
IF i + 1 < 100 THEN
    BEGIN
        IF $CARRY THEN ... ! ERROR: invalid to test $CARRY here
    END
```

You can test \$OVERFLOW or \$CARRY by evaluating, in a separate assignment statement, the expression in which overflow or carry could occur, then test \$OVERFLOW or \$CARRY.

```
INT temp;
temp := i + 1;           ! Carry could occur here
IF NOT $CARRY THEN       ! OK to test $CARRY here
    BEGIN
        IF temp < 100 THEN ...
    END
ELSE ...                 ! Handle $CARRY condition
```

- Except as noted in the following item, the conditional expression in each IF statement in a nest of IF statements can test only the value of the condition code, optionally preceded by the NOT operator. The conditional expression cannot include any other operator or operand.

```
I := i + 1;
IF <= THEN
    IF NOT = THEN ... ! OK
```

- The conditional expression of the innermost IF statement can be a complex expression, but the condition code must be the first operand in the expression.

```
I := j + 1;
IF >= THEN
    IF = AND (j + 4) / 5 * 5 > 0 THEN ... ! OK
```

- If the root operator in the conditional expression of an IF statement is a relational operator, the first statement in the THEN or ELSE clause of the IF statement can be an IF statement that tests the condition code set by the root operator of the encompassing IF statement.

```
IF (i + 10) <= (m - 2) THEN ! Root operator (<=) is
    BEGIN                  ! relational operator
        IF < THEN          ! OK: test if condition was
            ...             ! "less than"
        ELSE
    END;
```

```
IF (i < -1) OR (i > 1) THEN
    BEGIN
        IF < THEN          ! ERROR: root operator of IF
            ...             ! statement is Boolean,
        ELSE                ! not relational
    END;
```

- If an outer IF statement's conditional expression uses a signed operator (= or <>) to compare two 16-bit addresses, an inner IF statement's THEN or ELSE clause cannot test the condition code established by the outer IF statement's conditional expression.

```

WADDR    w1, w2;
EXTADDR  e1, e2;
IF e1 <> e2 THEN
    BEGIN
        IF < THEN ...      ! OK: e1 and e2 are EXTADDR values
    END;

IF w1 '<>' w2 THEN
    BEGIN
        IF < THEN ...      ! OK: Original test is unsigned
    END;

IF w1 <> w2 THEN
    BEGIN
        IF < THEN ...      ! ERROR: cannot test condition code
                           ! set by signed comparison of
                           ! 16-bit addresses

```

Using Hardware Indicators Across Procedures

Topics:

- Testing a Hardware Indicator Set in the Calling Procedure (page 244)
- Returning a Condition Code to the Calling Procedure (page 244)
- Returning the Value of \$OVERFLOW or \$CARRY to the Calling Procedure (page 245)

Testing a Hardware Indicator Set in the Calling Procedure

A called procedure cannot test the value of a hardware indicator that was set in the procedure that called the hardware indicator. To achieve this effect:

1. In the calling procedure:
 - a. Test the value of the hardware indicator and set a variable to reflect its value.
 - b. Pass the variable to the called procedure.
2. In the called procedure, test the variable that you passed to the procedure in [FIX_THIS_LINK](#).

Example 198 Testing a Hardware Indicator Set in a Calling Procedure

```

PROC b(status);                ! Called procedure
    INT status;
BEGIN
    IF status <> 0 THEN ...      ! Test parameter value from PROC a
END;

PROC a;                        ! Calling procedure
BEGIN
    INT i, j, k;
    ...
    j := i;
    IF <> THEN k := 1           ! Test hardware indicator and set k
    ELSE k := 0;k
    CALL b(k);                 ! Call PROC b, passing k
END;

```

Returning a Condition Code to the Calling Procedure

A called procedure can return a condition code value to its caller by using the RETURNSCC procedure attribute in its procedure or subprocedure declaration and a RETURN statement.

For more information:

Topic	Source
Procedure declarations	Procedure Declarations (page 246)
Subprocedure declarations	Subprocedure Declarations (page 257)
RETURNSCC procedure attribute	Procedure Attributes (page 248)
RETURN statement	RETURN (page 223)

Returning the Value of \$OVERFLOW or \$CARRY to the Calling Procedure

A called procedure cannot return the value of \$OVERFLOW or \$CARRY to its caller. To achieve this effect, set variables with the values of these indicators and return the variables' values using either parameters, global variables, or return values.

Example 199 Returning the Value of \$OVERFLOW in a Reference Parameter

```
PROC p;                                ! Calling procedure
BEGIN
  INT rtn_ovfl;
  CALL q(rtn_ovfl);                    ! q returns rtn_ovfl
  IF rtn_ovfl = 0 THEN ...             ! Test value of rtn_ovfl
END;
PROC q(ovfl);                          ! Called procedure
  INT .ovfl;
BEGIN
  INT i := 32767;
  i := i + 1;
  IF $OVERFLOW THEN                   ! Test hardware indicator and
    ovfl := 1                         ! set ovfl
  ELSE
    ovfl := 0;
END;                                  ! Return ovfl to caller
```

14 Procedures, Subprocedures, and Procedure Pointers

Procedures are program units that contain the executable portions of a pTAL program and that are callable from anywhere in the program. Procedures allow you to segment a program into discrete parts that each perform a particular task such as I/O or error handling.

An executable program contains at least one procedure. One procedure in the program has the attribute MAIN, which identifies it as the first procedure to execute when you run the program.

A procedure can contain subprocedures, which are callable from various points within the same procedure.

A function is a procedure or subprocedure that returns a value. A function is also known as a typed procedure or typed subprocedure.

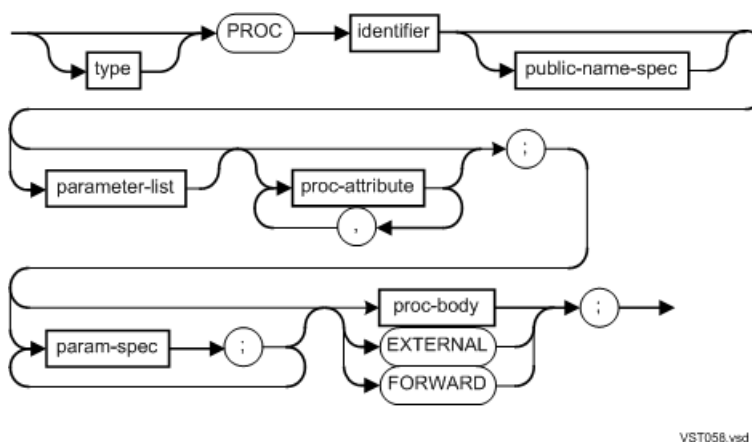
Topics:

- [Procedure Declarations \(page 246\)](#)
- [Procedure Attributes \(page 248\)](#)
- [Formal Parameter Specification \(page 251\)](#)
- [Procedure Body \(page 256\)](#)
- [Subprocedure Declarations \(page 257\)](#)
- [Subprocedure Body \(page 259\)](#)
- [Entry-Point Declarations \(page 260\)](#)
- [Procedure Pointers \(page 263\)](#)
- [Labels in Procedures \(page 273\)](#)

In this section, references to procedures refers to procedures and subprocedures unless otherwise specified.

Procedure Declarations

A procedure is a program unit that is callable from anywhere in the program. You declare a procedure as follows:



type

specifies that the procedure is a function that returns a result and indicates the data type of the returned result. *type* can be any data type described in [Chapter 3 \(page 46\)](#).

identifier

is the procedure identifier to use in the compilation unit.

public-name-spec



VST209.vsd

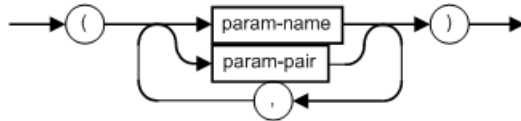
If a procedure declaration includes *public-name-spec*, it must also include [EXTERNAL](#).

If a procedure declaration includes [LANGUAGE](#), it must also include *public-name-spec*.

public-name

is the procedure name to use in the linker, not in the compilation unit. The default *public-name* is identifier. *public-name* must conform to the identifier rules of the language in which the external procedure is written. For all languages except HP C, the compiler upshifts *public-name* automatically.

parameter-list



VST210.vsd

param-name

is the identifier of a formal parameter. A procedure can have up to 32 formal parameters.

param-pair

is a pair of formal parameter identifiers that comprise a language-independent string descriptor in the form:



VST039.vsd

string

is the identifier of a standard or extended STRING simple pointer. The actual parameter is the identifier of a STRING array or simple pointer declared inside or outside a structure.

length

is the identifier of a directly addressed INT simple variable. The actual parameter is an INT expression that specifies the length of *string*, in bytes.

proc-attribute

is a procedure attribute, as described in [Procedure Attributes \(page 248\)](#).

param-spec

specifies the parameter type of a formal parameter and whether it is a value or reference parameter, as described in [Formal Parameter Specification \(page 251\)](#).

proc-body

is a BEGIN-END block that contains local declarations and statements, as described in [Procedure Body \(page 256\)](#).

FORWARD

specifies that the procedure body is declared later in the compilation.

EXTERNAL

specifies that the procedure body is either declared in another compilation unit or later in this compilation unit.

Example 200 Procedure Declaration

```
INT var;                                ! var is a global INT
WADDR PROC p(i), RETURNSCC,;           ! Attributes: empty, RETURNSCC,
                                      ! and empty

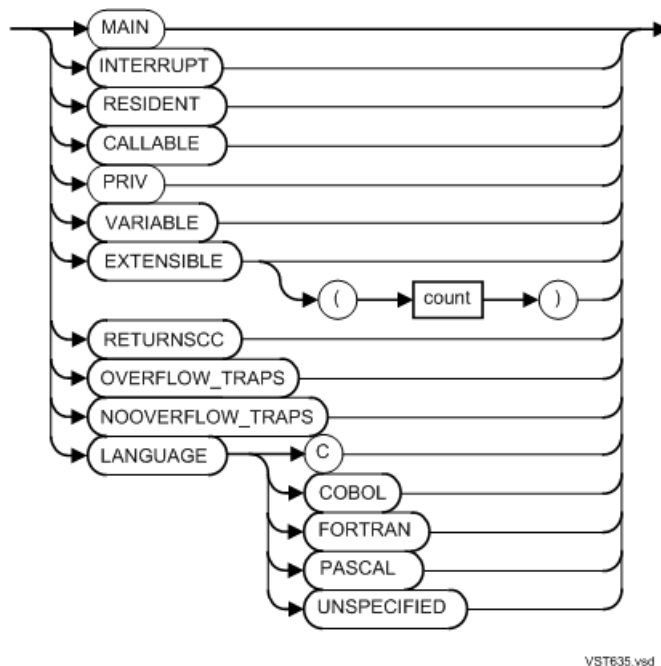
INT .i;
BEGIN
    RETURN @var, i+1;                  ! Return address and
END;                                   ! condition code value
```

Example 200 (page 248) illustrates the following procedure declarations:

- `p` specifies three attributes, the first and third of which are empty.
- The second attribute to `p`, `RETURNSCC`, is a valid procedure, subprocedure, or function attribute, which, if present, requires that the code execute a `RETURN` statement that specifies a value from which to determine the condition code to return to the caller. For more information about using `RETURNSCC`, see [RETURN \(page 223\)](#).
- The data type of the value returned by `p` is `WADDR`: namely, the address of the global variable `var`. The `RETURN` statement sets the condition code to `CCL`, `CCE`, or `CCG`, depending on whether the value of `i+1` is less than, equal to, or greater than 0.

Procedure Attributes

Procedures can have the following attributes:



MAIN

causes the procedure to execute first when you run the program. When the MAIN procedure completes execution, it passes control to the `PROCESS_STOP_` system procedure, rather than executing an `EXIT` instruction.

If more than one procedure in a compilation has the MAIN attribute, the compiler emits a warning and uses the first main procedure it sees as the main procedure. For example, in the following source code, procedures `main_proc1` and `main_proc2` have the MAIN attribute, but in the object file only `main_proc1` has the MAIN attribute:

```
PROC main_proc1 MAIN;    ! This MAIN procedure is MAIN
BEGIN                  ! in the object file
    CALL this_proc;
```



```

    CALL that_proc;
END;
PROC main_proc2 MAIN;    ! This MAIN procedure is not MAIN
BEGIN                    ! in the object file
    CALL some_proc;
END;

```

INTERRUPT

causes the pTAL compiler to generate an interrupt exit instruction instead of an EXIT instruction at the end of execution. Only operating system interrupt handlers use the INTERRUPT attribute. An example is:

```

PROC int_handler INTERRUPT;
BEGIN
    ! Do some work
END;

```

NOTE: The EpTAL compiler ignores INTERRUPT.

RESIDENT

causes procedure code to remain in main memory for the duration of program execution. The operating system does not swap pages of this code. The linker allocates storage for RESIDENT procedures as the first procedures in the code space. An example is:

```

PROC res_proc RESIDENT;
BEGIN
    ! Do some work
END;

```

CALLABLE

authorizes a procedure to call a PRIV procedure (described next). Nonprivileged procedures can call CALLABLE procedures, which can call PRIV procedures. Thus, nonprivileged procedures can only access PRIV procedures indirectly by first calling CALLABLE procedures. Normally, only operating system procedures have the CALLABLE attribute. In the following example, a CALLABLE procedure calls the PRIV procedure declared next:

```

PROC callable_proc CALLABLE;
BEGIN
    CALL priv_proc;
END;

```

PRIV

means the procedure can execute privileged instructions. Only PRIV or CALLABLE procedures can call a PRIV procedure. Normally, only operating system procedures have the PRIV attribute. PRIV protects the operating system from unauthorized (nonprivileged) calls to its internal procedures.

The following PRIV procedure is called by the preceding CALLABLE procedure:

```

PROC priv_proc PRIV;
BEGIN
    ! Privileged instructions
END;

```

For information about privileged mode, see [Privileged Mode \(page 274\)](#).

VARIABLE

means the compiler treats all parameters of the subprocedure as if they are optional, even if some are required by your code. If you add parameters to the VARIABLE subprocedure declaration, all procedures that call it must be recompiled. The following example declares a VARIABLE subprocedure:

```

SUBPROC v (a, b) VARIABLE;
    INT a, b;
BEGIN

```

```
! Lots of code
END;
```

When you call a VARIABLE subprocedure, the compiler allocates space in the parameter area for all the parameters. The value of the data for a missing parameter is unspecified.

EXTENSIBLE

lets you add new parameters to the procedure declaration without recompiling its callers. The compiler treats all parameters of the procedure as if they are optional, even if some are required by your code. The following example declares an EXTENSIBLE procedure:

```
PROC x (a, b) EXTENSIBLE;
    INT a, b;
BEGIN
    ! Do some work
END;
```

When you call an EXTENSIBLE procedure, the compiler allocates space in the parameter area for all the parameters. The values of missing parameters are unspecified.

Declare procedures EXTENSIBLE, but not subprocedures.

count

converts a VARIABLE procedure to an EXTENSIBLE procedure. The *count* value is the number of formal parameters in the VARIABLE procedure that you are converting to EXTENSIBLE. For the *count* value, specify an INT value in the range 1 through 15.

RETURNSCC

causes a procedure to return a condition code. The compiler reports an error if a procedure attempts to test the condition code after calling a procedure that does not specify RETURNSCC. Procedures declared with RETURNSCC cannot return 64-bit values.

NOTE: The EpTAL compiler issues a warning if a procedure that has this attribute returns both a traditional function value and a condition code value by means of [RETURN \(page 223\)](#). The reason for this warning is described in [Appendix D \(page 528\)](#).

OVERFLOW_TRAPS

enables overflow traps for a procedure.

NOOVERFLOW_TRAPS

disables overflow traps for a procedure.

LANGUAGE

specifies that the external routine is an HP C, HP COBOL, FORTRAN, or Pascal routine. If you do not know if the external routine is an HP C, HP COBOL, FORTRAN, or Pascal routine, use LANGUAGE UNSPECIFIED. The following example shows the LANGUAGE COBOL option and a public name "a_proc" (in HP COBOL identifier format):

```
PROC a_proc = "a-proc" (a, b, c) ! EXTERNAL declaration for
LANGUAGE COBOL;                  ! HP COBOL procedure
    STRING .a, .b, .c;
EXTERNAL;
```

Specify no more than one LANGUAGE attribute in a declaration.

Because no FORTRAN or Pascal compilers exist especially for TNS/R or TNS/E architecture, LANGUAGE FORTRAN and LANGUAGE PASCAL have no meaning on TNS/R or TNS/E architecture.

If a procedure declaration includes LANGUAGE, it must also include *public-name-spec*.

Parameters and VARIABLE and EXTENSIBLE Procedures

To determine which parameters were passed by the caller, use the [\\$PARAM \(page 336\)](#).

Memory is allocated for all parameters to VARIABLE procedures or EXTENSIBLE procedures; therefore, your program can store default values for parameters the caller does not pass.

VARIABLE, EXTENSIBLE and RETURNSCC Procedures as Actual Parameters

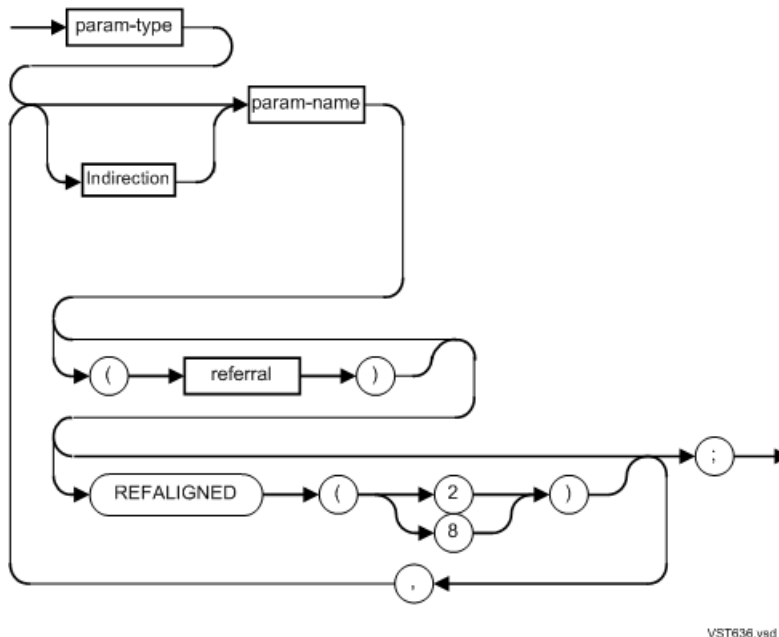
You can pass a procedure or procedure pointer that includes an EXTENSIBLE, VARIABLE, or RETURNSCC attribute as a parameter to a procedure whose formal parameter is a PROC, but you cannot reference the PROC formal parameter identifier in a CALL statement. Instead, you must assign the address from the formal parameter to a procedure pointer and then specify the procedure pointer in a CALL statement.

Example 201 EXTENSIBLE Procedures as Actual Parameters

```
PROC p1 (i, j) EXTENSIBLE;
  INT i, j;
EXTERNAL;
PROC p2( p );
PROC p;
BEGIN
  PROCPTR pp(a, b) EXTENSIBLE; INT a, b; END PROCPTR;
  INT i, j;
  ...
  pp := pi;
  CALL pp(i, j);
END;
PROC p3;
BEGIN
  CALL p2(p1);
END;
```

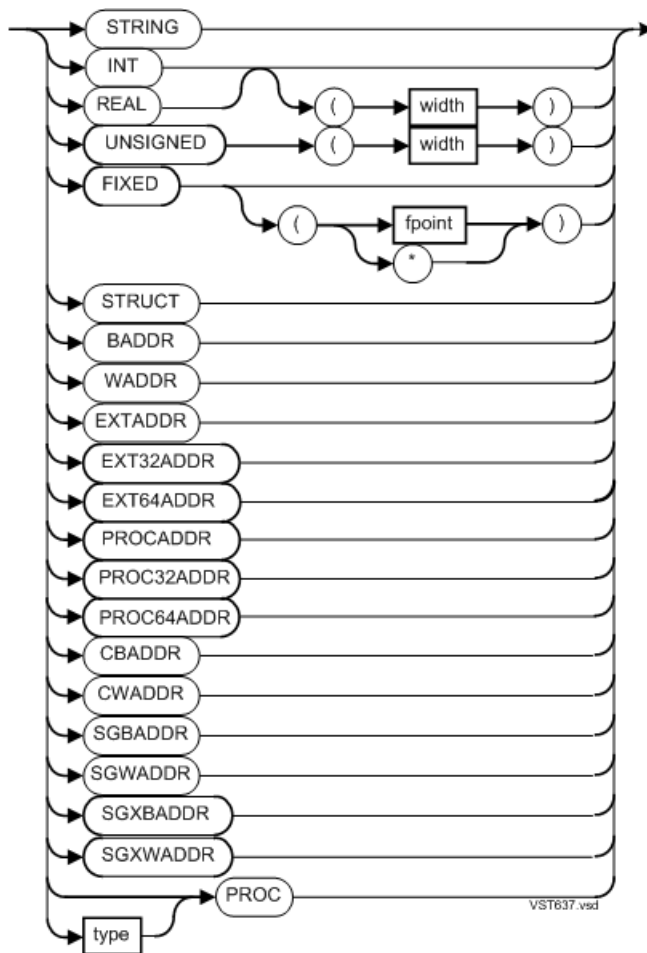
Formal Parameter Specification

A formal parameter specification defines the parameter type of a formal parameter and whether the parameter is a value parameter or a reference parameter.



param-type

is the parameter type of the formal parameter and can be one of the following:



Descriptions for STRUCT, PROC, PROC(32), and *type*, are included below. You can find descriptions of the remaining data types in [Chapter 3 \(page 46\)](#).

STRUCT

means the parameter is one of:

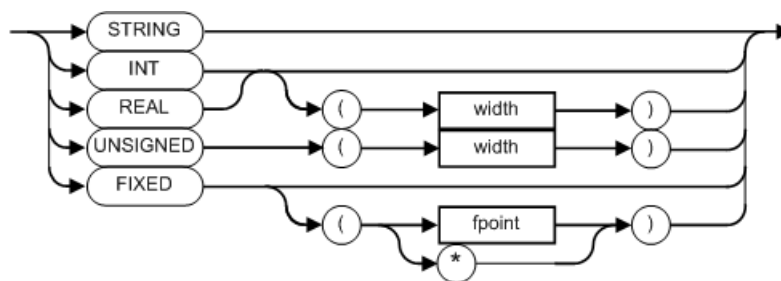
- A standard indirect or extended indirect definition structure (not supported in future software platforms)
- A standard indirect or extended indirect referral structure

PROC

is the address of the entry point of a procedure. You must assign PROC to a PROCPTR before you can call it.

type

specifies that the parameter is a function procedure, the return value of which is one of the following data types:



width

is a constant expression that specifies the number of bits in the variable. The result of the constant expression must be one of the following values:

Data Type	width
INT	16, 32, or 64
REAL	32 or 64
UNSIGNED	A value in the range 1 through 31

UNSIGNED parameters must be passed by value; you cannot use an indirection symbol (see [Table 14 \(page 41\)](#)) with UNSIGNED parameters.

fpoint

is an integer in the range -19 through 19 that specifies the implied decimal point position. The default is 0 (no decimal places). A positive *fpoint* specifies the number of decimal places to the right of the decimal point. A negative *fpoint* specifies a number of integer places to the left of the decimal point.

*

prevents scaling of the *fpoint* of a FIXED actual parameter to match the *fpoint* in the parameter specification. Such scaling might cause loss of precision. The called procedure treats the actual parameter as having an *fpoint* of 0.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

NOTE: “Indirection Symbols” (page 41), .EXT32 and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

param-name

is the identifier of a formal parameter. The identifier has local scope if declared in a procedure or sublocal scope if declared in a subprocedure.

referral

is the identifier of a previously declared structure or structure pointer. The diagram under *param-type* describes lists the kind parameter requiring a *referral*.

REFALIGNED

For simple pointers, the default for REFALIGNED is the value you specify in the [REFALIGNED \(page 510\)](#).

2

specifies that the variables and structures that identifier references are aligned as they would be aligned in TAL (and might not be well-aligned in pTAL).

8

specifies that the variables and structures are well-aligned in pTAL (and in TAL, that they might have more space).

For nonstructure pointers, the default for REFALIGNED is the value you specify in the [REFALIGNED \(page 510\)](#).

When a procedure is called, each actual parameter is bound to its corresponding formal parameter. Parameters passed by value must follow the same rules for assignment compatibility as do assignment statements. Each actual value parameter corresponds to the right side of an assignment statement.

[Table 57 \(page 254\)](#) lists the characteristics that you can declare in a formal parameter specification depending on the kind of actual parameter the procedure or subprocedure expects.

Table 57 Formal Parameter Specification

Expected Actual Parameter	Formal Parameter Characteristics			
	Declare Formal Parameter As:	Parameter Type	Indirection Symbol	Referral
Simple variable	A value or reference parameter	STRING* INT INT(32) REAL REAL(64) FIXED(<i>n</i>) FIXED(*)	Value, no; reference, yes	No
Simple variable	A value parameter	UNSIGNED	No	No
Array or simple pointer	A reference parameter	STRING INT INT(32) REAL REAL(64) FIXED(<i>n</i>)	Yes	No
Definition structure, referral structure, or structure pointer	A reference parameter	INT or STRING	Yes	Yes
Referral structure or structure pointer	A reference parameter	STRUCT	Yes	Yes
Constant expression** (including <i>@identifier</i>)	A value parameter	INT INT(32) UNSIGNED REAL REAL(64) FIXED(<i>n</i>)	No	No
Procedure	A value parameter	PROC PROC(32)	No	No

* You cannot declare a STRING value parameter. The compiler reports a syntax error if you declare a STRING value parameter.

** The data type of the expression and of the formal parameter must match, except that you can mix the STRING, INT, and UNSIGNED (1-16) data types, and you can mix the INT(32) and UNSIGNED(17-31) data types.

Any of the 13 address types can be used as formal parameters.

In [Example 202 \(page 255\)](#), the compiler treats `var1` as if it were a simple variable and treats `var2` as if it were a simple pointer.

Example 202 Function With Value and Reference Formal Parameters

```
PROC mult (var1, var2);
    INT var1,          ! Value parameter
      .var2;          ! Reference parameter
BEGIN
    var2 := var2 + var1; ! Manipulate parameters
END;
```

Example 203 Reference Structure as a Formal Reference Parameter

```
STRUCT template (*);      ! Template structure
BEGIN
    INT a;
    INT b;
END;
PROC .EXT p;
STRUCT ref_struct (template);
BEGIN
    ! Lots of code
END;
```

Topics:

- [Using STRUCT as a Formal Parameter \(page 255\)](#)
- [Passing an Extended Address Parameter to a Non-EXTENDED Reference Parameter \(page 255\)](#)
- [Using the PROC Formal Parameter \(page 256\)](#)
- [Referencing Parameters \(page 256\)](#)

Using STRUCT as a Formal Parameter

You cannot declare a definition STRUCT as a formal parameter. You can, however, achieve the same effect by using a referral STRUCT as a formal parameter, and having it reference a previously declared structure.

Example 204 Using a Referral STRUCT as a Formal Parameter

```
INT .EXT ea;
INT .EXT32 e32a;
INT .EXT64 e64a;
PROC p(a);
INT .a;
BEGIN
    ...
END;
...
p(ea);    ! OKAY
p(e32a); ! OKAY
p(e64a); ! ERROR: EXT64ADDR not assignment compatible with WADDR.
```

Passing an Extended Address Parameter to a Non-EXTENDED Reference Parameter

You can pass a variable declared with a .EXT or .EXT32 indirection symbol to a formal parameter declared with a "." indirection symbol. pTAL converts the extended address to a BADDR or WADDR, as appropriate. In the following example, pTAL converts the extended address of l to a WADDR address:

Example 205 Converting the extended address of I to a WADDR address

```
INT .EXT ea;  
INT .EXT32 e32a;  
INT .EXT64 e64a;  
PROC p(a);  
INT .a;  
BEGIN  
    ...  
END;  
...  
p(ea);    ! OKAY  
p(e32a); ! OKAY  
p(e64a); ! ERROR: EXT64ADDR not assignment compatible with WADDR
```

NOTE: The “Indirection Symbols” (page 41), .EXT32 and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Using the PROC Formal Parameter

The @ character is not allowed on the actual parameter if the formal parameter is a PROC.

Referencing Parameters

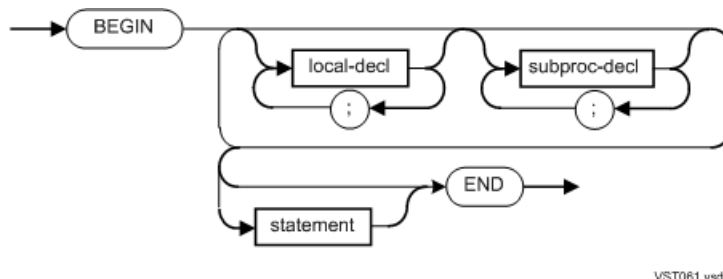
Do not depend on the order in which parameters are allocated in memory. You must refer to each parameter only as a named entity. Do not refer to one parameter as a base off of which you reference other parameters.

Guidelines:

- Do not treat a procedure’s formal parameters as an implied array or implied structure.
- Do not index a parameter to access another parameter or local variable.
- Do not perform block moves in which the source or destination spans more than one parameter.
- Do not pass the address of a value parameter to another procedure that expects the address of an array or structure.
- Do not proceed through a parameter list using indexing and address calculations.

Procedure Body

A procedure body can contain local declarations, subprocedure declarations, and statements.



local-decl

is a declaration for one of:

- simple variable
- array (direct, indirect, or read-only)
- structure (direct or indirect)

- simple pointer
- structure pointer
- equivalenced variable
- LITERAL
- DEFINE
- label
- entry point
- FORWARD subprocedure

subproc-decl

is a subprocedure declaration, as described in [Subprocedure Declarations \(page 257\)](#).

statement

is any statement described in [Chapter 12 \(page 199\)](#).

Example 206 Procedures

```

INT c;                ! Global declaration
PROC first;
BEGIN                ! Procedure body
  INT a,              ! Local declarations
    b;
  ...
END;
PROC second;
BEGIN                ! Procedure body
  ...
  CALL first;         ! Call first procedure
  ...
END;
```

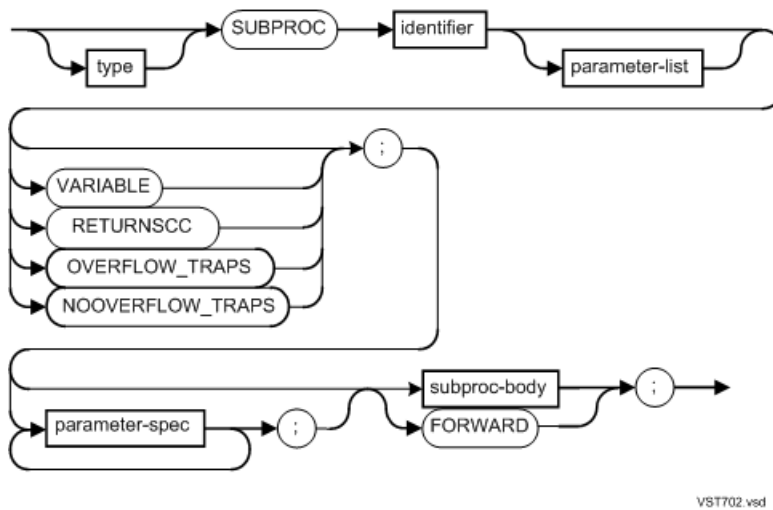
Example 207 FORWARD Declaration for a Procedure

```

INT g2;
PROC procb (param1); ! FORWARD declaration for procb
  INT param1;
FORWARD;
PROC proca;
BEGIN
  INT i1 := 2;
  CALL procb (i1);    ! Call procb
END;
PROC procb (param1); ! Body for procb
  INT param1;
BEGIN
  g2 := g2 + param1;
END;
```

Subprocedure Declarations

You can declare subprocedures within procedures, but not within subprocedures.



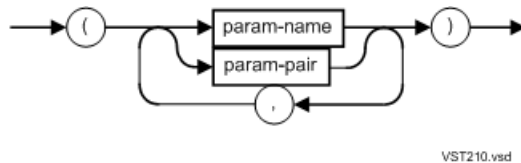
type

specifies that the subprocedure is a function that returns a result and indicates the data type of the returned result. *type* can be any data type described in [Chapter 3 \(page 46\)](#).

identifier

is the identifier of the subprocedure.

parameter-list



param-name

is the identifier of a formal parameter. The number of formal parameters a subprocedure can have is limited by space available in the parameter area of the subprocedure.

param-pair

is a pair of formal parameter identifiers that comprise a language-independent string descriptor in the form:



string

is the identifier of a standard or extended STRING simple pointer. The actual parameter is the identifier of a STRING array or simple pointer declare inside or outside a structure.

length

is the identifier of a directly addressed INT simple variable. The actual parameter is an expression that specifies the length of *string* in bytes.

VARIABLE

specifies that the compiler treats all parameters as optional, even if some are required by your code.

RETURN SCC

causes a subprocedure to return a condition code. The compiler reports an error if a subprocedure attempts to test the condition code after calling a subprocedure that does not specify RETURN SCC. Subprocedures declared with RETURN SCC cannot return 64-bit values.

OVERFLOW_TRAPS

enables overflow traps for a subprocedure.

NOOVERFLOW_TRAPS

disables overflow traps for a subprocedure.

parameter-spec

specifies the parameter type of a formal parameter and whether it is a value or reference parameter, as described in [Formal Parameter Specification \(page 251\)](#).

subproc-body

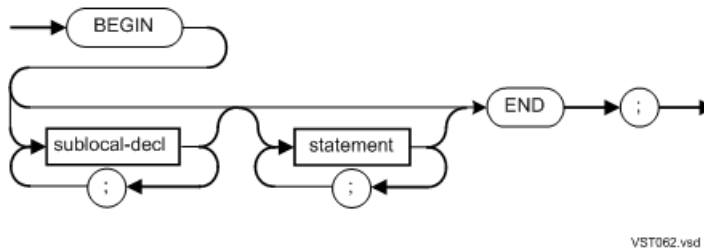
is a BEGIN-END block that contains sublocal declarations and statements—see [Subprocedure Body \(page 259\)](#).

FORWARD

means the subprocedure body is declared later in this procedure.

Subprocedure Body

A subprocedure body can contain sublocal declarations and statements.



sublocal-decl

is a declaration for one of:

- simple variable
- array (direct or read-only)
- structure (direct only)
- simple pointer
- structure pointer
- equivalenced variable
- LITERAL
- DEFINE
- label
- entry point

statement

is any statement described in [Chapter 12 \(page 199\)](#).

In subprocedures, declare pointers and directly addressed variables only. Here are examples:

Sublocal Variable	Example
Simple variable (always direct)	INT var;
Direct array	INT array[0:5];
Read-only array	INT ro_array = 'P' := [0,1,2,3,4,5];
Simple variable (always direct)	INT var;

Sublocal Variable	Example
Direct array	<code>INT array[0:5];</code>
Read-only array	<code>INT ro_array = 'P' := [0,1,2,3,4,5];</code>

Example 208 Function Subprocedure

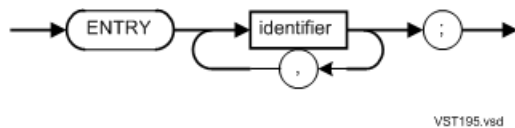
```

PROC p;
BEGIN
  SUBPROC p1;
  BEGIN
    INT .a[0:9];
    INT .ext b[0:9];
    a[0] := 1;
    b[9] := 2;
  END;
  CALL p1;
END;
PROC q;
BEGIN
  SUBPROC q1;
  BEGIN
    STRUCT .s;
    BEGIN
      INT i;
      INT j;
    END;
  END;
END;
END;

```

Entry-Point Declarations

The entry-point declaration associates an identifier with a secondary location in a procedure or subprocedure where execution can start.



identifier

is an entry-point identifier to be placed in the procedure or subprocedure body. It is an alternate or secondary point in the procedure or subprocedure at which to start executing.

Topics:

- [Procedure Entry-Point Identifiers \(page 260\)](#)
- [Subprocedure Entry-Point Identifiers \(page 262\)](#)

Procedure Entry-Point Identifiers

Here are guidelines for using procedure entry point identifiers:

- Declare all entry-point identifiers for a procedure within the procedure.
- Place each entry-point identifier and a colon (:) at a point in the procedure at which execution is to start.
- You can call a procedure entry-point identifier from anywhere in the program. (For functions, use the entry-point identifier in an expression; for other procedures, use a CALL statement.)
- Pass actual parameters as if you were calling the procedure identifier.

- You cannot use a GOTO statement to branch to a procedure entry-point identifier.
- To obtain the address of a procedure entry-point identifier, preface the identifier with @.
- You can specify FORWARD or EXTERNAL procedure entry-point declarations, which look like FORWARD procedure declarations and EXTERNAL procedure declarations.

Example 209 Procedure Entry-Point Identifiers

```

INT to_this := 314;      ! Declare global data
PROC add_3 (g2);
    INT .g2;
BEGIN
    ENTRY add_2;          ! Declare entry-point identifiers
    ENTRY add_1;
    INT m2 := 1;
    g2 := g2 + m2;
    add_2:                ! Location of entry-point identifier add_2
        g2 := g2 + m2;
    add_1:                ! Location of entry-point identifier add_1
        g2 := g2 + m2;
END;
PROC mymain MAIN;        ! Main procedure
BEGIN
    CALL add_1 (to_this); ! Call entry point add_1
END;

```

Example 210 FORWARD Declarations for Entry Points

```

INT to_this := 314;
PROC add_1 (g2);          ! FORWARD entry-point identifier
    INT .g2;              ! declaration
FORWARD;

PROC add_2 (g2);          ! FORWARD entry-point identifier
    INT .g2;              ! declaration
FORWARD;
PROC add_3 (g2);          ! FORWARD procedure declaration
    INT .g2;
FORWARD;
PROC mymain MAIN;        ! Main procedure declaration

BEGIN
    CALL add_1 (to_this); ! Call entry-point identifier
END;
PROC add_3 (g2);          ! Body for FORWARD procedure
    INT .g2;
BEGIN
    ENTRY add_2;          ! Declare entry-point identifiers
    ENTRY add_1;
    INT m2 := 1;
    g2 := g2 + m2;
    add_2:                ! Location of entry-point identifier
        g2 := g2 + m2;    ! add_2
    add_1:                ! Location of entry-point identifier
        g2 := g2 + m2;    ! add_1
END;

```

Subprocedure Entry-Point Identifiers

Here are guidelines for using subprocedure entry-point identifiers:

- Declare all entry-point identifiers for a subprocedure within the subprocedure.
- Place each entry-point identifier and a colon (:) at a point in the subprocedure at which execution is to start.
- You call a subprocedure entry-point identifier from anywhere in the encompassing procedure, including from within the same subprocedure. (For functions, use the entry-point identifier in an expression; for other subprocedures, use a CALL statement.)
- Pass actual parameters as if you were calling the subprocedure identifier.
- You cannot use a GOTO statement to branch to a subprocedure entry-point identifier.
- To obtain the code address of a subprocedure entry-point identifier, preface the identifier with @.
- You can specify FORWARD subprocedure entry-point declarations, which look like FORWARD subprocedure declarations.

Example 211 Subprocedure Entry-Point Identifiers

```
literal write_op,
        read_op,
        writeread_op,
        readwrite_op;
int proc io (op, buf);
    int      op;
    int .ext buf;
begin
    int subproc do_read_op (buf);
        int .ext buf;
    forward;
    int subproc do_write_op (buf);
        int .ext buf;
    forward;
    int subproc do_writeread_op (buf);
        int .ext buf;
    begin
        entry do_read_op;
        call do_write_op (buf);
        do_read_op:
            ! Perform read operation
    end;
    int subproc do_readwrite_op (buf);
        int .ext buf
    begin
        entry do_write_op;
        call do_read_op (buf);
        do_write_op:
            ! Perform write operation
    end;
    case op of
        begin
            ! write_op !      call do_write_op (buf);
            ! read_op !      call do_read_op (buf);
            ! writeread_op !  call writeread_op (buf);
            ! readwrite_op !  call readwrite_op (buf);
        end;
    end;
end;
```

Procedure Pointers

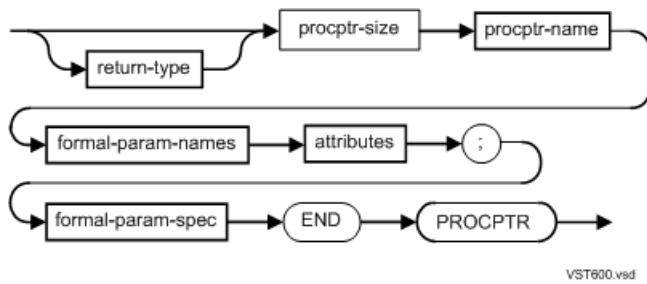
Procedure pointers allow a program to call a variable dynamically or to call an EXTENSIBLE procedure.

The syntax of procedure pointers is similar to the syntax of forward procedures; however, instead of the keyword PROC, you declare a procedure pointer using the keywords PROCPTR, PROC32PTR, or PROC64PTR. As with a forward procedure, a procedure pointer fully specifies the procedure's attributes and formal parameters but has no body—a procedure pointer does not include executable statements.

The size of PROCPTRs and PROC32PTRs is 32-bits in length. The size of PROC64PTRs is 64-bits in length.

You can declare procedure pointers as:

- Variables
- Formal parameters
- Structure fields



procptr-size

specifies the size of the procedure pointer and can be any one of:

- PROCPTR
- PROC32PTR
- PROC64PTR

PROCPTR and PROC32PTR are 32-bits in length and PROC64PTR is 64-bits in length.

return-type

specifies that the procedure is a function that returns a result and indicates the data type of the returned result, and can be any of:

- BADDR
- CBADDR
- CWADDR
- EXTADDR
- EXT32ADDR
- EXT64ADDR
- FIXED
- FIXED *[(scale)]*
- INT
- REAL
- REAL(64)
- PROCADDR
- PROC32ADDR
- PROC64ADDR
- SGWADDR
- SGBADDR
- SGXWADDR
- SGXBADDR
- STRING
- UNSIGNED *(width)*
- WADDR

NOTE: The address types and procedure pointers, EXT32ADDR, EXT64ADDR, PROC32ADDR, PROC64ADDR, PROC32PTR, and PROC64PTR are 64-bit addressing functionality added to the EptAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

scale

is a constant integer expression from -19 to 19.

width

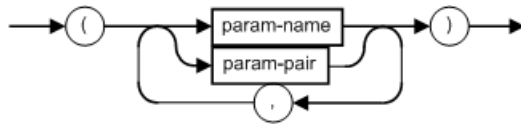
is a constant integer expression from 1 to 31.

procptr-name

is the name of the procedure pointer.

formal-param-names

is the identifier of a formal parameter. A procedure can have up to 32 formal parameters, with no limit on the number of words of parameters and has the form:



VST210.vsd

param-name

is the identifier of a formal parameter. A procedure can have up to 32 formal parameters, with no limit on the number of words of parameters.

param-pair

is a pair of formal parameter identifiers that comprise a language-independent string descriptor in the form:



VST039.vsd

string

is the identifier of a standard or extended STRING simple pointer. The actual parameter is the identifier of a STRING array or simple pointer declared inside or outside a structure.

length

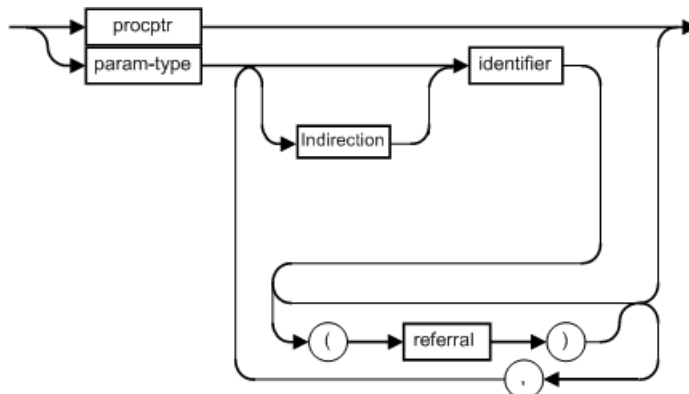
is the identifier of a directly addressed INT simple variable. The actual parameter is an INT expression that specifies the length of *string* in bytes.

attributes

is an attribute described in [Procedure Attributes \(page 248\)](#).

formal-param-spec

is a formal parameter and has the following form:



VST712.vsd

procptr

is a procedure pointer identifier.

param-type

is any data type described in the *data-type* parameter of this syntax description.

Indirection

., .EXT, .EXT32, .EXT64, .SG, and .SGX are indirection symbols (see [Table 14 \(page 41\)](#)).

NOTE: The “Indirection Symbols” (page 41), .EXT32 and .EXT64 are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

identifier

is an identifier (as described in [Identifiers \(page 42\)](#)).

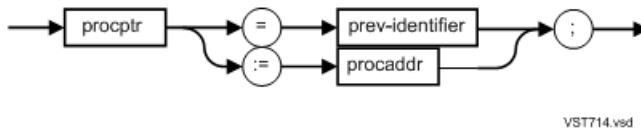
referral

is the name of a previously declared structure or structure pointer. You must include *referral* if the formal parameter *identifier* is the name of a structure.

Topics:

- [Declaring Procedure Pointer Variables \(page 266\)](#)
- [Declaring Procedure Pointers in Structures \(page 267\)](#)
- [Declaring PROCPTRs as Formal Parameters \(page 268\)](#)
- [Assignments to Procedure Pointers \(page 269\)](#)
- [Dynamically Selected Procedure Calls \(page 271\)](#)

Declaring Procedure Pointer Variables



procptr

is a procedure pointer identifier.

prev-identifier

is the identifier of a previously declared variable. On TNS architecture, *prev-identifier* must be 16 bits. On TNS/R and TNS/E architecture, *prev-identifier* must be 32 bits or more.

procaddr

is a constant or dynamic expression of type PROCADDR, PROC32ADDR, or PROC64ADDR. *procaddr* must be the name of a procedure, procedure pointer, or PROCADDR, PROC32ADDR, or PROC64ADDR variable. If *procaddr* is a procedure or procedure pointer, the parameters of *procptr* and *procaddr* must match and the following procedure attributes must match: EXTENSIBLE, VARIABLE, RETURNSCC, MAIN, and INTERRUPT; the following procedure attributes do not have to match: OVERFLOW_TRAPS, CALLABLE, PRIV, and RESIDENT.

You can declare a procedure pointer anywhere a data declaration is valid. For purposes of declarations, procedure pointers are treated as data, not as procedures.

The address type of a PROCPTR, PROC32PTR, and PROC64PTR is PROCADDR, PROC32ADDR, and PROC64ADDR, respectively.

The address type of a procedure pointer variable is WADDR.

The object data type of a reference to a function procedure pointer is the data type returned by the procedure pointer.

You can equivalence a procedure pointer (PROCPTR, PROC32PTR, or PROC64PTR) to any previously declared variable, if the width of the previous variable is greater than or equal to the width of the procedure pointer.

You can assign and pass procedure pointers of smaller or equal size to other procedure pointers, provided that the parameters and attributes match.

Example 212 Procedure Pointers as Variables and Formal Parameters

```

INT i;
INT .EXT j;
REAL k;
PROCADDR pa;
PROC32ADDR p32a;
PROC64addr p64a;
PROC p (i, j) EXTENSIBLE, CALLABLE;           ! Declare PROC p in a
    INT i, .EXT j;                             ! FORWARD declaration
FORWARD;
PROCPTR pp (i, j) EXTENSIBLE, CALLABLE;         ! Declare PROCPTR a and
    INT i, .EXT j;                             ! initialize it to point
END PROCPTR := @p;                             ! to PROC p
PROC64PTR p64pa (i, j) EXTENSIBLE, CALLABLE;    ! Declare PROC64PTR p64pa
    INT i, .EXT j;                             ! and initialize it to point
END PROCPTR := @p;                             ! to PROC p
PROC64PTR p64pb (i, j) EXTENSIBLE, CALLABLE;    ! Declare PROC64PTR p64pb
    INT i, .EXT j;                             ! and initialize it to point
END PROCPTR := @pp;                           ! to PROC p too
FIXED PROCPTR b (str : length);                ! Declare FIXED PROCPTR b
    STRING .str; INT length;                   ! with a parameter pair
END PROCPTR;
PROCPTR c (p);                                ! Declare PROCPTR c with one
    REAL PROC32PTR p(x); REAL x; END PROCPTR; ! one parameter, p, which is
END PROCPTR;                                  ! a PROC32PTR
REAL PROCPTR d (x); REAL x; END PROCPTR;       ! Declare REAL PROCPTR d
END PROCPTR;                                  ! with one REAL parameter
PROCPTR e(i);                                ! Declare PROCPTR e
    INT i;                                    ! Equivalence e to d
END PROCPTR = d;

```

Declaring Procedure Pointers in Structures

You can declare PROCPTR fields within structure declarations.



procptr

is a procedure pointer identifier.

previous-identifier

The identifier of a field at the same level as *procptr* in the same structure.

Example 213 (page 268) declares a REAL PROCPTR as a field in a structure array of 10 elements. Use an index to reference elements of array *s1*:

```
CALL s1[3].f(3.0e1);
```

Example 213 Procedure Pointers in a Structure

```
STRUCT s1 [0:9];
BEGIN
  REAL PROCPTR f(x); REAL x; END PROCPTR;
  PROC32PTR g; END PROCPTR;
  PROC64PTR h (x, y, z) EXTENSIBLE;
    INT x, y, z;
  END PROCPTR;
END;
```

Example 214 (page 268) declares a template structure `s2` with three components. When `s2` is the referent of a referral structure, `pTAL` allocates space for procedure pointer `f`. `pTAL` does not allocate space for procedure pointers `g` or `h` because they redefine procedure pointer `f`. Procedure pointers `f`, `g`, and `h` are the same except for the type of the parameter passed to the procedure.

Example 214 Equivalenced Procedure Pointers in a Structure

```
STRUCT s2 (*);
BEGIN
  REAL PROCPTR f(x);
    REAL x;
  END PROCPTR;
  REAL PROC32PTR g(x);
    INT x;
  END PROCPTR = f;
  REAL PROCPTR h(x);
    FIXED x;
  END PROCPTR = g;
END;
```

The code in Example 215 (page 268) uses the structure `s2` in Example 214 (page 268).

Example 215 Code That Uses the Structure in Example 214 (page 268)

```
STRUCT s(s2);
REAL my_real;
INT my_index := type_int;
CASE my_index OF
  BEGIN
    type_real -> my_real := s.f(3.0E1);
    type_int -> my_real := s.g(3);
    type_fixed -> my_real := s.h(3F);
  END;
```

Declaring PROCPTRs as Formal Parameters

The compiler:

- Ensures that the procedure attributes and parameter data types of procedures passed as actual parameters match those defined in the formal parameters of the called procedure
- Builds parameter masks for calls to VARIABLE procedures and EXTENSIBLE procedures

Example 216 Procedure Pointers as Formal Parameters

```
PROC a(i); INT i; EXTERNAL;
PROC b(p);
    PROCPTR p(a); INT a; END PROCPTR;
EXTERNAL;
PROC c(p);
    PROC64PTR p(a); INT a; END PROCPTR;
EXTERNAL;

PROC d(pa); PROCADDR pa; BEGIN END;
PROC e(pa); PROC32ADDR pa; BEGIN END;

PROC f;
BEGIN
    CALL b(a);      ! OK
    CALL b(@a);     ! ERROR: @ character is not valid
    CALL c(a);      ! OK
    CALL c(@a);     ! ERROR: @ character is not valid
    CALL d(a);      ! ERROR: @ character is required
    CALL d(@a);     ! OK
    CALL e(a);      ! ERROR: @ character is required
    CALL e(@a);     ! OK
END;
```

NOTE: Address type PROC32ADDR and procedure pointer type PROC64PTR are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

An @ character in front of the actual parameter is:

- Not allowed if the formal parameter is a PROC or a PROCPTR
- Required if the formal parameter is a PROCADDR

Assignments to Procedure Pointers

You can assign values to a procedure pointer variable in much the same way as you assign values to any variable; however, only values of data type procedure address can be assigned to a procedure pointer.

You can assign the following items to a procedure pointer:

- The address of a procedure or function
- The value of another procedure pointer
- The value of a variable whose data type is procedure address

Assignment statements involving procedure pointers fall into one of two categories:

- If the left side is a procedure pointer and right side is an @ character followed by the name of a procedure, subprocedure, or function—that is, neither the left side nor the right side is a procedure address variable—the attributes and the formal parameter types of each side of the assignment must match. The attributes specified must be the same but do not have to be presented in the same order.
- If either the left side or the right side of the assignment statement is a procedure address variable, the compiler does not attempt to match attributes or parameter types.
- Subject to the matching rules above, you can assign procedure pointers and procedure addresses to other procedure pointers and procedure addresses if the size of the target is equal to or larger than the size of the source.

Example 217 Assignments to procedure pointers, First Example

```
PROCPTR pp1 (a, b) RETURNSSC;
  INT a, b;
END PROCPTR;
PROCPTR pp2 (a) RETURNSSC;
  INT a;
END PROCPTR;
PROCPTR pp3 (a, b);
  INT a, b;
END PROCPTR;
PROC p(i, j) RETURNSSC;
  INT i, j;
BEGIN
  RETURN ,j;
END;
PROCADDR paddr;
paddr := @p;      ! OK: PROCADDR variable is assigned PROC addr
@pp1 := @p;       ! OK: Left side is PROCPTR, right side is PROC
@pp1 := @pp2;     ! ERROR: pp1 has two parameters, pp2 has one
@pp1 := @pp3;     ! ERROR: pp1 specifies RETURNSSC, pp3 does not
paddr := @pp2;    ! OK: paddr is a PROCADDR variable
@pp1 := paddr;    ! OK: paddr is a PROCADDR variable
```

Example 218 Assignments to procedure pointers, Second Example

```
REAL r;
INT i;
STRUCT s1 [0:9];
BEGIN
  REAL PROCPTR f(x);
  REAL x;
  END PROCPTR;
END;
PROC p (i, j) EXTENSIBLE, CALLABLE;      ! Declare PROC p in a
  INT i, .EXT j;                        ! FORWARD declaration
FORWARD;
PROCPTR a (i, j) EXTENSIBLE,CALLABLE;    ! Declare PROCPTR a and
  INT i, .EXT j;                        ! initialize it to
END PROCPTR;                            ! point to PROC p
PROCPTR c (p);
  REAL PROCPTR p (x);
  REAL x;
  END PROCPTR;
END PROCPTR;
REAL PROCPTR d (x);                      ! Declare REAL PROCPTR d
  REAL x;                                ! with REAL parameter a
END PROCPTR;
@a := @p;
@d := @s1[2].f;
@s1[3].f := @d;
CALL c(d);
```

Example 219 Assignments to Procedure Pointers, Third Example

```
PROCPTR pp; END PROCPTR;
PROC32PTR p32p; END PROCPTR;
PROC64PTR p64p; END PROCPTR;
PROCADDR pa;
PROC32ADDR p32a;
PROC64ADDR p64a;

@pp := @pp;      ! OK
@pp := pa        ! OK
@pp := @p32p;    ! OK
@pp := p32a;     ! OK
@pp := @p64p;    ! ERROR, @p64p is 64-bits long, @pp is 32-bits long
@pp := p64a;     ! ERROR, @p64a is 64-bits long, @pp is 32-bits long
@p32p := @p32p;  ! OK
@p32p := p32a;   ! OK
@p32p := @pp;    ! OK
@p32p := pa;     ! OK
@p32p := @p64p;  ! ERROR, @p64p is 64-bits long, @pp is 32-bits long
@p32p := p64a;   ! ERROR, @p64p is 64-bits long, @p32p is 32-bits long
@p64p := @p64p;  ! OK
@p64p := p64a;   ! OK
@p64p := @pp;    ! OK
@p64p := pa;     ! OK
@p64p := @p32p;  ! OK
@p64p := p32a;   ! OK
pa := @pp;       ! OK
pa := pa;        ! OK
pa := @p32p;     ! OK
pa := p32a;      ! OK
pa := @p64p;     ! ERROR, @p64p is 64-bits long, pa is 32-bits long
pa := p64a;      ! ERROR, p64a is 64-bits long, pa is 32-bits long
p32a := @pp;     ! OK
p32a := pa;      ! OK
p32a := @p32p;   ! OK
p32a := p32a;    ! OK
p32a := @p64p;   ! ERROR, @p64p is 64-bits long, p32a is 32-bits long
p32a := p64a;    ! ERROR, p64a is 64-bits long, p32a is 32-bits long
p64a := @pp;     ! OK
p64a := pa;      ! OK
p64a := @p32p;   ! OK
p64a := p32a;    ! OK
p64a := @p64p;   ! OK
p64a := p64a;    ! OK
```

NOTE: Address types PROC32ADDR and PROC64ADDR and procedure pointer types PROC32PTR and PROC64PTR are 64-bit addressing functionality added to the EptAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

Once you have set up a procedure pointer to point to a procedure, you can call the procedure by using the procedure pointer name in a CALL statement or, if the procedure pointer is typed, in an expression:

```
CALL a(1, 2);
r := d(r);
IF (sl[i].f(r)) < 1.0E0 THEN ...
```

Dynamically Selected Procedure Calls

You can use a procedure pointer to dynamically select a procedure to call.

Example 220 Dynamically Selected Procedure Call

```
LITERAL dev_6530, dev_3270, dev_dove;
INT device_type;
INT param1, param2;

PROC device_6530(i, j);
    INT i, j;
EXTERNAL;
PROC device_3270(i, j);
    INT i, j;
EXTERNAL;
PROC device_dove(i, j);
    INT i, j;
EXTERNAL;
PROCPTR p(i, j);
    INT i, j;

END PROCPTR;
CASE device_type of
    BEGIN
        dev_6530 -> @p := @device_6530;
        dev_3270 -> @p := @device_3270;
        dev_dove -> @p := @device_dove;
    END;
CALL p(param1, param2);
```

Although you cannot create an array of procedure pointers, you can create a structure that includes a procedure pointer field. You can choose dynamically which procedure pointer in the structure array to call.

Example 221 Dynamically Selected Procedure Call

```
LITERAL dev_6530, dev_3270, dev_dove;
STRUCT s1 [dev_6530:dev_dove]; ! Array of PROCPTRs
BEGIN
    PROCPTR d(device, param);

    INT device, param;
END PROCPTR;
END;
PROC device_6530(i, j);

    INT i, j;
EXTERNAL;
PROC device_3270(i, j);
    INT i, j;
EXTERNAL;
PROC device_dove(i, j);
    INT i, j;
EXTERNAL;
```

You must initialize the array `s1` to hold the addresses of the procedures that you want to dynamically call, as in the following example:

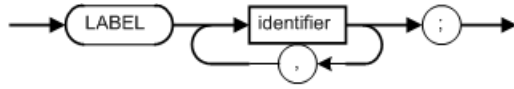
```
@s1[dev_6530].d := @device_6530;
@s1[dev_3270].d := @device_3270;
@s1[dev_dove].d := @device_dove;
```

Use an index to choose which element of array `s1` to call, as in the following example:

```
CALL s1[dev_6530].d(80, 2);
```


Labels in Procedures

A label is the target location of a GOTO statement.



VST196.vsd

identifier

is as described in [Identifiers \(page 42\)](#). It cannot be an entry-point identifier.

The following guidelines apply:

- LABEL is not a valid data type for a formal procedure parameter. You cannot pass a label to a procedure.
- A label is not a valid actual procedure parameter.
- If a GOTO statement in a subprocedure branches to a label in the containing procedure, the label must be declared in a LABEL declaration in the containing procedure, before the subprocedure that contains the GOTO statement (see [Nonlocal \(page 215\)](#)).

NOTE: This is not recommended in pTAL because it is very inefficient.

- The executable statement identified by a label cannot be an IF statement that tests the hardware indicator.

The conditional expression in an IF statement that is identified by a label cannot test a hardware indicator.

Example 222 IF Statements Identified by Labels

```
INT i, j := 0;
i := i + 1;
IF < THEN ... ! OK
i := i + 1
label_a:
IF < THEN ... ! ERROR: label cannot immediately precede an
               ! IF statement that tests a hardware indicator
```

15 Built-In Routines

Topics:

- [Privileged Mode \(page 274\)](#)
- [Parameters \(page 275\)](#)
- [Hardware Indicators \(page 276\)](#)
- [Atomic Operations \(page 276\)](#)
- [Nonatomic Operations \(page 281\)](#)

Built-in routine calls whose results do not depend on the values of variables (such as `$LEN(n)` or `$INT(10D)`) can be used wherever constant values are allowed.

The syntax descriptions in this section use these terms:

Term	Definition
sINT	Signed 16-bit integer. Range is -32,768 through 32,767.
uINT	Unsigned 16-bit integer. Range is 0 through 65,535. Must be an INT variable, not a STRING or UNSIGNED variable.
word	16-bit word unless otherwise specified

“sINT” and “uINT” are not pTAL data types. This section uses them only to specify how built-in routines use INT parameters.

Privileged Mode

Many built-in routines can be executed only by processes running in privileged mode.

Routines that operate in privileged mode can:

- Call other routines that operate in privileged mode
- Perform privileged operations by means of calls to system procedures
- Execute privileged instructions that can affect other programs or the operating system
- Use system global pointers and 'SG' equivalencing to:
 - Access system tables (which are described in the system description manual for your system)
 - Access the system data area
 - Compare and move data between the system data area and the user data area
 - Initiate certain input-output transfers

(Only procedures that operate in privileged mode can access system global data.)

Routines that operate in privileged mode must be specially licensed, because they might (if improperly written) adversely affect the status of the processor in which they are running.

The following execute in privileged mode:

- CALLABLE procedures (that is, procedures declared with the attribute [CALLABLE](#))
- PRIV procedures (that is, procedures declared with the attribute [PRIV](#))
- Nonprivileged procedures that are called by CALLABLE or PRIV procedures
- [pTAL Privileged Routines \(page 281\)](#)

Parameters

Parameters of built-in routines are always passed by value.

Topics:

- [Addresses as Parameters \(page 275\)](#)
- [Expressions as Parameters \(page 275\)](#)

Addresses as Parameters

If a parameter of a built-in routine is an address, the address must have the correct address type—whether the parameter is an input parameter, an output parameter, or both.

In [Example 223 \(page 275\)](#), the built-in routine `$BUI<_IN_1` has one formal parameter whose data type is `BADDR`. The corresponding actual parameter must be either a `BADDR` variable or the address field of a `STRING` pointer.

Example 223 Built-In Routine With Address Parameter

```
BADDR    b;
STRING   .s;
$BUI<_IN_1(b);    ! OK: data type of b is BADDR
$BUI<_IN_1(@s);   ! OK: address type of @s is BADDR
$BUI<_IN_1(s);    ! ERROR: data type of s is STRING
```

If an output parameter of a built-in routine is an address, the corresponding actual parameter must not be an indirect array pointer or an indirect structure pointer.

In [Example 224 \(page 275\)](#), the built-in routine `$BUI<_IN_2` has one formal output parameter whose data type is `BADDR`.

Example 224 Built-In Routine With Address Output Parameter

```
STRING   .s[0:99];
$BUI<_IN_2(@s);   ! ERROR: s has no address container
                  !   in which to store a new address
```

Expressions as Parameters

Many built-in routines accept expressions as parameters (see their individual syntax descriptions). If a parameter of a built-in routine is an expression:

- The value of the expression can be any data type except `STRING` or `UNSIGNED`.
- Except in `INT` and `INT(32)` expressions, all operands must be of the same data type.
- An `INT` expression can include `STRING`, `INT`, and `UNSIGNED(1-16)` operands.

The system treats `STRING` and `UNSIGNED(1-16)` operands as if they were 16-bit values; that is, the system:

- Places a `STRING` operand in the right byte of a word and sets the left byte to 0.
- Places an `UNSIGNED(1-16)` operand in the right bits of a word and sets the unused left bits to 0.

- An INT(32) expression can include INT(32) and UNSIGNED(17-31) operands.
The system treats UNSIGNED(17-31) operands as if they were 32-bit values. Before evaluating the expression, the system places an UNSIGNED(17-31) operand in the right bits of a doubleword and sets the unused left bits to 0.
- The built-in routine, not the expression or its data type, determines whether the value of the parameter is signed or unsigned:
 - Built-in routines that expect signed arguments treat unsigned expressions as if they were signed.
 - Built-in routines that expect unsigned arguments treat signed expressions as if they were unsigned.

Hardware Indicators

The description of each built-in routine specifies which hardware indicators (condition code, \$CARRY, and \$OVERFLOW) the built-in routine sets. If the description does not specify the conditions for which the built-in routine sets the value of a hardware indicator, see the system description manual for your system.

If a built-in routine does not set a particular hardware indicator, then the value of that hardware indicator is undefined after the built-in routine completes. If you reference a hardware indicator when its value is undefined, the compiler reports a syntax error.

If the value of \$OVERFLOW would be nonzero after executing a built-in routine, an overflow trap occurs if overflow traps are enabled. If overflow traps are disabled, you must test \$OVERFLOW explicitly in your program.

For general information about hardware indicators, see [Chapter 13 \(page 234\)](#).

Atomic Operations

The built-in routines in [Table 58 \(page 276\)](#) perform atomic operations. No other process can access the memory referenced by an atomic operation until the atomic operation completes; for example, \$ATOMIC_ADD is equivalent to the following algorithm:

```
var := var + value;
```

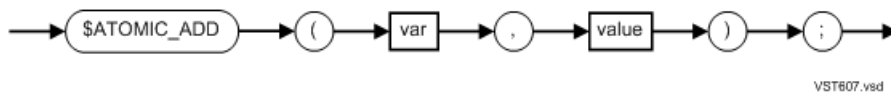
After the atomic operation reads *var*, no other process can access the memory location associated with *var* until the read completes. The read, add, and store operations are performed without interruption, as if the three operations were one.

Table 58 Built-In Routines for Atomic Operations

Routine	Atomic Operation	Can Set ...
\$ATOMIC_ADD	Adds two INT values	Condition code \$CARRY \$OVERFLOW
\$ATOMIC_AND	Performs a LAND on two INT values	Condition code
[EN DIS]ABLE_OVERFLOW_TRAPS Block Attribute	Deposits bits into an INT variable	Condition code
\$ATOMIC_GET	Gets (returns) the value of a variable	Condition code
Substructure Alignment	Performs a LOR on two INT values	Condition code
\$ATOMIC_PUT	Puts a value into a variable	

\$ATOMIC_ADD

\$ATOMIC_ADD atomically adds two INT values.



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	Yes, if traps are disabled
Sets \$OVERFLOW	Yes, if traps are disabled; otherwise, traps on overflow

var
 input,output
 sINT:variable
 is the variable that \$ATOMIC_ADD increments.

value
 input
 sINT:value
 is the value \$ATOMIC_ADD adds to *var*.

\$ATOMIC_ADD performs the following operation:

var := *var* + *value*

The read, add, and store operations are performed without interruption, as if the three operations were one.

Example 225 \$ATOMIC_ADD Routine

```

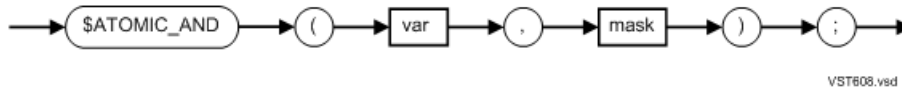
INT var;
INT value;
$ATOMIC_ADD (var, value);
  
```

The following table shows examples of \$ATOMIC_ADD:

var	value	result
%H1234	%HAAAA	%HBCDE
%H1234	%H5555	%H6789
%H6789	%HAAAA	%H1233
%H6789	%H5555	%HBCDE

\$ATOMIC_AND

\$ATOMIC_AND performs an atomic LAND on two INT values.



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

var
 input,output

`sINT:variable`
 is the variable to which `$ATOMIC_AND` applies *mask*.

mask
 input
 INT:value
 is a 16-bit mask that `$ATOMIC_AND` applies to *var*.

`$ATOMIC_AND` performs the following operation:

`var := var LAND mask`

The read, LAND, and store operations are performed without interruption, as if the three operations were one.

Example 226 \$ATOMIC_AND Routine

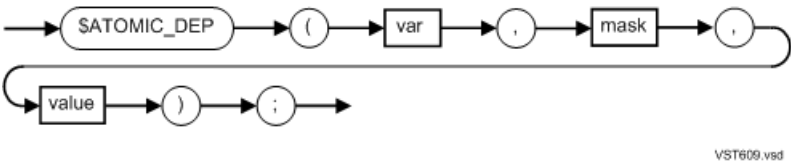
```
INT var;
INT mask;
$ATOMIC_AND(var, mask);
```

The following table shows examples of `$ATOMIC_AND`:

var	value	result
%H1234	%HAAAA	%HBCDE
%H1234	%H5555	%H6789
%H6789	%HAAAA	%H1233
%H6789	%H5555	%HBCDE

\$ATOMIC_DEP

`$ATOMIC_DEP` atomically deposits bits into an INT variable.



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

var
 input,output
 INT:variable
 is the variable into which `$ATOMIC_DEP` deposits bits from *value*.

mask
 input
 INT:value
 is a 16-bit mask word that determines which bits of *value* to deposit into *var*.
`$ATOMIC_DEP` stores into each bit position of *var*. The corresponding bit in *value* after performing an “and” operation between the corresponding bits in *value* and *mask*.

`value`
 input
`INT:value`
 holds the bits that, after being masked, `$ATOMIC_DEP` deposits in `var`.
`$ATOMIC_DEP` performs the following operation:
`var := (var LAND $COMP(mask)) LOR (value LAND mask)`
 All the operations are performed without interruption, as if they were one.

Example 227 \$ATOMIC_DEP Routine

```

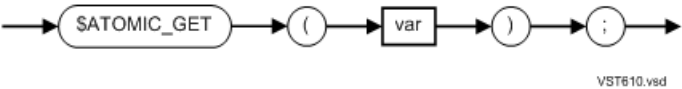
INT var;
INT mask;
INT value;
$ATOMIC_DEP(var, mask, value);
  
```

The following table shows examples of `$ATOMIC_DEP`:

var	value	mask	result
%H0000	%H1234	%HAAAA	%H0220
%H0000	%H1234	%H5555	%H1010
%H0000	%H6789	%HAAAA	%H2288
%H0000	%H6789	%H5555	%H1010

\$ATOMIC_GET

`$ATOMIC_GET` atomically gets (returns) the value of a variable.



Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

`var`
 input
`type:variable`
 is the variable whose value `$ATOMIC_GET` returns. `var` must be one of:

- A well-aligned byte, 2-byte, or 4-byte variable whose address is an integral multiple of its width.
- A bit field fully contained in a 1-byte, 2-byte, or 4-byte variable that is aligned on an even-byte boundary.

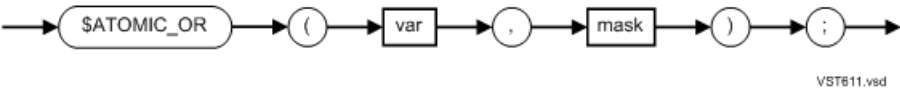
 If `var` is not well aligned, an error occurs.
 The operation is performed without interruption.

Example 228 \$ATOMIC_GET Routine

```
INT var1;
INT var2;
var1 := $ATOMIC_GET(var2);
if < then ... ! OK: $ATOMIC_GET sets condition code
```

\$ATOMIC_OR

\$ATOMIC_OR performs an atomic LOR on two INT values.



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

var
input,output
INT:variable
is the variable to which \$ATOMIC_OR applies *mask*.

mask
input
INT:value
is a 16-bit mask that \$ATOMIC_OR applies to *var*.

\$ATOMIC_OR performs the following statement:

```
var := var LOR mask
```

The read, LOR, and store operations are performed without interruption, as if the three operations were one.

Example 229 \$ATOMIC_OR Routine

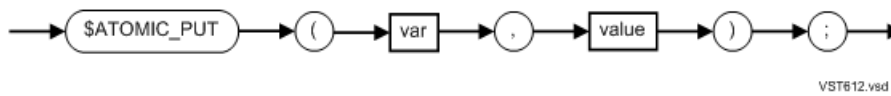
```
INT var;
INT mask;
$ATOMIC_OR(var, mask);
```

The following table shows examples of \$ATOMIC_OR:

var	mask	result
%H1234	%HAAAA	%HBABC
%H1234	%H5555	%H5775
%H6789	%HAAAA	%HEFAB
%H6789	%H5555	%H77DB

\$ATOMIC_PUT

\$ATOMIC_PUT atomically puts a value into a variable.



Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

var
 output
 type:variable
 the variable into which \$ATOMIC_PUT stores *value*. *var* must be one of:

- A 1-byte, 2-byte, or 4-byte variable whose address is an integral multiple of its width.
- A bit field fully contained in a 1-byte, 2-byte, or 4-byte variable that is aligned on an even-byte boundary.

value
 input
 type:value
 the value \$ATOMIC_PUT stores in *var*. *value* must be assignment-compatible with *var*.
 \$ATOMIC_PUT performs the following action:
var := *value*

Example 230 \$ATOMIC_PUT Routine

```

INT var;
INT value;
$ATOMIC_PUT(var, value);

```

Nonatomic Operations

- [pTAL Privileged Routines \(page 281\)](#)
- [Type-Conversion Routines \(page 282\)](#)
- [Address-Conversion Routines \(page 283\)](#)
- [Character-Test Routines \(page 284\)](#)
- [Minimum and Maximum Routines \(page 285\)](#)
- [Arithmetic Routines \(page 285\)](#)
- [Carry and Overflow Routines \(page 285\)](#)
- [FIXED-Expression Routines \(page 285\)](#)
- [Variable-Characteristic Routines \(page 285\)](#)
- [Procedure-Parameter Routines \(page 286\)](#)
- [Miscellaneous Routines \(page 286\)](#)

[Table 70 \(page 286\)](#) lists the built-in routines for nonatomic operations alphabetically and shows which hardware indicators they can set.

pTAL Privileged Routines

pTAL privileged routines execute in privileged mode (see [Privileged Mode \(page 274\)](#)). The pTAL compiler supports all pTAL privileged routines except \$TRIGGER.

The EpTAL compiler supports no pTAL privileged routines except \$TRIGGER.

Table 59 pTAL Privileged Routines

Procedure	Description
\$AXADR	Converts a standard address or a relative extended address to an absolute extended address
\$EXECUTEIO	Executes an I/O operation
\$FREEZE	Freezes (halts) the processor in which its process is running and any other processes on the same node that have FREEZE enabled
VOLATILE Attribute	Halts the processor in which its process is running
\$INTERROGATEIO	Stores cause and status information from an I/O interrupt
\$LOCKPAGE	Locks one page of memory
\$READBASELIMIT	Returns the base and limit of the current extended segment
\$TRIGGER	Replaces \$FREEZE and \$HALT, which are available only for code generated for the TNS/R architecture
\$UNLOCKPAGE	Unlocks one page of memory
\$WRITEPTE	Writes a segment-page-table entry

Type-Conversion Routines

A type-conversion routine converts its argument or arguments from one data type to another data type.

Table 60 Built-In Type-Conversion Routines

Routine	Converts ...	To ...
\$ASCIITOFIXED	ASCII value	FIXED value
\$DBL	INT, INT(32), FIXED, REAL, or REAL(64), or UNSIGNED(1-31) value EXTADDR or PROCADDR address	INT(32) value
\$DBLL	Two INT values	INT(32) value
\$DBLR	INT, INT(32), FIXED, REAL, or REAL(64) value	Rounded INT(32) value
\$DFIX	INT(32) value	FIXED(<i>fpoint</i>) value
\$EFLT	INT, INT(32), FIXED(<i>fpoint</i>), REAL, or REAL(64) value	REAL(64) value
\$EFLTR	INT, INT(32), FIXED(<i>fpoint</i>), or REAL, or REAL(64) value	Rounded REAL(64) value
\$FIX	INT, INT(32), REAL, REAL(64), FIXED, or EXT64ADDR ¹ value	FIXED value
\$FIXD	FIXED value	INT(32) value
\$FIXEDTOASCII	Absolute value of a FIXED value	ASCII value
\$FIXEDTOASCIIRESIDUE	Same as \$FIXEDTOASCII but returns the value of the residue	
\$FIXI	FIXED value	Signed INT value
\$FIXL	FIXED value	Unsigned INT value

Table 60 Built-In Type-Conversion Routines *(continued)*

Routine	Converts ...	To ...
\$FIXR	INT, INT(32), REAL, REAL(64), or FIXED value	Rounded FIXED value
\$FLT	INT, INT(32), FIXED(<i>fpoint</i>), REAL, or REAL(64) value	REAL value
\$FLTR	INT, INT(32), FIXED(<i>fpoint</i>), REAL, or REAL(64) value	Rounded REAL value
\$HIGH	Upper 16 bits of an INT(32) or EXTADDR value	INT value
\$IFIX	Signed INT value	FIXED(<i>fpoint</i>) value
\$INT	INT, INT(32), FIXED, UNSIGNED (1-31), REAL, or REAL(64) value Some address types	INT value
\$INT_OV	Same as \$INT, but sets \$OVERFLOW in some cases	
\$INTR	Low-order 16 bits of an INT, INT(32), or FIXED value REAL or REAL(64) value	Rounded INT value
\$LFIX	Unsigned INT value	FIXED(<i>fpoint</i>) value
\$UDBL	Unsigned INT value	INT(32) value
\$UFIX ¹	INT(32)	FIXED ²

¹ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

² Zero extends the INT(32) value to FIXED; does not sign extend.

Type-conversion routines that convert an argument from a smaller data type to a larger data type, such as \$DFIX, perform a sign extension of the expression to the high bits.

Type-conversion routines whose names end in R, such as \$DBLR, round their results. All other type-transfer routines truncate their results.

Type-conversion routines round values as follows:

$(\text{IF } value < 0 \text{ THEN } value - 5 \text{ ELSE } value + 5) / 10$

That is:

1. If *value* is negative, 5 is subtracted; if *value* is positive, 5 is added.
2. Integer division by 10 truncates the result; therefore, if the absolute value of the least significant digit of the result after initial truncation is 5 or more, one is added to the absolute value of the final least significant digit.

Rounding has no effect on INT, INT(32), or FIXED expressions.

Address-Conversion Routines

An address-conversion routine converts one address type to another address type.

Table 61 Built-In Address-Conversion Routines

Routine	Converts ...	To ...
\$BADDR_TO_EXTADDR	BADDR address	EXTADDR address
\$BADDR_TO_WADDR	BADDR address	WADDR address

Table 61 Built-In Address-Conversion Routines *(continued)*

Routine	Converts ...	To ...
\$EXTADDR_TO_BADDR	EXTADDR address	BADDR address
\$EXTADDR_TO_WADDR	EXTADDR address	WADDR address
\$EXT64ADDR_TO_EXTADDR ¹	EXT64ADDR ¹	EXTADDR address
\$EXTADDR_TO_EXT64ADDR ¹	EXTADDR	EXT64ADDR address ¹
\$EXT64ADDR_TO_EXT32ADDR ¹	EXT64ADDR ¹	EXT32ADDR address ¹
\$EXT64ADDR_TO_EXT32ADDR_OV ¹	EXT64ADDR ¹	EXT32ADDR address ^{1, 2}
\$IS_32BIT_ADDR ¹	Extended address	INT ³
\$PROCADDR	Procedure address or INT(32)	PROCADDR address
\$PROC32ADDR ¹	Procedure address or INT(32)	PROC32ADDR address ¹
\$PROC64ADDR ¹	Procedure address or FIXED	PROC64ADDR address ¹
\$SGBADDR_TO_EXTADDR	SGBADDR or SGXBADDR address	EXTADDR address
\$SGBADDR_TO_SGWADDR	SGBADDR or SGXBADDR address	SGWADDR address
\$SGWADDR_TO_EXTADDR	SGWADDR or SGXWADDR address	EXTADDR address
\$SGWADDR_TO_SGBADDR	SGWADDR or SGXWADDR address	SGBADDR address
\$WADDR_TO_BADDR	WADDR address	BADDR address
\$WADDR_TO_EXTADDR	WADDR address	EXTADDR address
\$XADR	Variable or struct	EXTADDR address ⁴
\$XADR32 ¹	Variable or struct	EXT32ADDR address ^{1, 4}
\$XADR64 ¹	Variable or struct	EXT64ADDR address ^{1, 4}

¹ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

² If the specified address cannot be represented in 32-bits, an overflow trap occurs. This trap cannot be disabled using the arithmetic trap controls (for example, NO_OVERFLOW_TRAPS, DISABLE_OVERFLOW_TRAPS, etc.)

³ Returns -1 if the specified address can be represented as a 32-bit address otherwise, returns 0.

⁴ Returns the address of the specified variable or struct in the desired extended address type.

The pTAL privileged routine **\$AXADR** (page 293), supported only by the pTAL compiler, is also an address-conversion routine.

Character-Test Routines

A character-test routine tests the right byte of an INT value for an alphabetic, numeric, or special character, returning a true value if the character is there and a false value otherwise.

Table 62 Built-In Character-Test Routines

Routine	Tests for ...
\$ALPHA	Alphabetic character
\$NUMERIC	Numeric character
\$SPECIAL	Special (ASCII nonalphanumeric) character (see Table 8 (page 36))

Minimum and Maximum Routines

Minimum routines return the minimum of two arguments. Maximum routines return the maximum of two arguments.

Table 63 Built-In Minimum and Maximum Routines

Arguments are of the type ...	Minimum	Maximum
Unsigned INT	\$LMIN	\$LMAX
Signed INT, INT(32), FIXED(<i>fpoint</i>), REAL, or REAL(64)	SRL	\$MAX

Arithmetic Routines

Table 64 Built-In Arithmetic Routines

Routine	Description
\$ABS	Returns the absolute value of its argument
\$COMP	Returns the one's complement of its argument
\$UDIVREM16	Divides an INT(32) dividend by an INT divisor to produce an INT quotient and an INT remainder
\$UDIVREM32	Divides an INT(32) dividend by an INT divisor to produce an INT(32) quotient and an INT remainder

Carry and Overflow Routines

Table 65 Built-In Carry and Overflow Routines

Routine	Indicates whether ...
\$CARRY	An arithmetic carry occurred during certain arithmetic operations or during execution of a SCAN or RSCAN statement
\$OVERFLOW	An overflow occurred during certain arithmetic operations

FIXED-Expression Routines

Table 66 Built-In FIXED-Expression Routines

Routine	Description
\$POINT	Returns the <i>fpoint</i> value of a FIXED expression
\$SCALE	Moves the position of the implied decimal point by changing a FIXED(<i>fpoint</i>) value

Variable-Characteristic Routines

Variable-characteristic routines return INT values that represent various characteristics of variables.

Table 67 Built-In Variable-Characteristic Routines

Routine	Returns an INT value that is the ...
\$BITLENGTH	Length, in bits, of a variable
\$BITOFFSET	Offset, in bits, of a structure data item from the address of the zeroth structure occurrence
\$LEN	Length, in bytes, of a variable
\$OCCURS	Number of elements in an array

Table 67 Built-In Variable-Characteristic Routines *(continued)*

Routine	Returns an INT value that is the ...
\$OFFSET	Offset, in bytes, of a structure item from the beginning of the structure
\$TYPE	Data type of a variable

Procedure-Parameter Routines

Table 68 Built-In Procedure-Parameter Routines

Routine	Description
\$OPTIONAL	Controls whether a given parameter or parameter pair is passed to a VARIABLE procedure or EXTENSIBLE procedure
\$PARAM	Checks for the presence or absence of an actual parameter in the call that called the current procedure or subprocedure

Miscellaneous Routines

Table 69 Miscellaneous Built-In Routines

Routine	Description
\$CHECKSUM	Returns the checksum of data
\$COUNTDUPS	Returns the number of duplicate words in a buffer
\$EXCHANGE	Exchanges the values of two variables of the same data type
\$FILL8, \$FILL16, and \$FILL32	Fill an array or structure with repetitions of an 8-bit, 16-bit, and 32-bit value, respectively
\$INTERROGATEHIO*,**	Stores cause and status information from a high-priority I/O interrupt
\$LOCATESPTHDR*,**	Returns the address of the Segment Page Table (SPT)
Declaring Arrays in Structures	Moves bytes from one memory location to another and computes a checksum (bitwise exclusive “or”) on them
\$MOVENONDUP	Moves words until it encounters two adjacent identical words
\$READCLOCK	Returns the current setting of the system clock
\$READSPT*,**	Returns (copies) an entry from the Segment Page Table (SPT)
\$READTIME	Returns the number of microseconds since the last cold load
\$STACK_ALLOCATE	Allocates a block of memory on the stack and returns the address of the block

* Only procedures executing in privileged mode can call this routine (see [Privileged Mode \(page 274\)](#))

** The EpTAL compiler does not support this routine

Table 70 Built-In Routines for Nonatomic Operations

Routine	Description	Can Set ...
\$ABS	Returns the absolute value of its argument	\$OVERFLOW
\$ALPHA	Tests for an alphabetic character	
\$ASCIITOFIXED	Converts an ASCII value to a FIXED value	Condition code \$OVERFLOW

Table 70 Built-In Routines for Nonatomic Operations *(continued)*

Routine	Description	Can Set ...
\$AXADR ^{1, 2, 3}	Converts a standard address or a relative extended address to an absolute extended address	
\$BADDR_TO_EXTADDR	Converts a BADDR address to an EXTADDR address	Condition code
\$BADDR_TO_WADDR	Converts a BADDR address to a WADDR address	Condition code
\$BITLENGTH	Returns the length, in bits, of a variable	
\$BITOFFSET	Returns the offset, in bits, of a structure data item from the address of the zeroth structure occurrence	
\$CARRY	Indicates whether an arithmetic carry occurred during certain arithmetic operations or during execution of a SCAN or RSCAN statement	
\$CHECKSUM	Returns the checksum of data	
\$COMP	Returns the one's complement of its argument	
\$COUNTDUPS	Returns the number of duplicate words in a buffer	
\$DBL	Converts its argument to an INT(32) value	\$OVERFLOW
\$DBLL	Converts two INT values to an INT(32) value	
\$DBLR	Converts its argument to a rounded INT(32) value	
\$DFIX	Converts an INT(32) value to a FIXED(<i>fpoint</i>) value	\$OVERFLOW
\$EFLT	Converts its argument to a REAL(64) value	
\$EFLTR	Converts its argument to a rounded REAL(64) value	
\$EXCHANGE	Exchanges the values of two variables of the same data type	
\$EXECUTEIO ^{1, 2, 3}	Executes an I/O operation	Condition code
\$EXTADDR_TO_BADDR	Converts a EXTADDR address to a BADDR address	
\$EXTADDR_TO_WADDR	Converts a EXTADDR address to a WADDR address	
\$EXT64ADDR_TO_EXTADDR ⁴	Converts address of type EXT64ADDR ⁴ to EXTADDR	
\$EXT64ADDR_TO_EXT32ADDR ⁴	Converts address of type EXT64ADDR ⁴ to EXT32ADDR ⁴	
\$EXT64ADDR_TO_EXT32ADDR_OV ⁴	Converts address of type EXT64ADDR ⁴ to EXT32ADDR ⁴ .	

Table 70 Built-In Routines for Nonatomic Operations *(continued)*

Routine	Description	Can Set ...
	Overflow trap occurs if the address cannot be represented by 32-bits	
<code>\$EXTADDR_TO_EXT64ADDR</code> ⁴	Converts address of type <code>EXTADDR</code> to <code>EXT64ADDR</code> ⁴	
<code>\$FILL8</code> , <code>\$FILL16</code> , and <code>\$FILL32</code>	Fill an array or structure with repetitions of an 8-bit, 16-bit, and 32-bit value, respectively	
<code>\$FIX</code>	Converts its argument to a <code>FIXED</code> value	
<code>\$FIXD</code>	Converts a <code>FIXED</code> value to an <code>INT(32)</code> value	<code>\$OVERFLOW</code>
<code>\$FIXEDTOASCII</code>	Converts the absolute value of a <code>FIXED</code> value to an ASCII value	<code>\$OVERFLOW</code>
<code>\$FIXEDTOASCIIRESIDUE</code>	Converts the absolute value of a <code>FIXED</code> value to an ASCII value and returns the value of the residue	<code>\$OVERFLOW</code>
<code>\$FIXI</code>	Converts a <code>FIXED</code> value to a signed <code>INT</code> value	<code>\$OVERFLOW</code>
<code>\$FIXL</code>	Converts a <code>FIXED</code> value to an unsigned <code>INT</code> value	<code>\$OVERFLOW</code>
<code>\$FIXR</code>	Converts its argument to a rounded <code>FIXED</code> value	<code>\$OVERFLOW</code>
<code>\$FLT</code>	Converts its argument to a <code>REAL</code> value	
<code>\$FLTR</code>	Converts its argument to a rounded <code>REAL</code> value	<code>\$OVERFLOW</code>
<code>\$FREEZE</code> ^{1, 2, 3}	Freezes (halts) the processor in which its process is running and any other processes on the same node that have <code>FREEZE</code> enabled	
<code>\$HALT</code> ^{1, 2, 3}	Halts the processor in which its process is running	
<code>\$HIGH</code>	Converts the high-order (leftmost) 16 bits of an <code>INT(32)</code> or <code>EXTADDR</code> value to an <code>INT</code> value	
<code>\$IFIX</code>	Converts a signed <code>INT</code> value to a <code>FIXED(fpoint)</code> value	
<code>\$INT</code>	Converts its argument to an <code>INT</code> value	
<code>\$INT_OV</code>	Same as <code>\$INT</code> , but sets overflow indicator in some cases	<code>\$OVERFLOW</code>
<code>\$INTERROGATEHIO</code> ^{2, 3}	Stores cause and status information from a high-priority I/O interrupt	Condition code
<code>\$INTERROGATEIO</code> ^{1, 2, 3}	Stores cause and status information from an I/O interrupt	Condition code
<code>\$INTR</code>	Converts the low-order 16 bits of an <code>INT</code> , <code>INT(32)</code> , or <code>FIXED</code> value to an <code>INT</code> value	<code>\$OVERFLOW</code>

Table 70 Built-In Routines for Nonatomic Operations *(continued)*

Routine	Description	Can Set ...
	Converts a REAL or REAL(64) value to a rounded INT value	
<code>\$IS_32BIT_ADDR</code> ⁴	Returns INT typed value -1 if the specified address can be represented by 32-bits and 0 otherwise.	
<code>\$LEN</code>	Returns the length, in bytes, of a variable	
<code>\$LFIX</code>	Converts an unsigned INT value to a <code>FIXED(<i>fpoint</i>)</code> value	
<code>\$LMAX</code>	Returns the maximum of two unsigned INT values	
<code>\$LMIN</code>	Returns the minimum of two unsigned INT values	
<code>\$LOCATESPTHDR</code> ^{2, 3}	Returns the address of the Segment Page Table (SPT)	<code>\$CARRY</code>
<code>\$LOCKPAGE</code> ^{1, 2, 3}	Locks one page of memory	Condition code <code>\$CARRY</code>
<code>\$MAX</code>	Returns the maximum of two signed values	
<code>\$MIN</code>	Returns the minimum of two signed values	
<code>\$MOVEANDCXSUMBYTES</code>	Moves bytes from one memory location to another and computes a checksum (bitwise exclusive “or”) on them	
<code>\$MOVENONDUP</code>	Moves words until it encounters two adjacent identical words	Condition code
<code>\$NUMERIC</code>	Tests for a numeric character	
<code>\$OCCURS</code>	Returns the number of elements in an array	
<code>\$OFFSET</code>	Returns the offset, in bytes, of a structure item from the beginning of the structure	
<code>\$OPTIONAL</code>	Controls whether a given parameter or parameter pair is passed to a <code>VARIABLE</code> procedure or <code>EXTENSIBLE</code> procedure	
<code>\$OVERFLOW</code>	Indicates whether an overflow occurred during certain arithmetic operations	
<code>\$PARAM</code>	Checks for the presence or absence of an actual parameter in the call that called the current procedure or subprocedure	
<code>\$POINT</code>	Returns the <i>fpoint</i> value of a <code>FIXED</code> expression	
<code>\$PROCADDR</code>	Converts an procedure address to a <code>PROCADDR</code> address	

Table 70 Built-In Routines for Nonatomic Operations *(continued)*

Routine	Description	Can Set ...
\$PROC32ADDR ⁴	Converts procedure address to PROC32ADDR ⁴	
\$PROC64ADDR ⁴	Converts procedure address to PROC64ADDR ⁴	
\$READBASELIMIT ^{1, 2}	Returns the base and limit of the current extended segment	
\$READCLOCK	Returns the current setting of the system clock	
\$READSPT ^{2, 3}	Returns (copies) an entry from the Segment Page Table (SPT)	\$CARRY
\$READTIME	Returns the number of microseconds since the last cold load	
\$SCALE	Moves the position of the implied decimal point by changing a FIXED(<i>fpoint</i>) value	\$OVERFLOW
\$SGBADDR_TO_EXTADDR	Converts a SGBADDR or SGXBADDR address to an EXTADDR address	
\$SGBADDR_TO_SGWADDR	Converts a SGBADDR or SGXBADDR address to a SGWADDR address	
\$SGWADDR_TO_EXTADDR	Converts a SGWADDR or SGXWADDR address to an EXTADDR address	
\$SGWADDR_TO_SGBADDR	Converts a SGWADDR or SGXWADDR address to a SGBADDR address	
\$SPECIAL	Tests for a special (ASCII nonalphanumeric) character	
\$STACK_ALLOCATE	Allocates a block of memory on the stack and returns the address of the block	
\$TRIGGER ^{1, 2, 5}	Replaces \$FREEZE and \$HALT, which are available only for code generated for the TNS/R architecture	
\$TYPE	Returns an INT value that represents the data type of a variable	
\$UDBL	Converts an unsigned INT value to an INT(32) value	
\$UDIVREM16	Divides an INT(32) dividend by an INT divisor to produce an INT quotient and INT remainder	\$OVERFLOW
\$UDIVREM32	Divides an INT(32) dividend by an INT divisor to produce an INT(32) quotient and INT remainder	\$OVERFLOW
\$UFIX ⁴	converts INT(32) to FIXED, zero extended	
\$UNLOCKPAGE ^{1, 2, 3}	Unlocks one page of memory	Condition code
\$WADDR_TO_BADDR	Converts a WADDR address to a BADDR address	

Table 70 Built-In Routines for Nonatomic Operations *(continued)*

Routine	Description	Can Set ...
<code>\$WADDR_TO_EXTADDR</code>	Converts a WADDR address to an EXTADDR address	
<code>\$WRITEPTE</code> ^{1, 2, 3}	Writes a segment-page table entry	<code>\$CARRY</code>
<code>\$XADR</code>	Returns the address of the specified variable or struct as type EXTADDR. ⁶	
<code>\$XADR32</code> ⁴	Returns the address of the specified variable or struct as type EXT32ADDR4. ⁶	
<code>\$XADR64</code> ⁴	Returns the address of the specified variable or struct as type EXT64ADDR4. ⁶	

¹ pTAL privileged procedure (see [Privileged Mode \(page 274\)](#))

² Only procedures operating in privileged mode can execute this routine (see [Privileged Mode \(page 274\)](#)).

³ The EpTAL compiler does not support this routine.

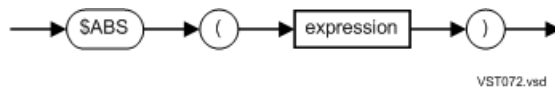
⁴ 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

⁵ The pTAL compiler does not support this routine.

⁶ The desired address is returned only if there exists a valid, explicit type conversion from @var or @struct to the desired extended address type.

\$ABS

`$ABS` returns the absolute value of its argument. The returned value has the same data type as the argument.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets <code>\$CARRY</code>	No
Sets <code>\$OVERFLOW</code>	Yes

expression

is an expression (as described in [Chapter 5 \(page 69\)](#)).

If the absolute value of a negative INT, INT(32), or FIXED expression cannot be represented in two’s complement form (for example, if *expression* has the INT value -32,768), `$ABS` traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, `$ABS` ignores the problem.

Example 231 `$ABS` Routine

```

INT int_val := -5;
INT abs_val;
abs_val := $ABS(int_val); ! Return 5, the absolute value of -5
  
```

\$ALPHA

`$ALPHA` tests the right byte of an INT value for the presence of an alphabetic character.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an INT expression.

\$ALPHA inspects bits <8:15> of *int-expression* and ignores bits <0:7>. It tests for an alphabetic character according to the following criteria:

int-expression >= "A" AND *int-expression* <= "Z" OR

int-expression >= "a" AND *int-expression* <= "z"

If an alphabetic character occurs, \$ALPHA sets the condition code indicator to CCE (condition code equal to). If you plan to check the condition code, do so before an arithmetic operation or assignment occurs.

If the character passes the test, \$ALPHA returns a -1 (true); otherwise, it returns a 0 (false).

int-expression can include STRING and UNSIGNED(1-16) operands, as described in "Expression Arguments" at the beginning of this section.

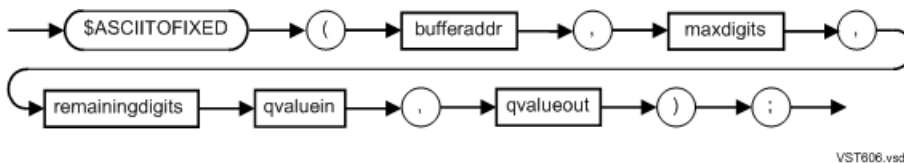
Example 232 \$ALPHA Routine

```

STRING some_char;
IF $ALPHA (some_char) THEN ... ; ! Test for alphabetic character
  
```

\$ASCIITOFIXED

\$ASCIITOFIXED converts an ASCII value to a FIXED value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

bufferaddr

input,output

BADDR:variable

is the byte address from which \$ASCIITOFIXED reads ASCII digits. When \$ASCIITOFIXED completes, *bufferaddr* contains the address following the last byte read.

maxdigits

input

uINT:value

is the maximum number of ASCII digits to read from *bufferaddr*.

remainingdigits

output

uINT:variable

is the number of bytes that \$ASCIITOFIXED did not convert because it encountered a nonnumeric ASCII byte. *remainingdigits* must be an INT variable; it cannot be a STRING, UNSIGNED, or USE variable or a bit field.

qvaluein

input

FIXED(*) :value

is a value that \$ASCIITOFIXED adds to the result of converting the bytes at *bufferaddr*. \$ASCIITOFIXED multiplies *qvaluein* by 10 for each digit it converts from ASCII to FIXED. After it converts the last digit at *bufferaddr*, \$ASCIITOFIXED adds *qvaluein* to the result of the conversion to establish the value that it returns *qvalueout*.

qvalueout

output

FIXED(*) :variable

is a quadrupleword integer value that holds the final result of the conversion.

\$ASCIITOFIXED converts a string of ASCII-coded digits at *bufferaddr* to a binary-coded FIXED value, adds *qvaluein* times 10^n , where n is the number of digits converted, and stores the result in *qvalueout*.

If a nondigit ASCII code is encountered, \$ASCIITOFIXED ends the conversion. \$ASCIITOFIXED converts only the digits before the nondigit ASCII code. CCG indicates that \$ASCIITOFIXED converted only part of the ASCII number. CCE indicates \$ASCIITOFIXED converted the entire string. If overflow traps are enabled and the result is greater than 263-1 or less than 263, \$ASCIITOFIXED sets \$OVERFLOW and *qvalueout* is undefined.

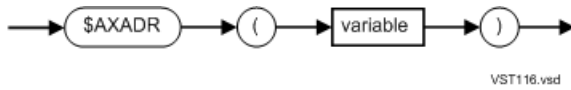
Example 233 \$ASCIITOFIXED Routine

```
LITERAL    buffer_len = 100;
STRING     .buffer[ 0:buffer_len - 1 ];    ! Buffer to convert
STRING     .ptr := @buffer;                ! pointer to buffer
INT        maxdigits;
INT        remainingdigits;
FIXED      qvaluein;
FIXED      qvalueout;
$ASCIITOFIXED (@ptr, maxdigits, remainingdigits, qvaluein,
               qvalueout);
```

\$AXADR

NOTE: The EpTAL compiler does not support this routine. (The EpTAL compiler does allow \$AXADR as a DEFINE name.)

\$AXADR converts a standard address or a relative extended address to an absolute extended address.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

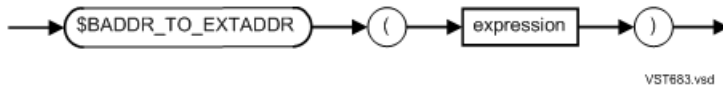
is the identifier of a simple variable, pointer, array element, structure, or structure data item. If *variable* is a pointer, \$AXADR returns the absolute extended address of the item to which the pointer points, not the address of the pointer itself.

Example 234 \$AXADR Routine

```
PROC myproc PRIV;
BEGIN
  STRING .EXT str;
  INT intr;
  ! Lots of code
  @str := $AXADR (intr);    ! Convert standard address of intr
                           ! to an absolute extended address
  !More code
END;
```

\$BADDR_TO_EXTADDR

\$BADDR_TO_EXTADDR converts a BADDR address to an EXTADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

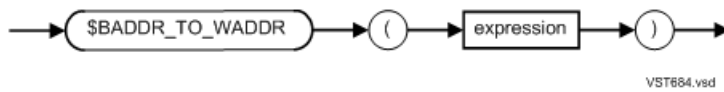
is an expression whose value is a BADDR address.

Example 235 \$BADDR_TO_EXTADDR Routine

```
STRING .EXT s;
STRING t;
@s := $BADDR_TO_EXTADDR(@t);    ! @t is a BADDR address
```

\$BADDR_TO_WADDR

\$BADDR_TO_WADDR converts a BADDR address to a WADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is a BADDR address.

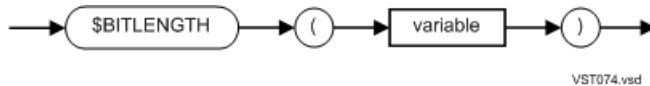
The result of \$BADDR_TO_WADDR is undefined if the least significant bit of *expression* is 1. The least significant bit of an address is not truncated when a byte address is converted to a word address—the address is not rounded down to the preceding even-byte address.

Example 236 \$BADDR_TO_WADDR Routine

```
INT    .i;
STRING s;
@i := $BADDR_TO_WADDR(@s); ! @s is a BADDR address
```

\$BITLENGTH

\$BITLENGTH returns an INT value that is the length, in bits, of a variable.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the identifier of a simple variable, array element, pointer, structure, or structure data item.

\$BITLENGTH returns the length, in bits, of a single occurrence of a simple variable, array element, structure, structure item, or item to which a pointer points.

The length of a structure or substructure occurrence is the sum of the lengths of all items contained in the structure or substructure. Complete the structure before you use \$BITLENGTH to obtain the length of any of the items in the structure.

To compute the total number of bits in an entire array or substructure, multiply the value returned by \$BITLENGTH by the value returned by \$OCCURS. To compute the total number of bits in a structure, first round up the value returned by \$BITLENGTH to the word boundary and then multiply the rounded value by the value returned by \$OCCURS.

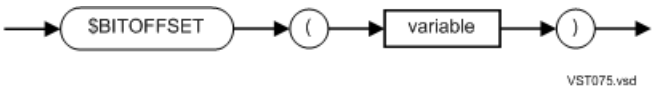
You can use \$BITLENGTH in LITERAL expressions and global initializations, because it always returns a constant value.

Example 237 \$BITLENGTH Routine

```
INT s_len;
STRUCT .s[0:3];           ! Declare four occurrences of a
BEGIN                     ! structure
    UNSIGNED(1) flags[0:15];
    UNSIGNED(2) status;
    BIT_FILLER 14;
END;
s_len := $BITLENGTH (s);   ! Return 32, the number of bits
                           ! in one structure occurrence
```

\$BITOFFSET

\$BITOFFSET returns an INT value that is the offset, in bits, of a structure data item from the address of the zeroth structure occurrence.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the fully qualified identifier of a structure item.

The zeroth structure occurrence has an offset of 0. For items other than substructure, simple variable, array, or pointer declared within a structure, \$BITOFFSET returns a 0.

When you qualify the identifier of *variable*, you can use constant indexes but not variable indexes; for example:

```
$BITOFFSET (struct1.subst[1].item) !1 is a constant index
```

To find the offset of an item in a structure, complete the structure before you use \$BITOFFSET.

You can use \$BITOFFSET in LITERAL expressions and global initializations, because it always returns a constant value.

Example 238 \$BITOFFSET Routine

```
STRUCT a;  
BEGIN  
  INT array[0:40];  
  STRUCT ab[0:9];  
  BEGIN  
    UNSIGNED(1) flag;  
    UNSIGNED(15) offset;  
  END;  
END;  
INT c;  
c := $BITOFFSET (a.ab[2]); ! Return offset of 3rd occurrence  
                           ! of ab
```

\$CARRY

\$CARRY returns a value that indicates whether an arithmetic carry occurred during certain arithmetic operations or during execution of a SCAN or RSCAN statement.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

The value returned by \$CARRY is based on instructions emitted by the compiler that determine whether a carry occurred. \$CARRY returns -1 if a carry occurred, 0 otherwise.

Procedures cannot return \$CARRY.

You can test \$CARRY only after one of the following statements:

- An assignment statement in which the final operator executed in the expression on the right side of the assignment is one of the following:
 - Signed integer add, subtract, or negate
 - Unsigned integer add, subtract, or negate
- A SCAN or RSCAN statement.

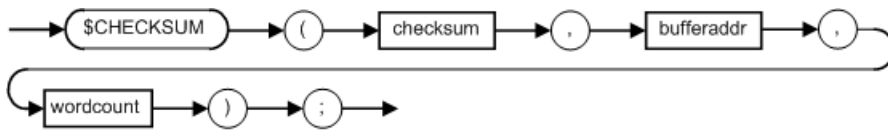
\$CARRY cannot be an actual parameter. If it is important to pass the value of \$CARRY to a procedure, use code similar to that in [Example 239](#).

Example 239 \$CARRY Routine

```
INT a, carry_flag;  
carry_flag := 0;  
a := a + 1;  
IF $CARRY THEN carry_flag := 1;  
CALL pl(carry_flag, .... );
```

\$CHECKSUM

\$CHECKSUM returns the checksum of data.



VST613.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

checksum

input,output

uINT:variable

the initial value ("seed" value) of the checksum. When \$CHECKSUM completes, *checksum* holds the final checksum. *checksum* must be an INT variable. It cannot be a STRING, UNSIGNED, or USE variable or a bit field.

bufferaddr

input

EXTADDR:value

the address of the first 16-bit word to include in the *checksum*.

wordcount

input

uINT:value

the number of 16-bit words to include in the *checksum*.

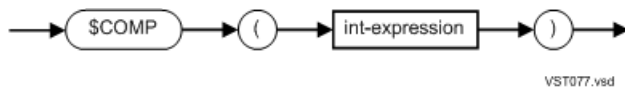
\$CHECKSUM accumulates the *checksum* by performing an exclusive-or operation on the accumulated *checksum* and *wordcount* successive 16-bit words, starting at *bufferaddr*. When \$CHECKSUM completes, *checksum* holds the accumulated *checksum* and *bufferaddr* is unchanged.

Example 240 \$CHECKSUM Routine

```
LITERAL buffer_len = 100;
INT      c_sum_val;
INT .EXT buffer1 [ 0:buffer_len - 1 ];
INT .EXT buffer2 [ 0:buffer_len - 1 ];
buffer1 '[:=' [%H0123, %H4567, %H89AB];
c_sum_val:= 3;
$CHECKSUM(c_sum_val, @buffer1, buffer_len);
! Value of c_sum_val is now %HCDEF
! Checksum buffer2 in same checksum word as buffer1
$CHECKSUM(c_sum_val, @buffer2, buffer_len);
! c_sum_val now has the combined checksum of buffer1 & buffer2
```

\$COMP

\$COMP returns the one's complement of its argument.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an expression whose value is an INT or INT(32) value.

The data type of the expression returned by \$COMP is the same as the data type of its argument.

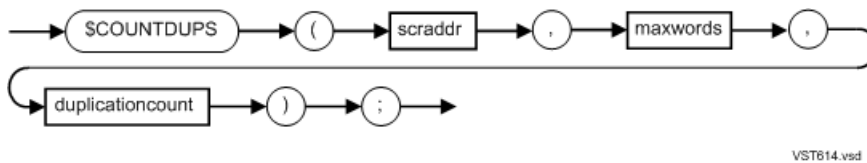
Example 241 \$COMP Routine

```

INT      i;
INT(32)  j;
i := $COMP(i);
j := $COMP(j);
  
```

\$COUNTDUPS

\$COUNTDUPS returns the number of consecutive words, starting at the beginning of a buffer, that are equal to the first word in the buffer.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

srcaddr

input,output

EXTADDR:variable

an address. Starting at *srcaddr*, \$COUNTDUPS scans 16-bit words until it encounters two adjacent words that are not equal. At the end of the operation, *srcaddr* points to the word that differs from the first word and which, therefore, terminated the scan. If there are no duplicates in the buffer, *srcaddr* points immediately after the last two words it compared—that is, at the first word \$COUNTDUPS did not examine.

maxwords

input,output

uINT:variable

the maximum number of 16-bit words to scan at *srcaddr*. At the end of the operation, *maxwords* contains:

- 0 if \$COUNTDUPS scanned the entire buffer.
- The number of words \$COUNTDUPS did not scan because it found a nonduplicate pair.

maxwords must be an INT variable; it cannot be a STRING, UNSIGNED, or USE variable or a bit field.

duplicationcount

input,output

uINT:variable

holds an initial value. At the end of the operation, *duplicationcount* contains its original value plus the number of duplicate words found by \$COUNTDUPS.

duplicationcount must be an INT variable; it cannot be a STRING, UNSIGNED, or USE variable or a bit field.

\$COUNTDUPS scans a buffer from left to right until it encounters two adjacent unequal words or until it reads *maxwords* words.

Example 242 \$COUNTDUPS Routine

```
LITERAL buffersize = 100;
INT .EXT buffer[ 0:buffersize-1 ];
INT      maxwords;
INT      duplication_count;
maxwords := maxbuff;
$COUNTDUPS(@buffer, maxwords, duplication_count);
```

\$DBL

\$DBL converts its argument to an INT(32) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes, if <i>expression</i> is a fixed value

expression

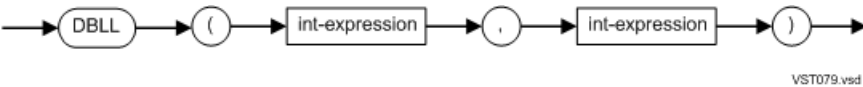
is an expression whose value is an INT, INT(32), FIXED, REAL, REAL(64), UNSIGNED(1-16), UNSIGNED(17-31), EXTADDR, or PROCADDR value.

Example 243 \$DBL Routine

```
INT .EXT i;  
EXTADDR e;  
INT(32) j;  
j := $DBL(e);      ! OK: e is type EXTADDR  
j := $DBL(@i);     ! OK: @i is type EXTADDR  
j := $DBL(i);      ! OK: i is type INT  
j := $DBL(@j);     ! ERROR: @j is type WADDR  
j := $DBL(@e);     ! ERROR: @e is type WADDR
```

\$DBLL

\$DBLL converts to INT values an INT(32) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an INT expression.

To form the INT(32) value, \$DBLL places the first *int-expression* in the high-order 16 bits and the second *int-expression* in the low-order 16 bits.

Example 244 \$DBLL Routine

```
INT first_int, second_int;  
INT(32) some_double;  
  
INT .EXT p;                ! 32-bit simple pointer  
some_double := $DBLL (first_int, second_int);  
                ! Return INT(32) value  
@p := ($DBLL (2, 7)) '<<' 1; ! Return 32-bit address in  
                ! user code segment
```

\$DBLR

\$DBLR converts its argument to an INT(32) value and rounds the result.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
is an INT, INT(32), FIXED, REAL, or REAL(64) expression.

If *expression* is too large to be represented by a 32-bit two's complement integer, \$DBLR traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$DBLR ignores the problem.

Example 245 \$DBLR Routine

```
REAL r2 := 1.5e0;
INT(32) b32;

REAL realnum := 123.456E0;
INT(32) dblnum;
b32 := $DBLR (r2);           ! Return 2d
dblnum := $DBLR (realnum);   ! Return 123D
```

\$DFIX

\$DFIX converts an INT(32) value to a FIXED(*fpoint*) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

dbl-expression
is an INT(32) expression.

fpoint
is a value in the range -19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

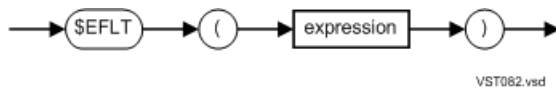
\$DFIX converts an INT(32) expression to a FIXED(*fpoint*) expression by performing the equivalent of a signed right shift of 32 positions from the left 32 bits into the right 32 bits of a quadrupleword unit.

Example 246 \$DFIX Routine

```
FIXED(2) fixnum;
INT(32) dblnum := -125D;
fixnum := $DFIX (dblnum, 2); ! Return -1.25
```

\$EFLT

\$EFLT converts its argument to a REAL(64) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression.

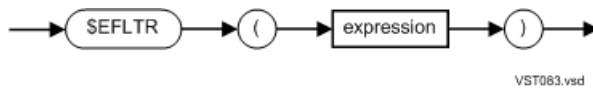
If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

Example 247 \$EFLT Routine

```
REAL(64) dbrlnum;
FIXED(3) fixnum := 12345.678F;
dbrlnum := $EFLT (fixnum);      ! Return 12345678L-3
```

\$EFLTR

\$EFLTR converts its argument to a REAL(64) value and rounds the result.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression.

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

Example 248 \$EFLTR Routine

```
REAL(64) rndnum;
FIXED(3) fixnum := 12345.678F;
rndnum := $EFLTR (fixnum);      ! Return rounded REAL(64) value
```

\$EXCHANGE

\$EXCHANGE exchanges the values of two variables of the same data type.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

var1

input,output

anytype: var

a variable whose contents are exchanged with *var2*.

var2

input,output

anytype: var

a variable whose contents are exchanged with *var1*.

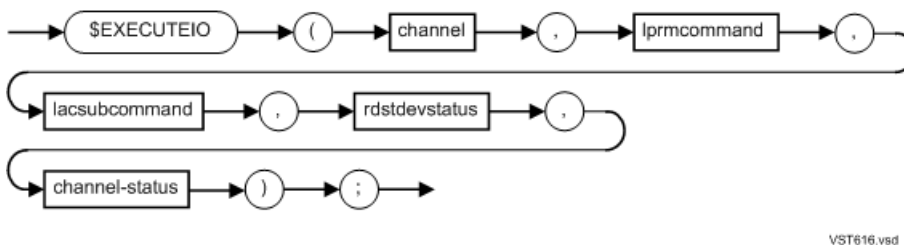
var1 and *var2* must meet the following requirements:

- *var1* and *var2* must both be INT variables or both be INT(32) variables.
- Neither *var1* nor *var2* can be a structure, but they can be fields of structures.
- Neither *var1* nor *var2* can be STRING, UNSIGNED, or USE variables, nor can they be bit strings.
- *var1*, *var2*, or both can be array elements.
- If *var1* or *var2* names an entire array, \$EXCHANGE exchanges element 0 of the array.

\$EXECUTEIO

NOTE: The EpTAL compiler does not support this procedure.

\$EXECUTEIO executes an I/O operation.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

channel

input

uINT:value

is the channel number to which the I/O is initialized.-

lprmcommand

input

uINT:value

is the load parameter.

lacsubcommand

input

sINT:value

is the load address and the command word.

rdstdevstatus

output

uINT:variable

is the controller and device status.

channel-status

output

sINT:variable

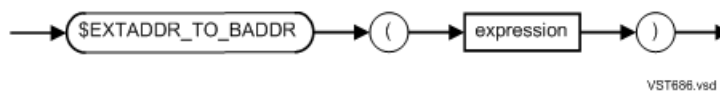
See the system description manual for your system for details.

Example 249 \$EXECUTEIO Routine

```
INT channel;  
INT lprm_command;  
INT lac_subcommand;  
INT rdst_dev_status;  
INT channel_status;  
$EXECUTEIO (channel, lprm_command, lac_subcommand,  
            rdst_dev_status, channel_status);
```

\$EXTADDR_TO_BADDR

\$EXTADDR_TO_BADDR converts an EXTADDR address to a BADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

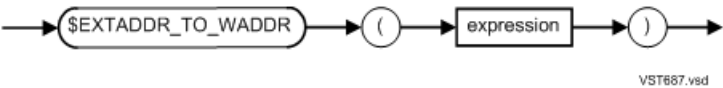
is an expression whose value is an EXTADDR address.

Example 250 \$EXTADDR_TO_BADDR Routine

```
PROC p(x);
  STRING .EXT x;
BEGIN
  STRING .j;
  @j := $EXTADDR_TO_BADDR(@x);
  @j := $EXTADDR_TO_BADDR(x);    ! ERROR: x is STRING,
END;                             ! not EXTADDR
```

\$EXTADDR_TO_WADDR

\$EXTADDR_TO_WADDR converts an EXTADDR address to an WADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
is an expression whose value is an EXTADDR address.

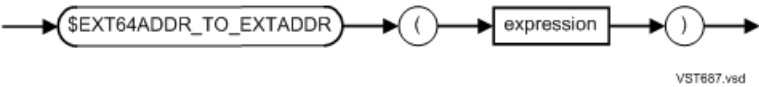
Example 251 \$EXTADDR_TO_WADDR Routine

```
PROC p(x);
  INT .EXT x;
BEGIN
  INT .j;
  @j := $EXTADDR_TO_WADDR(@x);
  @j := $EXTADDR_TO_WADDR(x);    ! ERROR: x is INT, not EXTADDR
END;
```

\$EXT64ADDR_TO_EXTADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXT64ADDR_TO_EXTADDR converts an EXT64ADDR address to an EXTADDR address. No check is performed to see if the resulting EXTADDR address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is an EXT64ADDR address.

\$EXT64ADDR_TO_EXT32ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXT64ADDR_TO_EXT32ADDR converts an EXT64ADDR address to an EXT32ADDR address. No check is performed to see if the resulting EXT32ADDR address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

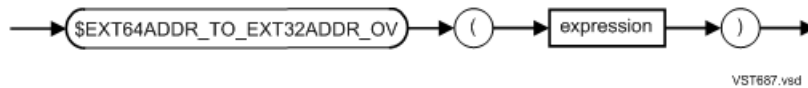
expression

is an expression whose value is an EXT64ADDR address.

\$EXT64ADDR_TO_EXT32ADDR_OV

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXT64ADDR_TO_EXT32ADDR_OV converts an EXT64ADDR address to an EXT32ADDR address. If the address cannot be represented as an EXT32ADDR value, an overflow trap occurs. This trap cannot be disabled using the existing overflow trap controlling mechanisms.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

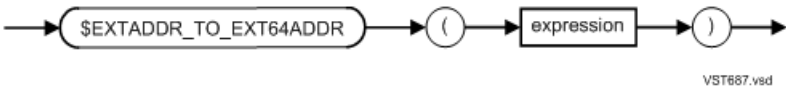
expression

is an expression whose value is an EXT64ADDR address.

\$EXTADDR_TO_EXT64ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXTADDR_TO_EXT64ADDR converts an EXTADDR address to an EXT64ADDR address.

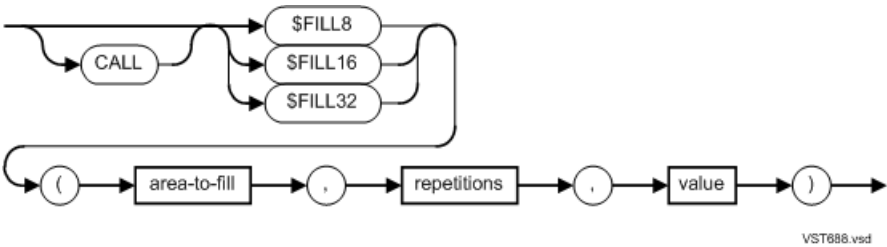


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
is an expression whose value is an EXTADDR or EXT32ADDR address.

\$FILL8, \$FILL16, and \$FILL32

\$FILL8, \$FILL16, and \$FILL32 fill an array or structure with repetitions of an 8-bit, 16-bit, or 32-bit value, respectively (sometimes called a “smear” operation).



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

area-to-fill
is a variable of any data type. The address of *area-to-fill* specifies the beginning of the area to fill.

repetitions
is an INT expression whose value specifies the number of times to write.

value
is an expression whose value is a STRING value for \$FILL8, to an INT value for \$FILL16, and to an INT(32) value for \$FILL32.

\$FILL16 and \$FILL32 cause an alignment trap if *area-to-fill* is not aligned to at least a 2-byte boundary.

\$FILL32 performance is significantly degraded if *area-to-fill* is not aligned to at least a 4-byte boundary.

None of the fill procedures (\$FILL8, \$FILL16, \$FILL32) perform bounds-checking on their parameters. If you write more bytes than the size of *area-to-fill*, the results are undefined. You might overwrite other data in your program with no immediate error, or you might cause any of several addressing errors, such as attempting to write in an area for which you do not have write permission, attempting to write in an unmapped page, and so forth.

Example 252 \$FILL8 Procedure

```
PROC a MAIN;
BEGIN
    STRUCT s(s_t);
    ...
    CALL $FILL8(s, $LEN(s), 0);
END;
```

\$FIX

\$FIX converts its argument to a FIXED value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
is an INT, INT(32), FIXED, REAL, EXT64ADDR, or REAL(64) expression.

If *expression* is too large in magnitude to be represented by a 64-bit two's complement integer, \$FIX traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$FIX ignores the problem.

Example 253 \$FIX Routine

```
FIXED fixnum;
INT intnum := 5;
fixnum := $FIX (intnum); ! Return 5F
```

\$FIXD

\$FIXD converts a FIXED value to an INT(32) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	Yes

fixed-expression
 is a FIXED expression, which \$FIXD treats as a FIXED expression, ignoring any implied decimal point.

If the result cannot be represented in a signed doubleword, \$FIXD traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$FIXD ignores the problem.

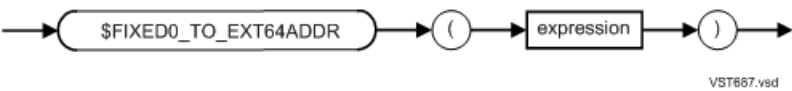
Example 254 \$FIXD Routine

```
INT(32) dblnum;
FIXED fixnum := 1234F;
dblnum := $FIXD (fixnum); ! Return 1234D
```

\$FIXED0_TO_EXT64ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “[64-bit Addressing Functionality](#)” (page 531).

\$FIXED0_TO_EXT64ADDR converts a FIXED value to an EXT64ADDR address.

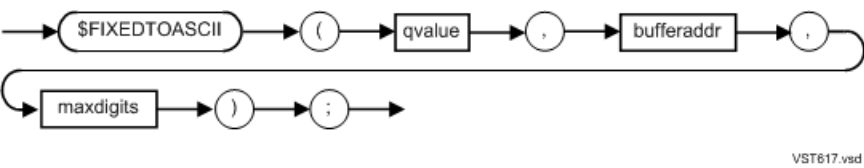


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
 is an expression whose value is FIXED.

\$FIXEDTOASCII

\$FIXEDTOASCII converts the absolute value of a FIXED value to an ASCII value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

qvalue
input
FIXED(*) :value
is a quadrupleword integer value to convert to ASCII digits.

bufferaddr
input
BADDR:value
is the byte address at which to write the ASCII digits.

maxdigits
input
uINT:value
is the maximum number of ASCII digits to write at *bufferaddr*.

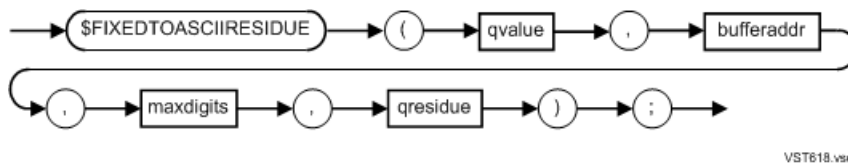
If \$FIXEDTOASCII converts *maxdigits* bytes but leading digits in *qvalue* are not converted, and \$OVERFLOW can be checked, \$FIXEDTOASCII sets \$OVERFLOW; otherwise, it resets \$OVERFLOW.

Example 255 \$FIXEDTOASCII Routine

```
LITERAL    buffer_len = 100;
FIXED      val;
STRING     .buffer[ 0:buffer_len - 1 ];
$FIXEDTOASCII(val, @buffer, buffer_len);
```

\$FIXEDTOASCIIRESIDUE

\$FIXEDTOASCIIRESIDUE converts the absolute value of a FIXED value to an ASCII value and returns the value of the residue.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

qvalue
input
FIXED(*) :value
is a quadrupleword integer value to convert to ASCII digits.

bufferaddr
input
BADDR:value
is the byte address at which to write the ASCII digits.

maxdigits

input

uINT:value

is the maximum number of ASCII digits to write at *bufferaddr*.

qresidue

output

FIXED(*):variable

holds any of the original value that was not converted because *maxdigits* bytes were converted without converting all of *qvalue*.

\$FIXEDTOASCIIRESIDUE returns in *qresidue* any portion of *qvalue* that it does not convert because *maxdigits* digits were written but *qvalue* was not fully converted.

If \$FIXEDTOASCIIRESIDUE converts *maxdigits* bytes but leading digits in *qvalue* are not converted, and \$OVERFLOW can be checked, \$FIXEDTOASCIIRESIDUEI sets \$OVERFLOW; otherwise, it resets \$OVERFLOW.

Example 256 \$FIXEDTOASCIIRESIDUE Routine

```
LITERAL    buffer_len = 100;
FIXED      val;
STRING     .buffer[ 0:buffer_len - 1 ];
FIXED      residue;
$FIXEDTOASCIIRESIDUE(val, @buffer, buffer_len, residue);
```

\$FIXI

\$FIXI converts a FIXED value to a signed INT value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

fixed-expression

is a FIXED expression, which \$FIXI treats as a FIXED expression, ignoring any implied decimal point.

If the result cannot be represented in a signed 16-bit integer, \$FIXI traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$FIXI ignores the problem.

Example 257 \$FIXI Routine

```
INT intnum;
FIXED fixnum := %177777F;
intnum := $FIXI (fixnum); ! Return -1
```

\$FIXL

\$FIXL converts a FIXED value to an unsigned INT value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

fixed-expression

is a FIXED expression, which \$FIXL treats as a FIXED expression, ignoring any implied decimal point.

If the result cannot be represented in an unsigned 16-bit integer, \$FIXL traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$FIXL ignores the problem.

Example 258 \$FIXL Routine

```

INT intnum;
FIXED fixnum := 32767F;
intnum := $FIXL (fixnum);  ! Return 32,767
  
```

\$FIXR

\$FIXR converts its argument to a FIXED value and rounds the result.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

expression

is an INT, INT(32), FIXED, REAL, or REAL(64) expression.

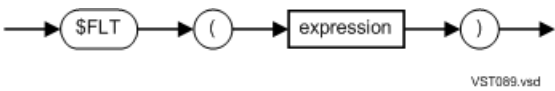
If *expression* is too large in magnitude to be represented by a 64-bit two's complement integer, \$FIXR traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$FIXR ignores the problem.

Example 259 \$FIXR Routine

```
FIXED rfixnum;  
REAL(64) bigrealnum := -1.5L0;  
FIXED rndfnum;  
REAL realnum := 123.456E0;  
rfixnum := $FIXR (bigrealnum); ! Return -1F  
rndfnum := $FIXR (realnum);    ! Return 123F
```

\$FLT

\$FLT converts its argument to a REAL value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression

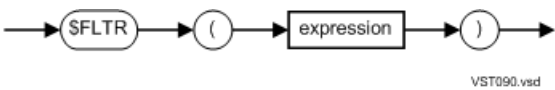
If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

Example 260 \$FLT Routine

```
REAL realnum;  
INT(32) dblnum := 147D;  
realnum := $FLT (dblnum); ! Return 147E0
```

\$FLTR

\$FLTR converts its argument to a REAL value and rounds the result.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

expression

is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

Example 261 \$FLTR Routine

```
REAL rrlnum;  
INT(32) dblnum := 147D;  
rrlnum := $FLTR (dblnum); ! Return rounded REAL value
```

\$FREEZE

NOTE:

- The EpTAL compiler does not support this procedure. Use [\\$TRIGGER \(page 345\)](#) instead. (The EpTAL compiler does allow \$FREEZE as a DEFINE name.)
- Execution does not return from this call.

\$FREEZE halts the processor in which its process is running and any other processors on the same node that have FREEZE enabled.



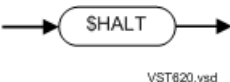
pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

\$HALT

NOTE:

- The EpTAL compiler does not support this procedure. Use [\\$TRIGGER \(page 345\)](#) instead. (The EpTAL compiler does allow \$HALT as a DEFINE name.)
- Execution does not return from this call.

\$HALT halts the processor in which its process is running.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

\$HIGH

\$HIGH converts the high-order (leftmost) 16 bits of an INT(32) or EXTADDR value to an INT value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

dbl-expression

is an expression whose value is INT(32) or EXTADDR.

\$HIGH returns the high-order 16 bits of *dbl-expression* and preserves the sign bit. \$HIGH does not cause overflow.

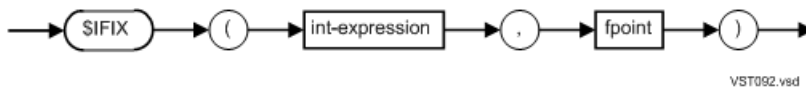
Example 262 \$HIGH Routine

```

INT      a;INT(32)  b;
INT .EXT c;
EXTADDR  d;
a := $HIGH(b);      ! OK: b is INT(32)
a := $HIGH(@c);      ! OK: @c is EXTADDR
a := $HIGH(@b);      ! ERROR: @b is WADDR
a := $HIGH(c);       ! ERROR: c is INT
a := $HIGH(d);       ! OK: d is EXTADDR
  
```

\$IFIX

\$IFIX converts a signed INT value to a FIXED(*fpoint*) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is a signed INT expression.

fpoint

is a value in the range -19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

When \$IFIX converts the signed INT expression to a FIXED value, it performs the equivalent of a signed right shift of 48 positions in a quadrupleword unit.

In [Example 263 \(page 317\)](#), \$IFIX returns a FIXED(2) value from a signed INT expression and an *fpoint* of 2.

Example 263 \$IFIX Routine

```
FIXED(2) fixnum;  
INT intnum := 12345;  
fixnum := $IFIX (intnum, 2); ! Return 123.45
```

\$INT

\$INT converts its argument to an INT value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is an INT, INT(32), UNSIGNED(1-16), UNSIGNED(17-31), FIXED, REAL, REAL(64), SGBADDR, SGWADDR, SGXBADDR, SGXWADDR, or EXTADDR value.

If *expression* is not a FIXED, INT (64), REAL, or REAL(64) value, \$INT returns the low-order (rightmost) 16 bits of *expression*. \$INT never causes overflow. \$INT does not explicitly maintain the sign of *expression*. In [Example 264 \(page 317\)](#), \$INT returns -1 although the argument to \$INT is a positive number.

Example 264 \$INT Routine

```
INT i;  
i := $INT(%HFFFFFF%D);
```

If the value of the expression in [Example 264 \(page 317\)](#) is a FIXED, REAL, or REAL(64) value, \$INT returns the result of converting expression arithmetically to an INT value—\$INT does not just truncate an expression. If the converted value of expression is too large to fit in 16 bits, an exception trap occurs.

For details on SG and SGX variables, see [Chapter 3 \(page 46\)](#).

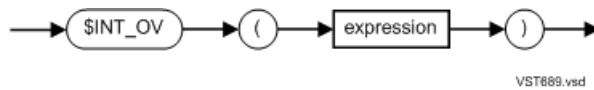
Example 265 \$INT Routine

```
PROC p;
BEGIN
  INT .SG a;
  SGXBADDR b;
  INT i;
  INT .EXT e;
  i := $INT(a);      ! OK: a is INT
  i := $INT(@a);     ! OK: @a is SGWADDR
  i := $INT(b);      ! OK: b is SGXBADDR
  i := $INT(i);      ! OK: i is INT
  i := $INT(@b);     ! ERROR: @b is WADDR
  i := $INT(@i);     ! ERROR: @i is WADDR
  i := $INT(@e);     ! OK: @e is EXTADDR
END;
```

\$INT_OV

NOTE: \$INT_OV is supported in the D40 and later product versions.

\$INT_OV converts its argument to an INT value and sets \$OVERFLOW in some cases.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

expression

is an expression whose value is an INT, INT(32), UNSIGNED(1-31), FIXED, REAL, REAL(64), SGBADDR, SGWADDR, SGXBADDR, SGXWADDR, or EXTADDR value.

If the data type of its argument is an INT(32) value greater than 32767 or less than -32768, \$INT_OV traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$INT_OV ignores the problem.

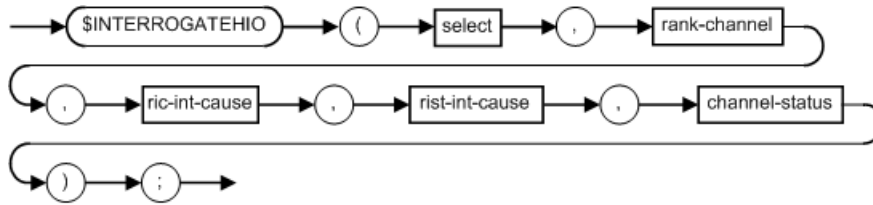
Example 266 Difference Between \$INT and \$INT_OV

```
INT i;
INT(32) j := 32767;
INT(32) k := 32768;
i := $INT(j);          ! INT never sets overflow
IF $OVERFLOW THEN ... ! $OVERFLOW is false
i := $INT(k);          ! INT never sets overflow
IF $OVERFLOW THEN ... ! $OVERFLOW is false
i := $INT_OV(j);
IF $OVERFLOW THEN ... ! $OVERFLOW is false
i := $INT_OV(k);
IF $OVERFLOW THEN ... ! $OVERFLOW is true
```

\$INTERROGATEHIO

NOTE: The EpTAL compiler does not support this procedure.

\$INTERROGATEHIO stores cause and status information from a high-priority I/O interrupt, which the operating system uses to reset the interrupt



VST643.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

select

output

uINT:variable

is an integer variable that is always set to 0.

rank-channel

output

uINT:variable

is an integer variable that is always set to 0.

ric-int-cause

output

uINT:variable

is the read interrupt cause received from the controller holding the completed I/O.

rist-int-cause

output

uINT:variable

is the read interrupt status received from the controller holding the completed I/O.

channel-status

output

uINT:variable

is an integer variable that holds the status returned by the controller.

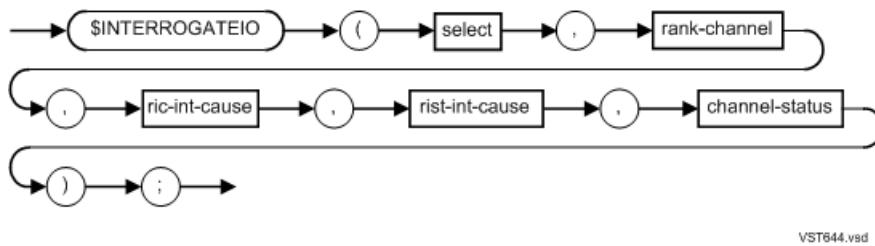
Example 267 \$INTERROGATEHIO Routine

```
INT select;
INT rank_channel;
INT ric_interrupt_status;
INT rist_interrupt_cause;
INT channel_status;
$INTERROGATEHIO(select, rank_channel, ric_interrupt_status,
                 rist_interrupt_status, channel_status);
```

\$INTERROGATEHIO

NOTE: The EpTAL compiler does not support this procedure.

\$INTERROGATEHIO stores cause and status information from an I/O interrupt, which the operating system uses to reset the interrupt.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

select

output

sINT:variable

is an integer variable that is always set to 0.

rank-channel

output

sINT:variable

is an integer variable that is always set to 0.

ric-int-cause

output

sINT:variable

is the read interrupt cause received from the controller holding the completed I/O.

rist-int-cause

output

sINT:variable

is the read interrupt status received from the controller holding the completed I/O.

channel-status

output

sINT:variable

is an integer variable that holds the status returned by the controller.

Example 268 \$INTERROGATEIO Routine

```
INT select;
INT rank_channel;
INT ric_interrupt_status;
INT rist_interrupt_cause;
INT channel_status;
$INTERROGATEIO(select, rank_channel, ric_interrupt_status,
               rist_interrupt_status, channel_status);
```

\$INTR

\$INTR converts:

- The low-order 16 bits of an INT, INT(32), or FIXED value to an INT value
- A REAL or REAL(64) value to a rounded INT value



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

expression

is an INT, INT(32), FIXED, REAL, or REAL(64) expression.

If *expression* is type INT, INT(32) or FIXED, \$INTR returns the low-order (least significant) 16 bits and does not explicitly maintain the sign. No overflow occurs.

If *expression* is type REAL or REAL(64), \$INTR returns a fully converted and rounded INT value, not a truncation. If the converted value of *expression* is too large to be represented by a 16-bit two's complement integer, an overflow trap occurs.

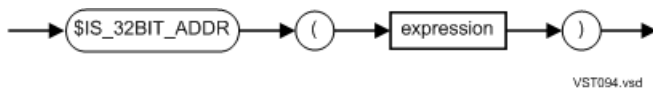
Example 269 \$INTR Routine

```
INT rndnum;
REAL realnum := 12345E-2;
rndnum := $INTR (realnum); ! Return 123
```

\$IS_32BIT_ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$IS_32BIT_ADDR returns the INT-typed value -1 if the specified address value can be represented as a 32-bit extended address; otherwise, it returns 0.



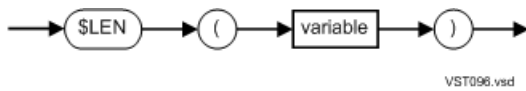
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is any of the address types, except SGWADDR and SGBADDR which are 16-bits in length.

\$LEN

\$LEN returns an INT value that is the length, in bytes, of a variable.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the identifier of a simple variable, array element, pointer, structure, or structure data item.

The compiler reports an error if you apply the \$LEN routine to a structure that consists of an odd number of bytes, exclusive of a pad byte.

You can avoid this error by using one of the following solutions:

- Declare explicitly a 1-byte filler item at the end of structures that consist of an odd number of bytes.
- Use [\\$BITLENGTH \(page 295\)](#) instead of \$LEN.

The compiler reports an error if you apply \$LEN to an UNSIGNED variable or structure field. Use \$BITLENGTH to obtain the length of an UNSIGNED variable or structure.

Example 270 \$LEN Routine

```
INT b;  
INT a [0:11];  
b := $LEN (a); ! Return 2
```

Example 271 \$LEN Routine

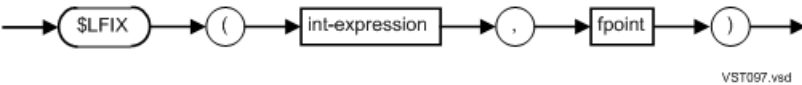
```
INT s_len;  
STRUCT .s[0:99];  
BEGIN  
    INT(32) array[0:2];  
END;  
s_len := $LEN (s); ! Return 12
```

Example 272 \$LEN Routine

```
INT array_length;  
INT(32) array[0:2];  
array_length := $LEN (array) * $OCCURS (array);  
! Return 12, the length of the entire array in bytes
```

\$LFIX

\$LFIX converts an unsigned INT value to a FIXED(*fpoint*) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression
is an unsigned INT expression.

fpoint
is a value in the range -19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

\$LFIX places the INT value in the low-order (least significant) word of the quadrupleword and sets the three high-order (most significant) words to 0.

Example 273 \$LFIX Routine

```
FIXED(2) fixnum;  
INT intnum := 125;  
fixnum := $LFIX (intnum, 2); ! Return 1.25
```

\$LMAX

\$LMAX returns the maximum of two unsigned INT values.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression
is an unsigned INT expression.

Example 274 \$LMAX Routine

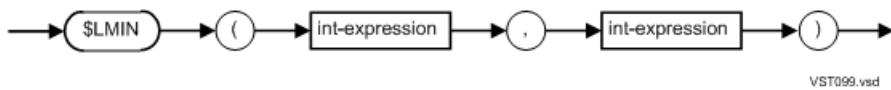
```

INT intval := 3;
max := $LMAX (intval, 5); ! Return 5

```

\$LMIN

\$LMIN returns the minimum of two unsigned INT values.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression
is an unsigned INT expression.

Example 275 \$LMIN Routine

```

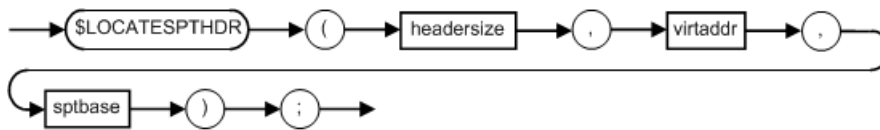
INT intval := 3;
min := $LMIN (intval, 5); ! Return 3

```

\$LOCATESPTHDR

NOTE: The EpTAL compiler does not support this procedure.

\$LOCATESPTHDR returns the address of the Segment Page Table (SPT).



VST645.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

headersize

input

uINT:value

is the unsigned byte offset from the beginning of the SPT to the beginning of the header. Because the SPT header always precedes the SPT, *headersize* is subtracted from the address of the SPT to obtain the address of the start of the header.

virtaddr

input

EXTADDR:value

is the address of the SPT.

sptbase

output

EXTADDR:variable

is the address of the segment-page-table header associated with *virtaddr*.

\$LOCATESPTHDR returns in *sptbase* the address of the segment-page table for the address in *virtaddr*.

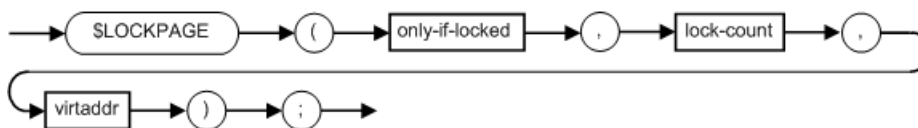
Example 276 \$LOCATESPTHDR Routine

```
INT    headersize;
EXTADDR addr;
EXTADDR seg_page_table_base;
$LOCATESPTHDR(headersize, addr, seg_page_table_base);
```

\$LOCKPAGE

NOTE: The EpTAL compiler does not support this procedure.

\$LOCKPAGE locks one page of memory.



VST646.vsd

pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes

Sets condition code	Yes
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

only-if-locked

input

sINT:value

is an INT value. If *only-if-locked* is greater than or equal to zero, the page will always be locked. If *only-if-locked* is less than zero, the page will be locked (that is, lock count will be incremented) only if it is already locked.

lock-count

input

sINT:value

is the total number of bytes to lock in the page.

virtaddr

input

EXTADDR:value

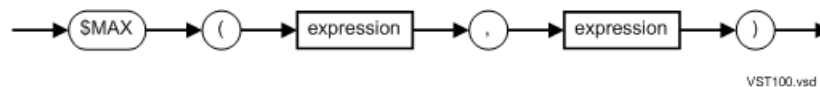
is the beginning virtual address to lock. \$LOCKPAGE calculates the page associated with *virtaddr*.

Example 277 \$LOCKPAGE Routine

```
INT    only_if_locked;
INT    lock_count;
EXTADDR virtaddr;
$LOCKPAGE(only_if_locked, lock_count, virtaddr);
```

\$MAX

\$MAX returns the maximum of two signed values.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is a signed INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression. Both expressions must be of the same data type.

Example 278 \$MAX Routine

```
REAL realval := -3E0;  
max := $MAX (realval, 5E0); ! Return 5E0
```

\$MIN

\$MIN returns the minimum of two signed values.



VST101.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

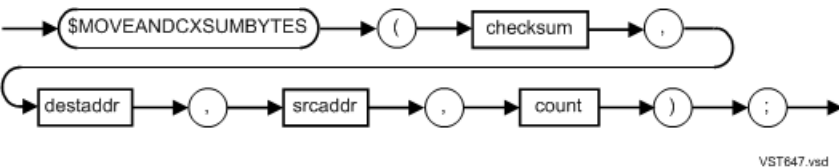
is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression. Both expressions must be of the same data type.

Example 279 \$MIN Routine

```
FIXED fixval := -3F;  
min := $MIN (fixval, 5F); ! Return -3F
```

\$MOVEANDCXSUMBYTES

\$MOVEANDCXSUMBYTES moves a specified number of bytes from one memory location to another and computes a checksum (bitwise exclusive “or”) on the bytes moved.



VST647.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

checksum

input,output

uINT:variable

contains an initial value for the *checksum*. When \$MOVEANDCXSUMBYTES completes, *checksum* contains the newly computed value.

destaddr

input,output

EXTADDR:variable

is the address to which \$MOVEANDCXSUMBYTES moves data. When \$MOVEANDCXSUMBYTES completes, *destaddr* points to the memory location following the last byte written.

srcaddr

input,output

EXTADDR:variable

is the address from which bytes are read. When \$MOVEANDCXSUMBYTES completes, *srcaddr* points to the memory location following the last byte read.

count

input

uINT:value

is the number of bytes to move.

\$MOVEANDCXSUMBYTES transfers *count* bytes from *srcaddr* to *destaddr* and computes a checksum (bitwise exclusive “or”) on the data moved. When \$MOVEANDCXSUMBYTES completes, *srcaddr* points to the immediate right of the last byte read, *destaddr* points to the immediate right of the last byte written, and *checksum* holds the newly computed checksum.

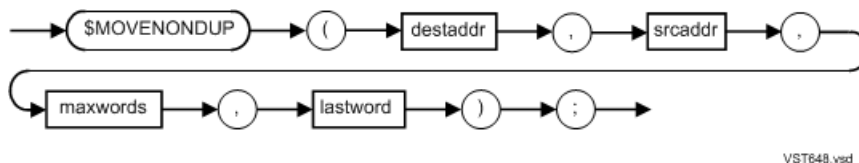
\$MOVEANDCXSUMBYTES does not ensure that the source and destination buffers do not overlap.

Example 280 \$MOVEANDCXSUMBYTES Routine

```
INT      checksum;
INT      .EXT source;
INT      .EXT dest;
INT(32)   count;
checksum := 0;
$MOVEANDCXSUMBYTES(checksum, @dest, @source, count);
```

\$MOVENONDUP

\$MOVENONDUP moves words from one location to another until it encounters two adjacent identical words.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

destaddr

input,output

EXTADDR:variable

is the address to which words are moved. When \$MOVENONDUP completes, *destaddr* is the address after which \$MOVENONDUP stored the last byte.

srcaddr

input,output

EXTADDR:variable

is the address from which 16-bit words are moved. When \$MOVENONDUP completes, *srcaddr* is the address after which \$MOVENONDUP read the last byte it moved.

maxwords

input,output

sINT:variable

is the maximum number of 16-bit words to move. When \$MOVENONDUP completes, *maxwords* is the number of words not moved because \$MOVENONDUP found a duplicate, or, if a duplicate was not found, *maxwords* is zero.

lastword

input,output

uINT:variable

holds the 16-bit word against which the first word at *srcaddr* is compared. When \$MOVENONDUP completes, *lastword* contains the last word moved.

Example 281 \$MOVENONDUP Routine

```
INT .EXT source;
INT .EXT destination;
INT     maxword;
INT     latestword;
$MOVENONDUP(@destination, @source, maxword, latestword);
```

\$NUMERIC

\$NUMERIC tests the right byte of an INT value for the presence of a numeric character.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an INT expression.

\$NUMERIC inspects bits <8:15> of *int-expression* and ignores bits <0:7>. It tests for a numeric character according to the criterion:

int-expression >= "0" AND *int-expression* <= "9"

If a numeric character occurs, \$NUMERIC sets the condition code to CCL (condition code less than). If you plan to test the condition code, do so before an arithmetic operation or assignment occurs.

If the character passes the test, \$NUMERIC returns a -1 (true); otherwise, it returns a 0 (false).

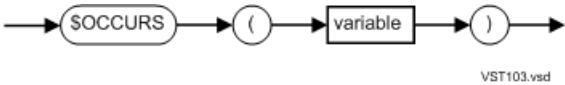
int-expression can include STRING and UNSIGNED(1-16) operands, as described in "Expression Arguments" at the beginning of this section.

Example 282 \$NUMERIC Routine

```
STRING char;  
IF $NUMERIC (char) THEN ... ; ! Test for numeric character
```

\$OCCURS

\$OCCURS returns an INT value that is the number of elements in an array.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the name of a variable, array, structure, or structure field. *variable* cannot be the name of a structure template.

If *identifier* is the identifier of an explicitly declared array—that is, it is not a reference parameter—and identifier is the unindexed name of the array, \$OCCURS returns the number of array elements specified in the array’s declaration; otherwise, \$OCCURS returns 1.

Table 71 \$OCCURS for Nonstructure Arrays

\$OCCURS Argument	Example	\$OCCURS Returns
Entire array	INT a[0:9]; \$OCCURS (a);	10
Value parameter or simple variable	INT a; \$OCCURS (a);	1
Reference parameter or pointer	INT .a; \$OCCURS (a);	1
Array element using constant index	INT a[0:9]; \$OCCURS (a[3]);	1
Array element using expression in index	INT a[0:9]; \$OCCURS (a[j]);	1

Table 72 \$OCCURS for Structure Arrays and Arrays Within Structures

\$OCCURS Argument	Example	\$OCCURS Returns
Unindexed structure array or substructure array	STRUCT s [0:9]; BEGIN	
or	STRUCT	
	BEGIN	
an element of a structure array or substructure array,	INT	
	END;	
or	END;	
an array that is a field within a structure or substructure	\$OCCURS (s);	10
	\$OCCURS (a[7]);	1
	\$OCCURS (a[7].t);	8
	\$OCCURS (a[7].t[3]);	1

Table 72 \$OCCURS for Structure Arrays and Arrays Within Structures *(continued)*

\$OCCURS Argument	Example	\$OCCURS Returns
	\$OCCURS (a[7].t[3].i);	5
	\$OCCURS (a[7].t[3].i[v]);	1
Entire structure or nonarray field of a structure or substructure	STRUCT s; BEGIN INT f; END;	
	\$OCCURS (s);	1
	\$OCCURS (s.f);	1
Structure template	STRUCT s; BEGIN INT f[0:9]; END;	
	\$OCCURS (s);	Compile-time err
	\$OCCURS (s.f);	10
	\$OCCURS (s.f[5]);	1

Example 283 \$OCCURS Routine With Nonstructure Arrays

```

PROC p(x, y);
    INT x,
        .y;
BEGIN
    INT a[0:9];
    INT i;
    INT .r;
    i := $OCCURS(a);      ! OK: a is an entire array
    i := $OCCURS(i);      ! OK: $OCCURS returns 1
    i := $OCCURS(x);      ! OK: $OCCURS returns 1
    i := $OCCURS(a[3]);   ! WARNING: $OCCURS returns 1
    i := $OCCURS(a[i]);   ! WARNING: $OCCURS returns 1
    i := $OCCURS(r);      ! WARNING: $OCCURS returns 1
    i := $OCCURS(y);      ! WARNING: $OCCURS returns 1
END;
```

Example 284 \$OCCURS Routine With Structure Arrays

```

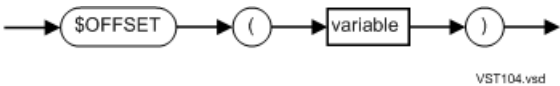
INT i;
STRUCT s[0:9];
BEGIN
    STRUCT t[0:7];
    BEGIN
        INT i[0:4];
        INT f;
    END;
END;
i := $OCCURS(s);          ! OK: s is an entire array
i := $OCCURS(s[7].t);     ! OK: t is an entire array
i := $OCCURS(s[7].t[3].i); ! OK: i is an entire array
i := $OCCURS(s[7].t[3].f); ! OK: $OCCURS returns 1
i := $OCCURS(s[7]);       ! WARNING: $OCCURS returns 1
i := $OCCURS(s[7].t[3]);  ! WARNING: $OCCURS returns 1
i := $OCCURS(s[7].t[3].i[v]); ! WARNING: $OCCURS returns 1
```

Example 285 \$OCCURS Routine With Template Structure Arrays

```
INT i;
STRUCT s(*);
BEGIN
    INT f[0:9];
END;
i := $OCCURS(s);           ! ERROR: Template structure not OK
i := $OCCURS(s.f);         ! OK: f is an array
i := $OCCURS(s.f[5]);      ! WARNING: $OCCURS returns 1
```

\$OFFSET

\$OFFSET returns an INT value that is the offset, in bytes, of a structure item from the beginning of the structure.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the fully qualified identifier of a structure field.

The compiler reports an error for the following uses of \$OFFSET:

- \$OFFSET applied to an UNSIGNED field. Use \$BITOFFSET instead of \$OFFSET.
- \$OFFSET applied to an item that is not a field in a structure.
- \$OFFSET applied to a structure array whose lower bound is nonzero; however, \$OFFSET applied to a substructure array whose lower bound is nonzero returns the appropriate offset.
- \$OFFSET for which the result would be greater than 216-1.

Example 286 \$OFFSET Routine

```
STRUCT a;
BEGIN
    INT array[0:40];
    STRUCT ab[0:9];
    BEGIN
        ! Lots of declarations
    END;
END;
INT c;
! Some code
c := $OFFSET (a.ab[2]); ! Return offset of third
                        ! occurrence of substructure
```

Example 287 \$OFFSET Routine

```
STRUCT .tt;
BEGIN
```

```

INT      i;
INT(32)  d;
STRING   s;
END;
STRUCT .st;
BEGIN
  INT i;
  INT j;
  INT .st_ptr(tt); ! Declare structure pointer
END;          ! that points to structure tt
INT x;
x := $OFFSET (st.j);           ! x gets 2
x := $OFFSET (tt.s);           ! x gets 6
x := $OFFSET (st.st_ptr.s);    ! x gets 6

```

Example 288 \$OFFSET Routine Applied to a Template Structure

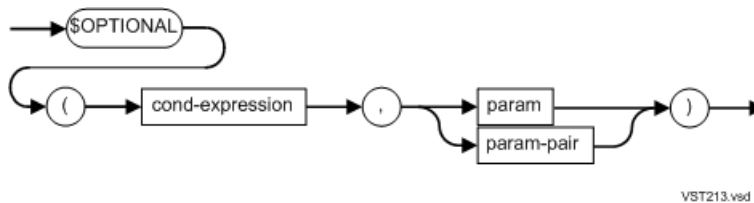
```

INT x;
STRUCT st[-1:1];
BEGIN
  INT item;
  FIXED(2) price;
END;
x := $OFFSET (st[-1].item); !x gets -10

```

\$OPTIONAL

\$OPTIONAL controls whether a given parameter or parameter pair is passed to a VARIABLE procedure or EXTENSIBLE procedure.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

cond-expression

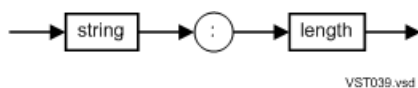
is a conditional expression. If *cond-expression* is true, *param* or *param-pair* is passed. If *cond-expression* is false, *param* (or *param-pair*) is not passed.

param

is an a variable identifier or an expression that defines an actual parameter to pass to a formal parameter declared in the called procedure if *cond-expression* is true.

param-pair

is an actual parameter pair to pass to a formal parameter pair declared in the called procedure if *cond-expression* is true. *param-pair* has the form:



string

is the identifier of a STRING array or simple pointer declared inside or outside a structure.

length

is an INT expression that specifies the length, in bytes, of *string*.

A call to a VARIABLE or EXTENSIBLE procedure can omit some or all parameters. \$OPTIONAL lets your program pass a parameter (or parameter-pair) based on a condition at execution time.

\$OPTIONAL is evaluated as follows each time the encompassing CALL statement is executed:

- If *cond-expression* is true, the parameter is passed; \$PARAM, if present, is set to true for the corresponding formal parameter.
- If *cond-expression* is false, the parameter is not passed; \$PARAM, if present, is set to false for the corresponding formal parameter.

A called procedure cannot distinguish between a parameter that is passed conditionally and one that is passed unconditionally. Passing parameters conditionally, however, is slower than passing them unconditionally. In the first case, the EXTENSIBLE mask is computed at execution time; in the second case, the mask is computed at compilation time.

Example 289 Parameters Passed Conditionally and Unconditionally

```
PROC p1 (i) EXTENSIBLE;
  INT i;
BEGIN
  ! Lots of code
END;
PROC p2;
BEGIN
  INT n := 1;
  CALL p1 ($OPTIONAL (n > 0, n) );    ! These two calls are
  CALL p1 (n);                       ! indistinguishable
END;
```

Example 290 Parameters Omitted Conditionally and Unconditionally

```
PROC p1 (i) EXTENSIBLE;
  INT i;
BEGIN
  ! Lots of code
END;
PROC p2;
BEGIN
  INT n := 1;
  CALL p1 ($OPTIONAL (n < 0, n) );    ! These two calls are
  CALL p1 ( );                       ! indistinguishable
END;
```

Example 291 Parameters Passed Conditionally

```
PROC p1 (str:len, b) EXTENSIBLE;
  STRING .str;
  INT len;
  INT b;
BEGIN
  ! Lots of code
END;
PROC p2;
BEGIN
  STRING .s[0:79];
  INT i:= 1;
  INT j:= 1;
  CALL p1 ($OPTIONAL (i < 9, s:i),    ! Pass s:i if i < 9
```

```

                                $OPTIONAL ( j > 2, j ) ); ! Pass j if j > 2
END;

```

You can use \$OPTIONAL when one procedure provides a front-end interface for another procedure that does the actual work, as [Example 292 \(page 335\)](#) shows.

Example 292 \$OPTIONAL Routine for a Front-End Interface

```

PROC p1 (i, j) EXTENSIBLE;
  INT .i;
  INT .j;
BEGIN

  ! Lots of code
END;

PROC p2 (p, q) EXTENSIBLE;
  INT .p;
  INT .q;
BEGIN
  ! Lots of code
  CALL p1 ($OPTIONAL ($PARAM (p), p ),
           $OPTIONAL ($PARAM (q), q ));
  ! Lots of code
END;

```

\$OVERFLOW

\$OVERFLOW returns a value indicating whether an overflow occurred during certain arithmetic operations.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

\$OVERFLOW indicates whether an overflow occurred. You can test \$OVERFLOW only if overflow traps are disabled and only following an assignment statement in which the final operator executed on the right side of the assignment is one \$FIX of a REAL or REAL(64) value of the following operators or built-in routines:

- Negate (unary -), +, -, *, /, '//
- \$DBL of an INT, FIXED, REAL, or REAL(64) value
- \$FLTR of a REAL(64) value
- \$FIX of a REAL or REAL(64) value
- \$FIXD
- \$FIXI
- \$FIXL
- \$FIXR of a REAL or REAL(64) value

- \$INT of a FIXED, REAL, or REAL(64) value
- \$INT of a FIXED, REAL, or REAL(64) value
- \$INTR of a FIXED, REAL, or REAL(64) value
- \$SCALE, for which: 1 <= exponent <= 4

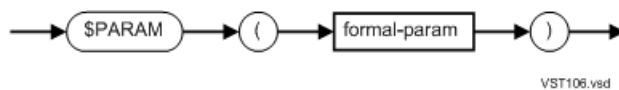
Example 293 \$OVERFLOW Routine

```
I := i + 1;
IF $OVERFLOW THEN ...
```

For more information about overflow, see [Chapter 13 \(page 234\)](#).

\$PARAM

\$PARAM checks for the presence or absence of an actual parameter in the call that called the current procedure or subprocedure.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

formal-param

is the identifier of a formal parameter as specified in the procedure or subprocedure declaration. If the actual parameter corresponding to *formal-param* is present in the CALL statement, \$PARAM returns 1 (not -1 as other Boolean operations do). If the actual parameter is absent from the CALL statement, \$PARAM returns 0.

Only a VARIABLE procedure or subprocedure or an EXTENSIBLE procedure can use \$PARAM. If such a procedure or subprocedure has required parameters, it must check for the presence or absence of each required parameter in CALL statements. The procedure or subprocedure can also use \$PARAM to check for optional parameters.

Example 294 \$PARAM Routine

```
PROC var_proc (buffer,length,key) VARIABLE;
  INT .buffer, length, ! Required parameters
      key;              ! Optional parameter

BEGIN
  ...
  IF NOT $PARAM (buffer) OR NOT $PARAM (length) THEN RETURN;
  ! Return 1 or 0 for each required parameter
  IF $PARAM (key) THEN ... ;
  ! Return 1 if optional parameter is present
END;
```

\$POINT

\$POINT returns the *fpoint* value (as an integer) of a FIXED expression.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

fixed-expression

is a FIXED expression.

The compiler emits no instructions when evaluating *fixed-expression*; therefore, *fixed-expression* cannot call a routine and cannot be an assignment expression.

Example 295 \$POINT Routine

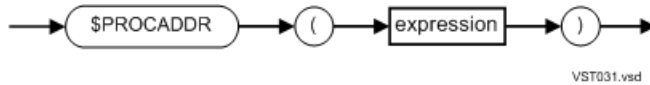
```

FIXED(3) result;
FIXED(3) a;
FIXED(3) b;
result := $SCALE (a, $POINT (b)) / b;
! Return fpint of FIXED expression & scale value by that factor

```

\$PROCADDR

\$PROCADDR converts a PROCADDR address, PROC32ADDR address, PROC64ADDR address, or INT(32) expression to a PROCADDR address. No check is performed to see if the resulting PROCADDR address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

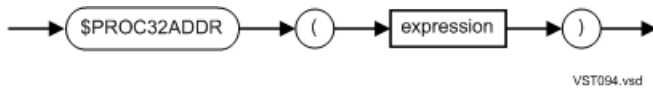
expression

is an expression whose value is an INT(32), PROCADDR, or PROC32ADDR address.

\$PROC32ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, [“64-bit Addressing Functionality” \(page 531\)](#).

\$PROC32ADDR converts a PROCADDR address, PROC32ADDR address, PROC64ADDR address, or INT(32) expression to a PROC32ADDR address. No check is performed to see if the resulting PROC32ADDR address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

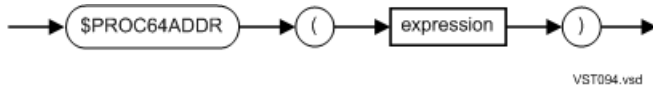
expression

is an expression whose value is an INT(32), PROCADDR, PROC32ADDR, or PROC64ADDR address.

\$PROC64ADDR

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$PROC64ADDR converts a PROCADDR address, PROC32ADDR address, PROC64ADDR address, or FIXED value to a PROC64ADDR address. No check is performed to see if the resulting PROC64ADDR address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is a FIXED, PROCADDR, PROC32ADDR, or PROC64ADDR address.

\$READBASELIMIT

NOTE: The EpTAL compiler does not support this procedure.

\$READBASELIMIT returns the base and limit of the current extended segment.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	No

xbase

INT(32):variable

is the base address of the current extended segment.

xlimit

output

INT(32):variable

is the limit of the current extended segment.

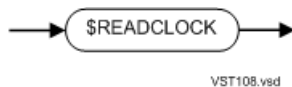
Consult the system description manual for your system for the format in which the base and limit values are returned.

Example 296 \$READBASELIMIT Routine

```
INT(32) xbase;
INT(32) xlimit;
$READBASELIMIT(xbase, xlimit);
```

\$READCLOCK

\$READCLOCK returns the current setting of the system clock as a FIXED value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

Example 297 \$READCLOCK Routine

```
FIXED the_time;
the_time := $READCLOCK; ! Return current clock time
```

\$READSPT

NOTE: The EpTAL compiler does not support this procedure.

\$READSPT returns (copies) an entry from the Segment Page Table (SPT).



pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	No

Sets \$CARRY	Yes
Sets \$OVERFLOW	No

virtaddr

input

EXTADDR:value

is the virtual address of the SPT entry to copy.

sptentryaddr

output

EXTADDR:variable

is the address at which \$READSPT stores the SPT entry.

Example 298 \$READSPT Routine

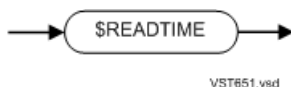
```
EXTADDR virtual_addr;
INT .EXT spt_entry(spt_template) := spt_entry_addr;
$READSPT(virtual_addr, @spt_entry);
```

\$READTIME

\$READTIME returns the number of microseconds since the last cold load.

NOTE: \$READTIME is not affected by the TACL command SETTIME; therefore, \$READTIME does not always return the value [JULIANTIMESTAMP(0) - JULIANTIMESTAMP(1)].

For a description of the SETTIME command, see the *TACL Reference Manual*. For a description of the JULIANTIMESTAMP function, see the *Guardian Procedure Calls Reference Manual*.



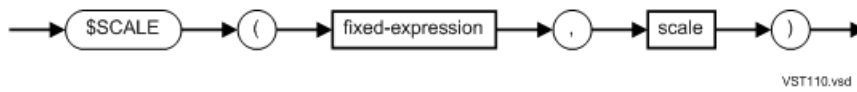
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

Example 299 \$READTIME Routine

```
FIXED time_now;
time_now := $READTIME;
```

\$SCALE

\$SCALE moves the position of the implied fixed-point (decimal point) by changing a FIXED(*fpoint*) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

fixed-expression

is the FIXED expression whose implied decimal point is to be changed.

scale

is an INT constant in the range -19 to +19 that specifies the number of positions to move the implied decimal point with respect to the least significant digit. If *scale* is negative, the implied decimal point moves to the left; if *scale* is positive, the implied decimal point moves to the right.

\$SCALE adjusts the implied decimal point of the stored FIXED value by multiplying or dividing the value by 10 to the *scale* power. Some precision might be lost with negative *scale* values.

If the result of the scale operation exceeds the range of a FIXED expression, \$SCALE traps if overflow traps are enabled (see [Chapter 13 \(page 234\)](#)); otherwise, \$SCALE ignores the problem.

Example 300 \$SCALE Routine

```

FIXED(3) a := 9.123F;
FIXED(7) result;
result := $SCALE (a, 4); ! Return FIXED(7) value from
                        ! FIXED(3) value
  
```

To retain precision when you divide operands that have nonzero *fpoint* settings, use the \$SCALE built-in routine to scale up the *fpoint* of the dividend by a factor equal to the *fpoint* of the divisor, as in [Example 301 \(page 341\)](#).

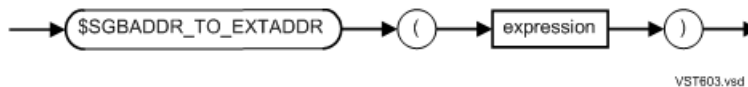
Example 301 Using the \$SCALE Routine to Maintain Precision

```

FIXED(3) num, a, b;      ! fpoint of 3
num := $SCALE (a,3) / b; ! Scale a to FIXED(6); result is a
                        ! FIXED(3) value
  
```

\$SGBADDR_TO_EXTADDR

\$SGBADDR_TO_EXTADDR converts an SGBADDR or SGXBADDR address to an EXTADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is an SGBADDR or SGXBADDR address.

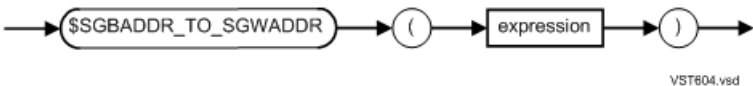
\$SGBADDR_TO_EXTADDR returns *expression* converted to an EXTADDR address.

Example 302 \$SGBADDR_TO_EXTADDR Routine

```
STRING .SG s;
INT    .EXT i;
INT     j;
@i := $SGBADDR_TO_EXTADDR(@s[j]);  !??: OK if @s[j] is at an
                                     ! even-byte offset;
                                     ! otherwise, @i is undefined.
```

\$SGBADDR_TO_SGWADDR

\$SGBADDR_TO_SGWADDR converts an SGBADDR or SGXBADDR address to an SGWADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is an SGBADDR or SGXBADDR address.

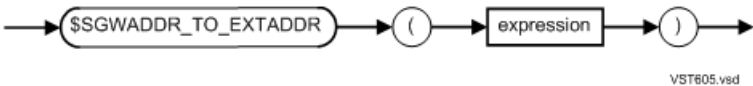
\$SGBADDR_TO_SGWADDR returns *expression* converted to an SGWADDR address. The result is undefined if the least significant bit of *expression* is 1.

Example 303 \$SGBADDR_TO_SGWADDR Routine

```
STRING .SG s;
INT    .SG i;
INT     j;
@i := $SGBADDR_TO_SGWADDR(@s[j]);  !??: OK if @s[j] is at an
                                     ! even-byte offset;
                                     ! otherwise, @i is undefined.
```

\$SGWADDR_TO_EXTADDR

\$SGWADDR_TO_EXTADDR converts an SGWADDR or SGXWADDR address to an EXTADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is *n* SGWADDR or SGXWADDR address.

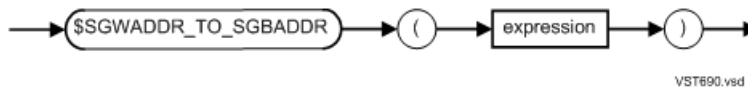
\$SGWADDR_TO_EXTADDR returns *expression* converted to an EXTADDR address.

Example 304 \$SGWADDR_TO_EXTADDR Routine

```
STRING .EXT s;
INT    .SG i;
@s := $SGWADDR_TO_EXTADDR(@i);
```

\$SGWADDR_TO_SGBADDR

\$SGWADDR_TO_SGBADDR converts an SGWADDR or SGXWADDR address to an SGBADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is an SGWADDR or SGXWADDR address.

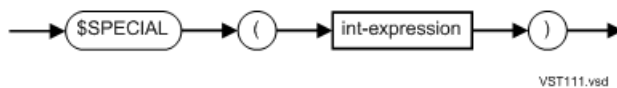
If *expression* is not an address in the lower half of the 64K word segment, the address returned by \$SGWADDR_TO_SGBADDR is undefined.

Example 305 \$SGWADDR_TO_SGBADDR Routine

```
STRING .SG s;
INT    .SG i;
@s := $SGWADDR_TO_SGBADDR(@i); !OK: OK if i is in the
                                ! lower half of system globals
```

\$SPECIAL

\$SPECIAL tests the right byte of an INT value for the presence of an ASCII special (nonalphanumeric) character (see [Table 8 \(page 36\)](#)).



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an INT expression.

\$SPECIAL inspects bits <8:15> of the *int-expression* and ignores bits <0:7>. \$SPECIAL (*int-expression*) has the same value as:

NOT \$NUMERIC(*int-expression*) AND NOT \$ALPHABETIC(*int-expression*)

If the character passes the test, \$SPECIAL returns a -1 (true); otherwise, \$SPECIAL returns a 0 (false).

int-expression can include STRING and UNSIGNED(1-16) operands (see [Expressions as Parameters \(page 275\)](#)).

In [Example 306](#), \$SPECIAL tests for the presence of a special character in a STRING argument, which the system places in the right byte of a word and treats as an INT value.

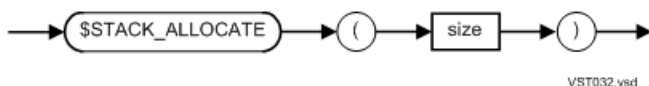
Example 306 \$SPECIAL Routine

```
STRING char;
IF $SPECIAL (char) THEN ... ; ! Test for special character
```

\$STACK_ALLOCATE

NOTE: The pTAL and EpTAL compilers behave differently.

\$STACK_ALLOCATE allocates a block of memory on the stack and returns the address of the block.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

size

is an INT expression that specifies the number of bytes to allocate. *size* is an unsigned value from 0 through 65534.

Difference between pTAL and EpTAL compilers:

pTAL Compiler	EpTAL Compiler
If <i>size</i> is not an integral multiple of 8, \$STACK_ALLOCATE rounds <i>size</i> up to the next integral multiple of 8.	If <i>size</i> is not an integral multiple of 32, \$STACK_ALLOCATE rounds <i>size</i> up to the next integral multiple of 32.
The returned value is aligned to an 8-byte boundary.	The returned value is aligned to a 32-byte boundary.

Blocks returned by multiple calls to \$STACK_ALLOCATE are not necessarily contiguous.

\$STACK_ALLOCATE returns a WADDR address, which is the lowest address in the allocated memory.

\$STACK_ALLOCATE does not clear the allocated data area.

\$STACK_ALLOCATE does not return error conditions, but stack overflow can occur within \$STACK_ALLOCATE or on a subsequent procedure call from within the procedure that calls \$STACK_ALLOCATE.

When a procedure or routine returns to its caller, the system deallocates all memory allocated by \$STACK_ALLOCATE within that procedure.

pTAL does not support calls to \$STACK_ALLOCATE from subprocedures and reports a syntax error if it encounters one. From within a subprocedure, however, you can reference data in a block allocated in the encompassing procedure.

Example 307 \$STACK_ALLOCATE Routine

```
INT .p(template);
INT(32) .a;
INT(32) i32;
...
@p := $STACK_ALLOCATE ($LEN(template));
@a := $STACK_ALLOCATE ($LEN(i32) * 10);
```

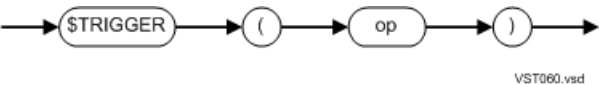
For more information about \$STACK_ALLOCATE, see the *pTAL Conversion Guide*.

\$TRIGGER

NOTE:

- The TAL and pTAL compilers do not support this routine.
- Execution does not return from this call.

\$TRIGGER replaces \$FREEZE (page 315) and \$HALT (page 315), which are available only for code generated for the TNS/R architecture.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

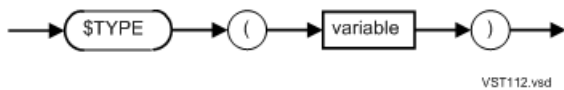
op
is an INT(32) value.

Example 308 \$TRIGGER Routine

```
INT(32) op;
$TRIGGER (op); ! or
call $TRIGGER (op);
```

\$TYPE

\$TYPE returns an INT value that represents the data type of a variable.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is the identifier of a simple variable, array, simple pointer, structure, structure data item, or structure pointer.

\$TYPE returns an INT value that has a meaning as follows:

Value	Meaning	Value	Meaning
0	Undefined	5	REAL
1	STRING	6	REAL(64)
2	INT	7	Substructure
3	INT(32)	8	Structure
4	FIXED	9	UNSIGNED

For a structure pointer, \$TYPE returns the value 8, regardless of whether the structure pointer points to a structure or to a substructure.

You can use \$TYPE in LITERAL expressions and global initializations, because \$TYPE always returns a constant value.

Example 309 \$TYPE Routine

```
REAL(64) var1;
INT type1;
type1 := $TYPE (var1);  ! Return 6 for REAL(64)
```

\$UDBL

\$UDBL converts an unsigned INT value to an INT(32) value.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

int-expression

is an unsigned INT expression.

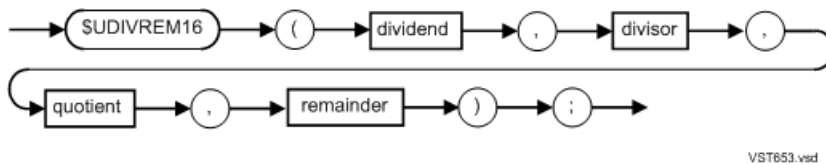
\$UDBL places the INT value in the low-order 16 bits of an INT(32) variable and sets the high-order 16 bits to 0.

Example 310 \$UDBL Routine

```
INT a16 := -1;s
INT(32) a32;
a32 := $UDBL (a16); ! Return 65535D
```

\$UDIVREM16

\$UDIVREM16 divides an INT(32) dividend by an INT divisor to produce an INT quotient and INT remainder.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes, if the divisor is 0 or the quotient is too large

dividend

input

INT(32):value

divisor

input

sINT:value

quotient

output

sINT:variable

remainder

output

sINT:variable

The compiler checks the following conditions during compilation:

- If the value of *divisor* is a constant value of zero, the compiler reports an error that division by zero is not valid:
`$UDIVREM16(dividend, 2 / 2 - 1, quot, rem); ! Report error`
- If both *dividend* and *divisor* have constant values whose unsigned quotient is greater than 16 bits, the compiler reports overflow:
`INT quot, rem;`
`$UDIVREM16(65536 * 1024, 256, quot, rem); ! Report error`
- If both *dividend* and *divisor* are constants, and you test \$OVERFLOW following the call to \$UDIVREM16, the compiler reports a warning that overflow cannot occur:
`$UDIVREM16(32767, 256, quot, rem);`

IF \$OVERFLOW THEN ...

! Report warning

If the compiler reports an error because overflow occurs for constant dividend and constant divisor, it does not report a warning if you test \$OVERFLOW in the following IF statement:

\$UDIVREM16(65536 * 1024, 256, quot, rem); ! Report error

IF \$OVERFLOW THEN....

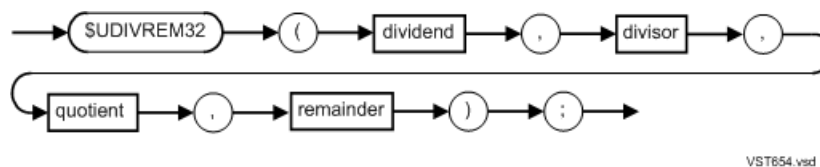
! No warning or error

Example 311 \$UDIVREM16 Routine

```
INT(32) dividend;
INT      divisor;
INT      quotient;
INT      remainder;
$UDIVREM16(dividend, divisor, quotient, remainder);
```

\$UDIVREM32

\$UDIVREM32 divides an INT(32) dividend by an INT divisor to produce an INT(32) quotient and INT remainder.



VST654.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes, if and only if the divisor is 0

dividend

input

INT(32):value

divisor

input

sINT:value

quotient

output

INT(32):variable

remainder

output

sINT:variable

The compiler checks the following conditions during compilation:

- If the value of *divisor* is a constant value of zero, the compiler reports an error that division by zero is not valid:
\$UDIVREM32(dividend, 2 / 2 - 1, quot, rem); ! Report error
- If both *dividend* and *divisor* are constants, and you test \$OVERFLOW following the call to \$UDIVREM32, the compiler reports a warning that overflow cannot occur:

```

$UDIVREM32(32767, 256, quot, rem);
IF $OVERFLOW THEN ...                                ! Report warning

```

If the compiler reports an error because overflow occurs for constant dividend and constant divisor, it does not report a warning if you test \$OVERFLOW in the following IF statement:

```

$UDIVREM32(65536 * 1024, 256, quot, rem); ! Report error
IF $OVERFLOW THEN....                     ! No warning or error

```

Example 312 \$UDIVREM32 Routine

```

INT(32) dividend;
INT      divisor;
INT(32) quotient;
INT      remainder;
$UDIVREM32(dividend, divisor, quotient, remainder);

```

\$UFIX

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$UFIX returns the FIXED-type zero-extended value of the specified INT(32)-typed expression.



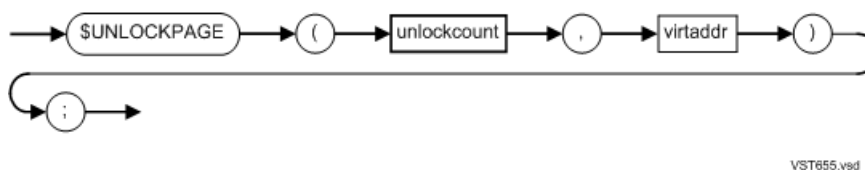
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression
is INT(32) expression.

\$UNLOCKPAGE

NOTE: The EpTAL compiler does not support this procedure.

\$UNLOCKPAGE unlocks one page of memory.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes

Sets \$CARRY	No
Sets \$OVERFLOW	No

.

unlockcount
input
sINT:value
is the total number of bytes to unlock in the page.

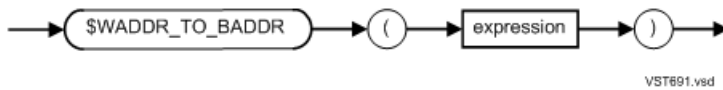
virtaddr
input
EXTADDR:value
is the beginning virtual address to unlock. \$UNLOCKPAGE calculates the page associated with virtaddr.

Example 313 \$UNLOCKPAGE Routine

```
INT      unlockcount;
EXTADDR  addr;
$UNLOCKPAGE(unlockcount, addr);
```

\$WADDR_TO_BADDR

\$WADDR_TO_BADDR converts a WADDR address to a BADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

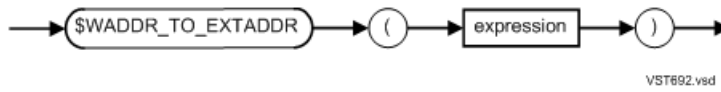
expression
is an expression whose value is a WADDR address.

Example 314 \$WADDR_TO_BADDR Routine

```
STRING .s;
INT     t;
@s := $WADDR_TO_BADDR(@t); ! @t is a WADDR address
```

\$WADDR_TO_EXTADDR

\$WADDR_TO_EXTADDR converts a WADDR address to an EXTADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

expression

is an expression whose value is a WADDR address.

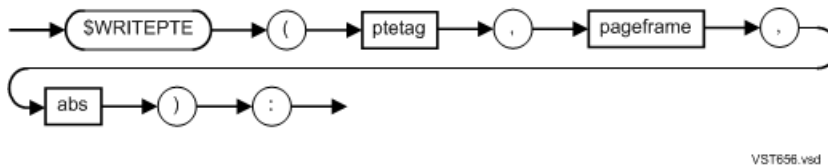
Example 315 \$WADDR_TO_EXTADDR Routine

```
STRING .EXT s;
INT      t;
@s := $WADDR_TO_EXTADDR(@t); ! @t is a WADDR address
```

\$WRITEPTE

NOTE: The EpTAL compiler does not support this procedure.

\$WRITEPTE writes a segment-page-table entry.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

ptetag

input

uINT:value

are the page attribute bits associated with *pageframe*.

pageframe

input

INT(32):value

is the frame number of the physical frame associated with *abs*.

abs

input

EXTADDR:value

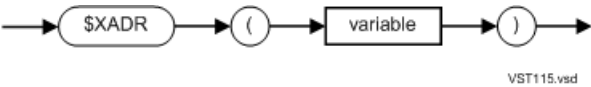
is the virtual address to which \$WRITEPTE maps *pageframe*.

Example 316 \$WRITEPTE Routine

```
INT      ptetag;  
INT(32)  pageframe;  
EXTADDR  abs;  
$WRITEPTE(ptetag, pageframe, abs);
```

\$XADR

\$XADR converts a standard address to an EXTADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is a variable that has a standard, extended, or system-global address.

\$XADR returns an EXTADDR address. If the argument to \$XADR is not a variable, the compiler reports an error.

\$XADR returns an absolute extended EXTADDR address in absolute segment 1 if variable is a system global address (an SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR address).

Variable can be the name of a pointer preceded by an “@” operator. In this case, \$XADR returns the absolute address of the pointer, as in the following example.

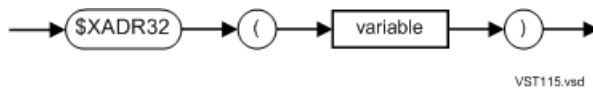
Example 317 \$XADR Routine

```
PROC p;  
BEGIN  
  INT .p;  
  INT .EXT e;  
  ...  
  @e := $XADR(@p);  
  ...  
END;
```

\$XADR32

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$XADR converts a standard extended, or extended system-global address to an EXT32ADDR address. No check is performed to determine if the resulting address is valid.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is a variable that has a standard, extended, or system-global address.

\$XADR returns an EXT32ADDR address. The compiler reports an error if there is no explicit conversion defined from the address type of the variable to EXT32ADDR or EXTADDR.

\$XADR32 returns an absolute extended EXT32ADDR address in absolute segment 1 if variable is an extended system global address (an SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR).

Variable can be the name of a pointer preceded by an "@" operator. In this case, \$XADR32 returns the EXT32ADDR address of the pointer.

Example 318 \$XADR32 Routine

```

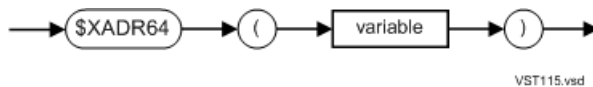
PROC p;
BEGIN
    INT .p;
    INT .EXT32 e;
    ...
    @e := $XADR32(@p);
    ...
END;

```

\$XADR64

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, ["64-bit Addressing Functionality"](#) (page 531).

\$XADR64 converts a standard extended, or extended system-global address to an EXT64ADDR address.



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

variable

is a variable that has a standard, extended, or system-global address.

\$XADR64 returns an EXT64ADDR address. The compiler reports an error if there is no explicit conversion defined from the address type of the variable to EXT64ADDR.

\$XADR64 returns an absolute extended EXT64ADDR address in absolute segment 1 if variable is an extended system global address (an SGBADDR, SGWADDR, SGXBADDR, or SGXWADDR).

Variable can be the name of a pointer preceded by an "@" operator. In this case, \$XADR64 returns the EXT64ADDR address of the pointer.

Example 319 \$XADR64 Routine

```
PROC p;
BEGIN
  INT .p;
  INT .EXT64 e;
  ...
  @e := $XADR64(@p);
  ...
END
```

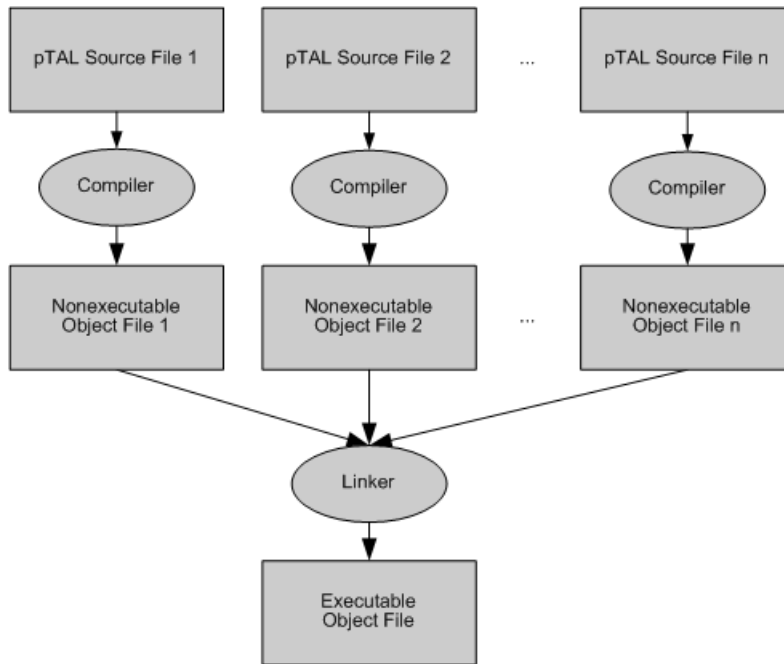
16 Compiling and Linking pTAL Programs

Input to the compiler is a source file containing pTAL source text (such as data declarations, statements, compiler directives, and comments).

Output from the compiler is a linkfile consisting of relocatable code and data blocks.

To produce an executable pTAL program, link one or more linkfiles into a single loadfile (see [Figure 13](#)).

Figure 13 Compiling and Linking pTAL Programs



VST699.vsd

Topics:

- [Compiling Source Files \(page 355\)](#)
- [Linking Object Files \(page 358\)](#)
- [Creating a Dynamic Linked Library \(DLL\) \(page 362\)](#)
- [Compiling With Global Data Blocks \(page 362\)](#)
- [Compiling With Saved Global Data \(page 366\)](#)
- [Using the Code Profiling Utilities \(page 366\)](#)

NOTE: The remainder of this section applies only to Guardian platforms. To compile and link pTAL programs on Windows platforms, see [Chapter 18 \(page 426\)](#).

Compiling Source Files

The compiler reads input files, produces output files, and uses swap files and temporary files as needed. On Guardian platforms, you use HP TACL commands to compile source files. The compiler accepts information you specify in HP TACL commands (DEFINE, PARAM, and ASSIGN) if you issue them before you run the compiler. For a summary of HP TACL commands, see [Appendix B \(page 518\)](#).

Example 320 Compiler Command Lines

```
ptal / in test, out $s.#test, nowait/ testobj; symbols  
eptal / in test, out $s.#test, nowait/ testobj; symbols
```

The compiler reads input only from a single edit-format disk file. You can use the [SOURCE \(page 514\)](#) in this input file to read code from other source files during compilation. The input file and code read from other source files comprise a compilation unit.

In general, the compiler opens each source file as it needs the source file and keeps the source file open until the end of the compilation. This behavior ensures that the contents of the file cannot change between the time the compiler reads the file and creates a listing. You can open a source file for read access, but generally not for write access, while the source file is compiling.

When the number of files read exceeds the maximum number of files that Guardian allows to be open, the compiler closes the least recently used file (unless that file is the primary source file, which is always kept open) in order to continue to open and read source files.

If you edit a file before the compiler creates an output listing, the source code in the listing will not match the code in the source file. The compiler reports a warning if it discovers that part of a source file has changed. Do not alter the source files until the compilation ends.

Topics:

- [Input Files \(page 356\)](#)
- [Output Files \(page 356\)](#)
- [Running the Compiler \(page 357\)](#)
- [Completion Codes Returned by the Compiler \(page 358\)](#)

Input Files

The compiler reads input only from an edit-format disk file up to a maximum of 132 characters for each record, ignoring characters after the 132nd (and issuing a warning for each such line). The compiler does not read input from a terminal or from any other source or file format.

Output Files

You can direct list output from the compiler to any of the following types of files:

- Spooler
- Entry-sequenced file
- Relative file
- Terminal
- Process
- Printer
- Edit-format file
- HP TACL variable

If you direct output to a disk file that does not exist, the compiler creates an edit-format file and writes the compiler listing to the newly created file.

If you direct output to an edit-format file that already exists, the compiler removes the existing file from your current compilation and creates a new file using the file name you specified.

Difference between pTAL and EpTAL compilers:

pTAL Compiler	EpTAL Compiler
On Guardian platforms, object files have the file code 700	On Guardian platforms, object files have the file code 800

Running the Compiler

To run the compiler on Guardian platforms, issue a compilation command at the HP TACL prompt. Options that you can specify in the compilation command are:

- [IN File Option \(page 357\)](#)
- [OUT File Option \(page 357\)](#)
- [HP TACL Run Options \(page 357\)](#)
- [Target File Option \(page 358\)](#)

You can include one or more compiler directives in the compilation command (see [Compilation Command \(page 367\)](#)).

IN File Option

The IN file is the primary source file. You can specify a file name or a DEFINE as described in [Appendix B \(page 518\)](#). In this example, the IN file is `mysource`.

```
pTAL /IN mysource/ myobject
EpTAL /IN mysource/ myobject
```

The IN file must be an edit-format disk file. The compiler reads the file as 132-byte records.

OUT File Option

The OUT file receives the compiler listings. The OUT file can be any of the files listed in [Output Files \(page 356\)](#).

In an unstructured disk file, each record has 132 characters; partial lines are filled with blanks through column 132. You can specify a file name or a DEFINE name. The OUT file is often a spooler location, such as `$s.#lists` in the following example:

```
PTAL /IN mysource, OUT $s.#lists/ myobject
EpTAL /IN mysource, OUT $s.#lists/ myobject
```

If you omit OUT and the HP TACL product is in interactive mode, the listings go to the home terminal. In noninteractive mode, the listings go to the current HP TACL OUT file:

```
PTAL /IN mysource/ myobject
EpTAL /IN mysource/ myobject
```

HP TACL Run Options

You can include one or more HP TACL run options in the compilation command, such as:

- A process name
- A CPU number
- A priority level
- The NOWAIT option
- A swap volume

For example, you can specify CPU 3 and NOWAIT when you run the compiler:

```
pTAL /IN mysource, CPU 3, NOWAIT/ myobject
EpTAL /IN mysource, CPU 3, NOWAIT/ myobject
```

For information about HP TACL run options, see the RUN command in the *TACL Reference Manual*.

By default, the compiler and its processes can run at a high PIN. If your compilation accesses files on systems running C-series software, you must run the compiler at a low PIN.

To run the compiler at a low PIN, set HIGHPIN OFF, as shown in the following HP TACL command:

```
SET HIGHPIN OFF
```

When you set HIGHPIN OFF in the HP TACL program, the program runs all processes at a low PIN except processes that explicitly specify the HIGHPIN ON option when the process is created.

To ensure that the compiler runs at a low PIN without affecting other processes, specify the run command's HIGHPIN OFF option, as in the following example:

```
PTAL / HIGHPIN OFF .../ ...  
EPAL / HIGHPIN OFF .../ ...
```

Target File Option

The target file is the disk file that is to receive the object code. You can specify a file name or a DEFINE name as described in [Appendix B \(page 518\)](#).

These examples write the object code to a disk file named myobject:

```
pTAL /IN mysource/ myobject  
EpTAL /IN mysource/ myobject
```

If you omit the target file, the compiler creates a file named object on your current default subvolume.

If an existing file has the name object or the name you specify, and the existing file has the correct filecode (700 for the pTAL compiler, 800 for EpTAL compiler), the compiler overwrites the existing file. (The compiler overwrites the existing file by purging it and then creating a new file that has the same name and filecode.)

If the compiler cannot purge the existing file, the compiler creates a file named ZZPTnnnn, where nnnn is a different number each time.

Completion Codes Returned by the Compiler

When the compiler compiles a source file, it either completes the compilation normally or stops abnormally. It then returns a process-completion code to the HP TACL product indicating the status of the compilation.

Table 73 Completion Codes

Code	Termination	Meaning
0	Normal	The compiler found no errors or unsuppressed warnings in the source file. (Warnings suppressed by the NOWARN directive do not count.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
1	Normal	The compiler found at least one unsuppressed warning. (Warnings suppressed by the NOWARN directive have no effect.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
2	Normal	The compiler found at least one compilation error and did not create an object file, but completed processing the source.
3	Abnormal	The compiler exhausted an internal resource such as symbol table space or could not access an external resource such as a file. The compiler did not create an object file.
8	Normal	The compiler could not use the object file name you specified, so it chose the name reported in the summary. The object file is complete and valid.

Linking Object Files

The linker links one or more linkfiles to produce either a loadfile or another linkfile.

The compiler and compiler directives you use determine the linker you must use and the kind of executable object code that is produced:

Compiler	Compiler Directive	Linker	Object Code
EpTAL	CALL_SHARED (default)	eld	PIC
	NOCALL_SHARED (error)		
pTAL	CALL_SHARED	ld	PIC
	NOCALL_SHARED (default)	nld	Non-PIC

The linker can also strip nonessential information from an object file and modify the object file's process attributes (such as HIGHPIN). For more information, see:

- *eld Manual*
- *ld Manual*
- *nld Manual*

The simplest cases are:

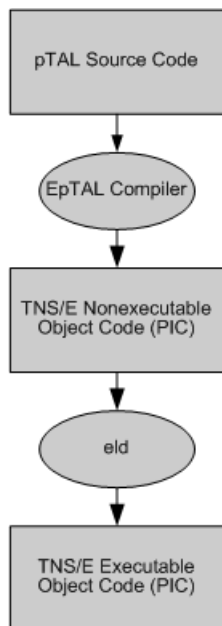
- On TNS/E, use the EpTAL compiler and the `eld` utility to create an object file that executes on TNS/E (see [Figure 14 \(page 359\)](#)).
- On TNS/R, use the pTAL compiler and either the `ld` or `nld` utility to create an object file that executes on TNS/R (see [Figure 15 \(page 360\)](#)).

Also, TNS allows you to create object files that execute on TNS/R. Use the TAL compiler and Binder on TNS and the Accelerator (AXCEL) on either TNS or TNS/R to create an object file that executes on TNS/R (see [Figure 16 \(page 361\)](#)). (In this case, you begin with TAL source code rather than pTAL source code.)

You can input some kinds of loadfiles to the Accelerator (AXCEL) and the Object Code Accelerator (OCA) to produce hybrid loadfiles (see [Figure 17 \(page 362\)](#)).

You cannot link PIC and non-PIC object files into a single object file.

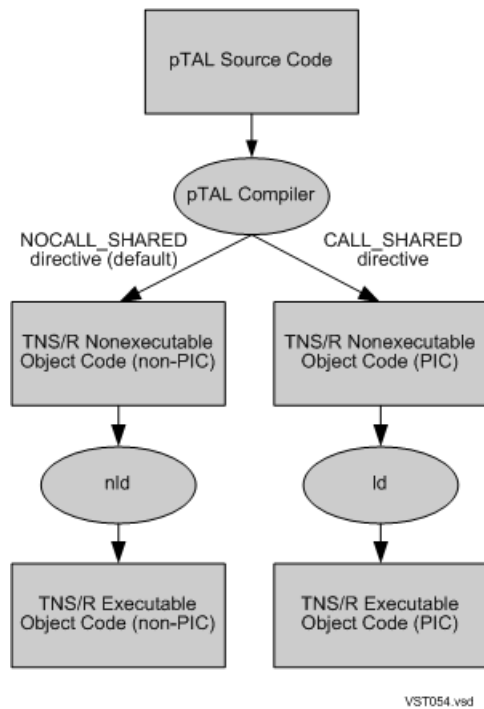
Figure 14 Creating a Loadfile on TNS/E for TNS/E



VST045.vsd

The source code can be in one or more files. From each source code file, the compiler generates a single nonexecutable object code file. Input these object code files to the linker to produce a single loadfile. (See [Figure 13 \(page 355\)](#).)

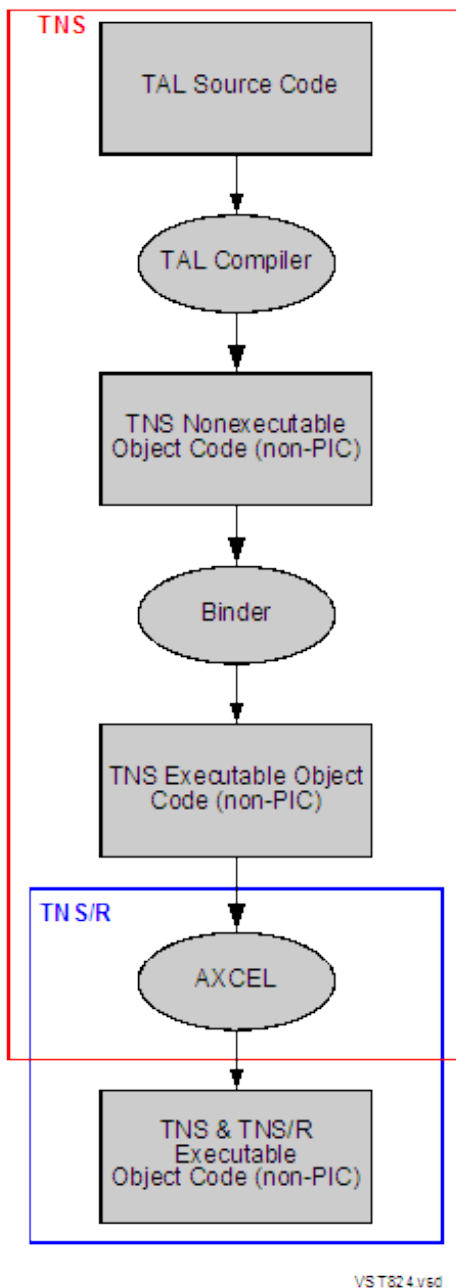
Figure 15 Creating Loadfiles on TNS/R for TNS/R



The source code can be in one or more files. From each source code file, the compiler generates a single nonexecutable object code file.

If you compile multiple source files, either compile all of them using the `CALL_SHARED` directive or all of them without using the `CALL_SHARED` directive (you cannot link PIC and non-PIC object files into a single object file). Input these object code files to the appropriate linker to produce a single loadfile. (See [Figure 13 \(page 355\)](#).)

Figure 16 Creating a Loadfile on TNS for TNS/R



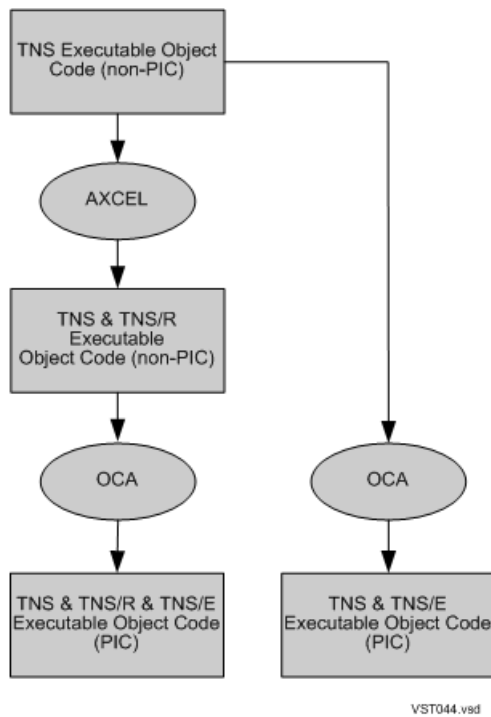
The source code can be in one or more files. From each source code file, the compiler generates a single nonexecutable object code file. Input these object code files to Binder to produce a single loadfile. (Figure 13 (page 355) illustrates this concept, but uses a linker instead of Binder.)

As Figure 16 (page 361) shows, the Accelerator (AXCEL) is available on both TNS/R and TNS processors; therefore, you can do either of the following:

- Accelerate your TNS executable object code while it is on a TNS processor and then move the resulting executable object code to a TNS/R processor.
- Move your TNS executable object code to a TNS/R processor and then accelerate it.

In both cases, the resulting executable object code executes only on TNS/R processors.

Figure 17 Producing Hybrid Loadfiles



AXCEL is available on TNS/E, TNS/R, and TNS processors.

OCA is available on TNS/E and TNS/R processors.

Non-PIC hybrid loadfiles run on the TNS/R architecture. PIC hybrid loadfiles run on the TNS/E architecture.

Creating a Dynamic Linked Library (DLL)

To create a dynamic-link library (DLL) from pTAL source files, compile the pTAL source files by using the `CALL_SHARED` directive (in the Guardian environment) or the `-call_shared` flag (in the Windows environment), and then use `ld` or `elld` to link the pTAL source files through the `-shared` option.

The compiler does not automatically export program names. You must specify `-export_all` or `-export` to the linker.

Compiling With Global Data Blocks

When you compile modules of a program separately or link pTAL code with code written in other languages, the linking process relocates some of your global data.

Topics:

- [Declaring Global Data \(page 362\)](#)
- [Allocating Global Data Blocks \(page 365\)](#)
- [Address Assignments \(page 365\)](#)
- [Sharing Global Data Blocks \(page 365\)](#)

Declaring Global Data

You can declare blocked and unblocked global data (variables, LITERALS, and DEFINES).

Blocked global data declarations are those appearing within `BLOCK` declarations. `BLOCK` declarations let you group global data declarations into named or private blocks. Named blocks are shareable among all compilation units in a program. The private block is private to the current

compilation unit. If you include a BLOCK declaration in a compilation unit, you must assign an identifier to the compilation unit by using a NAME declaration.

Unblocked global data declarations are those appearing outside a BLOCK declaration. Such declarations are also relocatable and shareable among all compilation units in a program.

If you do not use the BLOCKGLOBALS directive, then all separate compilations must specify exactly the same list of unblocked global data declarations.

If present in a compilation unit, global declarations must appear in the following order:

1. NAME declaration
2. Unblocked global data declarations
3. BLOCK declarations
4. PROC declarations

Topics:

- [Naming Compilation Units \(page 363\)](#)
- [Declaring Named Data Blocks \(page 363\)](#)
- [Declaring Private Data Blocks \(page 364\)](#)
- [Declaring Unblocked Data \(page 364\)](#)

Naming Compilation Units

To assign an identifier to a compilation unit, specify the NAME declaration as the first declaration in the compilation unit. (If no BLOCK declaration appears in the compilation unit, you need not include the NAME declaration.) In the NAME declaration, specify an identifier that is unique among all BLOCK and NAME declarations in the target file.

Example 321 Naming a Compilation Unit

```
NAME input_module;    ! Name the compilation unit
```

Declaring Named Data Blocks

A named data block is a global data block that is shareable among all compilation units in a program. You can include any number of named data blocks in a compilation unit. To declare a named data block:

- Put a NAME declaration in the compilation (see [Naming Compilation Units \(page 363\)](#)).
- Specify an identifier in the BLOCK declaration that is unique among all BLOCK and NAME declarations in the target file.

Example 322 Declaring a Named Data Block

```
BLOCK globals;                ! Declare named data block
  INT .vol_array[0:7];         ! Declare global data
  INT .out_array[0:34];
  DEFINE xaddr = INT(32)#;
END BLOCK;
```

A variable declared in a named data block can have the same name as the data block. Modules written in pTAL can share global variables with modules written in HP C by placing each shared variable in its own block and giving the variable and the block the same name.

Example 323 Data Block and Variable With the Same Name

```
BLOCK c_var;  
    INT c_var;  
END BLOCK;
```

Declaring Private Data Blocks

A private data block is a global data block that is shareable only among the procedures within a compilation unit. You can include only one private data block in a compilation unit. The private data block inherits the identifier you specify in the NAME declaration; therefore, the NAME declarations in all compilations that you use to assemble an executable program must have unique names. To declare a private global data block, specify the PRIVATE option of the BLOCK declaration.

Example 324 Declaring a Private Data Block

```
BLOCK PRIVATE;                ! Declare private global data block  
    INT term_num;             ! Declare global data  
    LITERAL msg_buf = 79;  
END BLOCK;
```

Declaring Unblocked Data

Place all unblocked global declarations (those not contained in BLOCK declarations) before the first BLOCK declaration. Unblocked declarations are relocatable and shareable among all compilation units in a program. The linking name of the private data block is derived from the NAME declaration.

Example 325 Declaring Unblocked Data

```
INT a;  
INT .b[0:9];  
INT .EXT c[0:14];  
LITERAL limit = 32;
```

The compiler places unblocked data declarations in implicit primary data blocks, created as follows:

1. When you use named blocks, private blocks, or the BLOCKGLOBALS directive, each data item becomes its own block. All the other unblocked data items are grouped into a block named `_GLOBAL` and `$_GLOBAL`.
2. Each block so created is split into two blocks to separate “large” data from “small” data. “Large” data means arrays or structures declared with “.” or “.EXT” notation. “Small” data is everything else. When both blocks exist, the “large” data block has a \$ in front of its name.

For example, if you have the following global data declarations:

```
INT x;  
INT .y;  
INT .z [0:113]
```

Variables `x` and `y` are placed in the block named `_GLOBAL`, and `z` is placed in the block named `$_GLOBAL`.

Named data blocks are split the same way. For example:

```
BLOCK blk;  
    INT x;  
    INT .y;  
    INT .ext z [0:99];  
END BLOCK;
```

Two data blocks are created. Variables `x` and `y` are placed in the block named `BLK` and `z` is placed in the block named `$BLK`.

You can link object files compiled with and without template blocks with no loss of information.

A referral structure and the structure layout to which it refers can appear in different data blocks. The structure layout must appear first.

In all other cases, a data declaration and any data to which it refers must appear in the same data block. The following declarations, for example, must appear in the same data block:

```
INT var;                ! Declare var
INT .ptr := @var;       ! Declare ptr by referring to var
```

If the reference is not in the same block, the compiler issues an error message.

Allocating Global Data Blocks

When you compile a program, the compiler constructs relocatable blocks of code and data that are linked into the object file. The compiler:

- Allocates each read-only array in its own data block in the code segment of the object file
- Allocates all other variables in relocatable global data blocks in the data segment (except LITERALS and DEFINES, which require no storage space)

Data is divided between “large” and “small” data sections.

The compiler associates the symbol information for the allocated variables with that data block. The compiler also associates the symbol information for any LITERALS, DEFINES, or read-only arrays declared in that data block, but allocates 0 words of storage for such declarations.

Address Assignments

The compiler assigns each direct variable and each pointer an offset from the beginning of the encompassing global data block. Within the data block, it allocates storage for each data declaration according to its data type and size.

Sharing Global Data Blocks

Because the length of any shared data block must match in all compilation units, it is recommended that you declare all shareable global data in one source file. You can then share that global data block with other source files as follows:

1. In the source file that declares the data block, specify the SECTION directive at the beginning of the data block to assign a section name to the data block. The SECTION directive remains active until another SECTION directive or the end of the source file occurs:

```
NAME calc_unit;
?SECTION unblocked_globals ! Name first section
  LITERAL true  = -1,      ! Implicit data block
        false  =  0;
  STRING read_only_array = 'P' := [ " ", "COBOL", "FORTRAN",
                                     "PASCAL", "pTAL" ];
?SECTION default           ! Name second section
  BLOCK default_vol;       ! Declare named block
    INT .vol_array [0:7],
        .out_array [0:34];
  END BLOCK;
?SECTION msglits           ! Name third section
  BLOCK msg_literals;      ! Declare named block
    LITERAL msg_eof  = 0,
          msg_open   = 1,
          msg_read    = 2;
  END BLOCK;               ! End msglits section
?SECTION end_of_data_sections
```

2. In each source file that needs to include the sections, specify the file name and the section names in a SOURCE directive:

```
NAME input_file;  
?SOURCE calcsrc(unblocked_globals)    ! Specify implicit block  
?SOURCE calcsrc(default)              ! Specify named block
```

3. If you then change any declaration within a data block that has a section name, you must recompile all source files that include SOURCE directives listing the changed data block.

Compiling With Saved Global Data

NOTE: This topic applies only to the pTAL compiler. If you are using the EpTAL compiler, see [Migrating from TNS/R to TNS/E \(page 375\)](#).

During program development or maintenance, you often need to change procedural code or data without changing the global declarations. You can save the global data in a file during a compilation session and then use the saved global data during a subsequent compilation. You can shorten the compile time by not compiling global declarations each time. For more information, see [Saving and Using Global Data Declarations \(page 372\)](#).

Using the Code Profiling Utilities

The Code Profiling Utilities provide these capabilities

- Evaluate the code coverage provided by application test cases. The utilities use information provided by a specially-instrumented object file to produce a report that indicates which functions and blocks were executed, and how many times each was executed.
- Optimize an application through a process called profile-guided optimization. In profile-guided optimization, a specially-instrumented object file is executed to produce a data file containing code profiling information. That data file, along with the original source code, is input to the compiler to generate more efficient object code.

Using the Code Profiling Utilities requires a special compilation to produce an object file containing the required instrumentation. To create such an object file, specify the CODECOV or PROFGEN option on the compiler command line. Several other compiler options are related to code profiling. These are the PROFDIR, PROFUSE, and BASENAME options.

NOTE: The Code Profiling Utilities are intended for data generation and collection in a test environment only. The use of instrumented object code is not recommended for production environments. Applications compiled with code profiling instrumentation will experience greatly reduced performance.

For details on using the Code Profiling Utilities, see the *Code Profiling Utilities Manual*.

17 Compiler Directives

Topics:

- [Specifying Compiler Directives \(page 367\)](#)
- [File Names as Compiler Directive Arguments \(page 368\)](#) (Guardian platforms only)
- [Directive Stacks \(page 369\)](#)
- [Toggles \(page 370\)](#)
- [Saving and Using Global Data Declarations \(page 372\)](#)
- [Summary of Compiler Directives \(page 377\)](#)
- Topics for individual compiler directives, beginning with [ASSERTION \(page 381\)](#)

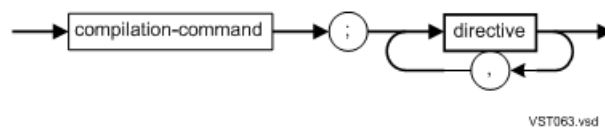
Specifying Compiler Directives

You can specify compiler directives either in the compilation command or in a directive line in the source code, unless otherwise specified. The compiler interprets and processes each directive at the point of occurrence.

Topics:

- [Compilation Command \(page 367\)](#)
- [Directive Line \(page 367\)](#)

Compilation Command



compilation-command

is as described in [Running the Compiler \(page 357\)](#).

directive

is a directive listed in [Table 74 \(page 377\)](#) or [Table 75 \(page 379\)](#), except the following, which can appear only in the source file (see [Compilation Command \(page 367\)](#)):

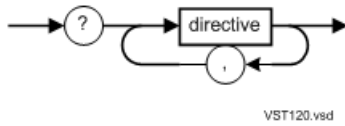
- [ASSERTION \(page 381\)](#)
- [BEGINCOMPILE \(page 382\)](#) (not recommended)
- [ENDIF \(page 390\)](#)
- [IF and IFNOT \(page 398\)](#)
- [PAGE \(page 407\)](#)
- [SECTION \(page 414\)](#)
- [SOURCE \(page 416\)](#)

Example 326 Compilation Commands With Compiler Directives

```
EPTAL /IN mysrc, OUT $s.#lists/ myobj; NOMAP, NOLIST
pTAL /IN mysrc, OUT $s.#lists/ myobj; NOMAP, NOLIST
```

Directive Line

The general form of a directive line is:



?

indicates a directive line, and can appear only in column 1.

directive

is a directive listed in [Table 74 \(page 377\)](#) or [Table 75 \(page 379\)](#), except OPTIMIZEFILE, which can appear only in the command line (see [Compilation Command \(page 367\)](#)).

Rules for directive lines:

- Begin each directive line by specifying ? in column 1. (? is not part of the directive name.)
- Place the name of the directive and its arguments on the same line unless the directive description says you can use continuation lines.
- Do not put extra characters (such as semicolons) at the end of a directive line.
- Do not use an equal sign (=) in the directive unless the directive's syntax includes one (as in [ASSERTION \(page 381\)](#)).

Rules for continuation lines:

- Begin each continuation line by specifying ? in column 1.

```
?NOLIST, SYMBOLS, NOMAP, GMAP
?INNERLIST
```
- Place the opening parenthesis of the argument list on the same line as the directive name.

```
?NOLIST, SOURCE $system.system.extdecs (
? process_getinfo_,
? process_stop_)
```

File Names as Compiler Directive Arguments

NOTE: This topic applies only to Guardian platforms, not Windows platforms.

The following directives accept [Disk File Names \(page 518\)](#), DEFINE names, and ASSIGN names as arguments:

- [ERRORFILE \(page 391\)](#)
- [SAVEGLOBALS \(page 413\)](#) (not recommended)
- [SOURCE \(page 416\)](#)
- [USEGLOBALS \(page 423\)](#) (not recommended)

A DEFINE name or an ASSIGN name is considered a logical file name (see [Logical File Names \(page 520\)](#)). The directives listed above accept a logical file name in place of a file name.

You can specify partial file names. If you specify a partial file name, the compiler uses default values as described in [Partial File Names \(page 519\)](#).

For the USEGLOBALS directive (not recommended) and the SOURCE and directive, the compiler can use the node (system), volume, and subvolume specified in ASSIGN SSV commands, as in [Example 328 \(page 372\)](#).

Directive Stacks

Each of these directives has a compile-time directive stack onto which you can push, and from which you can pop, directive settings:

- [CHECKSHIFTCOUNT](#) (page 384)
- [DEFEXPAND](#) (page 386)
- [DO_TNS_SYNTAX](#) (page 389)
- [GP_OK](#) (page 397)
- [INNERLIST](#) (page 400)
- [LIST](#) (page 401)
- [MAP](#) (page 402)
- [OVERFLOW_TRAPS](#) (page 406)
- [REFALIGNED](#) (page 410)

Each directive stack is 31 levels deep.

Topics:

- [Pushing Directive Settings](#) (page 369)
- [Popping Directive Settings](#) (page 369)
- [Example](#) (page 369)

Pushing Directive Settings

When you push the current directive setting onto a directive stack, the current directive setting of the source file remains unchanged until you specify a new directive setting.

To push a directive setting onto a directive stack, specify the directive name prefixed by PUSH. For example, to push the current setting of the LIST directive onto the LIST directive stack, specify PUSHLIST. The other values in the directive stack move down one level. If a value is pushed off the bottom of the directive stack, that value is lost. No diagnostic message is issued if too many items are pushed onto the stack.

Popping Directive Settings

To restore the top value from a directive stack as the current setting from the source file, specify the directive name prefixed by POP. For example, to restore the top value off the LIST directive stack, specify POPLIST. The remaining values in the directive stack move up one level, and the vacated level at the bottom of the stack is set to the off state. No diagnostic message is issued if too many items are popped from the stack.

Example

In [Example 327](#) (page 370):

1. LIST is the default setting for the source file.
2. PUSHLIST pushes the LIST directive setting onto the LIST directive stack.
3. NOLIST suppresses listing of procedures included by the SOURCE directive.
4. POPLIST pops the top value from the LIST directive stack and restores LIST as the current setting for the remainder of the source file.

Example 327 Pushing and Popping a Directive Stack

```
! LIST is the default setting for the source file
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
? PROCESS_GETINFO_, FILE_OPEN_, WRITEREADX, READX)
?POPLIST
```

Toggles

Toggles allow these directives to effect conditional compilation:

Directive	Description
DEFINETOG	Specifies toggles without changing their settings. If DEFINETOG is specifying a toggle for the first time, its setting is off.
SETTOG	Specifies toggles and turns them on
RESETTOG	Specifies toggles and turns them off
IF and IFNOT	Begin conditional compilation, based on the value of a specified toggle
ENDIF	Ends conditional compilation

Topics:

- [Named Toggles \(page 370\)](#)
- [Numeric Toggles \(page 370\)](#)
- [Examples \(page 371\)](#)

Named Toggles

Before you use a named toggle in an IF or IFNOT directive, you must specify that name in a DEFINETOG, SETTOG, or RESETTOG directive. Which of these directives you use depends on whether you want the setting of the toggle to be unchanged, turned on, or turned off.

Directive	Setting	
	New Toggle	Specified Existing Toggle
DEFINETOG	Off	Unchanged
SETTOG	On	On
RESETTOG	Off	Off

You can use DEFINETOG if you are not sure the toggles were created earlier in the compilation, possibly in a file that you included by using a SOURCE directive. If you specify toggles that already exist, DEFINETOG does not change their settings (as SETTOG and RESETTOG do).

Numeric Toggles

The numeric toggles are 1 through 15. All other toggles (including 16, 17, and so on) are considered named toggles.

You can use a numeric toggle in an IF or IFNOT directive even if that toggle has not been specified in a DEFINETOG, SETTOG, or RESETTOG directive.

By default, all numeric toggles not turned on by SETTOG are turned off. To turn off numeric toggles turned on by SETTOG, use RESETTOG.

Examples

- [Example 328 \(page 372\)](#)
- [Example 329 \(page 372\)](#)
- [Example 330 \(page 372\)](#)
- [Example 331 \(page 372\)](#)
- [Example 332 \(page 372\)](#)

Example 328 DEFINETOG, IF, and ENDIF Directives

```
?DEFINETOG scanner    ! Define toggle
...
?IF scanner            ! Test toggle for on state
PROC skipped;          ! Find it off, skip procedure
    BEGIN
        ...
    END;
?ENDIF scanner         ! End of skipped procedure
```

Example 329 DEFINETOG, IFNOT, and ENDIF Directives Directive

```
?DEFINETOG emitter    ! Define toggle
...
?IFNOT emitter         ! Test toggle for off state
PROC kept;            ! Find it off, compile procedure
    BEGIN
        ...
    END;
?ENDIF emitter         ! End of compiled procedure
```

Example 330 SETTOG, IF, and ENDIF Directives

```
?SETTOG keep          ! Create & turn on toggle
...
?IF keep              ! Test toggle for on state
PROC kept;            ! Find it on, compile procedure
    BEGIN
        ...
    END;
?ENDIF keep           ! End of compiled procedure
```

Example 331 SETTOG, IFNOT, and ENDIF Directives

```
?SETTOG (done, nested) ! Create & turn on toggles
?IFNOT done            ! Test toggle for off state
PROC skipped;          ! Find it on, skip procedure
    BEGIN
        ...
    END;
?ENDIF done            ! End of skipped procedure
```

Example 332 SETTOG, RESETTOG, IF, and ENDIF Directives

```
?SETTOG (versn1, versn2, 7, 4, 11) ! Turn on toggles
?SETTOG versn3                  ! Turn on toggle
?RESETTOG (versn2, 7)           ! Turn off toggles
...
?IF versn2                      ! Test toggle for on state
PROC version_2;                 ! Find it off,
    BEGIN                      ! skip procedure
        ...
    END;
?ENDIF versn2                   ! End of skipped procedure
```

Saving and Using Global Data Declarations

For the pTAL compiler, these directives allow you to compile and initialize global data declarations in one compilation and use them in subsequent compilations:

Directive	Description
-----------	-------------

Directive	Description
SAVEGLOBALS	Saves global data declarations and initial values in one file
USEGLOBALS	Reads global data declarations and initial values saved in a file
BEGINCOMPILE	Marks the point in the source file where compilation is to begin if the USEGLOBALS directive is active

NOTE:

- The EpTAL compiler does not accept the [SAVEGLOBALS](#) or [USEGLOBALS](#) directive.
- The EpTAL compiler ignores the [BEGINCOMPILE](#) directive.

Topics:

- [Saving Global Data Declarations \(page 373\)](#)
- [Retrieving Global Data Declarations \(page 374\)](#)
- [Examples \(page 374\)](#)
- [Migrating from TNS/R to TNS/E \(page 375\)](#)

Terms used in the following topics:

Term	Meaning
SAVEGLOBALS compilation	The compilation for which you specify SAVEGLOBALS
SAVEGLOBALS compilation file	The source file for the SAVEGLOBALS compilation
USEGLOBALS compilation	The compilation for which you specify USEGLOBALS
USEGLOBALS compilation file	The source file for the USEGLOBALS compilation

Saving Global Data Declarations

When you compile with [SAVEGLOBALS](#), the compiler saves the global data declarations—global data identifiers and their attributes (such as data type and kind of variable initialization)—in a file whose file code is 701.

If you make no changes in the global data declarations, you can use the saved declarations in subsequent [USEGLOBALS](#) compilations, reducing their compilation time.

[SAVEGLOBALS](#) does not save [FORWARD](#) procedure declarations or [EXTERNAL](#) procedure declarations. You must recompile these declarations in the [USEGLOBALS](#) compilation.

When you use the following directives in the [SAVEGLOBALS](#) compilation, they affect subsequent [USEGLOBALS](#) compilations as follows:

Directive in SAVEGLOBALS Compilation	Effect in Subsequent USEGLOBALS Compilations
SYNTAX	Negates the need for using the USEGLOBALS compilation because no object file was produced by the SAVEGLOBALS compilation
PRINTSYM	Continues to print symbols in the listing
SYMBOLS	Continues to make symbols available for all data blocks that had symbols during the SAVEGLOBALS compilation

You must use the same version of the compiler for the [SAVEGLOBALS](#) compilation and the [USEGLOBALS](#) compilation; otherwise, an error occurs in the [USEGLOBALS](#) compilation.

Whenever you switch to a new version of the compiler, you must recompile the source code using [SAVEGLOBALS](#) to create a new global declarations file.

Retrieving Global Data Declarations

After a SAVEGLOBALS compilation completes successfully, you can specify the following directives in a USEGLOBALS compilation to retrieve the global data declarations and initializations:

Directive in USEGLOBALS Compilation	Effect in Same USEGLOBALS Compilation
USEGLOBALS	<ul style="list-style-type: none">Retrieves global data declarationsSuppresses compilation of text lines and SOURCE directives (but not other directives) until BEGINCOMPILATION appears
BEGINCOMPILATION	Begins compilation of text lines and SOURCE directives

CAUTION: Be sure the global data declarations in both the SAVEGLOBALS and USEGLOBALS compilations are identical. If you include new or changed global data declarations anywhere in the USEGLOBALS source file, results are unpredictable.

The USEGLOBALS compilation terminates if the global declarations file:

- Cannot be found or opened by the compiler
- Was created using a different version of the compiler

Examples

The source file in [Example 333 \(page 374\)](#) (MYPROG) is compiled in examples [Example 334 \(page 375\)](#) through [Example 337 \(page 375\)](#), which show how the SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, and SYNTAX directives interact.

Example 333 MYPROG Source File for Example 334 Through Example 337

Source File MYPROG

```
! Source file MYPROG
! Unless USEGLOBALS is active, compile the entire source file.
?SOURCE SHARGLOB
?BEGINCOMPILATION    ! When USEGLOBALS is active, compile
                    !   following code
?PUSHLIST, NOLIST, SOURCE $system.system.extdecs
?POPLIST
PROC my_first_proc;
BEGIN
    ...
END;
PROC my_last_proc;
BEGIN
    ...
END;
```

File of Shared Global Data, SHARGLOB

```
?SOURCE glbfile1 (section1, section2)
?SOURCE moreglbs
    INT ignore_me1;
    INT ignore_me2;
```

The compilation command in [Example 334 \(page 375\)](#) compiles myprog (the source file in [Example 333](#)) and saves global data declarations and data initializations.

Example 334 Saving Global Data Declarations and Data Initializations

```
pTAL /IN myprog/ myobj; SAVEGLOBALS ptalsym
```

A USEGLOBALS compilation ([Example 335 \(page 375\)](#)) then produces object file `newobj` and retrieves global data declarations and initialization from `ptalsym` and global initializations from `myobj`. When USEGLOBALS is active, the compiler ignores text lines and SOURCE directives until BEGINCOMPILATION appears in the source file.

Example 335 Retrieving Global Data Declarations and Data Initializations

```
pTAL /IN myprog/ newobj; USEGLOBALS ptalsymj
```

You can check the syntax of global data declarations before saving them, as in [Example 336 \(page 375\)](#).

Example 336 Checking the Syntax of Global Data Declarations

```
pTAL /IN myprog/; SAVEGLOBALS ptalsym, SYNTAX
```

After you correct any errors, you can recompile `myprog` as in [Example 337 \(page 375\)](#).

Example 337 Recompiling MYPROG After Correcting Errors

```
pTAL /IN myprog/; USEGLOBALS ptalsym
```

Migrating from TNS/R to TNS/E

The EpTAL compiler does not accept the SAVEGLOBALS and USEGLOBALS directives.

To migrate a pTAL program that uses SAVEGLOBALS and USEGLOBALS from TNS/R to TNS/E:

1. Remove SAVEGLOBALS from the SAVEGLOBALS compilation command line.
2. Compile the file from [FIX_THIS_LINK](#) using the EpTAL compiler, omitting SAVEGLOBALS from the compilation command.
3. Remove USEGLOBALS from each USEGLOBALS compilation command line.
You can leave BEGINCOMPILATION in this file. The EpTAL compiler ignores BEGINCOMPILATION, and you need BEGINCOMPILATION if you want to compile the same files using the pTAL compiler.
4. Compile each file from [FIX_THIS_LINK](#) using the EpTAL compiler, omitting USEGLOBALS from each compilation command.

If all files compile without errors, the migration is done. (To compile the same files using the pTAL compiler, specify SAVEGLOBALS in the SAVEGLOBALS compilation command and USEGLOBALS in each USEGLOBALS compilation command.)

If some files do not compile successfully because of missing global data declarations, the source code files were not set up correctly and you must modify one or more of them.

For example:

1. Suppose that the original SAVEGLOBALS compilation source file is COMP1 in [Example 338 \(page 375\)](#).

Example 338 Original SAVEGLOBALS Compilation Source File

```
! COMP1
?FIELDALIGN (SHARED2)
name x;
?source FILE1
?source FILE2
...
```

```

?source FILEn
int il
struct s(*);
begin
    ...
end;
! All other common declarations and directives in the
! compilation ...
! End of global declarations
?BEGINCOMPILE
! All nonglobal declarations,
! including procedure declarations
! End of COM1

```

2. Extract all directives and declarations from the beginning of COMP1 to (but not including) BEGINCOMPILE. Put them in a new source file called GLOBALS (see [Example 339 \(page 376\)](#)).

Example 339 New GLOBALS Source File

```

! GLOBALS
?FIELDALIGN (SHARED2)
name x;
?source FILE1
?source FILE2
...
?source FILEn
int il

struct s(*);
begin
    ...
end;
! All other common declarations and directives in the
! compilation
! End of GLOBALS

```

3. Use a SOURCE directive to include GLOBALS in COMP1 (as in [Example 340 \(page 376\)](#)).

Example 340 Corrected SAVEGLOBALS Compilation Source File

```

! COMP1
?source GLOBALS
! End of global declarations
?BEGINCOMPILE
! All other non-global declarations,
! including procedure declarations ...
! End of COMP1

```

4. In each file that depended on the global data declarations file that the original COMP1 produced:
 - Use a SOURCE directive to include GLOBALS.
The SOURCE directive must appear before any other declarations and must be immediately followed by the BEGINCOMPILE directive.
 - After the BEGINCOMPILE directive, specify any additional directives that were originally specified in the compilation command.

Summary of Compiler Directives

Table 74 summarizes directives by categories.

Table 75 (page 379) lists directives by name in alphabetical order.

Table 74 Compiler Directives by Category

Category	Directive	Operation
Compiler input	BEGINCOMPILATION ¹	Marks the point in the source file where compilation is to begin if the USEGLOBALS directive is active
	COLUMNS	Treats as comments any text that appears beyond the specified column
	SAVEGLOBALS ²	Saves global data declarations and initial values in a file for subsequent use
	SECTION	Names a section of the source file
	SOURCE	Reads source code from another input file
	USEGLOBALS ²	Reads global data declarations and initial values from a file
Compiler listing	DEFEXPAND	Expands DEFINES in the compiler listing
	FMAP	Lists the file map in the compiler listing
	GMAP	Lists the global map in the compiler listing
	INNERLIST	Lists mnemonics after each source statement
	LINES	Skips to the top of form after a specified number of lines if the list file is a line printer or a process
	LIST	Lists the source code
	MAP	Lists the identifier map
	PAGE	Sets the string to be printed as part of the heading for each page. Each subsequent PAGE prints the heading and causes a page eject.
	PRINTSYM	Lists symbols in the compiler listing
	SUPPRESS	Suppresses all listings but the header, diagnostics, and trailer
Diagnostics	ERRORFILE	Writes error and warning messages to an error file
	ERRORS	Terminates compilation after the specified number of error messages
	DO_TNS_SYNTAX	Issues warnings for pTAL constructs that are not valid in TAL
	INVALID_FOR_PTAL	Causes errors for TAL constructs that are not valid in pTAL
	WARN	Suppresses compiler warnings
Object-file content	ASSERTION	Conditionally executes a debugging procedure

Table 74 Compiler Directives by Category (*continued*)

Category	Directive	Operation
	<code>BASENAME</code>	Specifies that the raw data file (used for code profiling) generated by the executing process is to contain only the base part of the file name.
	<code>BLOCKGLOBALS</code>	Determines how the compiler allocates global data that is not declared within the scope of a named data block or the private data block
	<code>CALL_SHARED</code> ³	Generates shared code (PIC)
	<code>CHECKSHIFTCOUNT</code>	Causes overflow traps for invalid bit-shift operations
	<code>CODECOV</code>	Generates instrumented object code for use by the Code Coverage Utility
	<code>EXPORT_GLOBALS</code>	Exports globals
	<code>FIELDALIGN</code>	Specifies the default memory alignment for structures
	<code>GLOBALIZED</code>	Generates preemptable object code for use when building DLLs that require such code
	<code>GP_OK</code> ¹	Generates code that has GP-relative addressing
	<code>OPTIMIZE</code>	Sets the object code's default optimization level
	<code>OPTIMIZEFILE</code>	Sets the optimization level for individual procedures and subprocedures
	<code>OVERFLOW_TRAPS</code> ³	Controls whether overflow traps are enabled
	<code>PROFDIR</code>	Specifies where an instrumented object file is to create the raw data file.
	<code>PROFGEN</code>	Generate an instrumented object file for use in profile-guided optimization.
	<code>PROFUSE</code>	Generates optimized object code based information in a DPI file.
	<code>REFALIGNED</code>	Specifies the default memory alignment for pointers to nonstructure items and procedure reference pointers
	<code>ROUND</code>	Rounds FIXED values assigned to FIXED variables with smaller <i>fpoint</i> values
	<code>SRL</code> ¹	Generates code that can be included in a user library
	<code>SYNTAX</code>	Checks the syntax, suppressing the object code
Conditional compilation	<code>DEFINETO</code>	Defines toggles without changing their settings
	<code>ENDIF</code>	Identifies the end of code that is to be conditionally compiled
	<code>IF</code> and <code>IFNOT</code>	Identifies the beginning of code that is to be conditionally compiled

Table 74 Compiler Directives by Category *(continued)*

Category	Directive	Operation
Run-time environment	RESETTOG	Turns toggles off
	SETTOG	Turns toggles on
	TARGET ³	Specifies the architecture on which the program will run
	SYMBOLS	Generates a symbol table for a symbolic debugger
Feature control	__EXT64	Enables 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see “64-bit Addressing Functionality” (page 531).

¹ The EpTAL compiler ignores this directive.

² The EpTAL compiler does not accept this directive.

³ The pTAL and EpTAL compilers treat this directive differently.

Table 75 Compiler Directives by Name

Directive	Operation
ASSERTION	Conditionally executes a debugging procedure
BASENAME	Specifies that the raw data file (used for code profiling) generated by the executing process is to contain only the base part of the file name.
BEGINCOMPILATION ¹	Marks the point in the source file where compilation is to begin if the USEGLOBALS directive is active
BLOCKGLOBALS	Determines how the compiler allocates global data that is not declared within the scope of a named data block or the private data block
CALL_SHARED ²	Generates shared code (PIC)
CHECKSHIFTCOUNT	Causes overflow traps for invalid bit-shift operations
CODECOV	Generates instrumented object code for use by the Code Coverate Tool
COLUMNS	Treats as comments any text that appears beyond the specified column
DEFEXPAND	Expands DEFINES in the compiler listing
DEFINETOG	Defines toggles without changing their settings
DO_TNS_SYNTAX	Issues warnings for pTAL constructs that are not valid in TAL
ENDIF	Identifies the end of code that is to be conditionally compiled
ERRORFILE	Writes error and warning messages to an error file
ERRORS	Terminates compilation after the specified number of error messages
EXPORT_GLOBALS	Exports globals
__EXT64	Directs the compiler to recognize the 64-bit keywords, indirection symbols, and built-in routines are 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see “64-bit Addressing Functionality” (page 531).
FIELDALIGN	Specifies the default memory alignment for structures
FMAP	Lists the file map in the compiler listing
GLOBALIZED	Generates preemptable object code for use when building DLLs that require such code

Table 75 Compiler Directives by Name *(continued)*

Directive	Operation
GMAP	Lists the global map in the compiler listing
GP_OK ¹	Generates code that has GP-relative addressing
IF and IFNOT	Identifies the beginning of code that is to be conditionally compiled
INNERLIST	Lists mnemonics after each source statement
INVALID_FOR_PTAL	Causes errors for TAL constructs that are not valid in pTAL
LINES	Specifies the maximum number of output lines per page if the list file is a line printer or a process
LIST	Lists the source code
MAP	Lists the identifier map
OPTIMIZE	Sets the object code's default optimization level
OPTIMIZEFILE	Sets the optimization level for individual procedures and subprocedures
OVERFLOW_TRAPS ²	Controls whether overflow traps are enabled.
PAGE	Sets the string to be printed as part of the heading for each page. Each subsequent PAGE prints the heading and causes a page eject.
PRINTSYM	Lists symbols in the compiler listing
PROFDIR	Specifies where an instrumented object file is to create the raw data file.
PROFGEN	Generates an instrumented object file for use in profile-guided optimization.
PROFUSE	Generates an optimized object file based on information in a DPI file.
REFALIGNED	Specifies the default alignment for pointers to nonstructure items and procedure reference pointers
RESETTOG	Turns off toggles
ROUND	Rounds FIXED values assigned to FIXED variables with smaller <i>fpoint</i> values
SAVEGLOBALS ³	Saves global data declarations and initial values in a file for subsequent use
SECTION	Names a section of the source file
SETTOG	Turns on toggles
SOURCE	Reads source code from another input file
SRL ¹	Generates code that can be included in a user library
SUPPRESS	Suppresses all listings but the header, diagnostics, and trailer
SYMBOLS	Generates a symbol table for a symbolic debugger
SYNTAX	Checks the syntax, suppressing the object code
TARGET ²	Specifies the architecture on which the program will run
USEGLOBALS ³	Reads global data declarations and initial values from a file
WARN	Suppresses compiler warnings

¹ The EpTAL compiler ignores this directive.

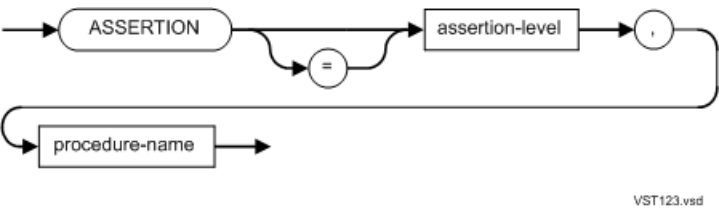
² The pTAL and EpTAL compilers treat this directive differently.

³ The EpTAL compiler does not accept this directive.

NOTE: In the following directive topics, “**Default:**” identifies the default for the compiler directive itself, not for its optional parameter(s). This default applies if a program does not contain the compiler directive at all.

ASSERTION

ASSERTION executes a procedure when the condition specified in the active ASSERT statement is true.



assertion-level

is an unsigned decimal constant in the range 0 through 32,767.

procedure-name

is the name of the procedure to execute if both:

- The *condition* defined in the active ASSERT statement is true.
- *assertion-level* is less than the *assert-level* in the active ASSERT statement.

This procedure must not have parameters.

Default:	None
Placement:	<ul style="list-style-type: none">• Anywhere in the source file (not in the compilation command)• Must be the last directive on the directive line
Scope:	Applies until another ASSERTION overrides it
Dependencies:	Has no effect without the ASSERT statement
References:	ASSERT (page 200)

[ASSERT \(page 200\)](#) explains how to use the ASSERTION directive and the ASSERT statement together.

BASENAME

This directive can be used only with the EpTAL compiler.

BASENAME specifies that when an instrumented object file is run, the raw data file created by the running process will contain only the base part of the source file name and not the full file path. For detailed information about using the BASENAME option when performing profile-guided optimization, see the *Code Profiling Utilities Manual*.



Default:	The raw data file contains the full path name of the source file
Placement:	Only on the command line
Scope:	Applies to the compilation unit

Dependencies:	Use the BASENAME option only with the PROGEN option
----------------------	---

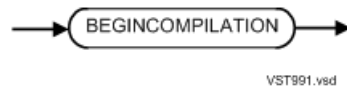
References:	PROGEN
--------------------	--------

BEGINCOMPILE

NOTE: The EpTAL compiler ignores this directive. See [Migrating from TNS/R to TNS/E \(page 375\)](#).

BEGINCOMPILE marks the point in the source file where:

- The information saved by the SAVEGLOBALS operation ends
- Compilation is to begin if the USEGLOBALS directive is active.



Default:	None
-----------------	------

Placement:	<ul style="list-style-type: none">• In the source file between the last global data declaration and the first procedure declaration, including any EXTERNAL and FORWARD declarations• Can appear only once in a compilation unit
-------------------	---

Scope:	Applies to all source code that follows it in the compilation unit
---------------	--

Dependencies:	<ul style="list-style-type: none">• Has no effect without the USEGLOBALS directive• If you specify either SAVEGLOBALS or USEGLOBALS, your compilation unit must have exactly one BEGINCOMPILE directive• Interacts with SAVEGLOBALS and USEGLOBALS (see Saving and Using Global Data Declarations (page 372))
----------------------	---

References:	<ul style="list-style-type: none">• SAVEGLOBALS (page 413)• USEGLOBALS (page 423)
--------------------	--

BLOCKGLOBALS

BLOCKGLOBALS determines how the compiler allocates global data that is not declared within the scope of a named data block or the private data block.



Default:	The compiler allocates data items in the _GLOBAL and \$_GLOBAL data blocks
-----------------	--

Placement:	Before the first data declaration in a compilation
-------------------	--

Scope:	Applies to the compilation unit
---------------	---------------------------------

Dependencies:	None
----------------------	------

If you specify BLOCKGLOBALS, the compiler allocates its own data block for each global variable that is not declared in the scope of a named data block or the private data block. The name of the data block is the same as the name of the variable contained in the data block.

Table 76 Data Block Names

Declaration	Without BLOCKGLOBALS	With BLOCKGLOBALS
INT a;	_GLOBAL	A
INT .a;	_GLOBAL	A
INT .EXT a;	_GLOBAL	A
INT a[0:9]	_GLOBAL	A
STRUCT a; BEGIN INT i; END	_GLOBAL	A
int .ext a [0:9]	\$_GLOBAL	A
struct .ext a; begin int i; end;	\$_GLOBAL	A

Separately compiled modules can share access to a data block only if both modules allocate the block in the small data area or both modules allocate the block in the large data area.

References to data in the small data area are faster than references to data in the large data area.

All data blocks in a shared run-time library must be allocated in the large data area.

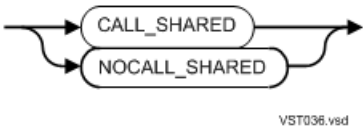
If the name of a variable is the same as the name of the data block in which the variable is located, and the block only contains one variable, the compiler allocates the data block in the small data area if the length of the block is eight or fewer bytes; otherwise, the compiler allocates the data block in the large data area. (This is the allocation strategy used by the native HP C compiler.)

The compiler does not allocate memory for LITERALS, DEFINES, or templates and, therefore, does not create an implicit global data block for these items.

CALL_SHARED

NOTE:

- This directive is useful only for the pTAL compiler. The EpTAL compiler ignores it (and issues a warning).
- You cannot link PIC and non-PIC object files into a single object file.



CALL_SHARED

generates shared code (PIC), the only option for the EpTAL compiler.

NOCALL_SHARED

causes the pTAL compiler to generate nonshared code (non-PIC).

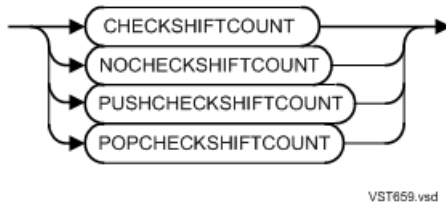
Default:	pTAL compiler:	NOCALL_SHARED
EpTAL compiler:	CALL_SHARED	
Placement:	Anywhere	
Scope:	Applies to the compilation unit	

Dependencies:

- If both `CALL_SHARED` and `NOCALL_SHARED` appear in the same compilation unit, the compiler uses the one that appears last
- Do not use `CALL_SHARED` with `GP_OK`

References:[GP_OK \(page 397\)](#)

CHECKSHIFTCOUNT

**CHECKSHIFTCOUNT**

generates code that causes an overflow trap if the number of positions in a bit-shift operation is too large, as in:

```
INT j := 20;  
INT i;  
I := i << j;
```

(For more information about bit shifts, see [Bit Shifts \(page 94\)](#).)

NOCHECKSHIFTCOUNT

suppresses the generation of code that causes an overflow trap if the number of positions in a bit-shift operation is too large.

⚠ CAUTION: If such a bit-shift operation occurs, subsequent program behavior is undefined.

PUSHCHECKSHIFTCOUNT

pushes the current setting (`CHECKSHIFTCOUNT` or `NOCHECKSHIFTCOUNT`) onto the `CHECKSHIFTCOUNT` directive stack. Does not change the current setting.

POPCHECKSHIFTCOUNT

pops the top value from the `CHECKSHIFTCOUNT` directive stack and changes the current setting to that value.

For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:

`NOCHECKSHIFTCOUNT`

Placement:

Anywhere

Scope:

- `CHECKSHIFTCOUNT` applies to the shift operators that follow it until it is overridden by `NOCHECKSHIFTCOUNT`
- `NOCHECKSHIFTCOUNT` applies to the shift operators that follow it until it is overridden by `CHECKSHIFTCOUNT`

Dependencies:

None

NOTE:

- This directive can be used only with the EpTAL compiler.
- Instrumented object code can result in greatly reduced performance. Therefore, the CODECOV directive should be used only in a test environment. See the caution under [Debugging](#) (page 429), which indicates how CODECOV affects debugging applications.

CODECOV causes the compiler to generate instrumented object code for use by the Code Coverage Utility. For detailed information about the Code Coverage Utility, see the *Code Profiling Utilities Manual*.



Default:	No code coverage instrumentation in object code
Placement:	Only on the command line
Scope:	Applies to the compilation unit
Dependencies:	None

COLUMNS

COLUMNS causes the compiler to treat any text beyond the specified column as comments.



columns-value is an unsigned decimal constant in the range 12 through 132, the column beyond which the compiler is to treat text as comments.

If *columns-value* is smaller than 12 or larger than 132, the compiler issues an error message.

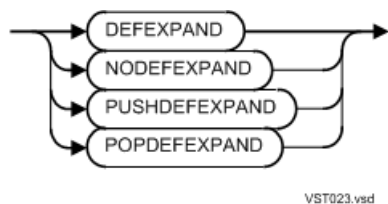
Default:	COLUMNS 132
Placement:	<ul style="list-style-type: none">• Anywhere, but if COLUMNS appears in the source code, it must be the only directive on the directive line• Typically specified before any SECTION directive
Scope:	Applies to all source code that follows it unless overridden by: <ul style="list-style-type: none">• Another COLUMNS directive in the same source file (not recommended)• A COLUMNS directive in a source file included by means of a SOURCE directive• A COLUMNS directive in a section identified by a SECTION directive For details, see the explanation that follows this table.
Dependencies:	None
References:	<ul style="list-style-type: none">• SECTION (page 414)• SOURCE (page 416)

The *columns-value* active at any given time depends on the context, as follows:

- The main input file initially has the *columns-value* set by the last COLUMNS directive in the compilation command. If there was no COLUMNS directive in the compilation command, the main input file initially has the default *columns-value* of 132.
- At each SOURCE directive, each included file initially has the *columns-value* active when the SOURCE directive appeared.
- At each SECTION directive, *columns-value* is set by the last COLUMNS directive before the first SECTION directive in the included file. If there is no such COLUMNS directive, each SECTION initially has the *columns-value* active at the beginning of the included file.
- Within a section, a COLUMNS directive sets the *columns-value* only until the next COLUMNS or SECTION directive or the end of the file.
- After a SOURCE directive completes execution (that is, after all sections listed in the SOURCE directive are read or the end of the file is reached), the compiler restores *columns-value* to what it was when the SOURCE directive appeared.
- In all other cases, *columns-value* is set by the most recently processed COLUMNS directive.

If a SOURCE directive lists sections, the compiler processes no source code outside the listed sections except any COLUMNS directives that appear before the first SECTION directive in the included file. For more information about including files or sections, see [SOURCE \(page 416\)](#) and [SECTION \(page 414\)](#).

DEFEXPAND



DEFEXPAND

expands DEFINEs in the compiler listing.

NOTE: MAP DEFINEs are available only on Guardian platforms.

NODEFEXPAND

suppresses the expansion of DEFINEs in the compiler listing.

PUSHDEFEXPAND

pushes the current setting (DEFEXPAND or NODEFEXPAND) onto the DEFEXPAND directive stack. Does not change the current setting.

POPDEFEXPAND

pops the top value from the DEFEXPAND directive stack and changes the current setting to that value.

For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	NODEFEXPAND
Placement:	Anywhere

Scope:	<ul style="list-style-type: none"> • DEFEXPAND applies to subsequent code until it is overridden by NODEFEXPAND • NODEFEXPAND applies to subsequent code until it is overridden by DEFEXPAND
Dependencies:	DEFEXPAND has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none"> • LIST (page 401) • SUPPRESS (page 420)

In the DEFEXPAND listing, the DEFINE body appears on lines following the DEFINE identifier. In the listing:

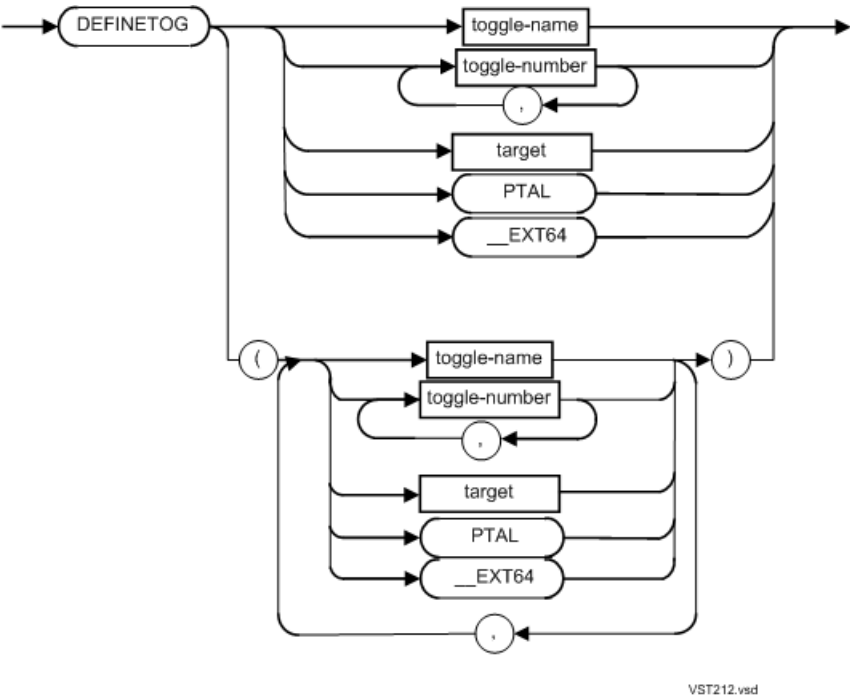
- All letters are uppercase.
- No comments, line boundaries, or extra blanks appear.
- The lexical level of the DEFINE appears in the left margin, starting at 1.
- Parameters to the DEFINE appear as #*n*, where *n* is the sequence number of the parameter, starting at 1.

Example 341 DEFEXPAND Directive

```
?DEFEXPAND                                ! List expanded DEFINES
DEFINE increment (x) = x := x + 1#;         ! Expanded DEFINE
DEFINE decrement (y) = y := y - 1#;         ! Expanded DEFINE
! Other global data declarations
```

DEFINETOG

DEFINETOG specifies toggles for use in conditional compilation. If DEFINETOG is specifying a toggle for the first time, its setting is off. DEFINETOG has no effect on toggles already in use.



toggle-name

is an identifier with a maximum of 31 characters in length.

The only characters allowed in a toggle-name are alphabetic (“A” through “Z” and “a” through “z”), numeric (“0” through “9”), underscore (“_”), and circumflex (“^”); the first character must be alphabetic.

Names are case-insensitive (For example, abc is the same as Abc.)

toggle-number

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored.

target

is as defined in “TARGET” (page 423).

PTAL

is a toggle implicitly defined and set by the TAL, pTAL and EpTAL compilers. It is set on if the compiler in use is any pTAL or EpTAL compiler, otherwise it is set off.

It can be used with the directives “IF and IFNOT” (page 398) to conditionally compile code. Source code enclosed within the IF PTAL directive is compiled only when using the pTAL or EpTAL compilers. Likewise, source code enclosed within the IFNOT PTAL directive is compiled only when using the TAL compiler.

Compiler	IF pTAL	IFNOT pTAL
pTAL or EpTAL	True	False
TAL	False	True

__EXT64

is a toggle implicitly defined and set by the EpTAL compiler starting with SPR T0561H01 ^AAP. It is set on if the corresponding __EXT64 directive has been specified otherwise, it is set off.

The `__EXT64` directive controls the availability of 64-bit addressing functionality; for more details, see Appendix E “64-bit Addressing Functionality” (page 531).

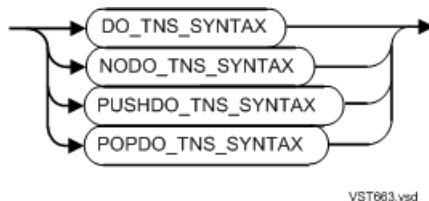
The toggle `__EXT64` is used with the directives “IF and IFNOT” (page 398) to conditionally compile source code containing 64-bit addressing functionality.

This toggle is not supported by the EpTAL compilers prior to SPR T0561H01 ^AAP nor is it supported by any pTAL or TAL compiler. If you need to compile using earlier versions of EpTAL, pTAL, or TAL compiler, explicitly specify `__EXT64` in a `DEFINETO` directive which explicitly defines and sets the toggle off in these compilers.

You can specify `DEFINETO __EXT64` using EpTAL compilers starting with SPR T0561H01 ^AAP. However, doing so has no effect on the implicitly defined `__EXT64` toggle setting.

Default:	None
Placement:	<ul style="list-style-type: none"> • With a parenthesized list, it can appear anywhere • Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none"> • SETTOG • RESETTOG • IF and IFNOT • ENDIF • TARGET • <code>__EXT64</code>
References:	<ul style="list-style-type: none"> • SETTOG (page 415) • RESETTOG (page 411) • IF and IFNOT (page 398) • ENDIF (page 390) • “TARGET” (page 423) • “<code>__EXT64</code>” (page 394) • Toggles (page 370)

DO_TNS_SYNTAX



DO_TNS_SYNTAX

issues a warning for each occurrence of certain constructs that are valid in pTAL but not in TAL (for these constructs, see the *pTAL Conversion Guide*).

NODO_TNS_SYNTAX

suppresses warnings for each occurrence of a construct that is valid in pTAL but not in TAL.

PUSHTNS_SYNTAX

pushes the current setting (DOTNS_SYNTAX or NODOTNS_SYNTAX) onto the DOTNS_SYNTAX directive stack. Does not change the current setting.

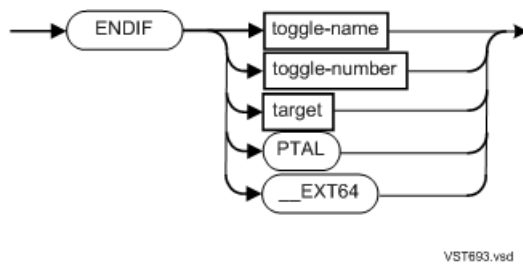
POPTNS_SYNTAX

pops the top value from the DOTNS_SYNTAX directive stack and changes the current setting to that value.

Default:	NODO_TNS_SYNTAX
Placement:	<ul style="list-style-type: none">• Can appear only once in a compilation• Must precede any TARGET directive and any nondirective lines
Scope:	Applies to the compilation unit
Dependencies:	None
References:	TARGET (page 423)

ENDIF

ENDIF identifies the end of code that is to be conditionally compiled.



toggle-name

is an identifier that was used as a *toggle-name* in an earlier IF or IFNOT directive.

The only characters allowed in a toggle-name are alphabetic ("A" through "Z" and "a" through "z"), numeric ("0" through "9"), underscore ("_"), and circumflex ("^"); the first character must be alphabetic.

Names are case-insensitive (For example, abc is the same as Abc.)

toggle-number

is an unsigned decimal constant in the range 1 through 15 that was used as a *toggle-name* in an earlier IF or IFNOT directive. Leading zeros are ignored.

target

is as defined in ["TARGET" \(page 423\)](#).

PTAL

is a toggle implicitly defined and set by the TAL, pTAL and EpTAL compilers. It is set on if the compiler in use is any pTAL or EpTAL compiler, otherwise it is set off. See ["DEFINETOG" \(page 388\)](#).

`__EXT64`

is a toggle implicitly defined and set by the EpTAL compiler starting with SPR T0561 H01 ^AAP. It is set on if the corresponding ["`__EXT64`" \(page 394\)](#) directive has been specified otherwise, it is set off. The `__EXT64` directive controls the availability of 64-bit addressing functionality; see ["DEFINETOG" \(page 388\)](#) and Appendix E, ["64-bit Addressing Functionality" \(page 531\)](#).

The next compiled ENDIF that matches the most recently compiled IF or IFNOT with the same toggle or target specified identifies the end of code to be conditionally compiled. For example:

```
?SETTOG tog1 -- Create and turn on tog1
?RESETTOG tog2 - Create and turn off tog2
?IF tog1
    -- Statements for true condition
```

```

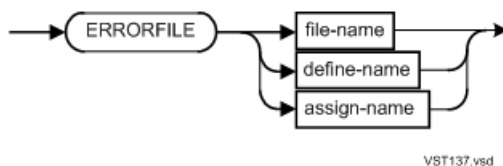
-- compiled because tog1 is on
?IF tog2
-- Statements for true condition
-- skipped because tog2 is off
?ENDIF tog1 -- Not compiled, part of skipped code for tog2
?ENDIF tog2 -- End of conditional code for tog2
?ENDIF tog1 -- End of conditional code for tog1

```

Default:	None
Placement:	<ul style="list-style-type: none"> • Anywhere in the source file (not in the compilation command) • Must be the only directive on the directive line
Scope:	Everything between ENDF and the most recently compiled IF or IFNOT directive that specifies the same toggle, target, or keyword
Dependencies:	Interacts with: <ul style="list-style-type: none"> • SETTOG • RESETTOG • IF and IFNOT • ENDF • TARGET • __EXT64
References:	<ul style="list-style-type: none"> • SETTOG (page 415) • RESETTOG (page 411) • IF and IFNOT (page 398) • ENDF (page 390) • "TARGET" (page 423) • "__EXT64" (page 394) • Toggles (page 370)

ERRORFILE

ERRORFILE writes compilation errors and warnings to an error file so you can use the HP TACL FIXERRS macro (available only on Guardian platforms) to view the diagnostic messages in one PS Text Edit window and correct the source file in another window.



file-name

is the name of either:

- An existing error file created by ERRORFILE. Such a file has file code 106 (an entry-sequenced disk file used only with the HP TACL FIXERRS macro). The compiler purges any data in it before logging errors and warnings.
- A new error file to be created by ERRORFILE if errors occur.

If a file with the same name exists but the file code is not 106, the compiler terminates compilation to prevent overwriting the file.

You can specify partial file names as described in [Partial File Names \(page 519\)](#). The compiler uses the current default volume and subvolume names as needed. For this directive, the compiler

does not use HP TACL ASSIGN SSV information (available only on Guardian platforms) to complete the file name.

define-name

is the name of a MAP DEFINE that refers to an error file.

NOTE: MAP DEFINES are available only on Guardian platforms.

assign-name

is a logical file name you have equated with an error file by issuing an ASSIGN command.

Default:	None
Placement:	<ul style="list-style-type: none">• In the compilation command or in the source code before any declarations• Can appear only once in a compilation unit
Scope:	Applies to the compilation unit
Dependencies:	None

The compiler writes a header record to the error file and then writes a record for each error or warning. Each record contains information such as:

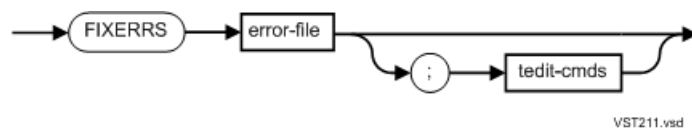
- The location of the error or warning—source file name, edit line number, and column number
- The message text of the error or warning

At the end of the compilation, the compiler prints the complete name of the error file in the trailer message of the compilation listing.

After the compiler logs messages to the error file, you can call the HP TACL FIXERRS macro and correct the source file. FIXERRS uses the PS Text Edit ANYHOW option to open the source file in a two-window session. One window displays a diagnostic message. The other window displays the source code to which the message applies. If you have write access to the file, you can correct the source code. If you have only read access, you can view the source code, but you cannot correct it.

Initially, the edit cursor is located in the source code at the first diagnostic. To move the cursor to the next or previous diagnostic, use the PS Text Edit NEXTERR or PREVERR command.

The HP TACL command for calling FIXERRS is:



error-file

is the name of the error file specified in the ERRORFILE directive.

tedit-cmds

is any PS Text Edit commands that are allowed on the PS Text Edit run line.

[Example 342 \(page 393\)](#) issues an HP TACL DEFINE command that calls FIXERRS and defines PS Text Edit function keys for NEXTERR and PREVERR.

Example 342 FIXERRS Macro

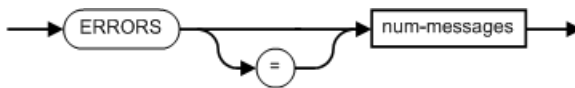
```
[#DEF MYFIXERRS MACRO |BODY|  
    FIXERRS %1%; SET <F9>, NEXTERR; SET <SF9>, PREVERR  
]
```

Example 343 ERRORFILE Directive

```
! MYSOURCE file  
?ERRORFILE myerrors ! Compiler reports errors and warnings  
                    ! to the file myerrors  
!Global declarations
```

ERRORS

ERRORS sets the maximum number of error messages to allow before the compiler terminates the compilation.



VST138.vsd

num-messages

is an unsigned decimal constant in the range 0 through 32,767 that represents the maximum number of error messages to allow before the compilation terminates.

Default:	Unlimited number of errors
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

A single error can cause many error messages. The compiler counts each error message separately. If the compiler's count exceeds the maximum you specify, the compiler terminates the compilation. (Warning messages do not affect the count.)

Example 344 ERRORS Directive

```
! MYSOURCE file  
?ERRORS 10 ! Stop compiling when 10 errors are found  
!Global declarations
```

EXPORT_GLOBALS



VST662.vsd

EXPORT_GLOBALS

causes the compiler to define (rather than only declare) global data blocks, allocating space for them and (optionally) giving them initial values, and causes the linker to include in the program file all global data blocks declared up to the next occurrence of NOEXPORT_GLOBALS or through the last declared global data block, whichever is first.

NOEXPORT_GLOBALS

causes the compiler to declare (rather than define) global data blocks.

PUSHEXPORT_GLOBALS

pushes the current setting (EXPORT_GLOBALS or NOEXPORT_GLOBALS) onto the EXPORT_GLOBALS directive stack. Does not change the current setting.

POPEXPORT_GLOBALS

pops the top value from the EXPORT_GLOBALS directive stack and changes the current setting to that value.

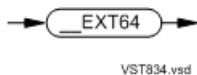
Default:	EXPORT_GLOBALS
Placement:	<ul style="list-style-type: none">• Can appear any number of times in a compilation unit• Must appear before the first procedure is compiled• Cannot appear within BLOCK declarations
Scope:	Applies to the compilation unit, except that NOEXPORT_GLOBALS does not affect a compilation's private data block, which is always exported
Dependencies:	<ul style="list-style-type: none">• You must specify NOEXPORT_GLOBALS when declaring a data block that belongs to an SRL• In a compilation that includes USEGLOBALS, the compiler exports the data blocks declared in the USEGLOBALS declarations file only if EXPORT_GLOBALS is active when the compiler encounters the BEGINCOMPILE directive.
References:	<ul style="list-style-type: none">• BEGINCOMPILE (page 382)• SRL (page 420)• USEGLOBALS (page 423)

You can export only whole data blocks. You cannot export individual variables declared within a data block.

The compiler exports initialization values for variables that specify them. If a data block is not being exported, the compiler ignores any specified initial values within the block.

You must export every data block in at least one compilation.

__EXT64



__EXT64 directive controls the accessibility of 64-bit addressing functionality support available in the EpTAL compiler starting with SPR T0561H01 ^AAP. See Appendix E, “[64-bit Addressing Functionality](#)” (page 531).

Starting with SPR T0561H01 ^AAP, the corresponding implicitly defined toggle __EXT64 is set on if the __EXT64 directive is specified, otherwise, it is set off. For example:

```
-- The ?__EXT64 directive is specified appropriately
-- on the EpTAL compiler command line

?DEFINETOGL __EXT64 -- For downward compatibility with
-- compilers that do not support
-- ?__EXT64 and the 64-bit address
-- functionality.

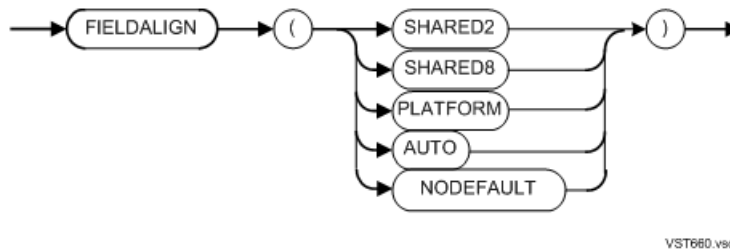
...
?IF __EXT64 -- EpTAL version is SPR AAP or newer
-- and 64-bit functionality is needed.
```

```
EXT64ADDR addr;    -- Use a 64-bit address type.
?ENDIF __EXT64
?IFNOT __EXT64     -- EpTAL prior to SPR AAP, pTAL or TAL
EXTADDR addr;
?ENDIF __EXT64
```

Default:	off
Placement:	Must appear either on the compiler command line or in the compiled source code before the first source code token is scanned by the compiler.
Scope:	Affects the entire compilation
Dependencies:	None
References:	<ul style="list-style-type: none"> • “DEFINETOG” (page 388) • “ENDIF” (page 390) • “IF and IFNOT” (page 398) • “RESETTOG” (page 411) • “SETTOG” (page 415) • Toggles (page 370)

FIELDALIGN

FIELDALIGN specifies the default alignment for structures.



SHARED2

specifies that the base of the structure and each field in the structure must begin at an even-byte address except STRING fields. For more information, see [SHARED2 Parameter \(page 128\)](#).

SHARED8

specifies that the offset of each field in the structure from the base of the structure must be begin at an address that is an integral multiple of the width of the field. For more information, see [SHARED8 Parameter \(page 129\)](#).

AUTO

specifies that the structure and the fields of the structure be aligned according to the optimal alignment for the architecture on which the program will run (this is not the same behavior as the AUTO attribute has in the native mode HP C compiler). For more information, see [AUTO \(page 118\)](#).

PLATFORM

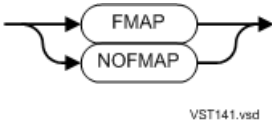
specifies that the structure and the fields of the structure must begin at addresses that are consistent across all languages on the same architecture. For more information, see [PLATFORM \(page 118\)](#).

NODEFAULT

specifies that every structure declaration must include a [FIELDALIGN](#) (page 395).

Default:	FIELDALIGN AUTO
Placement:	<ul style="list-style-type: none">• Can appear only once in a compilation unit• Must precede all declarations of data, blocks, and procedures
Scope:	Applies to the compilation unit
Dependencies:	None

FMAP



FMAP

lists the file map in the compiler listing.

NOFMAP

suppresses the file map in the compiler listing.

Default:	NOFMAP
Placement:	Anywhere, any number of times. The last FMAP or NOFMAP in the compilation unit determines whether the compiler lists the file map.
Scope:	Applies to the compilation unit
Dependencies:	FMAP has no effect if either NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none">• LIST (page 401)• SUPPRESS (page 420)

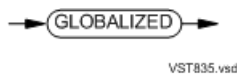
The file map:

- Appears after the map of global identifiers in the compilation listing
- Starts with the first file that the compiler encounters and includes each file introduced by SOURCE directives and (on Guardian platforms) HP TACL ASSIGN and DEFINE commands
- Shows the complete name of each file and the date and time when the file was last modified

GLOBALIZED

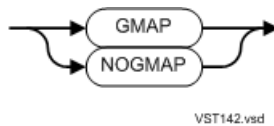
NOTE: This directive is valid only with the EpTAL compiler.

The GLOBALIZED directive directs the compiler to generate preemptable object code. Preemptable object code allows named references in a DLL to resolve to externally-defined code and data items instead of to the DLL's own internally-defined code and data items. You must specify the GLOBALIZED directive when compiling code that will be linked into a globalized DLL. By default, the compiler generates non-preemptable object code. Non-preemptable code is more efficient than preemptable code and results in faster compilation and execution, so you should specify GLOBALIZED only when required.



Default:	Generate non-preemptable object code
Placement:	On the command line
Scope:	Applies to the compilation unit
Dependencies:	None

GMAP



GMAP

lists the global map in the compiler listing.

NOGMAP

suppresses the global map in the compiler listing.

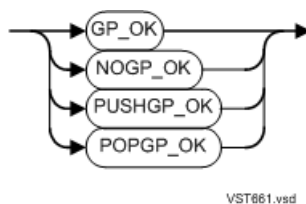
Default:	GMAP
Placement:	Anywhere, any number of times. The last GMAP or NOGMAP in the compilation unit determines whether the compiler lists the global map.
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none"> • GMAP has no effect if NOLIST, NOMAP, or SUPPRESS is active • NOGMAP suppresses the global map even if MAP is active
References:	<ul style="list-style-type: none"> • LIST (page 401) • MAP (page 402) • SUPPRESS (page 420)

The global map:

- Appears at the end of the compilation listing
- Lists all identifiers in the compilation unit and tells what kind of objects they are, including identifier class and type

GP_OK

NOTE: The EpTAL compiler ignores these directives.



GP_OK

causes the pTAL compiler to generate code that has GP-relative addressing (“small” data).

NOGP_OK
 suppresses the generation of code that has GP-relative addressing. (This is the only option for the EpTAL compiler.)

PUSHGP_OK
 pushes the current setting (GP_OK or NOGP_OK) onto the GP_OK directive stack. Does not change the current setting.

POPGP_OK
 pops the top value from the GP_OK directive stack and changes the current setting to that value.

For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	pTAL compiler:	GP_OK
	EpTAL compiler:	NOGP_OK
Placement:	Anywhere except inside a data block or inside a procedure declaration	
Scope:	<ul style="list-style-type: none"> GP_OK applies to subsequent code it until it is overridden by NOGP_OK NOGP_OK applies to subsequent code until it is overridden by GP_OK 	
Dependencies:	Do not use GP_OK with CALL_SHARED	
References:	CALL_SHARED (page 383)	

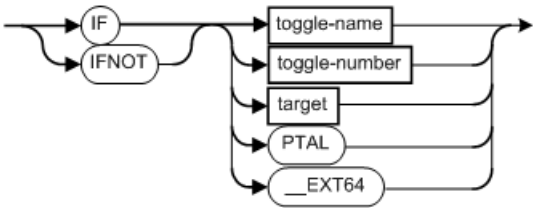
A pTAL program that references data in a shared run-time library (SRL) must specify NOGP_OK when it declares a data block that belongs to a shared run-time library. This behavior prevents the pTAL compiler from using GP-relative addressing for references to data in an SRL.

Example 345 GP_OK, NOGP_OK, PUSHGP_OK, and POPGP_OK Directive

```
?PUSHGP_OK
?NOGP_OK
?NOEXPORT_GLOBALS
BLOCK a_block;
...
END BLOCK;
?EXPORT_GLOBALS
?POPGP_OK
```

IF and IFNOT

IF and IFNOT identify the beginning of code that is to be conditionally compiled.



VST694.vsd

toggle-name
 is an identifier with a maximum of 31 characters in length

The only characters allowed in a toggle-name are alphabetic ("A" through "Z" and "a" through "z"), numeric ("0" through "9"), underscore ("_"), and circumflex ("^"); the first character must be alphabetic.

Names are case-insensitive (For example, abc is the same as Abc.)

toggle-number

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored.

target

is as defined in “[TARGET](#)” (page 516).

PTAL

is a toggle implicitly defined and set by the TAL, pTAL and EpTAL compilers. It is set on if the compiler in use is any pTAL or EpTAL compiler, otherwise it is set off. See “[DEFINETOG](#)” (page 388).

__EXT64

is a toggle implicitly defined and set by the EpTAL compiler starting with SPR T0561H01^AAP. It is set on if the corresponding “[__EXT64](#)” (page 394) directive has been specified otherwise, it is set off. The `__EXT64` directive controls the availability of 64-bit addressing functionality; see “[DEFINETOG](#)” (page 388) and Appendix E, “64-bit Addressing Functionality” (page 531).

The most recently compiled IF or IFNOT matches the next compiled ENDIF with the same toggle or target specified identifies the beginning of code to be conditionally compiled.

Example 346 IF Directive Without Matching ENDIF Directive

```
?RESETTOG flag ! Create & turn off flag
?IF flag
    ! Statements for true condition
    ! (skipped because flag is off)
?IFNOT flag
    ! Statements for false condition
    ! (also skipped, because no ENDIF appears for IF flag)
?ENDIF flag
```

If you insert an ENDIF for the IF in the code in [Example 346](#) (page 399), as in [Example 347](#) (page 399), the compiler skips only the first part.

Example 347 IF Directive With Matching ENDIF Directive

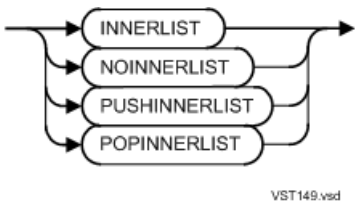
```
?RESETTOG flag ! Create & turn off flag
?IF flag
    ! Statements for true condition
    ! (skipped because flag is off)
?ENDIF flag ! ENDIF stops the skipping of statements
?IFNOT flag
    ! Statements for false condition
    ! (compiled because ENDIF appears for IF flag)
?ENDIF flag
```

Default:	None
Placement:	<ul style="list-style-type: none">Anywhere in the source file (not in the compilation command)Must be the last directive on the directive line
Scope:	Everything between IF or IFNOT and the next ENDIF that specifies the same toggle, target, or keyword
Dependencies:	Interacts with: <ul style="list-style-type: none">DEFINETOGENDIF__EXT64RESETTOG

	<ul style="list-style-type: none"> • SETTOG • TARGET
References:	<ul style="list-style-type: none"> • DEFINETO (page 388) • ENDIF (page 390) • “__EXT64” (page 394) • RESETTOG (page 411) • SETTOG (page 415) • TARGET (page 423) • Toggles (page 370)

An asterisk (*) appears in column 11 of the listing for any statements not compiled because of the IF or IFNOT directive.

INNERLIST



INNERLIST	lists mnemonics for each statement after that statement in the compiler listing.
NOINNERLIST	suppresses the mnemonics for each statement after that statement in the compiler listing.
PUSHINNERLIST	pushes the current setting (INNERLIST or NOINNERLIST) onto the INNERLIST directive stack. Does not change the current setting.
POPINNERLIST	pops the top value from the INNERLIST directive stack and changes the current setting to that value.
For an explanation of directive stacks, see Directive Stacks (page 369) .	

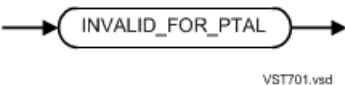
Default:	NOINNERLIST
Placement:	Anywhere
Scope:	<ul style="list-style-type: none"> • INNERLIST applies to subsequent statements it until it is overridden by NOINNERLIST • NOINNERLIST applies to subsequent statements until it is overridden by INNERLIST
Dependencies:	INNERLIST has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none"> • LIST (page 401) • SUPPRESS (page 420)

Example 348 INNERLIST and NOINNERLIST Directives

```
PROC any;
BEGIN
    INT x, y, z; ! No innerlisting here
    ! Statements that initialize variables
?INNERLIST      ! Start innerlisting here
    ! Statements that manipulate variables
?NOINNERLIST    ! Stop innerlisting here
END;
```

INVALID_FOR_PTAL

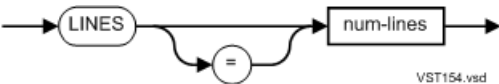
INVALID_FOR_PTAL forces the compiler to report an error message. Use it to identify a TAL source file that the pTAL or EpTAL compiler must not compile.



Default:	None
Placement:	After IF or IFNOT and before ENDIF
Scope:	Applies to code between itself and ENDIF
Dependencies:	None
References:	<ul style="list-style-type: none">• IF and IFNOT (page 398)• ENDIF (page 390)

LINES

LINES sets the maximum number of output lines per page if the list file is a line printer or a process.

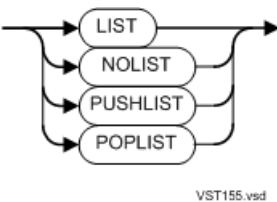


num-lines

is an unsigned decimal constant in the range 10 through 32,767.

Default:	LINES 60
Placement:	Anywhere
Scope:	Applies until overridden by another LINES directive
Dependencies:	Has no effect if the list file is a terminal

LIST



LIST

lists the source code in the compiler listing.

NOLIST

suppresses the source code the compiler listing.

PUSHLIST

pushes the current setting (LIST or NOLIST) onto the LIST directive stack. Does not change the current setting.

POPLIST

pops the top value from the LIST directive stack and changes the current setting to that value.

For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	LIST
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• LIST applies to subsequent code it until it is overridden by NOLIST• NOLIST applies to subsequent code until it is overridden by LIST
Dependencies:	LIST has no effect if SUPPRESS is active
References:	SUPPRESS (page 420)

Each line in the source listing consists of:

- An edit file number
- A lexical level:

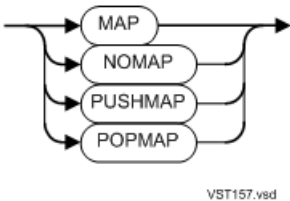
Lexical Level	Meaning
0	Global level
1	Procedure level
2	Subprocedure level

- A nesting level (only for BEGIN-END items such as structures, substructures, IF statements, and CASE statements)

Example 349 Listing Source Code But Not System Declarations

```
?NOLIST, SOURCE $system.system.extdecs (  
? process_getinfo_, process_stop_)  
?LIST
```

MAP



MAP

lists identifier maps in the compiler listing.

NOMAP

suppresses identifier maps in the compiler listing.

PUSHMAP

pushes the current setting (MAP or NOMAP) onto the MAP directive stack. Does not change the current setting.

POPMAP

pops the top value from the MAP directive stack and changes the current setting to that value.

For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	MAP
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• MAP applies to subsequent code it until it is overridden by NOMAP• NOMAP applies to subsequent code until it is overridden by MAP
Dependencies:	MAP has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none">• LIST (page 401)• SUPPRESS (page 420)

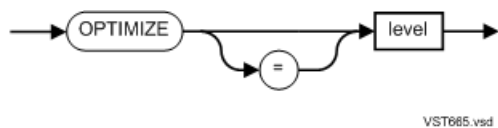
MAP lists:

- Sublocal identifiers after each subprocedure
- Local identifiers after each procedure
- Global identifiers after the last procedure in the source program

Each identifier map includes:

Item	Possible Values
Identifier class	<ul style="list-style-type: none">• VAR• SUBPROC• ENTRY• LABEL• DEFINE• LITERAL
Type	<ul style="list-style-type: none">• Data type• Structure• Substructure• Structure pointer
Addressing mode	<ul style="list-style-type: none">• Direct• Indirect
Subprocedure, entry, or label offset	
Text of LITERALS and DEFINES	

OPTIMIZE



level

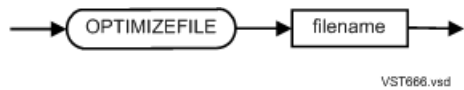
Level	Effect
0	Code is not optimized. Provided in case other optimization levels cause errors or interfere with debugging. Supports symbolic debugging; data is always in memory.
1	Code is optimized within statements and across statement boundaries. The resulting code is more efficient than that produced by lower levels of optimization and does not interfere with debugging.
2	Code is optimized within statements and across statement boundaries, and the resulting code is more efficient than code produced by lower levels.

NOTE: If your program compiles successfully at level 0 but runs out of memory at level 1 or 2, either compile your program only at level 0 or split your program into smaller subprograms and compile those at the same higher level.

Default:	OPTIMIZE 1
Placement:	Outside the boundary of a separately compiled program
Scope:	The optimization level active at the beginning of a separately compiled program determines the level of optimization for that program and any programs it contains
Dependencies:	None, but OPTIMIZEFILE can override OPTIMIZE in individual procedures
References:	OPTIMIZEFILE (page 404)

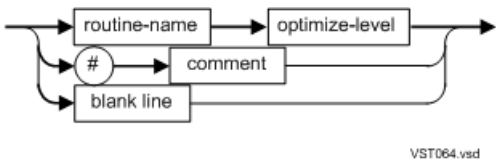
OPTIMIZEFILE

OPTIMIZEFILE sets the optimization level for individual procedures and subprocedures.



filename

is an EDIT file on Guardian platforms and a text file on Windows platforms. Each line of the file must have this syntax:



(See [Example 350 \(page 405\)](#).)

routine-name

is either a:

- procedure name
- subprocedure name of the form *procedure-name.subprocedure-name*

Each *routine-name* in *filename* must appear only once in *filename*.

optimize-level

is an integer. If it is not 0, 1, or 2, the compiler ignores the line. *optimize-level* must be preceded by white space and it can be followed by white space.

comment

is any text.

Default:	The optimization level that OPTIMIZE specified
Placement:	Only in the compilation command (not in the source file)
Scope:	Applies to the compilation unit
Dependencies:	None
References:	OPTIMIZE (page 404)

Example 350 File for OPTIMIZEFILE Directive

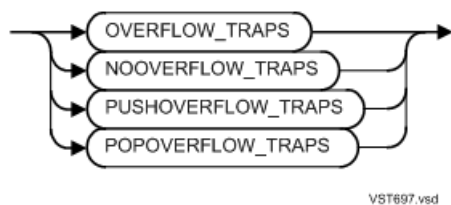
```
# This is the optimizefile for compilation xyz.  
abc.sub 0  
abc 2
```

```
def 1
```

Difference between pTAL and EpTAL compilers:

pTAL Compiler	EpTAL Compiler
Does not issue warnings for errors in <i>filename</i>	Issues a warning when <i>filename</i> : <ul style="list-style-type: none">• Does not exist• Cannot be opened• Is not an EDIT file (Guardian operating systems only)• Has the same <i>routine-name</i> on more than one line• Has a line that:<ul style="list-style-type: none">◦ Exceeds 511 characters (Windows operating systems only)◦ Has a <i>routine-name</i> that does not match any routine declaration in the source file◦ Has an <i>optimize-level</i> other than 0, 1, or 2◦ Has one or more characters other than spaces or tabs:<ul style="list-style-type: none">– Before <i>routine-name</i>– After <i>optimize-level</i>– Between <i>routine-name</i> and <i>optimize-level</i>

OVERFLOW_TRAPS



- OVERFLOW_TRAPS**
enables overflow traps throughout the program.
- NOOVERFLOW_TRAPS**
disables overflow traps throughout the program, except where you specify an overflow trapping procedure attribute or block attribute.
- PUSHOVERFLOW_TRAPS**
pushes the current setting (OVERFLOW_TRAPS or NOOVERFLOW_TRAPS) onto the OVERFLOW_TRAPS directive stack. Does not change the current setting.
- POPOVERFLOW_TRAPS**
pops the top value from the OVERFLOW_TRAPS directive stack and changes the current setting to that value.
- For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	pTAL compiler: OVERFLOW_TRAPS
	EpTAL compiler: NOOVERFLOW_TRAPS
Placement:	Before or between procedure declarations
Scope:	From where the directive it occurs in the compilation until the directive is overridden or the compilation ends, whichever occurs first
Dependencies:	OVERFLOW_TRAPS is overridden by: <ul style="list-style-type: none">• NOOVERFLOW_TRAPS procedure attribute• DISABLE_OVERFLOW_TRAPS block attributes NOOVERFLOW_TRAPS is overridden by: <ul style="list-style-type: none">• OVERFLOW_TRAPS procedure attribute• ENABLE_OVERFLOW_TRAPS block attributes
References:	See Managing Overflow Traps (page 234)

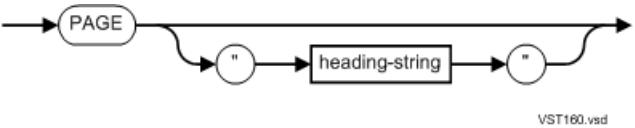
Example 351 OVERFLOW_TRAPS Compiler Directive

```
?OVERFLOW_TRAPS      ! Correct
PROC p;
  BEGIN
?NOOVERFLOW_TRAPS    ! Incorrect: OVERFLOW_TRAPS must appear
  ...                ! between procedure declarations
END;
?NOOVERFLOW_TRAPS    ! Correct
PROC q;
  BEGIN
  ...
END;
```

NOTE: OVERFLOW_TRAPS directive does not control the effects of the \$EXT64ADDR_TO_EXT32ADDR_OV directive (See directive “\$EXT64ADDR_TO_EXT32ADDR_OV” (page 307)).

PAGE

The first PAGE sets the string to be printed as part of the heading for each page. Each subsequent PAGE prints the heading and causes a page eject.



heading-string

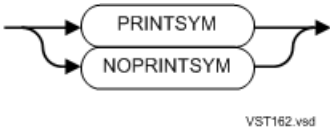
is a character string of at most 122 characters. The default is an empty string.

Default:	LINES determines page ejects and no heading is printed
Placement:	Only in the source file (not in the compilation command)
Scope:	Applies until overridden by another PAGE directive
Dependencies:	<ul style="list-style-type: none">• Has no effect if either:• NOLIST or SUPPRESS is active• The list file is a terminal• Interacts with SAVEGLOBALS and USEGLOBALS (see Saving and Using Global Data Declarations (page 372))
References:	<ul style="list-style-type: none">• LINES (page 401)• LIST (page 401)• SAVEGLOBALS (page 413)• SUPPRESS (page 420)• USEGLOBALS (page 423)

Example 352 PAGE Directive

```
! MYSOURCE file
?PAGE "Here are global declarations for MYSOURCE"
! Global declarations
?PAGE "Here are procedure declarations for MYSOURCE"
! Procedure declarations
```

PRINTSYM



PRINTSYM

lists symbols in the compiler listing.

NOPRINTSYM

suppresses symbols in the compiler listing.

Default:	PRINTSYM
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• PRINTSYM applies to subsequent declarations until overridden by NOPRINTSYM• NOPRINTSYM applies to subsequent declarations until overridden by PRINTSYM
Dependencies:	<ul style="list-style-type: none">• PRINTSYM has no effect if NOLIST or SUPPRESS is active• PRINTSYM interacts with SAVEGLOBALS and USEGLOBALS (see Saving Global Data Declarations (page 373))
References:	<ul style="list-style-type: none">• LIST (page 401)• SAVEGLOBALS (page 413)• SUPPRESS (page 420)• USEGLOBALS (page 423)

You can use PRINTSYM and NOPRINTSYM to list individual symbols or groups of symbols, such as global, local, or sublocal declarations.

Example 353 PRINTSYM Directive

```
?NOPRINTSYM ! Turn off symbol listing
  INT i;
  INT j;
?PRINTSYM   ! Turn on symbol listing
  INT k;
```

PROFDIR

This directive can be used only with the EpTAL compiler.

PROFDIR specifies where an instrumented process will create the raw data file. For detailed information about using the PROFDIR directive when performing profile-guided optimization, see the *Code Profiling Utilities Manual*.



Default:	Default subvolume
Placement:	Only on the command line
Scope:	Applies to the compilation unit
Dependencies:	PROFDIR is ignored if PROFGEN or CODEDOV is not also specified
References:	<ul style="list-style-type: none"> • PROFGEN (page 409) • CODECOV (page 385)

PROFGEN

This directive can be used only with the EpTAL compiler.

PROFGEN directs the compiler to generate instrumented object code for use in performing profile-guided optimization. For more information about profile-guided optimization, see the *Code Profiling Utilities Manual*.

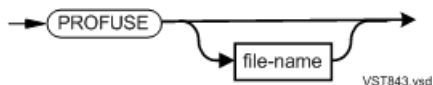


Default:	No instrumentation in object code
Placement:	Only on the command line
Scope:	Applies to the compilation unit
Dependencies:	None

PROFUSE

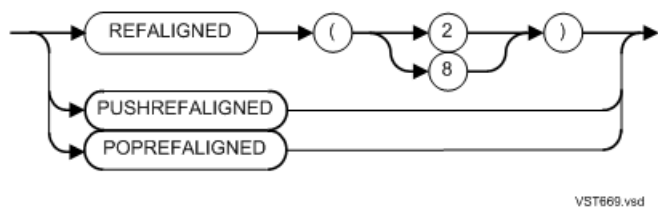
This directive can be used only with the EpTAL compiler.

PROFUSE directs the compiler to generate optimized object code based on information in a dynamic profiling information (DPI) file. For detailed information about the PROFUSE directive and profile-guided optimization, see the *Code Profiling Utilities Manual*.



Default:	None
Placement:	Only on the command line
Scope:	Applies to the compilation unit
Dependencies:	Cannot be specified with PROFGEN or CODECOV
References:	<ul style="list-style-type: none"> • PROFGEN (page 409) • CODECOV (page 385)

REFALIGNED

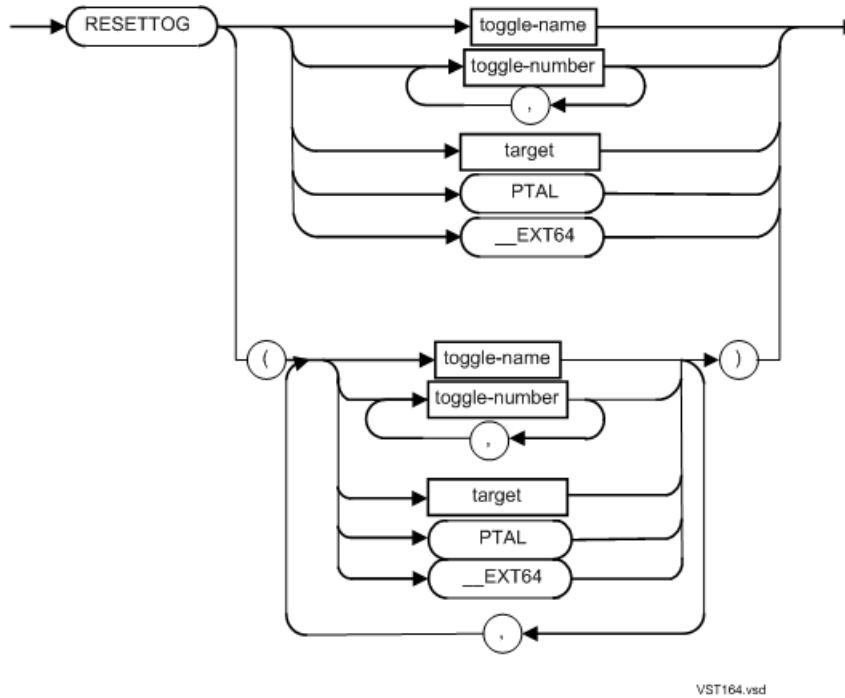


- REFALIGNED**
specifies the default alignment for pointers to nonstructure data items and procedure reference parameters.
- PUSHREFALIGNED**
pushes the current setting [REFALIGNED (2) or REFALIGNED (8)] onto the REFALIGNED directive stack. Does not change the current setting.
- POPREFALIGNED**
pops the top value from the REFALIGNED directive stack and changes the current setting to that value.
- For an explanation of directive stacks, see [Directive Stacks \(page 369\)](#).

Default:	REFALIGNED 8
Placement:	Anywhere
Scope:	Applies to subsequent pointers to nonstructure data items and procedure reference parameters until overridden by another REALIGN directive
Dependencies:	None

RESETTOG

RESETTOG turns off either specified toggles or all numeric toggles.



toggle-name

is an identifier with a maximum of 31 characters in length.

The only characters allowed in a toggle-name are alphabetic ("A" through "Z" and "a" through "z"), numeric ("0" through "9"), underscore ("_"), and circumflex ("^"); the first character must be alphabetic.

Names are case-insensitive (For example, abc is the same as Abc.)

toggle-number

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored.

target

is as defined in ["TARGET" \(page 423\)](#). In TAL, a warning is returned if a target is specified and the RESETTOG directive is ignored. In pTAL and EpTAL, RESETTOG can be applied to a target only if the target specified was not named in the compiled TARGET directive.

PTAL

is a toggle implicitly defined and set by the TAL, pTAL and EpTAL compilers. It is set on if the compiler in use is any pTAL or EpTAL compiler, otherwise it is set off. See ["DEFINETOG" \(page 388\)](#).

The TAL compiler emits a warning if PTAL is specified and the RESETTOG directive is ignored. In pTAL and EpTAL, an error is emitted if you specify PTAL in a RESETTOG directive.

__EXT64

is a toggle implicitly defined and set by the EpTAL compiler starting with SPR T0561H01^AAP. It is set on if the corresponding ["__EXT64" \(page 394\)](#) directive has been specified otherwise, it is set off. The __EXT64 directive controls the availability of 64-bit addressing functionality; see ["DEFINETOG" \(page 388\)](#) and Appendix E, ["64-bit Addressing Functionality" \(page 531\)](#).

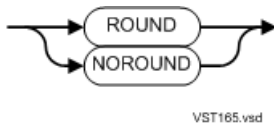
In TAL, pTAL and EpTAL prior to T0561H01^AAP, you can RESETTOG the __EXT64 toggle however, this is not recommended. In T0561H01^AAP EpTAL, RESETTOG can be applied to the __EXT64 toggle only if the implicit setting of the toggle is already off.

RESETTOG with no arguments turns off all numeric toggles but does not affect named toggles.

Default:	None
Placement:	<ul style="list-style-type: none">• With a parenthesized list, it can appear anywhere• Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none">• DEFINETOG• ENDIF• __EXT64• IF and ENDIF• SETTOG• TARGET
References:	<ul style="list-style-type: none">• DEFINETOG (page 388)• ENDIF (page 390)• "__EXT64" (page 394)• IF and IFNOT (page 398)• SETTOG (page 415)• "TARGET" (page 516)• Toggles (page 370)

ROUND

ROUND rounds FIXED values assigned to FIXED variables that have smaller *fpoint* values than the values you are assigning.



ROUND

turns on rounding. If the *fpoint* of the assignment value is greater than that of the variable, ROUND first truncates the assignment value so that its *fpoint* is one greater than that of the destination variable. The truncated assignment value is then rounded away from zero as follows:

$$\text{value} = (\text{IF } \text{value} < 0 \text{ THEN } \text{value} - 5 \text{ ELSE } \text{value} + 5) / 10$$

In other words, if the truncated assignment value is negative, 5 is subtracted; if positive, 5 is added. Then, an integer division by 10 is performed, and the result is truncated again, this time by a factor of 10. Thus, if the absolute value of the least significant digit of the initially truncated assignment value is 5 or more, a 1 is added to the absolute value of the final least significant digit.

NOROUND

turns off rounding. That is, rounding does not occur when a FIXED value is assigned to a FIXED variable that has a smaller *fpoint*. If the *fpoint* of the assignment value is greater than that of the variable, the assignment value is truncated and some precision is lost.

Default:	NOROUND
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• ROUND applies to subsequent code until overridden by NOROUND• NOROUND applies to subsequent code until overridden by ROUND
Dependencies:	None

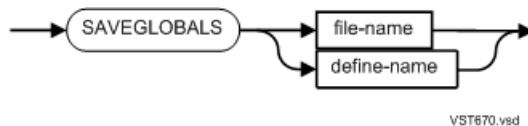
Example 354 ROUND Directive

```
?ROUND ! Request rounding
! Global declarations
PROC a;
BEGIN
    FIXED(2) f1;
    FIXED(3) f2;
    f1 := f2;
END;
```

SAVEGLOBALS

NOTE: The EpTAL compiler does not accept this directive. See [Migrating from TNS/R to TNS/E \(page 375\)](#).

SAVEGLOBALS saves all global data declarations in a file for use in subsequent compilations that specify the USEGLOBALS directive.



file-name

is the name of a disk file to which the compiler is to write the global data declarations.

If *file-name* already exists, the compiler purges the existing file and creates an unstructured global declarations file.

If the existing file is secured so that the compiler cannot purge it, the compilation terminates.

The compiler uses the current default volume and subvolume names as needed and lists the complete file name in the trailer message at the end of compilation. For this directive, the compiler does not use HP TACL ASSIGN SSV information (available only on Guardian platforms) to complete the file name.

define-name

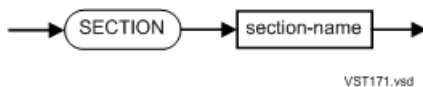
is the name of a MAP DEFINE that refers to the disk file to which you want the compiler to write the global data declarations.

NOTE: MAP DEFINEs are available only on Guardian platforms.

Default:	None
Placement:	Either in the compilation command or in the source code before any global data declarations
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none">• If SAVEGLOBALS and USEGLOBALS appear in the same compilation unit, the compiler uses only the one that appears first• The compilation unit must have exactly one BEGINCOMPILE directive• Interacts with the directives referenced in the next row (see Saving and Using Global Data Declarations (page 372))
References:	<ul style="list-style-type: none">• BEGINCOMPILE (page 382)• PRINTSYM (page 408)• SYMBOLS (page 421)• SYNTAX (page 422)• USEGLOBALS (page 423)

SECTION

SECTION gives a name to a section of a source file for use in a SOURCE directive.



section-name
is an identifier.

Default:	None
Placement:	<ul style="list-style-type: none">• Only in the source file (not in the compilation command)• Must be the only directive on the directive line
Scope:	Applies to subsequent code until another SECTION directive or the end of the file, whichever is first
Dependencies:	Interacts with SOURCE (see Section Names (page 417))
References:	SOURCE (page 416)

Example 355 SECTION Directive

APLLIB File

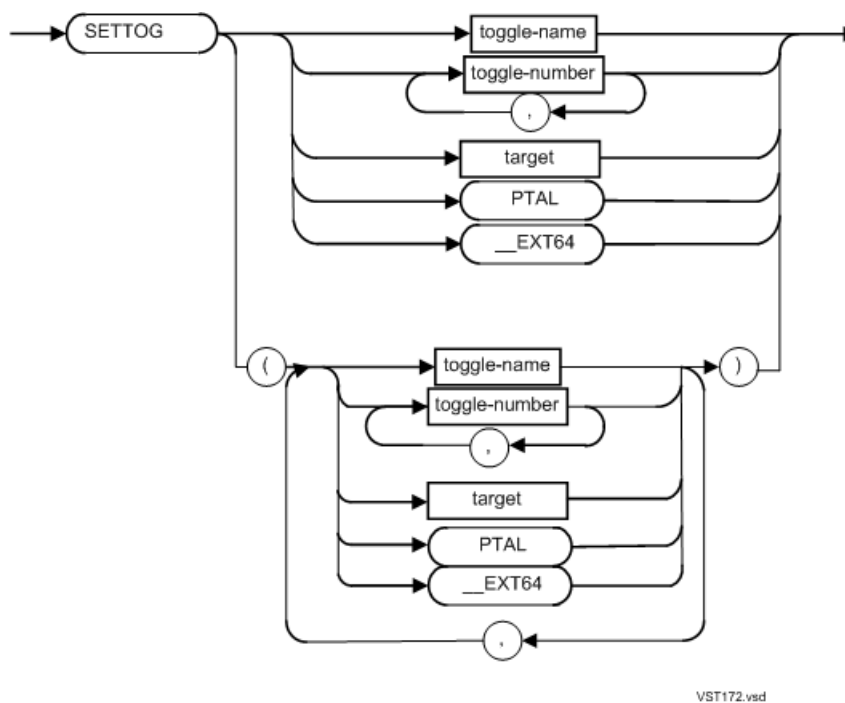
```
! File ID APPLLIB
?SECTION sort_proc
PROC sort_on_key(key1, key2, key3, length);
    INT .key1, .key2, .key3, length;
BEGIN
    ...
END;
?SECTION next_proc
```

SOURCE directive that includes section sort_proc of the preceding file:

```
?SOURCE appllib (sort_proc)
```

SETTOG

SETTOG turns on either specified toggles or all numeric toggles.



toggle-name

is an identifier with a maximum of 31 characters in length.

The only characters allowed in a toggle-name are alphabetic ("A" through "Z" and "a" through "z"), numeric ("0" through "9"), underscore ("_"), and circumflex ("^"); the first character must be alphabetic.

Names are case-insensitive (For example, abc is the same as Abc.)

toggle-number

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored.

target

is as defined in ["TARGET" \(page 423\)](#).

In TAL, a warning is returned if a target is specified and the SETTOG directive is ignored. In pTAL and EpTAL, SETTOG can only be applied to a target that was specified in a previously compiled TARGET directive.

PTAL

is a toggle implicitly defined and set by the TAL, pTAL and EpTAL compilers. It is set on if the compiler in use is any pTAL or EpTAL compiler, otherwise it is set off. See “DEFINETOG” (page 388).

The TAL compiler emits a warning if PTAL is specified and the SETTOG directive is ignored. In pTAL and EpTAL, an error is emitted if you specify PTAL in a SETTOG directive.

__EXT64

is a toggle implicitly defined and set by the EpTAL compiler starting with SPR T0561H01^AAP. It is set on if the corresponding “__EXT64” (page 394) directive has been specified otherwise, it is set off. The __EXT64 directive controls the availability of 64-bit addressing functionality; see “DEFINETOG” (page 388) and Appendix E, “64-bit Addressing Functionality” (page 531).

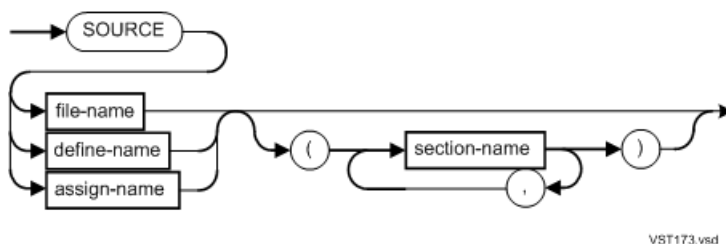
In TAL, pTAL and EpTAL prior to SPR T0561H01^AAP, you can SETTOG the __EXT64 toggle however, this is not recommended. In T0561H01^AAP EpTAL, SETTOG can be applied to the __EXT64 toggle only if the implicit setting of the toggle is already on.

SETTOG with no arguments turns on all numeric toggles but does not affect named toggles.

Default:	None
Placement:	<ul style="list-style-type: none">• With a parenthesized list, it can appear anywhere• Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none">• DEFINETOG• ENDIF• __EXT64• IF and ENDIF• RESETTOG• TARGET
References:	<ul style="list-style-type: none">• DEFINETOG (page 388)• ENDIF (page 390)• “__EXT64” (page 394)• IF and IFNOT (page 398)• RESETTOG (page 411)• “TARGET” (page 516)• Toggles (page 370)

SOURCE

SOURCE reads source code from another source file.



file-name

is the name of a disk file from which the compiler is to read source code. On Guardian platforms, the compiler uses HP TACL ASSIGN SSV information, if specified, to complete the file name; otherwise, the compiler uses the current default volume and subvolume names as needed.

define-name

is the name of a MAP DEFINE that refers to a disk file from which the compiler is to read source code.

NOTE: MAP DEFINES are available only on Guardian platforms.

assign-name

is a logical file name you have equated to a disk file (from which the compiler is to read source code) by issuing an ASSIGN command.

section-name

is an identifier specified within the included file by a SECTION directive. If the compiler does not find *section-name* in the specified file, it issues a warning.

The list of section names can extend to continuation lines.

Default:	None
Placement:	<ul style="list-style-type: none">• Only in the source file (not in the compilation command)• Must be the last directive on the directive line
Scope:	Applies to the source file
Dependencies:	<ul style="list-style-type: none">• Interacts with COLUMNS• Interacts with SECTION (see Section Names (page 417))• Interacts with the directives referenced in the next row (see Effect of Other Directives (page 418))
References:	<ul style="list-style-type: none">• BEGINCOMPILE (page 382)• COLUMNS (page 385)• LIST (page 401)• SECTION (page 414)• SUPPRESS (page 420)• USEGLOBALS (page 423)

Topics:

- [Section Names \(page 417\)](#)
- [Nesting Levels \(page 418\)](#)
- [Effect of Other Directives \(page 418\)](#)
- [Including System Procedure Declarations \(page 419\)](#)
- [Examples \(page 419\)](#)

Section Names

If you specify SOURCE with no section names, the compiler processes the specified source file until the end of that file. The compiler treats any SECTION directives in the source file as comments.

If you specify SOURCE with section names, the compiler processes the source file until it reads all the specified sections. A section begins with a SECTION directive and ends with another SECTION directive or the end of the file, whichever comes first.

The compiler reads the sections in order of appearance in the source file, not in the order specified in the SOURCE directive. If you want the compiler to read sections in a particular order, use a separate SOURCE directive for each section and place the SOURCE directives in the desired order.

Nesting Levels

You can nest SOURCE directives to a maximum of seven levels, not counting the original outermost source file. For example, the deepest nesting allowed is as follows:

1. The MAIN file F sources in file F1.
2. File F1 sources in file F2.
3. File F2 sources in file F3.
4. File F3 sources in file F4.
5. File F4 sources in file F5.
6. File F5 sources in file F6.
7. File F6 sources in file F7.

Effect of Other Directives

- COLUMNS (page 418)
- LIST and NOSUPPRESS (page 418)
- NOLIST (page 418)
- USEGLOBALS and BEGINCOMPILE (pTAL Compiler Only) (page 419)

COLUMNS

If a SOURCE directive specifies sections of a file, the compiler honors all COLUMNS directives in that file that precede the first section of that file. (The first section of the file might not be the first section of the file that the SOURCE directive specifies.) The compiler also honors COLUMN directives that appear in the sections that the SOURCE directive specifies.

After a SOURCE directive completes execution, the value of COLUMNS is restored to what it was before the SOURCE directive:

```
File1:
  ?COLUMNS 80
  ...
File2:
  ?COLUMNS 100
  ...
  ?SOURCE file1
  ! COLUMNS is restored to 100 at this point
```

LIST and NOSUPPRESS

If LIST and NOSUPPRESS are active after a SOURCE directive completes execution, the compiler prints a line identifying the source file to which it reverts and begins reading at the line following the SOURCE directive.

NOLIST

You can precede SOURCE with NOLIST to suppress the listings of procedures to be read in. Place NOLIST and SOURCE on the same line, because the line containing NOLIST is not suppressed:

```
?PUSHLIST, NOLIST, SOURCE $src.current.routines
! Suppress listings, read in external declarations of routines
?POPLIST
```

USEGLOBALS and BEGINCOMPILE (pTAL Compiler Only)

If USEGLOBALS is active, the compiler ignores all SOURCE directives until it encounters BEGINCOMPILE. For more information about how these directives interact, see [Saving and Using Global Data Declarations \(page 372\)](#).

Including System Procedure Declarations

You can use SOURCE directives to read in external declarations of system procedures from the EXTDECS files. In these files, the procedure name and the corresponding section name are the same. EXTDECS0 contains the current RVU of system procedures.

In [Example 356 \(page 419\)](#), a SOURCE directive specifies the current version of system procedures. A NOLIST directive suppresses the listings for the system procedures. Place NOLIST and SOURCE on the same line, because the line containing the NOLIST directive is not suppressed.

Example 356 SOURCE Directive Specifying System Procedure Declarations

```
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (  
?  PROCESS_DEBUG_, PROCESS_STOP_)  
! Suppress listings  
! Read external declarations of current system procedures  
?POPLIST
```

A procedure in the same source file can then call the procedures listed in the preceding SOURCE directive, as in [Example 357 \(page 419\)](#).

Example 357 Procedure That Calls Procedures Specified by SOURCE Directive

```
PROC a MAIN;  
BEGIN  
  INT x, y, z, error;  
  ! Code for manipulating x, y, and z  
  IF x = 5 THEN CALL PROCESS_STOP_;  
  CALL PROCESS_DEBUG_;           ! Call procedures listed  
END;                             ! in SOURCE directive
```

Examples

The SOURCE directive in [Example 358 \(page 419\)](#) instructs the compiler to process the file until an end of file occurs. (Any SECTION directives in the file ROUTINES are treated as comments.)

Example 358 SOURCE Directive

```
?SOURCE $src.current.routines
```

This SOURCE directive in [Example 359 \(page 419\)](#) reads three sections from the source file. It reads the files in the order in which they appear in the source file, not in the order specified in the SOURCE directive. (The specified files appear in the source file in the order sec3, sec2, and sec1, so they are read in that order.)

Example 359 SOURCE Directive

```
?SOURCE $src.current.routines (sec1, sec2, sec3)
```

[Example 360 \(page 420\)](#) shows how you can specify the order in which the compiler is to read the sections, regardless of their order in the source file.

Example 360 SOURCE Directive

```
?SOURCE $src.current.routines (sec1)
?SOURCE $src.current.routines (sec2)
?SOURCE $src.current.routines (sec3)
```

SRL

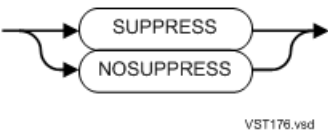
NOTE: The EpTAL compiler ignores this directive.

SRL causes the pTAL compiler to generate code that can be linked into a user library. You must specify SRL to be able to link the object file created by the compilation into a user library.



Default:	None
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	When declaring a data block that belongs to an SRL, you must specify NOEXPORT_GLOBALS and NOGP_OK.
References:	<ul style="list-style-type: none">• EXPORT_GLOBALS (page 393)• GP_OK (page 397)

SUPPRESS



SUPPRESS

suppresses all compilation listings except the compiler leader text, diagnostic messages, and the trailer text. (Does not alter the source code.)

NOSUPPRESS

allows all compilation listings.

Default:	NOSUPPRESS
Placement:	Anywhere
Scope:	Applies to the compilation unit

Dependencies:	Overrides all the listing directives (referenced in the next row)
References:	<ul style="list-style-type: none"> • DEFEXPAND (page 386) • FMAP (page 396) • GMAP (page 397) • INNERLIST (page 400) • LIST (page 401) • MAP (page 402) • PAGE (page 407) • PRINTSYM (page 408)

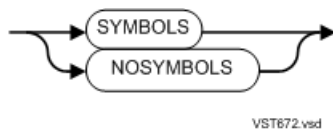
The compilation command in [Example 361 \(page 421\)](#) starts the compilation and suppresses all source code listings and maps from printing in the compiler output.

Example 361 SUPPRESS Directive

```
PTAL /IN mysrc, OUT $s.#lists/ myobj;
```

SYMBOLS

SYMBOLS saves symbols in a symbol table in the object file, enabling you use a symbolic debugger to debug the object file.



SYMBOLS

saves all symbols information.

NOSYMBOLS

saves information about:

- Procedure memos
- Global data block names
- Line numbers

Does not save information about parameters, local variables, data types, and so on.

Default:	NOSYMBOLS
Placement:	Before the first declaration in the compilation
Scope:	The last legally placed SYMBOLS or NOSYMBOLS applies to the compilation unit
Dependencies:	Interacts with SAVEGLOBALS and USEGLOBALS (see Saving Global Data Declarations (page 373))
References:	<ul style="list-style-type: none"> • SAVEGLOBALS (page 413) • USEGLOBALS (page 423)

- NOTE:** These linker options discard information that SYMBOLS saves:
- `-x` discards line number information.
 - `-s` discards information needed for future linking (use it only in building an executable file).

Usually you save symbols for the entire compilation by specifying SYMBOLS once at the beginning of the compilation unit. The symbol table then contains all the symbols generated by the source code.

Example 362 SYMBOLS Directive

```
! MYSOURCE file
?SYMBOLS ! Save symbols for compilation unit
! Declare global data
! Declare procedures
```

After debugging the program, you can use the linker to create a new, smaller object file without symbols. The executable portion of the old object file remains intact, but you dramatically reduce what you can do with a symbolic debugger.

```
nld -x -r oldobj -o newobj
ld -x -r oldobj -o newobj
eld -x -r oldobj -o newobj
```

Use the linker option `-s` when linking a loadfile, or use the `strip` utility after creating the loadfile.

```
STRIP oldobj
```

SYNTAX

SYNTAX checks the syntax of the source text without producing an object file.



Default:	The compiler produces an object file
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	Interacts with SAVEGLOBALS and USEGLOBALS (see Saving Global Data Declarations (page 373))
References:	<ul style="list-style-type: none">• SAVEGLOBALS (page 413)• USEGLOBALS (page 423)

The compilation command in [Example 363 \(page 422\)](#) checks the syntax of global data declarations in source file `myprog` and saves the declarations in file `ptalsym` for use in subsequent compilations.

Example 363 SYNTAX Directive

```
pTAL /IN myprog/; SAVEGLOBALS ptalsym, SYNTAX
```

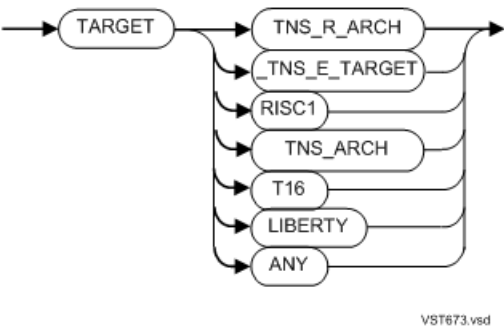
The compilation command in [Example 364 \(page 423\)](#) checks for the syntax of the code or data in source file `myprog`. In this compilation, USEGLOBALS retrieves global data declarations saved in the compilation shown in [Example 363 \(page 422\)](#).

Example 364 SYNTAX Directive

```
pTAL /IN myprog/; USEGLOBALS ptalsym, SYNTAX
```

TARGET

TARGET specifies the architecture on which you will run the object file produced by the current compilation.



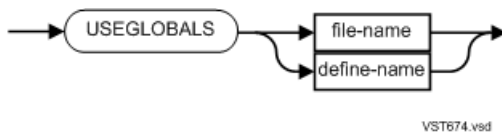
- RISC1 specifies the TNS/R architecture. This is the only option that the pTAL compiler accepts. It is also the default for the pTAL compiler.
- _TNS_E_TARGET specifies the TNS/E architecture. This is the only option that the EpTAL compiler accepts. It is also the default for the EpTAL compiler.
- TNS_ARCH specifies the TNS architecture. The compiler does not accept this option.
- T16 specifies the T16 architecture. The compiler does not accept this option.
- TNS_R_ARCH, LIBERTY specifies the Liberty architecture. The compiler does not accept this option.
- ANY specifies any architecture. The compiler does not accept this option.

Default:	pTAL compiler:	TNS_R_ARCH
EpTAL compiler:	_TNS_E_TARGET	
Placement:	Anywhere	
Scope:	Applies to the compilation unit	
Dependencies:	None	

USEGLOBALS

NOTE: The EpTAL compiler does not accept this directive. See [Migrating from TNS/R to TNS/E \(page 375\)](#).

USEGLOBALS reads global data declarations and initializations that were saved in a file by SAVEGLOBALS during a previous compilation.



file-name

is the name of the global declarations disk file created by SAVEGLOBALS in a previous compilation.

On Guardian platforms, the compiler uses HP TACL ASSIGN SSV information, if specified, to complete the file name; otherwise, the compiler uses the current default volume and subvolume names as needed.

define-name

is the name of a MAP DEFINE that refers to the global declarations file.

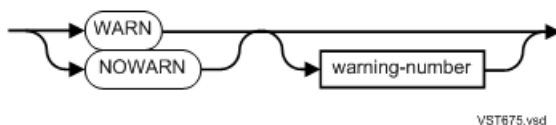
NOTE: MAP DEFINES are available only on Guardian platforms.

assign-name

is a logical file name you have equated to a disk file (that refers to the global declarations file) by issuing an ASSIGN command.

Default:	None
Placement:	Either in the compilation command or in the source code before any global data declarations
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none"> The compilation unit must have exactly one BEGINCOMPILATION directive. The compiler exports the data blocks declared in the USEGLOBALS declarations file only if EXPORT_GLOBALS is active when the compiler encounters the BEGINCOMPILATION directive. A module that specifies USEGLOBALS can export a global data block that was declared in the compilation that specified SAVEGLOBALS only if the SAVEGLOBALS compilation exported the data block. <p>Typically, a project that uses SAVEGLOBALS explicitly links globals into the object file and specifies NOEXPORT_GLOBALS (the default) for all individual compilations.</p> <ul style="list-style-type: none"> Interacts with the directives referenced in the next row (see Saving and Using Global Data Declarations (page 372))
References:	<ul style="list-style-type: none"> BEGINCOMPILATION (page 382) EXPORT_GLOBALS (page 393) PRINTSYM (page 408) SAVEGLOBALS (page 413) SYMBOLS (page 421) SYNTAX (page 422)

WARN



WARN

prints specific (or all) warning messages in the compiler listing.

NOWARN

suppresses specific (or all) warning messages in the compiler listing.

warning-number

is the number of a warning message. The default is all warning messages.

If *warning-number* is outside the range of all pTAL warnings and all TAL warnings, the compiler issues a warning. If *warning-number* is inside either range but not assigned warning text, the compiler ignores the WARN directive. For an explanation of how the compiler handles TAL warnings, see the *pTAL Conversion Guide*.

Default:	WARN
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• WARN applies to subsequent code until overridden by NOWARN• NOWARN applies to subsequent code until overridden by WARN; however: To print selected warnings, you must specify WARN before any NOWARN directives. If you specify NOWARN first, subsequent WARN <i>warning-number</i> directives have no effect.
Dependencies:	None

You can use NOWARN when a compilation produces a warning and you have determined that no real problem exists. Before the source line that produces the warning, specify NOWARN and the number of the warning you want suppressed. Following that source line, specify a WARN directive.

If NOWARN is active, the compiler records the number of suppressed and unsuppressed warnings. The compilation statistics at the end of the compiler listing include the following counts:

```
Number of unsuppressed compiler warnings = count
Number of warnings suppressed by NOWARN = count
```

Unsuppressed compiler warnings are compiler warnings that are not suppressed by NOWARN directives. The summary does not report the location of the last compiler warning.

If no compiler errors and no unsuppressed compiler warnings occur, the completion code is zero.

The following directive specifies that the compiler does not print warning message 12:

```
?NOWARN 12
```

18 pTAL Cross Compiler

The optional pTAL cross compiler runs on the PC platforms in [Table 77 \(page 426\)](#).

Table 77 pTAL Cross Compiler Platforms

Platform			Windows Operating System		
PC	Guardian	Cross Compiler Name	NT 4.0	2000	XP
ETK ¹	TNS/R	NonStop pTAL	Yes	Yes	Yes
	TNS/E ²	NonStop pTAL	No	Yes	Yes
PC command line	TNS/R ³	ptal	Yes	Yes	Yes
	TNS/E ²	eptal	No	Yes	Yes

¹ HP Enterprise Toolkit—NonStop Edition

² H06.01 and later RVUs

³ G06.14 and later RVUs

On all Windows platforms, valid pTAL cross compiler source files must have the extension `.tal`.

The pTAL cross compiler allows you to:

- Write, compile, and link NonStop RISC-based or Itanium-based server applications (NonStop Guardian executable files, static libraries, user libraries, and DLLs) on the PC and transfer them to the Guardian platform for use in production.
Object files built on the PC platform are functionally identical object files built in the NonStop RISC-based or Itanium-based server platform.
- Link pTAL, C/C++, and NMCOBOL or ECOBOL objects into a single object file.
- When multiple RVUs are installed, use any installed RVUs of the cross compilers and libraries. (Tools must come from the same RVU—HP does not test the interactions of tools used in one RVU with tools from other RVUs.)
- On the ETK platform, enter ADD, MODIFY, SET, and DELETE statements into a TACL DEFINE file (see [TACL DEFINE Tool \(ETK\) \(page 431\)](#)).

The pTAL cross compiler is delivered on a separate CD and is not available on the site update tape (SUT).

Topics:

- [NonStop pTAL \(ETK\) \(page 426\)](#)
- [pTAL or EpTAL \(PC Command Line\) \(page 427\)](#)
- [Compilation and Linking \(page 429\)](#)
- [Debugging \(page 429\)](#)
- [Tools and Utilities \(page 430\)](#)
- [Documentation \(page 431\)](#)

NonStop pTAL (ETK)

The optional pTAL cross compiler for use with the ETK, NonStop pTAL, is available for TNS/R and TNS/E.

The ETK is a GUI-based extension package to Visual Studio .NET that provides full application development functions targeted for NonStop servers. Development, editing, and building functions are very similar on Visual Studio .NET and the ETK.

NonStop pTAL components are:

Component Name	File	
	TNS/R	TNS/E
Driver executable	ptal.exe	eptal.exe
Driver DLL	ptaldvr.dll	(No driver)
Front end	ptalc.dll	eptalcom.exe
External declaration file	extdec.tal	eextdec.tal

The directory structure of NonStop pTAL is:

Directory	Files	
	TNS/R	TNS/E
bin	ptal.exe nld.exe ld.exe	eptal.exe eeld.exe
cmplr	ptaldvr.dll ptalc.dll uopt.dll ugen.dll as1.dll nld.dll ld.dll	eptalcom.exe
include	extdec.tal	extdec.tal

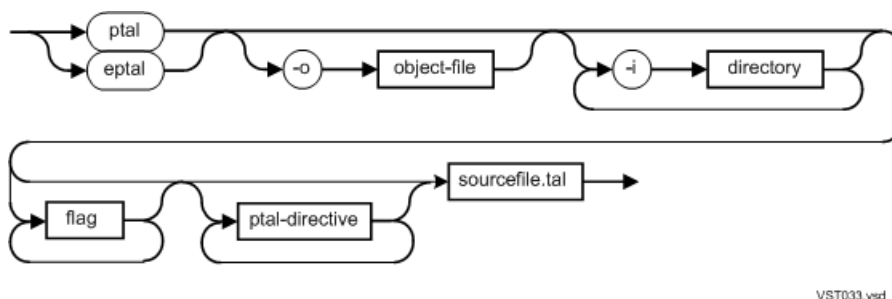
For PC and NonStop server hardware and software requirements, see the ETK online help. For instructions for accessing the online help, see [Documentation \(page 431\)](#).

pTAL or EpTAL (PC Command Line)

Beginning with RVU G06.14, you can call the pTAL cross compiler from the TNS/R command line (DOS prompt) on your PC by using the command `ptal`.

Beginning with RVU H06.01, you can call the pTAL cross compiler from the TNS/E command line (DOS prompt) on your PC by using the command `eptal`.

NOTE: Before you can use `eptal`, you must set the `COMP_ROOT` environment variable so that it points to the root of the directory location of the cross compiler. For instructions, see *Using the Command-Line Cross Compilers on Windows*.



`ptal`

calls the pTAL cross compiler from the command line. `ptal` is not case-sensitive.

eptal

calls the EpTAL cross compiler from the command line. *eptal* is not case-sensitive.

object-file

is the name of the object file to be created. The default is *sourcefile.o*.

directory

is the name of a directory for the compiler to search. If no directory is specified, the compiler searches only in the current working directory. If any directories are specified, the compiler searches them in the order in which they are listed, but does not search the current directory unless it is explicitly named.

flag

is one of the following:

flag	Directs the compiler to:
-Whelp	Display information about how to run the compiler. No compilation system components are run.
-Wusage	Display information about how to run the compiler. No compilation system components are run.
-Wverbose	Displays the command line used when the driver calls each component of the compiler.

ptal-directive

is one of the following:

Directives	Sources
-blockglobals	BLOCKGLOBALS (page 496)
-[no]call_shared	CALL_SHARED (page 496)
-[no]checkshiftcount	CHECKSHIFTCOUNT (page 497)
-codecov (eptal only)	CODECOV (page 497)
-columns= <i>n</i>	COLUMNS (page 498)
-[no]defexpand	DEFEXPAND (page 498)
-[no]do_tns_syntax	DO_TNS_SYNTAX (page 500)
-errors= <i>n</i>	ERRORS (page 500)
-export_globals	EXPORT_GLOBALS (page 501)
-fieldalign(<i>value</i>)	FIELDALIGN (page 502)
-[no]fmap	FMAP (page 502)
-globalized (eptal only)	GLOBALIZED (page 502)
-[no]gmap	GMAP (page 503)
-[no]gp_ok	GP_OK (page 503)
-[no]innerlist	INNERLIST (page 505)
-invalid_for_ptal	INVALID_FOR_PTAL (page 505)
-[no]list	LIST (page 506)
-[no]map	MAP (page 506)
-optimize= <i>n</i>	OPTIMIZE (page 507)

Directives	Sources
-[no]overflow_traps	OVERFLOW_TRAPS (page 508)
-[no]printsym	PRINTSYM (page 509)
-refaligned(<i>n</i>)	REFALIGNED (page 510)
-resettog(<i>value</i>)	RESETTOG (page 511)
-[no]round	ROUND (page 512)
-settog(<i>value</i>)	SETTOG (page 513)
-[no]symbols	SYMBOLS (page 515)
-[no]syntax	SYNTAX (page 516)
-warn= <i>n</i>	WARN (page 517)

The command-line interface allows you to create batch scripts for use on multiple platforms.

Compilation and Linking

The pTAL cross compiler can compile only one pTAL source file at a time.

Difference between platforms:

ETK Platform (NonStop pTAL)	Command-Line Platform (ptal or eptal)
Compilation and linking can be performed in one step.	Linking must be performed as a separate step after compilation.
Provides a GUI-based interface for you to select linker options.	You must specify the run-time libraries to the linker.

pTAL cross compiler linking is performed with one of the cross linkers:

Cross Linker	Cross Compilers	Directive	Object Code
nld	NonStop pTALptal	-nocall_shared*	Non-PIC
ld	NonStop pTALptal	-call_shared**	PIC
eld	NonStop pTALeptal	-call_shared	PIC
* Default for pTAL. EpTAL ignores it (and issues a warning).			
** Default for EpTAL.			

NOTE: You cannot link PIC and non-PIC object files into a single object file.

For more information:

Topic	Source
nld options	<i>nld Manual</i>
ld options	<i>ld Manual</i>
eld options	<i>eld Manual</i>

Debugging

On the ETK platform, debug pTAL source code using Visual Inspect. After Visual Inspect is installed on your workstation, you can configure Visual Inspect as an external tool.

On the command-line platform, debug loadfiles that were compiled through the pTAL cross compiler either by using Visual Inspect on Windows or by running Native Inspect on the NonStop RISC-based or Itanium-based server. To use Native Inspect, you must copy the loadfiles and the source files to the host (see [PC-to-NonStop-Host Transfer Tools \(page 431\)](#)).

For more information:

Topic	Source
Visual Inspect	Visual Inspect online help
Native Inspect	<i>Native Inspect Manual</i>

⚠ CAUTION: If you use the [CODECOV \(page 385\)](#) command line option to direct a TNS/E EpTAL compiler to generate instrumented object code, the Code Coverage Utility connects to the application program as a debugger to read its memory, and so on.

This causes all the debug requests to wait until the Code Coverage Utility (the active debugger) detaches from the application. The Code Coverage Utility only detaches after the instrumented application stops.

Therefore, if you must debug an instrumented application, it should be started in debug mode by using the TACL RUND or RUNV commands.

Tools and Utilities

The following tools and utilities allow you to use the pTAL cross compiler more efficiently:

- [NonStop ar Utility \(page 430\)](#)
- [TACL DEFINE Tool \(ETK\) \(page 431\)](#)
- [PC-to-NonStop-Host Transfer Tools \(page 431\)](#)

NonStop ar Utility

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

The `ar` utility accepts all OSS files, Guardian TNS code files, Guardian C text files (file code 180 files), TNS/R (PIC and non-PIC) native object files, and TNS/E native linkfiles or loadfiles as archive members.

You can mix one or more object file formats in one archive file; however, such an archive file will not contain the symbols table and cannot be used by the linker.

If an archive contains one or more native object files of the same format, the linker can use the archive as an object file library, replacing most functions provided by the Binder `SELECT SEARCH` command.

If an archive contains one or more ...	This cross linker can use the archive as an object file library ...	For more information, see ...
TNS/R non-PIC object files	<code>nld</code> on a G-Series system	<i>nld Manual</i>
TNS/R PIC object files	<code>ld</code>	<i>ld Manual</i>
TNS/E PIC object files	<code>eld</code>	<i>eld Manual</i>

TACL DEFINE Tool (ETK)

On the ETK platform, this GUI-based tool allows you to add ADD, MODIFY, SET, and DELETE statements to a DEFINE file. The TACL DEFINE tool automatically sets the first entry in the DEFINE obey file to be SET DEFMODE ON. You can leave this default or change it to SET DEFMODE OFF. Files created by the TACL DEFINE tool have the extension `.tdf`.

PC-to-NonStop-Host Transfer Tools

ETK

The Deploy command builds and copies each project in the active solution to the NonStop host. The Transfer Tool moves any kind of files to the NonStop host for execution and debugging. The Transfer Tool is better for transferring very large, complex applications to the NonStop host. For most applications, Deploy is more convenient.

PC Command Line

From the PC command line, you can use any FTP application to transfer executable and source files to the NonStop host.

Documentation

The ETK has online help that provides conceptual, reference, task-oriented, and error message information, as well as quick-start tutorials. To access the online help, do either of the following:

- From the Help menu, select Contents, Index, or Search.
- Click the Help button in any ETK dialog box.

The command-line documentation, *Using the Command-Line Cross Compilers on Windows*, is available:

- On the pTAL cross compiler CD
- On the EpTAL cross compiler CD
- In the ETK online help in the References chapter

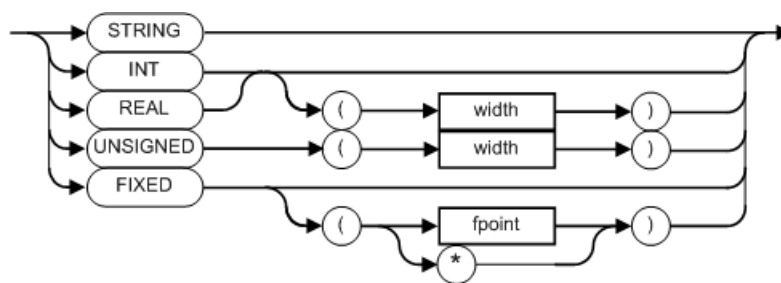
Syntax information for pTAL and EpTAL is also available from the command-line:

```
ptal -Whelp
eptal -Whelp
```

A Syntax Summary

- [Data Types \(page 432\)](#)
- [Constants \(page 432\)](#)
- [Expressions \(page 434\)](#)
- [Declarations \(page 436\)](#)
- [Statements \(page 455\)](#)
- [Overflow Traps \(page 460\)](#)
- [Built-in Routines \(page 460\)](#)
- [Compiler Directives \(page 494\)](#)

Data Types



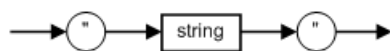
VST214.vsd

More information: [Specifying Data Types \(page 47\)](#)

Constants

- [Character String \(page 432\)](#)
- [STRING Numeric \(page 432\)](#)
- [INT Numeric \(page 433\)](#)
- [INT\(32\) Numeric \(page 433\)](#)
- [FIXED Numeric \(page 433\)](#)
- [REAL and REAL\(64\) Numeric \(page 433\)](#)
- [Constant List \(page 434\)](#)

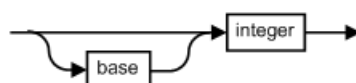
Character String



VST001.vsd

More information: [Character String \(page 57\)](#)

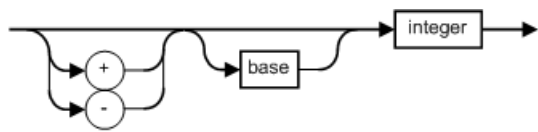
STRING Numeric



VST002.vsd

More information: [STRING Numeric \(page 58\)](#)

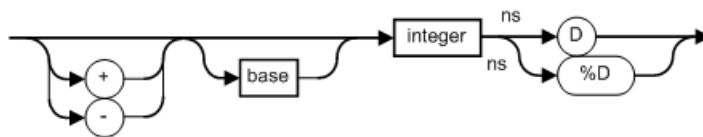
INT Numeric



VST027.vsd

More information: [INT Numeric \(page 58\)](#)

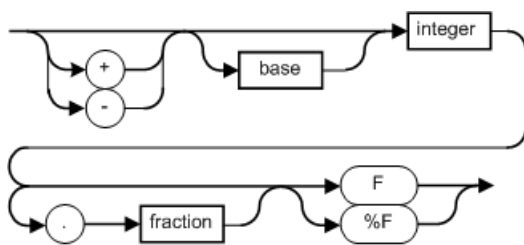
INT(32) Numeric



VST028.vsd

More information: [INT\(32\) Numeric \(page 59\)](#)

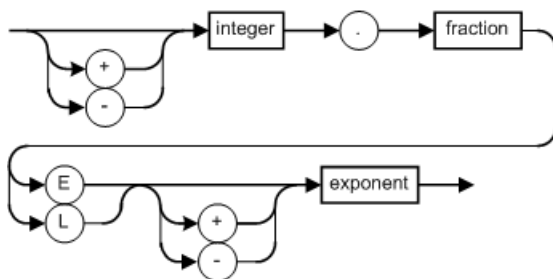
FIXED Numeric



VST005.vsd

More information: [FIXED Numeric \(page 61\)](#)

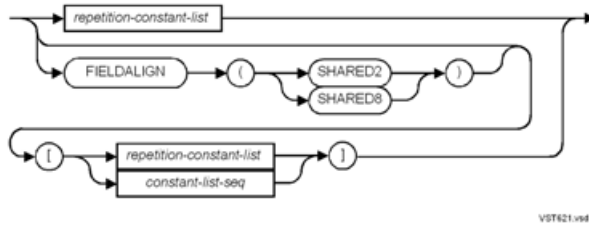
REAL and REAL(64) Numeric



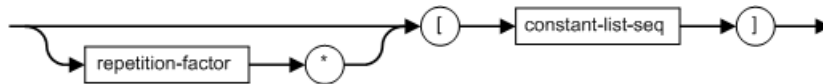
VST006.vsd

More information: [REAL and REAL\(64\) Numeric \(page 62\)](#)

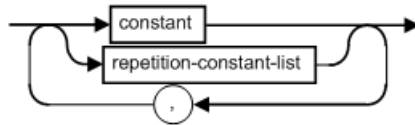
Constant List



repetition-constant-list



constant-list-seq



More information: [Constant Lists \(page 63\)](#)

Expressions

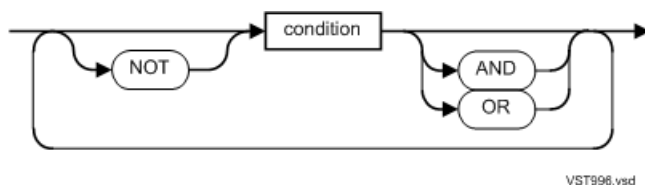
- [Arithmetic \(page 434\)](#)
- [Conditional \(page 435\)](#)
- [Assignment \(page 435\)](#)
- [CASE \(page 435\)](#)
- [IF \(page 435\)](#)
- [Group Comparison \(page 435\)](#)
- [Bit Extraction \(page 436\)](#)
- [Bit Shift \(page 436\)](#)

Arithmetic



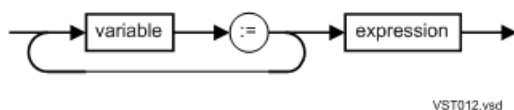
More information: [Arithmetic Expressions \(page 72\)](#)

Conditional



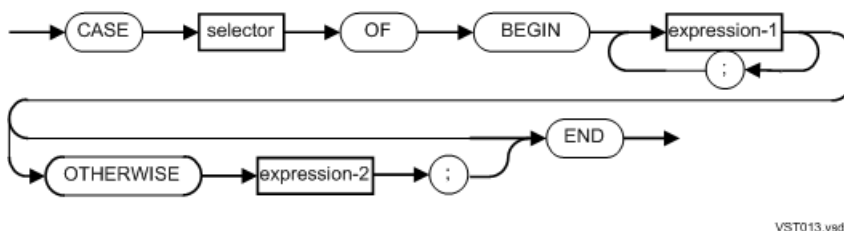
More information: [Conditional Expressions \(page 81\)](#)

Assignment



More information: [Assignment \(page 85\)](#)

CASE



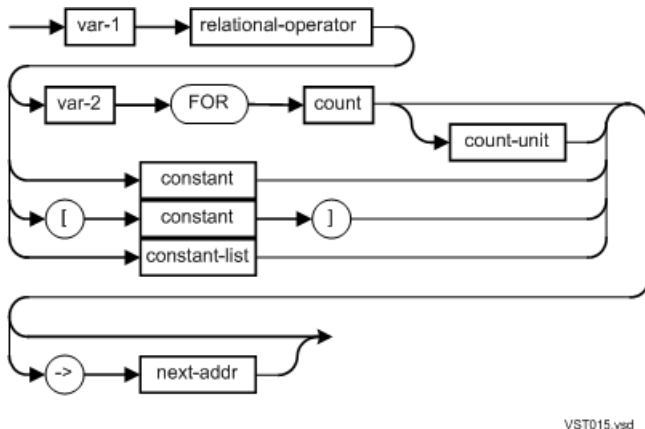
More information: [CASE \(page 86\)](#)

IF



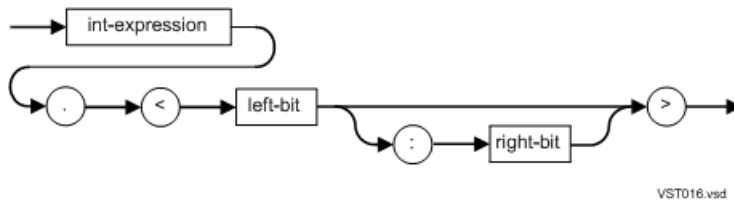
More information: [IF \(page 87\)](#)

Group Comparison



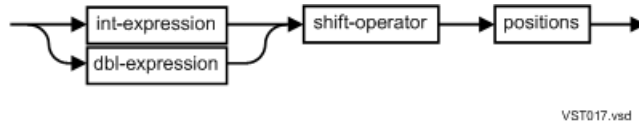
More information: [Group Comparison \(page 88\)](#)

Bit Extraction



More information: [Bit Extractions \(page 93\)](#)

Bit Shift

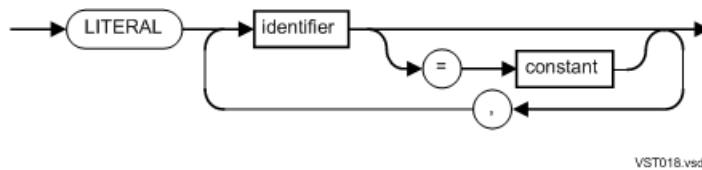


More information: [Bit Shifts \(page 94\)](#)

Declarations

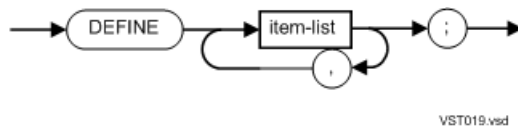
- [LITERAL \(page 436\)](#)
- [DEFINE \(page 436\)](#)
- [Simple Variable \(page 437\)](#)
- [Array \(page 437\)](#)
- [Read-Only Array \(page 438\)](#)
- [Structures \(page 438\)](#)
- [Redefinition \(page 442\)](#)
- [Pointer \(page 444\)](#)
- [Equivalenced Variable \(page 445\)](#)
- [Procedure and Subprocedure \(page 449\)](#)

LITERAL

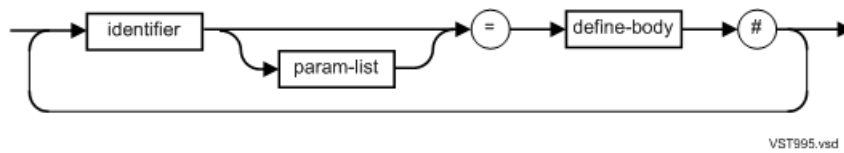


More information: [Declaring Literals \(page 97\)](#)

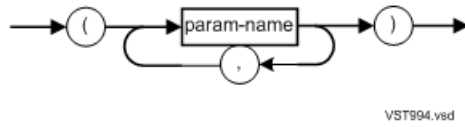
DEFINE



item-list

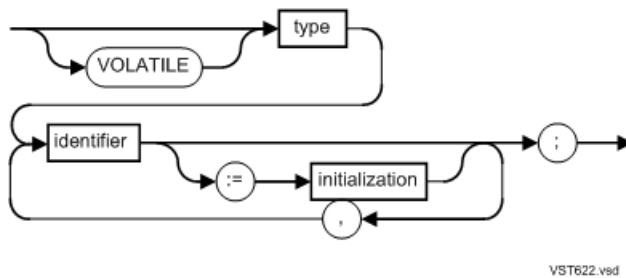


param-list



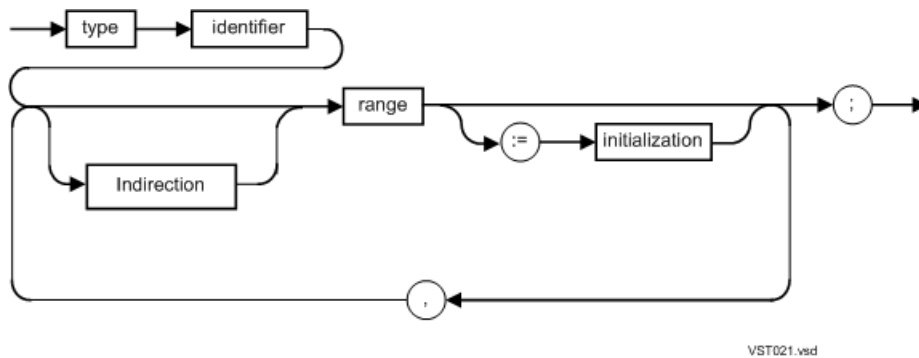
More information: [Declaring DEFINEs \(page 98\)](#)

Simple Variable

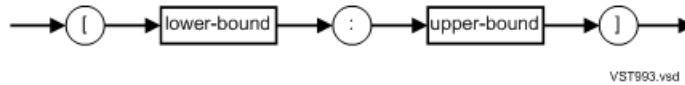


More information: [Declaring Simple Variables \(page 103\)](#)

Array

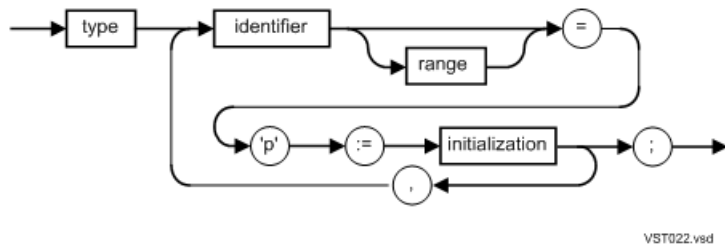


range

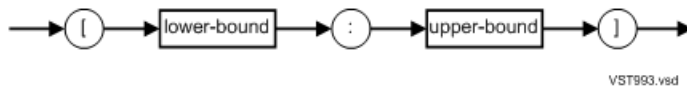


More information: [Declaring Arrays \(page 108\)](#)

Read-Only Array



range

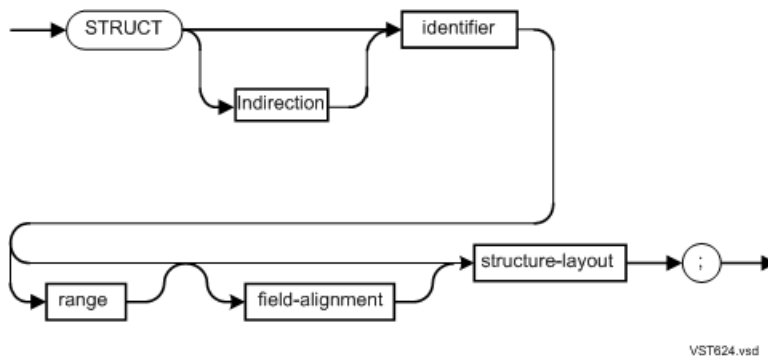


More information: [Declaring Read-Only Arrays \(page 111\)](#)

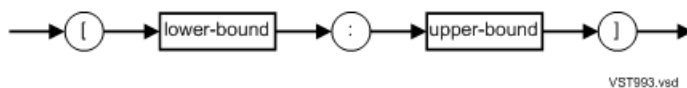
Structures

- [Definition Structure \(page 438\)](#)
- [Template Structure \(page 439\)](#)
- [Referral Structure \(page 439\)](#)
- [Simple Variables Declared in Structure \(page 440\)](#)
- [Arrays Declared in Structure \(page 440\)](#)
- [Definition Substructure \(page 440\)](#)
- [Referral Substructure \(page 440\)](#)
- [Filler in Structure \(page 441\)](#)
- [Simple Pointers Declared in Structure \(page 441\)](#)
- [Structure Pointers Declared in Structure \(page 441\)](#)

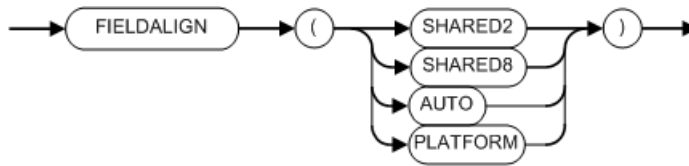
Definition Structure



range



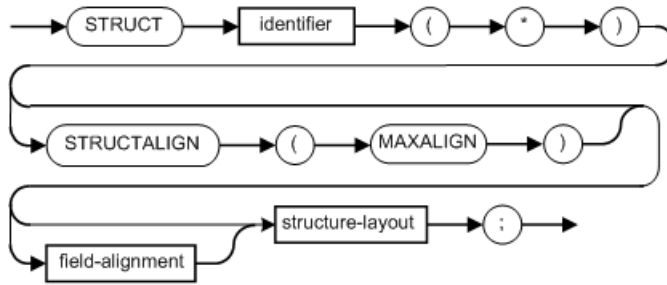
field-alignment



VST992.vsd

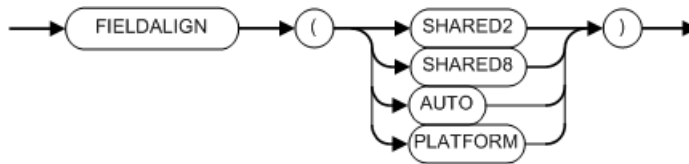
More information: [Declaring Definition Structures \(page 138\)](#)

Template Structure



VST625.vsd

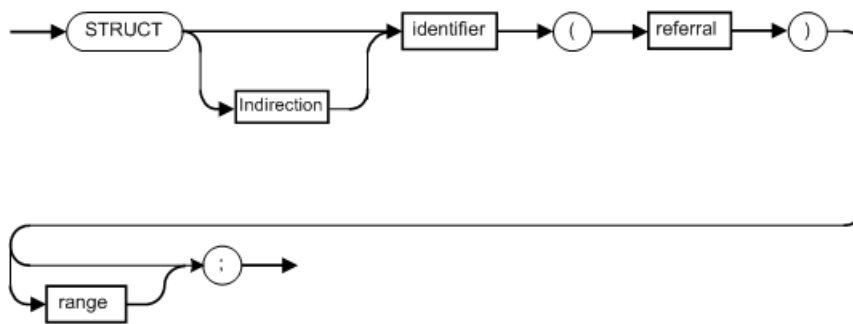
field-alignment



VST992.vsd

More information: [Declaring Template Structures \(page 139\)](#)

Referral Structure



VST025.vsd

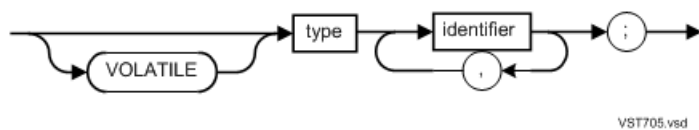
range



VST993.vsd

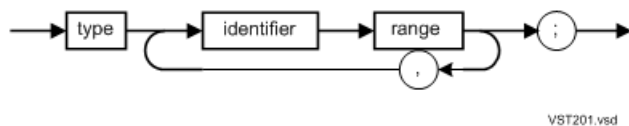
More information: [Declaring Referral Structures \(page 141\)](#)

Simple Variables Declared in Structure

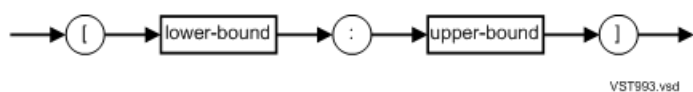


More information: [Declaring Simple Variables in Structures \(page 142\)](#)

Arrays Declared in Structure

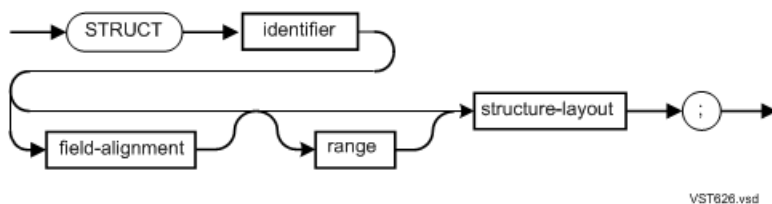


range

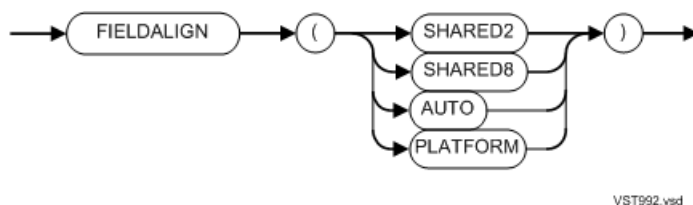


More information: [Declaring Arrays in Structures \(page 143\)](#)

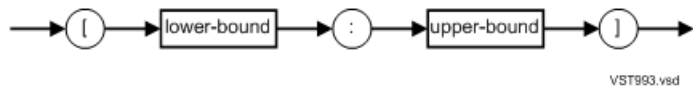
Definition Substructure



field-alignment

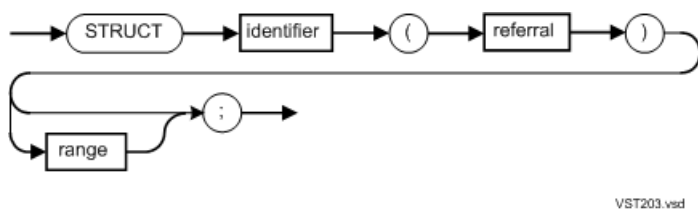


range



More information: [Definition Substructures \(page 144\)](#)

Referral Substructure



range

VST993 vsd

More information: [Referral Substructures \(page 146\)](#)

```

graph LR
    In(( )) --> FILLER
    In --> BIT_FILLER
    FILLER --> CE[constant-expression]
    BIT_FILLER --> CE
    CE --> Semicolon[;]
    Semicolon --> Out(( ))
  
```

VST026.vsd

More information: [Declaring Filler \(page 147\)](#)

```

graph LR
    Start(( )) --> VOLTILE
    VOLTILE --> type[type]
    type --> identifier[identifier]
    identifier --> Indirection[Indirection]
    Indirection --> REFALIGNED[REFALIGNED]
    REFALIGNED --> LP("(")
    LP --> 2((2))
    LP --> 8((8))
    2 --> RP(")")
    8 --> RP
    RP --> SEMI(";")
    SEMI --> End(( ))
    LP --> COMMA(",")
    COMMA --> Indirection
  
```

More information: [Declaring Simple Pointers in Structures \(page 148\)](#)

```

graph LR
    Input(( )) --> VOLATILE
    Input --> STRING
    Input --> INT
    VOLATILE --> Indirection
    STRING --> identifier
    INT --> referral
    Indirection --> identifier
    identifier --> LParen(( ))
    LParen --> referral
    referral --> RParen(( ))
    RParen --> Semicolon((;))
    Semicolon --> Output(( ))
    REFALIGNED --> LParen2(( ))
    LParen2 --> 2((2))
    LParen2 --> 8((8))
    2 --> RParen2(( ))
    8 --> RParen2
    RParen2 --> Comma((,))
    Comma --> Output
  
```

The diagram illustrates the structure of a C program. It shows the flow from input tokens (VOLATILE, STRING, INT) through various components like identifier, referral, and REFALIGNED, leading to the final output (semicolon). The flow starts with an input stream that branches to VOLATILE, STRING, and INT. VOLATILE leads to Indirection, which then leads to identifier. STRING and INT lead to referral. The flow then proceeds through identifier, LParen, referral, RParen, and finally to the semicolon (;). A separate path shows REFALIGNED leading to LParen, which then branches to 2 and 8, leading to RParen, and finally to a comma (,) before the semicolon.

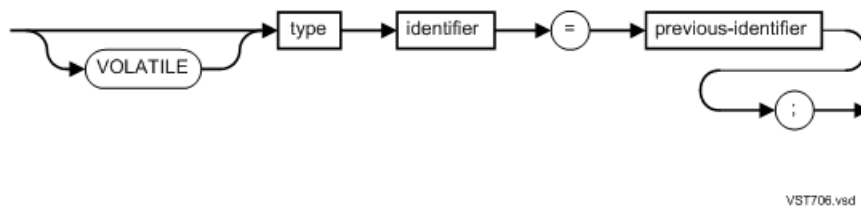
Declarations 441

More information: [Declaring Structure Pointers in Structures \(page 151\)](#)

Redefinition

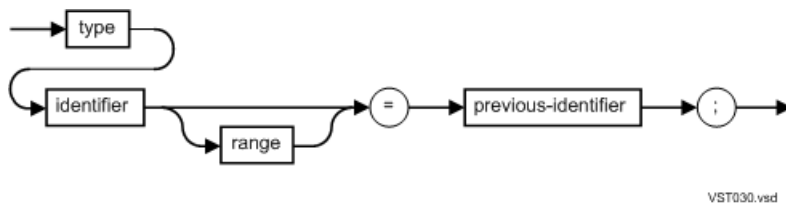
- [Simple Variable \(page 442\)](#)
- [Array \(page 442\)](#)
- [Definition Substructure \(page 442\)](#)
- [Referral Substructure \(page 443\)](#)
- [Simple Pointer \(page 443\)](#)
- [Structure Pointer \(page 444\)](#)

Simple Variable

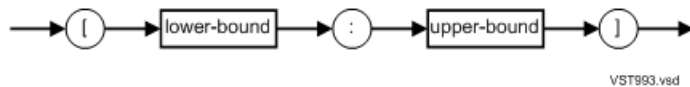


More information: [Simple Variable \(page 153\)](#)

Array

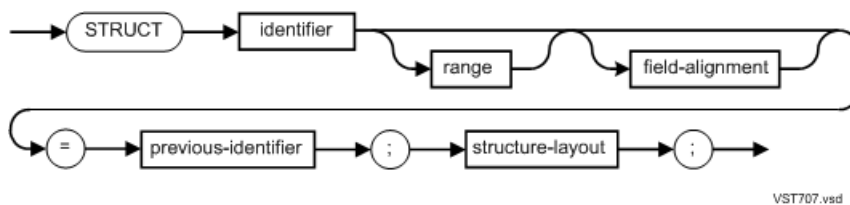


range

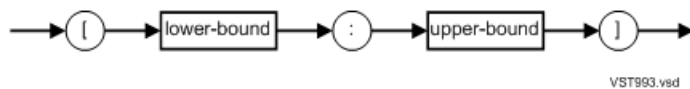


More information: [Array \(page 154\)](#)

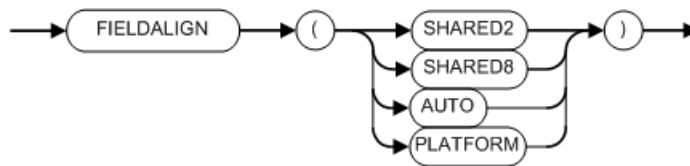
Definition Substructure



range



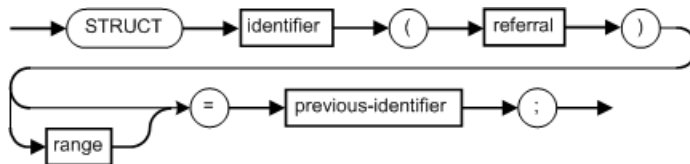
field-alignment



VST992.vsd

More information: [Definition Substructure \(page 155\)](#)

Referral Substructure



VST208.vsd

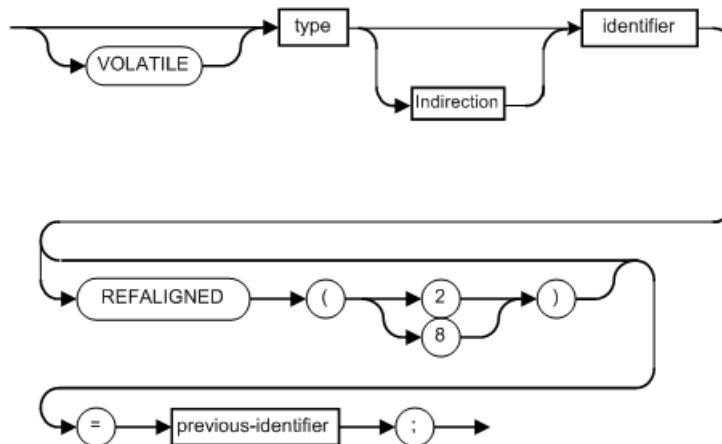
range



VST993.vsd

More information: [Referral Substructure \(page 157\)](#)

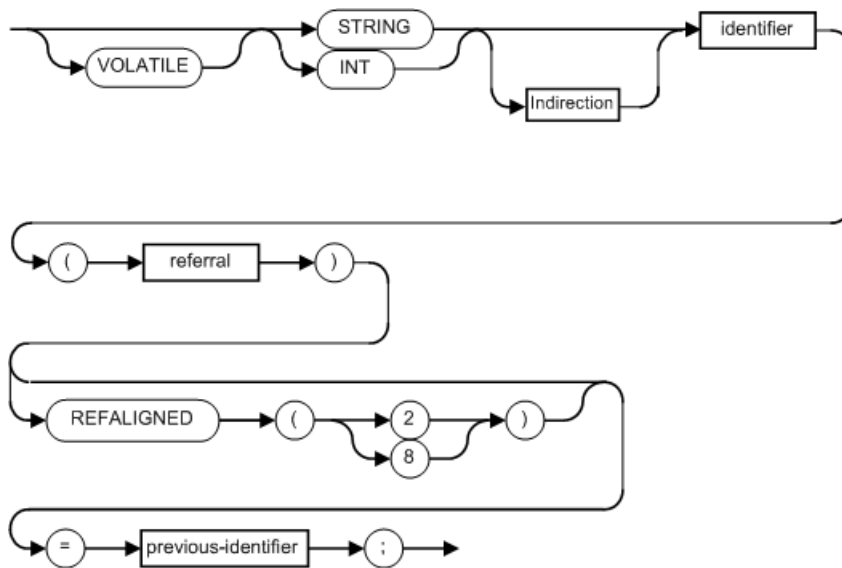
Simple Pointer



VST708.vsd

More information: [Simple Pointer \(page 158\)](#)

Structure Pointer



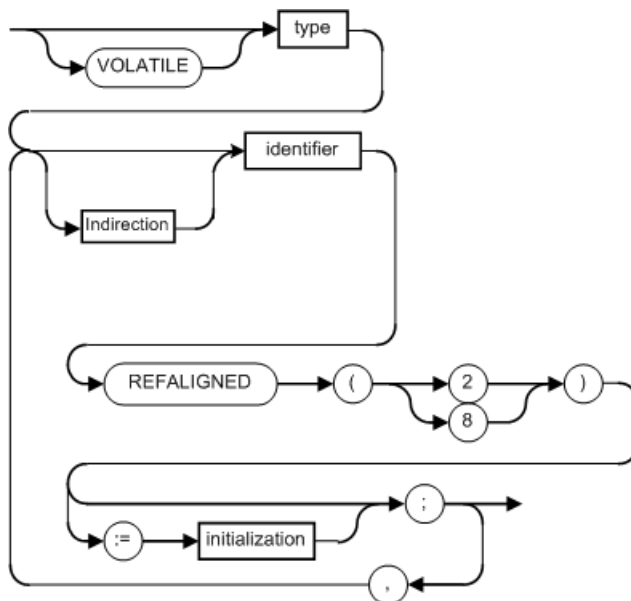
VST709.vsd

More information: [Structure Pointer \(page 159\)](#)

Pointer

- [Simple \(page 444\)](#)
- [Structure \(page 445\)](#)
- [System Global \(page 445\)](#)

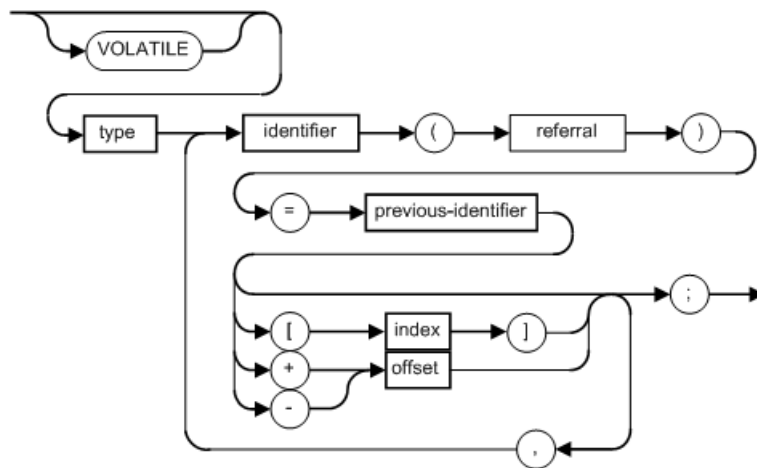
Simple



VST676.vsd

More information: [Declaring Simple Pointers \(page 170\)](#)

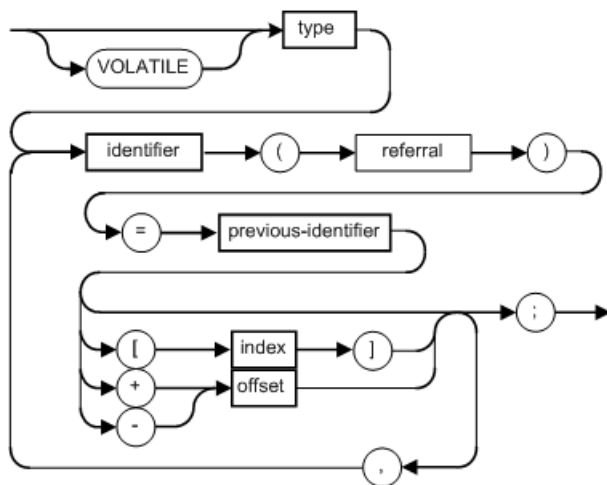
Nonstructure



VST629.vsd

More information: [Declaring Nonstructure Equivalenced Variables \(page 180\)](#)

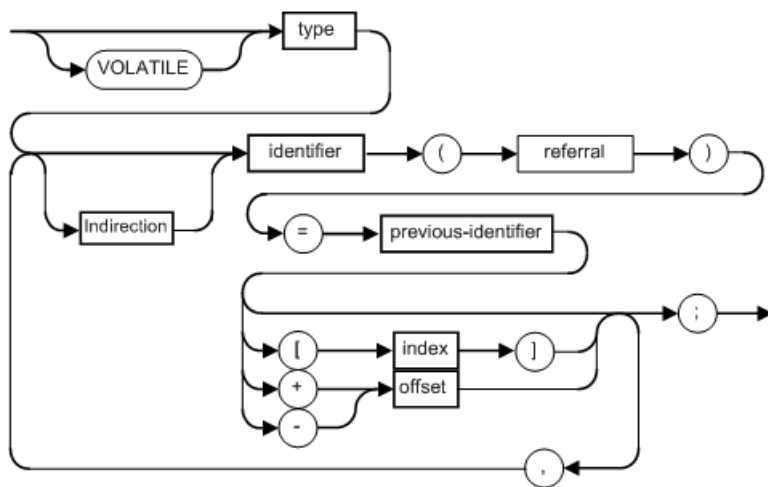
Simple Variable



VST004.vsd

More information: [Equivalenced Simple Variables \(page 182\)](#)

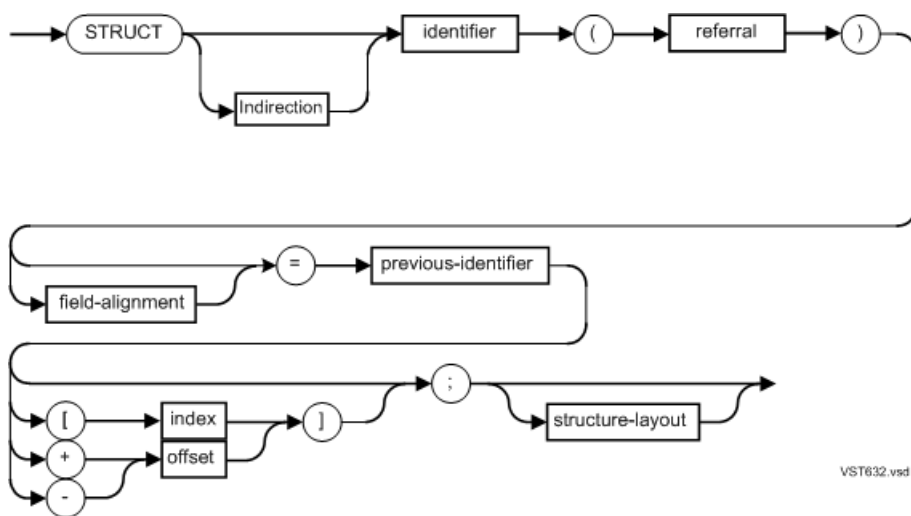
Simple Pointer



VST630.vsd

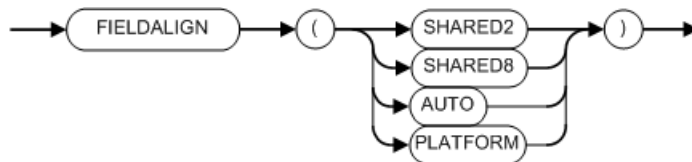
More information: [Equivalenced Simple Pointers \(page 183\)](#)

Definition Structure



VST632.vsd

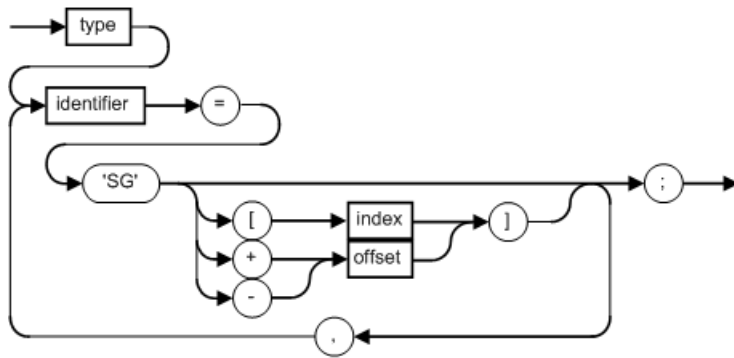
field-alignment



VST992.vsd

More information: [Declaring Equivalenced Definition Structures \(page 188\)](#)

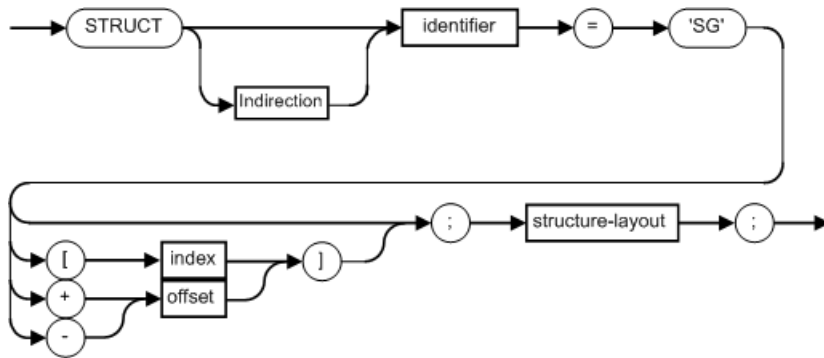
'SG'-Equivalenced Simple Variable



VST068.vsd

More information: [Equivalenced Simple Variables \(page 182\)](#)

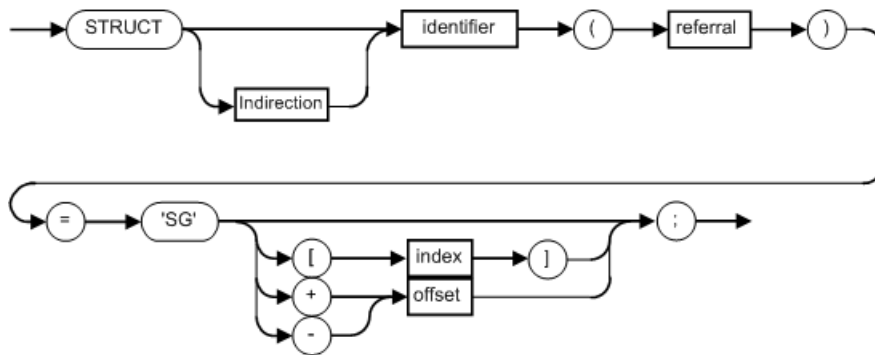
'SG'-Equivalenced Definition Structure



VST710.vsd

More information: [Equivalenced Definition Structure \(page 194\)](#)

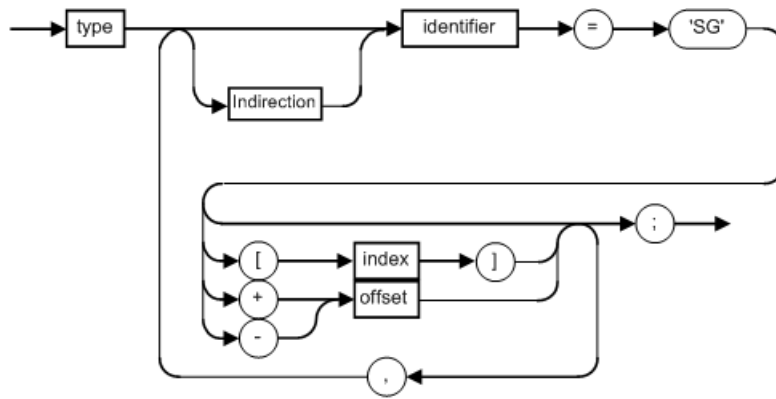
'SG'-Equivalenced Referral Structure



VST703.vsd

More information: [Equivalenced Referral Structure \(page 195\)](#)

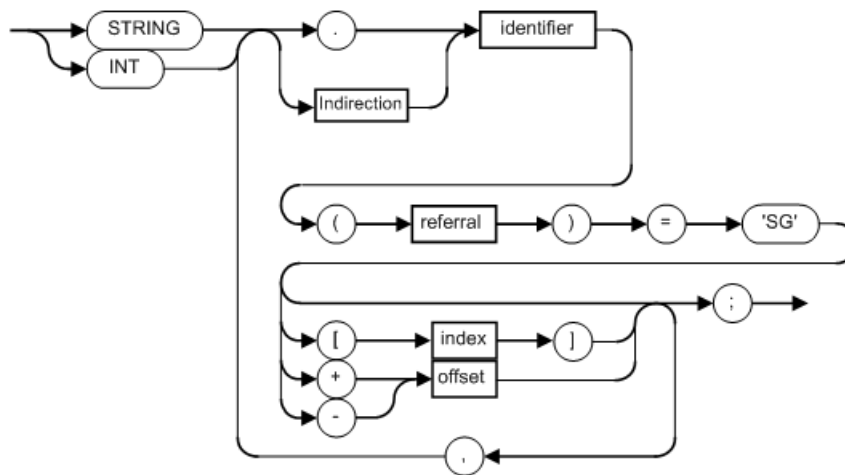
'SG'-Equivalenced Simple Pointer



VST704.vsd

More information: [Equivalenced Simple Pointer \(page 196\)](#)

'SG'-Equivalenced Structure Pointer



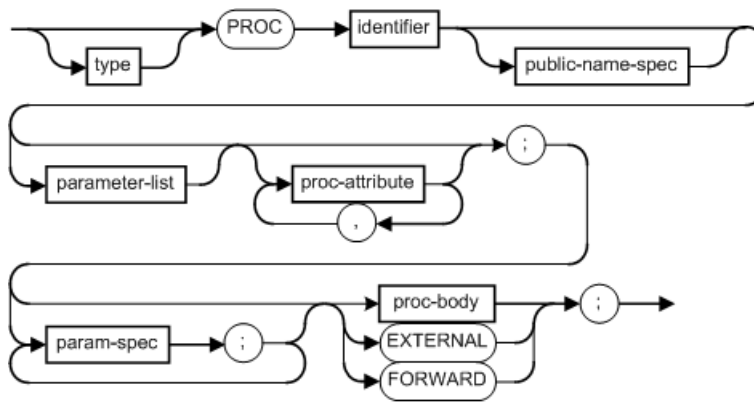
VST711.vsd

More information: [Equivalenced Structure Pointer \(page 197\)](#)

Procedure and Subprocedure

- [Procedure \(page 450\)](#)
- [Subprocedure \(page 452\)](#)
- [Formal Parameters \(page 453\)](#)
- [Entry Point \(page 454\)](#)
- [Label \(page 454\)](#)
- [Procedure Pointer \(page 454\)](#)

Procedure



VST058.vsd

type

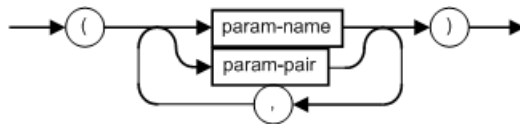
See [Data Types \(page 432\)](#).

public-name-spec



VST209.vsd

parameter-list



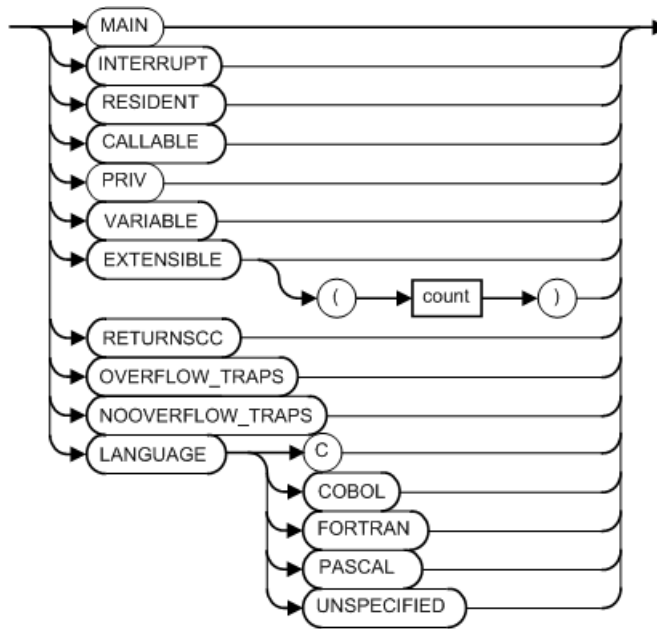
VST210.vsd

param-pair



VST039.vsd

proc-attribute



VST635.vsd

NOTE:

- The EpTAL compiler ignores INTERRUPT.
- Because no FORTRAN or Pascal compilers exist especially for TNS/R or TNS/E architecture, LANGUAGE FORTRAN and LANGUAGE PASCAL have no meaning on TNS/R or TNS/E architecture.

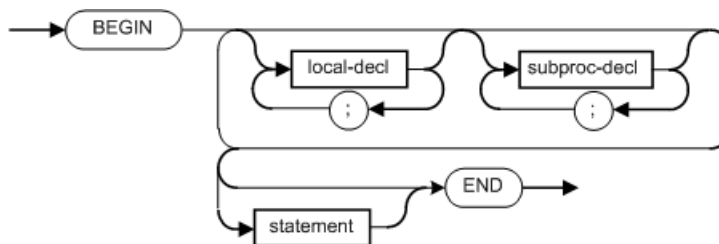
More information: [Procedure Attributes \(page 248\)](#)

param-spec

See [Formal Parameters \(page 453\)](#).

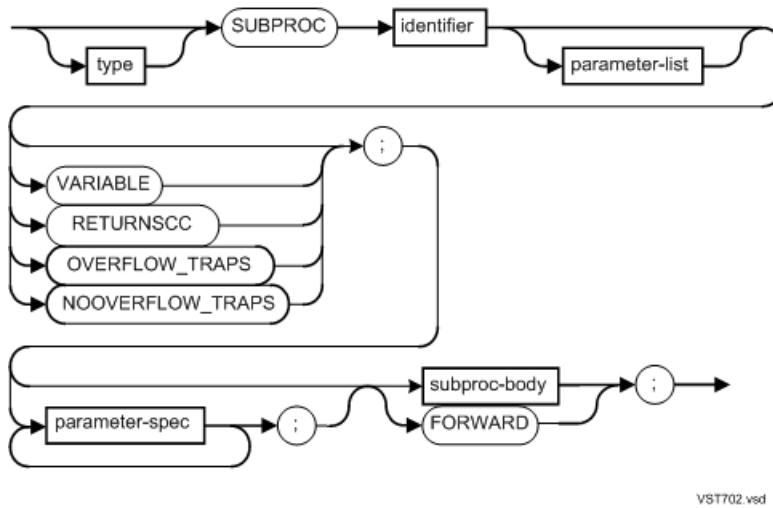
proc-body

More information: [Procedure Declarations \(page 246\)](#)



VST061.vsd

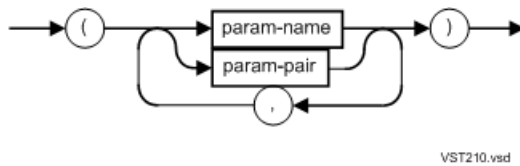
Subprocedure



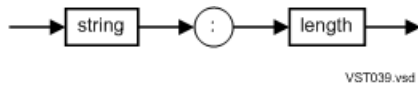
type

See [Data Types \(page 432\)](#).

parameter-list



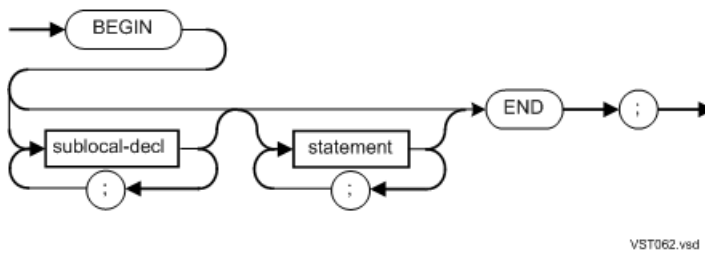
param-pair



param-spec

See [Formal Parameters \(page 453\)](#).

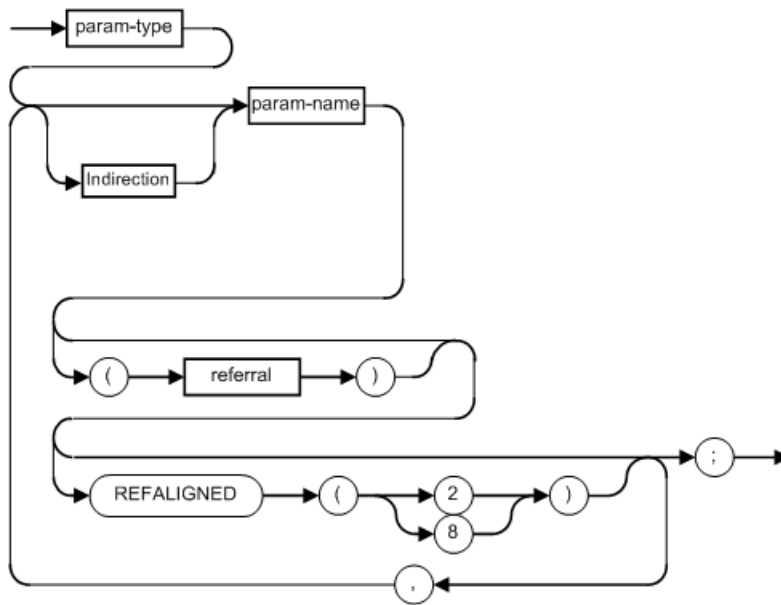
subproc-body



More information: [Subprocedure Body \(page 259\)](#)

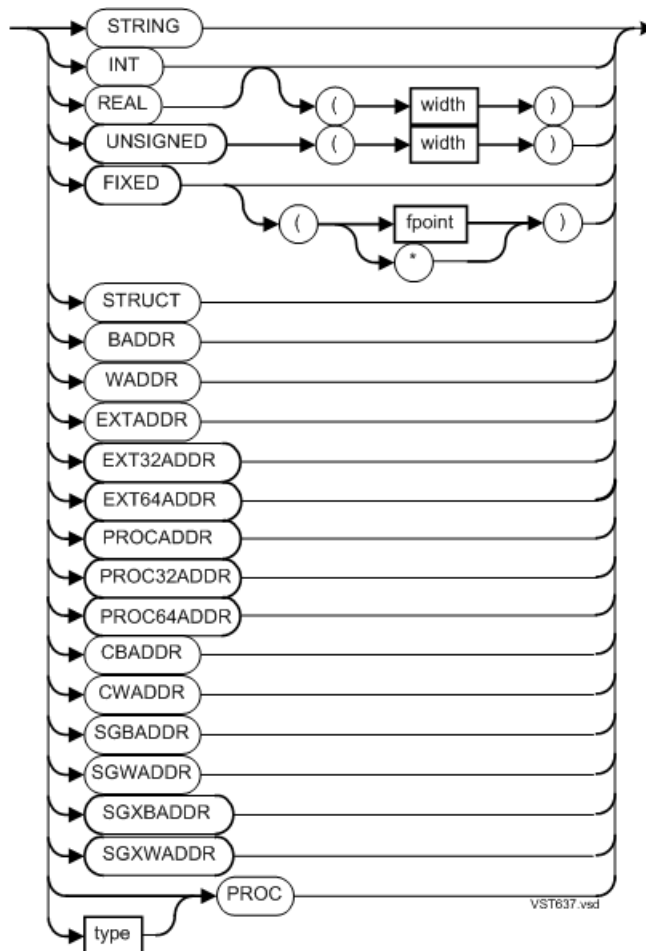
More information: [Subprocedure Declarations \(page 257\)](#)

Formal Parameters



VST636.vsd

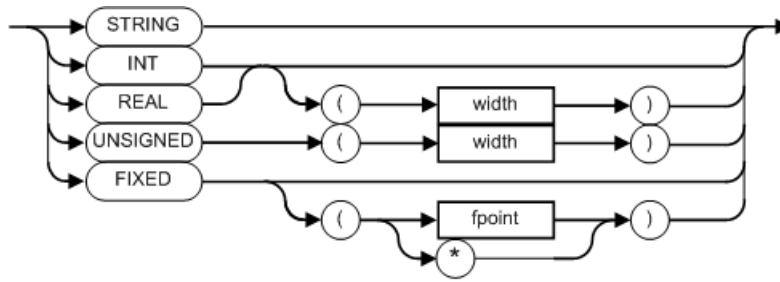
param-type



VST637.vsd

NOTE: The EpTAL compiler does not allow you to assign label or subprocedure addresses to `CBADDR` and `CWADDR` address types.

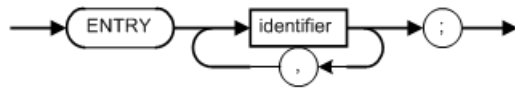
type



VST214.vsd

More information: [Formal Parameter Specification \(page 251\)](#)

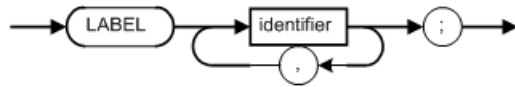
Entry Point



VST195.vsd

More information: [Entry-Point Declarations \(page 260\)](#)

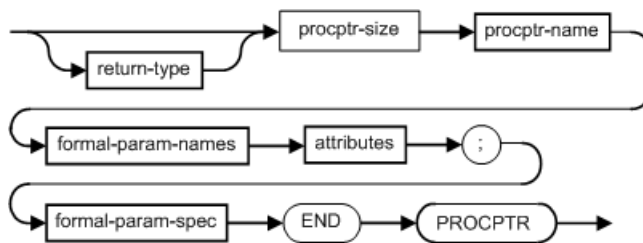
Label



VST196.vsd

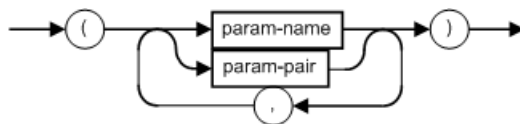
More information: [Labels in Procedures \(page 273\)](#)

Procedure Pointer



VST600.vsd

formal-param-names



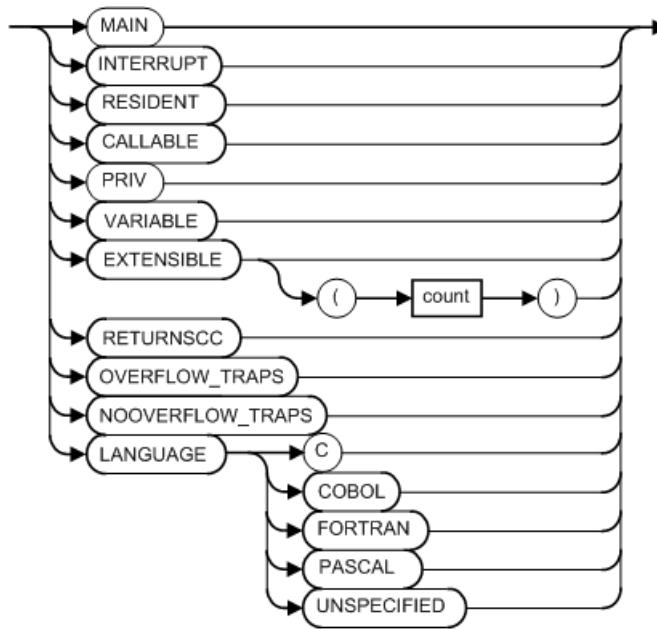
VST210.vsd

param-pair



VST039.vsd

attributes

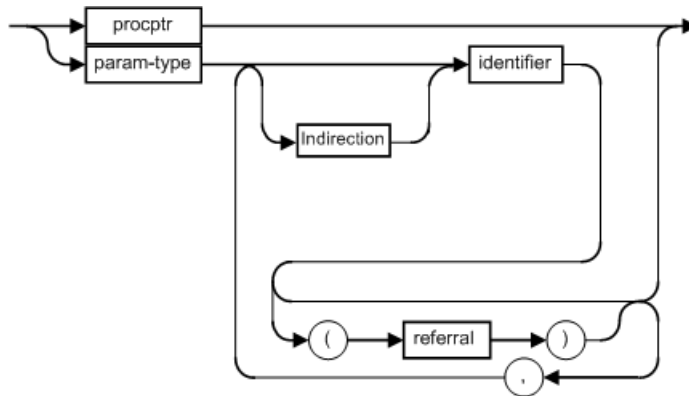


VST635.vsd

NOTE:

- The EpTAL compiler ignores INTERRUPT.
 - Because no FORTRAN or Pascal compilers exist especially for TNS/R or TNS/E architecture, LANGUAGE FORTRAN and LANGUAGE PASCAL have no meaning on TNS/R or TNS/E architecture.
-

formal-param-spec



VST712.vsd

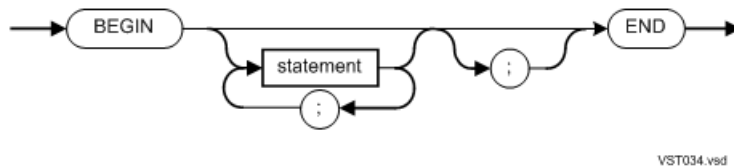
More information: [Procedure Pointers \(page 263\)](#)

Statements

- [Compound \(page 456\)](#)
- [ASSERT \(page 456\)](#)
- [Assignment \(page 456\)](#)
- [Bit Deposit Assignment \(page 456\)](#)
- [CALL \(page 457\)](#)

- Labeled CASE (page 457)
- Unlabeled CASE (page 457)
- DO-UNTIL (page 458)
- DROP (page 458)
- FOR (page 458)
- GOTO (page 458)
- IF (page 458)
- Move (page 459)
- RETURN (page 459)
- SCAN and RSCAN (page 459)
- USE (page 459)
- WHILE (page 460)

Compound



More information: [Compound Statements \(page 200\)](#)

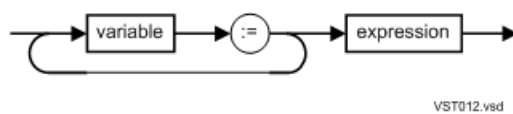
ASSERT



More information: [ASSERT \(page 200\)](#)

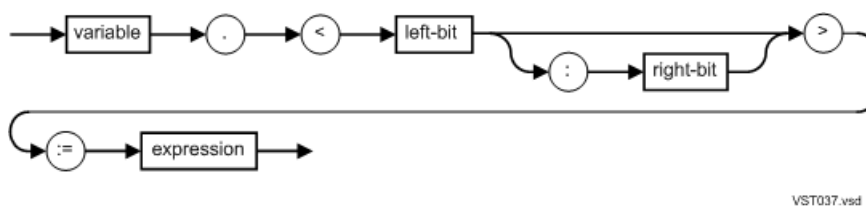
Assignment

The assignment statement assigns a value to a previously declared variable.



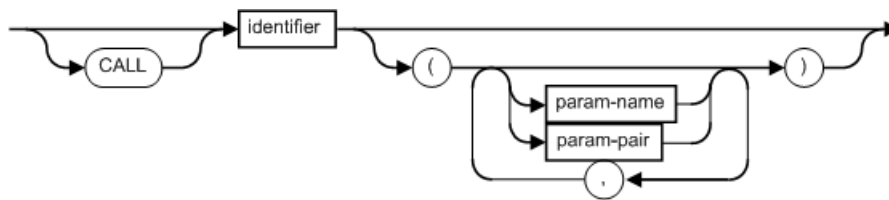
More information: [Assignment \(page 201\)](#)

Bit Deposit Assignment



More information: [Bit-Deposit Assignment \(page 204\)](#)

CALL



VST038.vsd

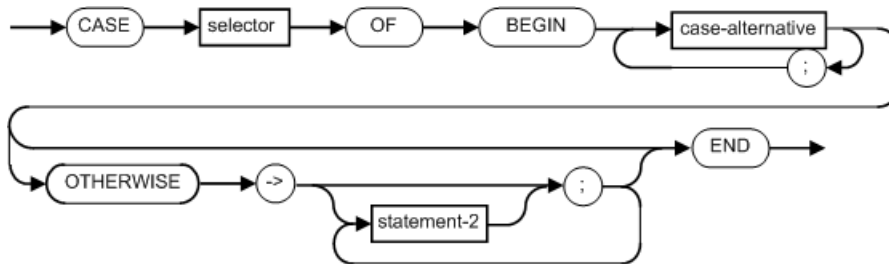
param-pair



VST039.vsd

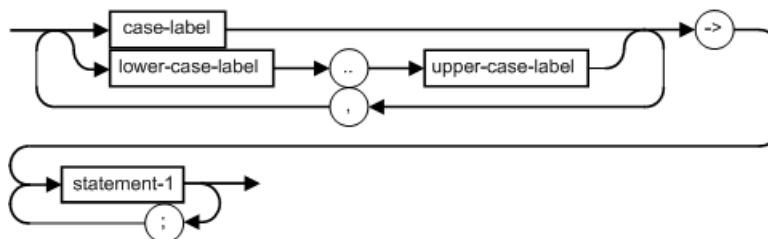
More information: [CALL \(page 205\)](#)

Labeled CASE



VST041.vsd

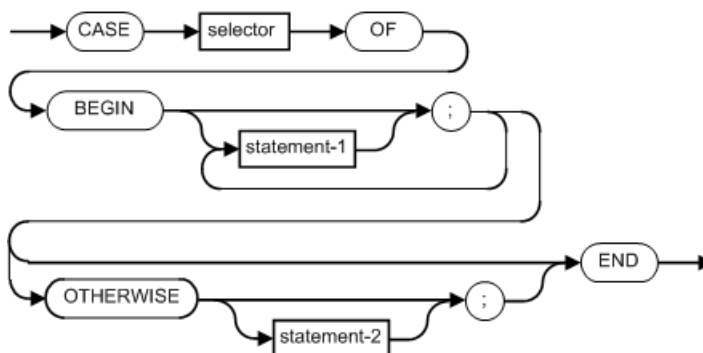
case-alternative



VST042.vsd

More information: [Labeled CASE \(page 207\)](#)

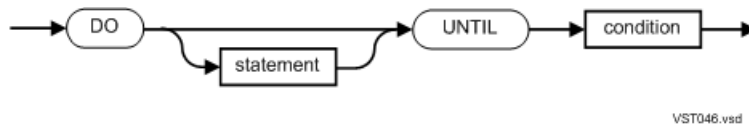
Unlabeled CASE



VST040.vsd

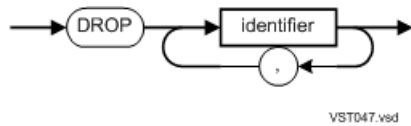
More information: [Unlabeled CASE \(page 209\)](#)

DO-UNTIL



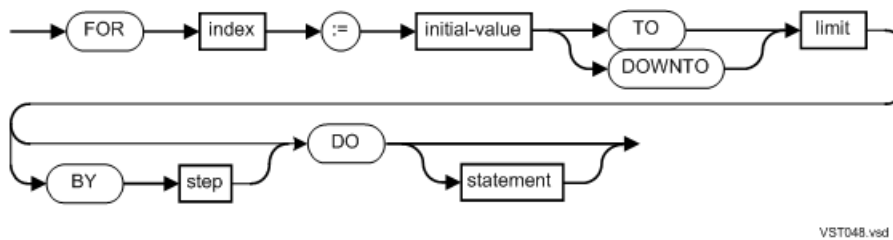
More information: [DO-UNTIL \(page 210\)](#)

DROP



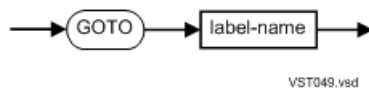
More information: [DROP \(page 212\)](#)

FOR



More information: [FOR \(page 212\)](#)

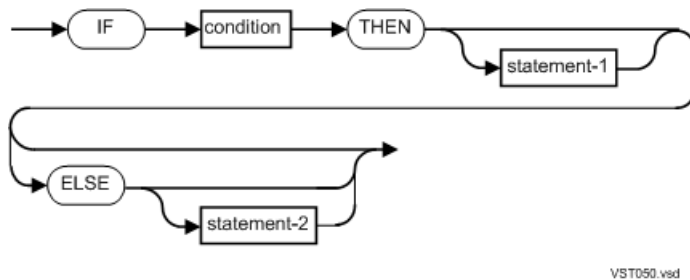
GOTO



NOTE: Nonlocal GOTO statements are are inefficient and not recommended.

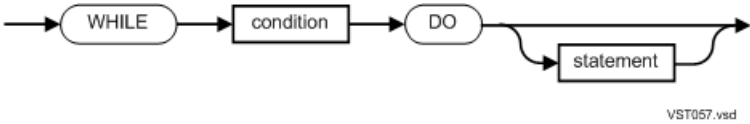
More information: [GOTO \(page 215\)](#)

IF



More information: [IF \(page 217\)](#)

WHILE



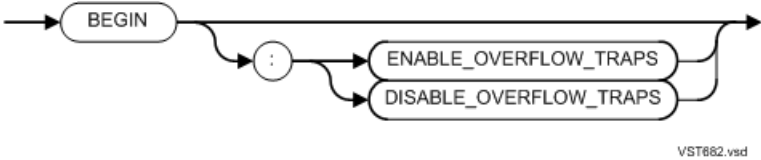
More information: [WHILE \(page 232\)](#)

Overflow Traps

OVERFLOW_TRAPS Directive

See [OVERFLOW_TRAPS \(page 508\)](#).

[EN | DIS]ABLE_OVERFLOW_TRAPS Block Attribute



More information: [\[EN | DIS\]ABLE_OVERFLOW_TRAPS Block Attribute \(page 235\)](#)

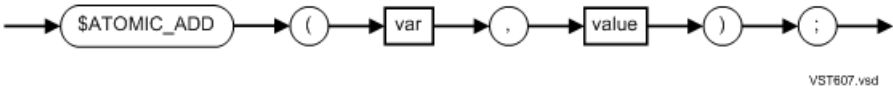
Built-in Routines

- [Atomic \(page 460\)](#)
- [Nonatomic \(page 462\)](#)

Atomic

- [\\$ATOMIC_ADD \(page 460\)](#)
- [\\$ATOMIC_AND \(page 461\)](#)
- [\\$ATOMIC_DEP \(page 461\)](#)
- [\\$ATOMIC_GET \(page 461\)](#)
- [\\$ATOMIC_OR \(page 461\)](#)
- [\\$ATOMIC_PUT \(page 462\)](#)

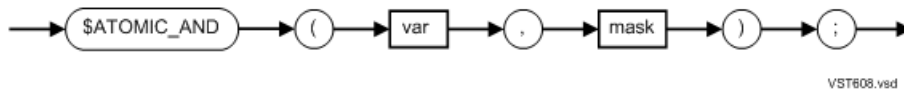
\$ATOMIC_ADD



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	Yes, if traps are disabled
Sets \$OVERFLOW	Yes, if traps are disabled; otherwise, traps on overflow

More information: [\\$ATOMIC_ADD \(page 276\)](#)

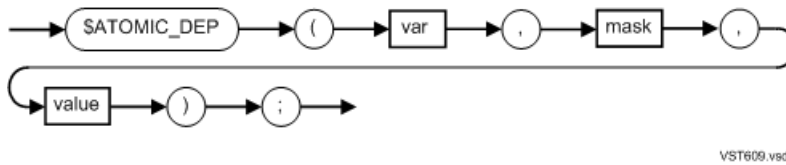
\$ATOMIC_AND



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ATOMIC_AND \(page 277\)](#)

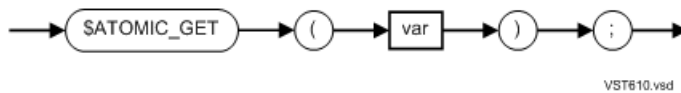
\$ATOMIC_DEP



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ATOMIC_DEP \(page 278\)](#)

\$ATOMIC_GET



Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ATOMIC_GET \(page 279\)](#)

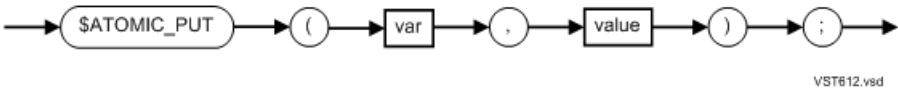
\$ATOMIC_OR



Sets condition code	Yes (according the final value of <i>var</i>)
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ATOMIC_OR \(page 280\)](#)

\$ATOMIC_PUT



Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: `$ATOMIC_PUT` (page 280)

Nonatomic

- `$ABS` (page 464)
- `$ALPHA` (page 464)
- `$ASCIITOFIXED` (page 465)
- `$AXADR` (page 465)
- `$BADDR_TO_EXTADDR` (page 465)
- `$BADDR_TO_WADDR` (page 466)
- `$BITLENGTH` (page 466)
- `$BITOFFSET` (page 466)
- `$CARRY` (page 467)
- `$CHECKSUM` (page 467)
- `$COMP` (page 467)
- `$COUNTDUPS` (page 468)
- `$DBL` (page 468)
- `$DBLL` (page 468)
- `$DBLR` (page 469)
- `$DFIX` (page 469)
- `$EFLT` (page 469)
- `$EFLTR` (page 470)
- `$EXCHANGE` (page 470)
- `$EXECUTEIO` (page 470)
- `$EXTADDR_TO_BADDR` (page 471)
- `$EXTADDR_TO_WADDR` (page 471)
- `$EXT64ADDR_TO_EXTADDR` (page 471)
- `$EXT64ADDR_TO_EXT32ADDR` (page 471)
- `$EXT64ADDR_TO_EXT32ADDR_OV` (page 472)
- `$EXTADDR_TO_EXT64ADDR` (page 472)
- `$FILL8`, `$FILL16`, and `$FILL32` (page 473)
- `$FIX` (page 473)
- `$FIXD` (page 473)
- `$FIXED0_TO_EXT64ADDR` (page 474)

- [\\$FIXEDTOASCII \(page 474\)](#)
- [\\$FIXEDTOASCIIRESIDUE \(page 474\)](#)
- [\\$FIXI \(page 475\)](#)
- [\\$FIXL \(page 475\)](#)
- [\\$FIXR \(page 475\)](#)
- [\\$FLT \(page 475\)](#)
- [\\$FLTR \(page 476\)](#)
- [\\$FREEZE \(page 476\)](#)
- [\\$HALT \(page 476\)](#)
- [\\$HIGH \(page 477\)](#)
- [\\$IFIX \(page 477\)](#)
- [\\$INT \(page 477\)](#)
- [\\$INT_OV \(page 478\)](#)
- [\\$INTERROGATEHIO \(page 478\)](#)
- [\\$INTERROGATEIO \(page 478\)](#)
- [\\$INTR \(page 479\)](#)
- [\\$IS_32BIT_ADDR \(page 479\)](#)
- [\\$LEN \(page 480\)](#)
- [\\$LFI \(page 480\)](#)
- [\\$LMAX \(page 480\)](#)
- [\\$LMIN \(page 480\)](#)
- [\\$LOCATESPTHDR \(page 481\)](#)
- [\\$LOCKPAGE \(page 481\)](#)
- [\\$MAX \(page 481\)](#)
- [\\$MIN \(page 482\)](#)
- [\\$MOVEANDCXSUMBYTES \(page 482\)](#)
- [\\$MOVENONDUP \(page 482\)](#)
- [\\$NUMERIC \(page 483\)](#)
- [\\$OCCURS \(page 483\)](#)
- [\\$OFFSET \(page 483\)](#)
- [\\$OPTIONAL \(page 484\)](#)
- [\\$OVERFLOW \(page 484\)](#)
- [\\$PARAM \(page 484\)](#)
- [\\$POINT \(page 485\)](#)
- [\\$PROCADDR \(page 485\)](#)
- [\\$PROC32ADDR \(page 485\)](#)
- [\\$PROC64ADDR \(page 486\)](#)
- [\\$READBASELIMIT \(page 486\)](#)
- [\\$READCLOCK \(page 486\)](#)
- [\\$READSPT \(page 486\)](#)
- [\\$READTIME \(page 487\)](#)

- [\\$SCALE \(page 487\)](#)
- [\\$SGBADDR_TO_EXTADDR \(page 487\)](#)
- [pTAL Privileged Routines \(page 281\)](#)
- [\\$SGWADDR_TO_EXTADDR \(page 488\)](#)
- [\\$SGWADDR_TO_SGBADDR \(page 488\)](#)
- [\\$SPECIAL \(page 489\)](#)
- [\\$STACK_ALLOCATE \(page 489\)](#)
- [\\$TRIGGER \(page 489\)](#)
- [\\$TYPE \(page 490\)](#)
- [\\$UDBL \(page 490\)](#)
- [\\$UDIVREM16 \(page 490\)](#)
- [\\$UDIVREM32 \(page 491\)](#)
- [\\$UNLOCKPAGE \(page 491\)](#)
- [\\$WADDR_TO_BADDR \(page 492\)](#)
- [\\$WADDR_TO_EXTADDR \(page 492\)](#)
- [\\$WRITEPTE \(page 492\)](#)
- [\\$XADR \(page 493\)](#)
- [\\$XADR32 \(page 493\)](#)
- [\\$XADR64 \(page 493\)](#)

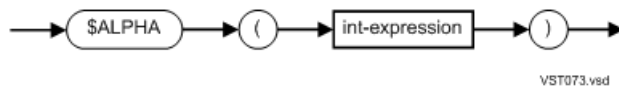
\$ABS



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ABS \(page 291\)](#)

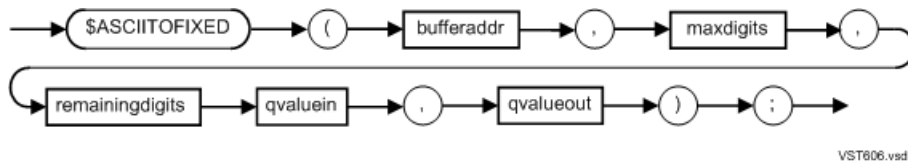
\$ALPHA



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$ALPHA \(page 291\)](#)

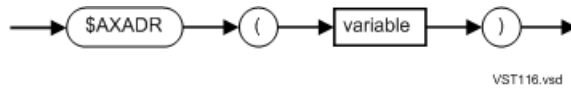
\$ASCIITOFIXED



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

More information: [\\$ASCIITOFIXED \(page 292\)](#)

\$AXADR

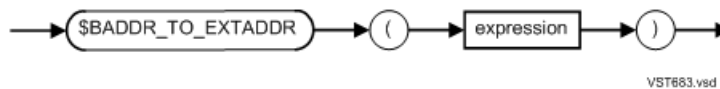


NOTE: The EpTAL compiler does not support this routine. (The EpTAL compiler does allow \$AXADR as a DEFINE name.)

pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$AXADR \(page 293\)](#)

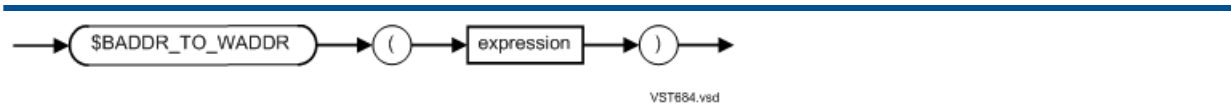
\$BADDR_TO_EXTADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$BADDR_TO_EXTADDR \(page 294\)](#)

\$BADDR_TO_WADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$BADDR_TO_WADDR \(page 294\)](#)

\$BITLENGTH



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$BITLENGTH \(page 295\)](#)

\$BITOFFSET



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$BITOFFSET \(page 296\)](#)

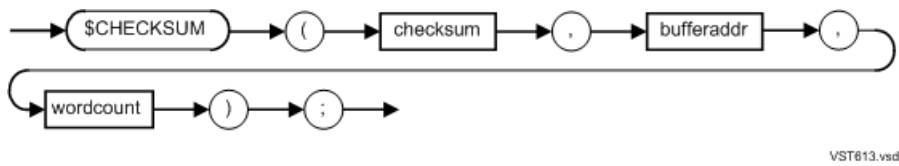
\$CARRY



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$CARRY \(page 297\)](#)

\$CHECKSUM



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$CHECKSUM \(page 297\)](#)

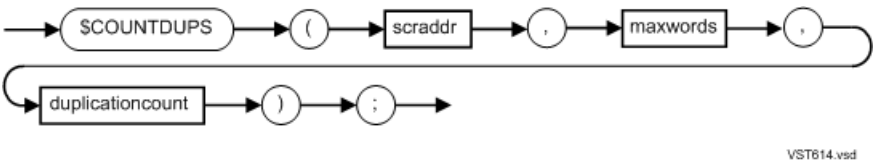
\$COMP



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$COMP \(page 298\)](#)

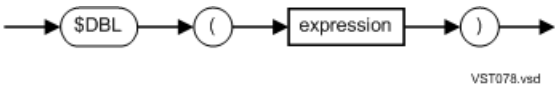
\$COUNTDUPS



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$COUNTDUPS \(page 299\)](#)

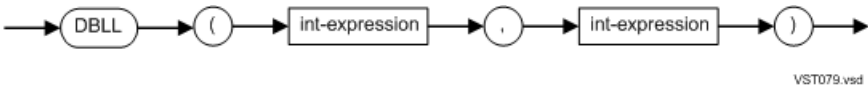
\$DBL



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$DBL \(page 300\)](#)

\$DBLL



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$DBLL \(page 301\)](#)

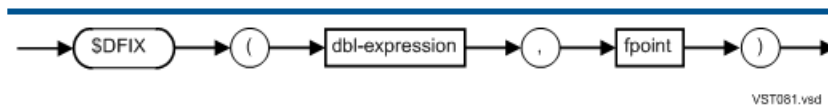
\$DBLR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$DBLR \(page 301\)](#)

\$DFIX



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$DFIX \(page 302\)](#)

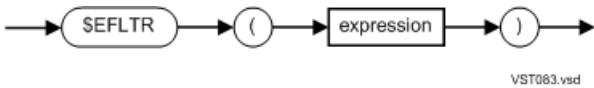
\$EFLT



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EFLT \(page 302\)](#)

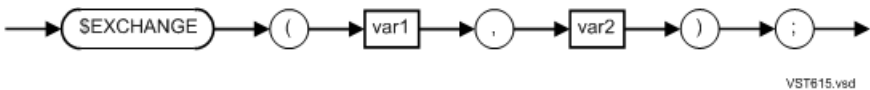
\$EFLTR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EFLTR \(page 303\)](#)

\$EXCHANGE

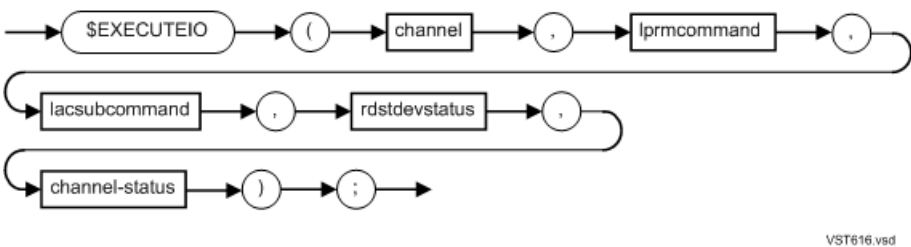


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXCHANGE \(page 303\)](#)

\$EXECUTEIO

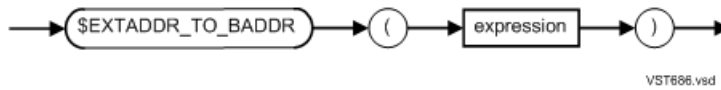
NOTE: The EpTAL compiler does not support this routine.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXECUTEIO \(page 304\)](#)

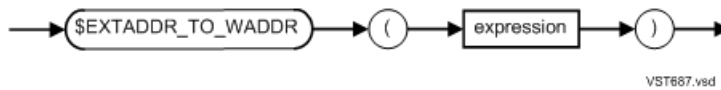
\$EXTADDR_TO_BADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXTADDR_TO_BADDR \(page 305\)](#)

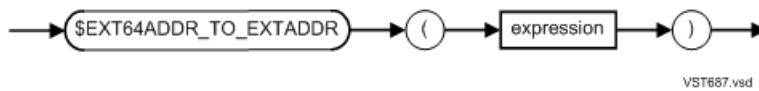
\$EXTADDR_TO_WADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXTADDR_TO_WADDR \(page 306\)](#)

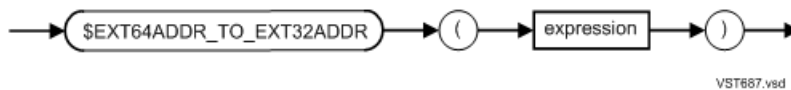
\$EXT64ADDR_TO_EXTADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXT64ADDR_TO_EXTADDR \(page 306\)](#)

\$EXT64ADDR_TO_EXT32ADDR



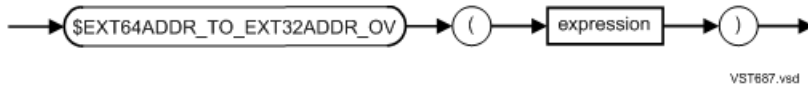
pTAL privileged procedure	No
Can be executed only by privileged procedures	No

Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXT64ADDR_TO_EXT32ADDR](#) (page 307)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXT64ADDR_TO_EXT32ADDR_OV

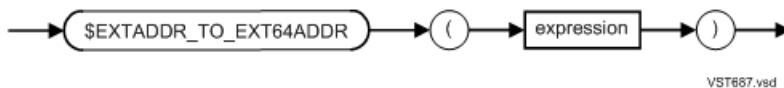


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXT64ADDR_TO_EXT32ADDR_OV](#) (page 307)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$EXTADDR_TO_EXT64ADDR

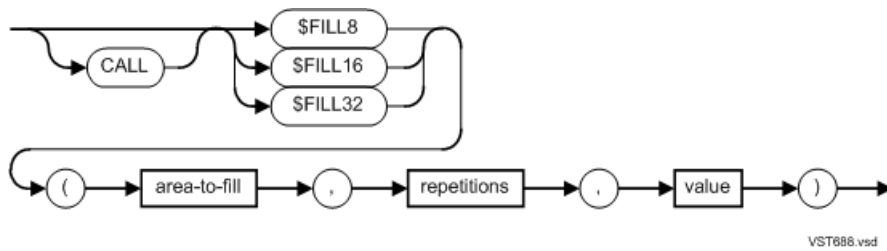


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$EXTADDR_TO_EXT64ADDR](#) (page 308)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$FILL8, \$FILL16, and \$FILL32



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FILL8, \\$FILL16, and \\$FILL32 \(page 308\)](#)

\$FIX



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIX \(page 309\)](#)

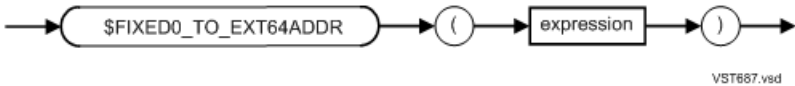
\$FIXD



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIXD \(page 309\)](#)

\$FIXED0_TO_EXT64ADDR

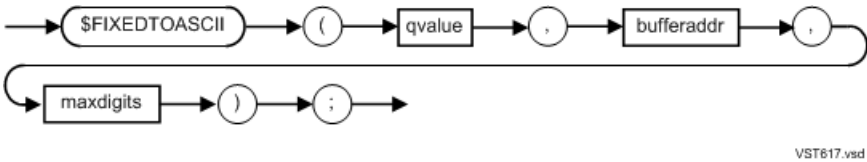


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIXED0_TO_EXT64ADDR \(page 310\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

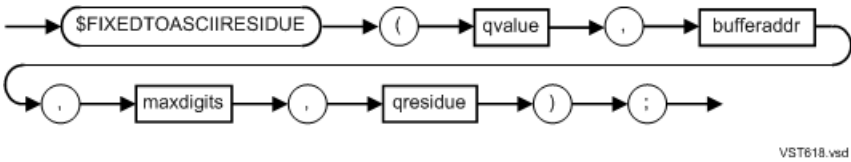
\$FIXEDTOASCII



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

More information: [\\$FIXEDTOASCII \(page 310\)](#)

\$FIXEDTOASCIIRESIDUE



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

More information: [\\$FIXEDTOASCIIRESIDUE \(page 311\)](#)

\$FIXI



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIXI \(page 312\)](#)

\$FIXL



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIXL \(page 312\)](#)

\$FIXR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FIXR \(page 313\)](#)

\$FLT

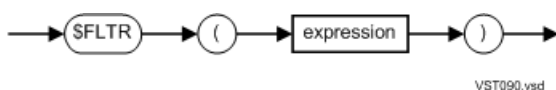


pTAL privileged procedure	No
Can be executed only by privileged procedures	No

Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FLT \(page 314\)](#)

\$FLTR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FLTR \(page 314\)](#)

\$FREEZE

NOTE:

- The EpTAL compiler does not support this procedure. Use [\\$TRIGGER \(page 345\)](#) instead. (The EpTAL compiler does allow \$FREEZE as a DEFINE name.)
- Execution does not return from this call.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$FREEZE \(page 315\)](#)

\$HALT

NOTE:

- The EpTAL compiler does not support this procedure. Use [\\$TRIGGER \(page 345\)](#) instead. (The EpTAL compiler does allow \$HALT as a DEFINE name.)
- Execution does not return from this call.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$HALT \(page 315\)](#)

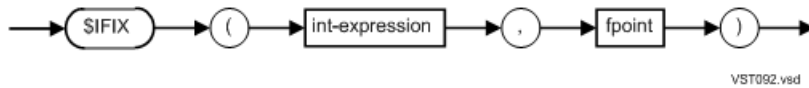
\$HIGH



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$HALT \(page 315\)](#)

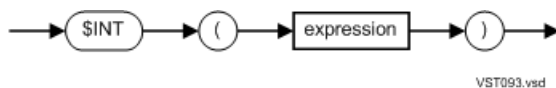
\$IFIX



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$IFIX \(page 316\)](#)

\$INT



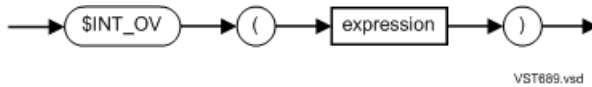
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$INT \(page 317\)](#)

\$INT_OV

NOTE: \$INT_OV is supported in the D40 and later RVUs.

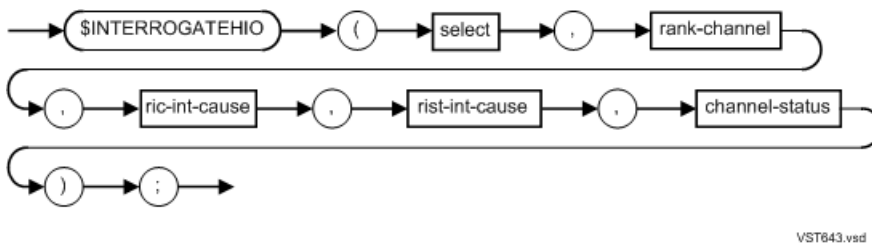


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes

More information: [\\$INT_OV \(page 318\)](#)

\$INTERROGATEHIO

NOTE: The EpTAL compiler does not support this routine.

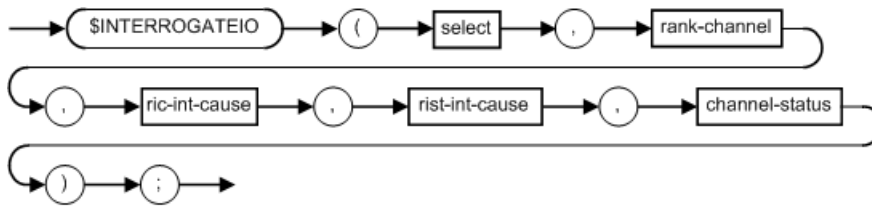


pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$INTERROGATEHIO \(page 318\)](#)

\$INTERROGATEIO

NOTE: The EpTAL compiler does not support this routine.



VST644.vsd

pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$INTERROGATEIO \(page 320\)](#)

\$INTR

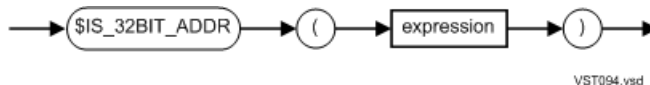


VST094.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$INTR \(page 321\)](#)

\$IS_32BIT_ADDR



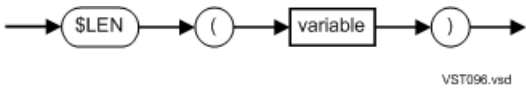
VST094.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$IS_32BIT_ADDR \(page 321\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

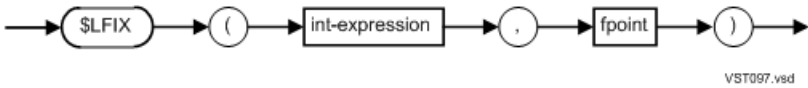
\$LEN



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$LEN \(page 322\)](#)

\$LFIX



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$LFIX \(page 323\)](#)

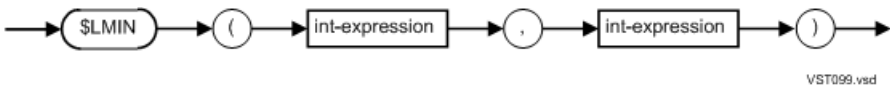
\$LMAX



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$LMAX \(page 323\)](#)

\$LMIN



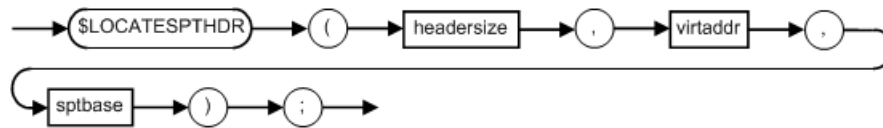
pTAL privileged procedure	No
Can be executed only by privileged procedures	No

Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$LMIN \(page 324\)](#)

\$LOCATESPTHDR

NOTE: The EpTAL compiler does not support this routine.



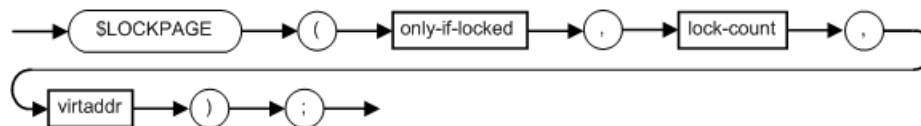
VST645.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

More information: [\\$LOCATESPTHDR \(page 324\)](#)

\$LOCKPAGE

NOTE: The EpTAL compiler does not support this routine.

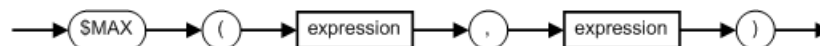


VST646.vsd

pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

More information: [\\$LOCKPAGE \(page 325\)](#)

\$MAX



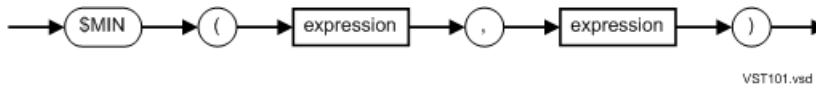
VST100.vsd

pTAL privileged procedure	No
Can be executed only by privileged procedures	No

Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$MAX \(page 326\)](#)

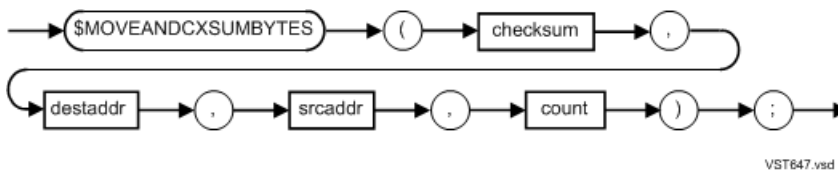
\$MIN



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$LMIN \(page 324\)](#)

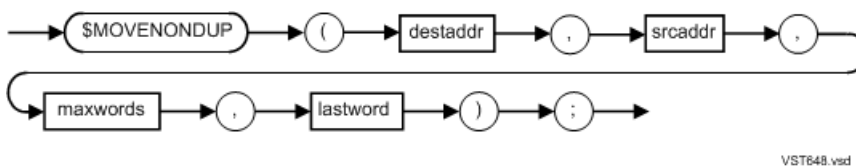
\$MOVEANDCXSUMBYTES



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$MOVEANDCXSUMBYTES \(page 327\)](#)

\$MOVENONDUP



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$MOVENONDUP \(page 328\)](#)

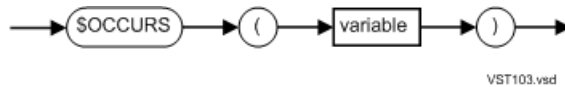
\$NUMERIC



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$NUMERIC \(page 329\)](#)

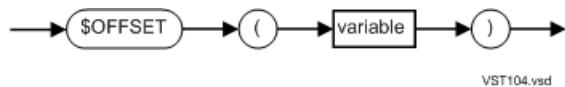
\$OCCURS



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$OCCURS \(page 330\)](#)

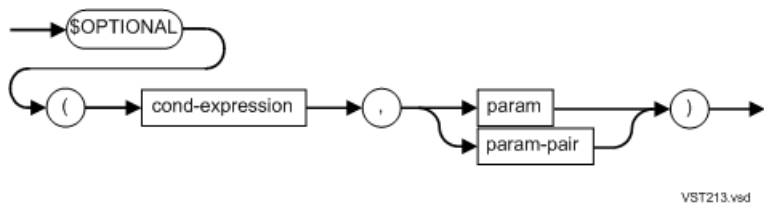
\$OFFSET



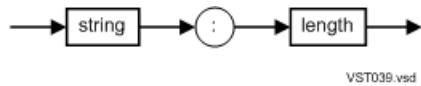
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$OFFSET \(page 332\)](#)

\$OPTIONAL



param-pair



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$OPTIONAL \(page 333\)](#)

\$OVERFLOW



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$OVERFLOW \(page 335\)](#)

\$PARAM



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$PARAM \(page 336\)](#)

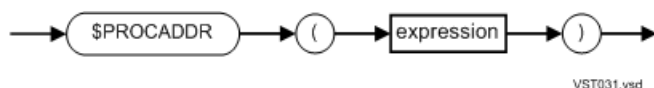
\$POINT



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$POINT \(page 336\)](#)

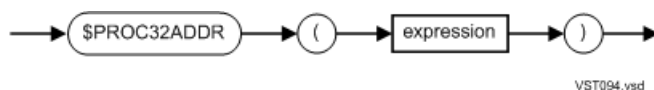
\$PROCADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$PROCADDR \(page 337\)](#)

\$PROC32ADDR

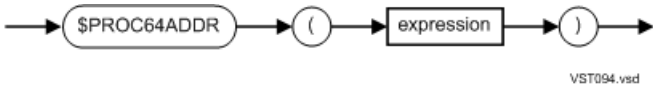


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$PROC32ADDR \(page 337\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$PROC64ADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$PROC64ADDR \(page 338\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$READBASELIMIT

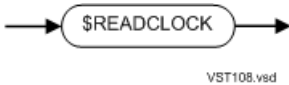
NOTE: The EpTAL compiler does not support this procedure.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$READBASELIMIT \(page 338\)](#)

\$READCLOCK



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$READCLOCK \(page 339\)](#)

\$READSPT

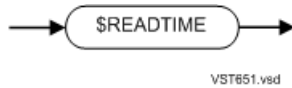
NOTE: The EpTAL compiler does not support this routine.



pTAL privileged procedure	No
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

More information: [\\$READSPT \(page 339\)](#)

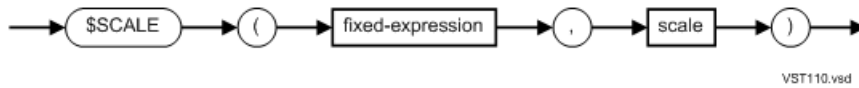
\$READTIME



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$READTIME \(page 340\)](#)

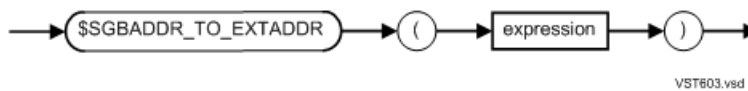
\$SCALE



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SCALE \(page 340\)](#)

\$SGBADDR_TO_EXTADDR

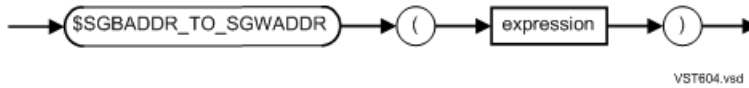


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No

Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SGBADDR_TO_EXTADDR \(page 341\)](#)

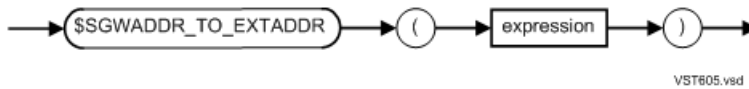
\$SGBADDR_TO_SGWADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SGBADDR_TO_SGWADDR \(page 342\)](#)

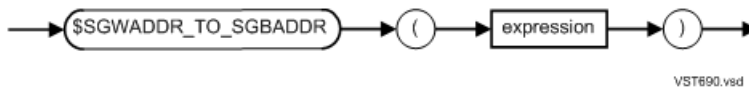
\$SGWADDR_TO_EXTADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SGWADDR_TO_EXTADDR \(page 342\)](#)

\$SGWADDR_TO_SGBADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SGWADDR_TO_SGBADDR \(page 343\)](#)

\$SPECIAL

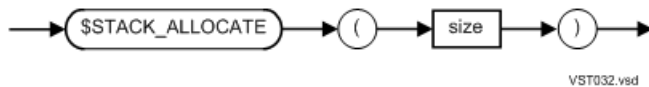


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$SPECIAL \(page 343\)](#)

\$STACK_ALLOCATE

NOTE: The pTAL and EpTAL compilers behave differently.



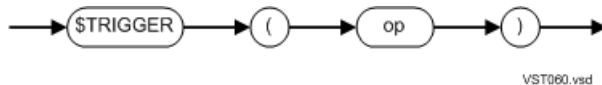
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$STACK_ALLOCATE \(page 344\)](#)

\$TRIGGER

NOTE:

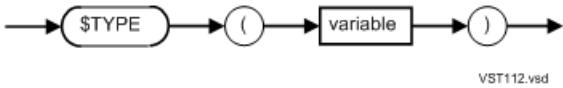
- The TAL and pTAL compilers does not support this routine.
- Execution does not return from this call.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$TRIGGER \(page 345\)](#)

\$TYPE



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$TYPE \(page 345\)](#)

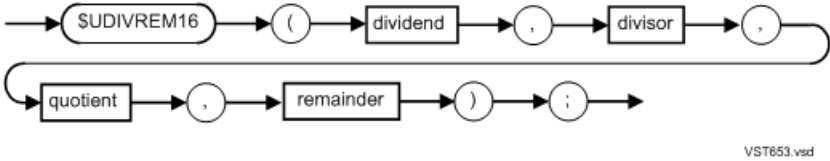
\$UDBL



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$UDBL \(page 346\)](#)

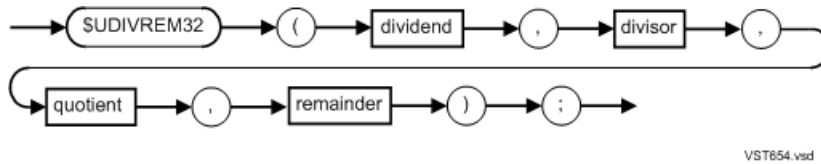
\$UDIVREM16



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes, if the divisor is 0 or the quotient is too large

More information: [\\$UDIVREM16 \(page 347\)](#)

\$UDIVREM32



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	Yes, if and only if the divisor is 0

More information: [\\$UDIVREM32 \(page 348\)](#)

\$UFIX



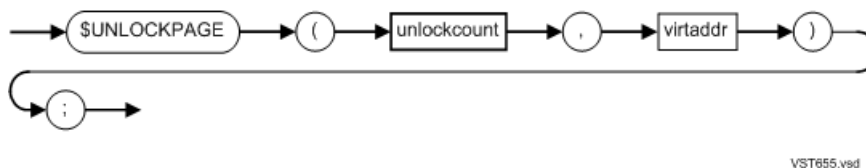
pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$UFIX \(page 349\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$UNLOCKPAGE

NOTE: The EpTAL compiler does not support this routine.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	Yes
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$UNLOCKPAGE \(page 349\)](#)

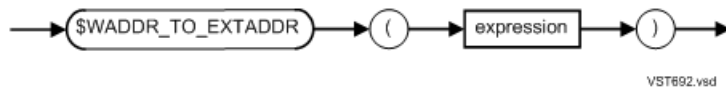
\$WADDR_TO_BADDR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$WADDR_TO_BADDR \(page 350\)](#)

\$WADDR_TO_EXTADDR

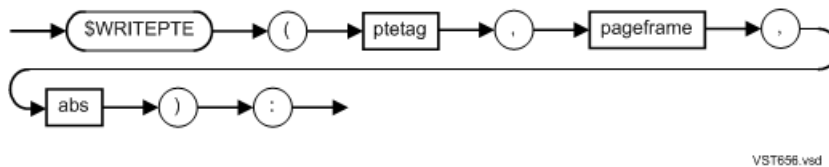


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$WADDR_TO_EXTADDR \(page 350\)](#)

\$WRITEPTE

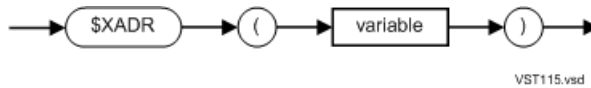
NOTE: The EpTAL compiler does not support this routine.



pTAL privileged procedure	Yes
Can be executed only by privileged procedures	Yes
Sets condition code	No
Sets \$CARRY	Yes
Sets \$OVERFLOW	No

More information: [\\$WRITEPTE \(page 351\)](#)

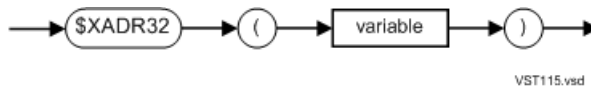
\$XADR



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$XADR \(page 352\)](#)

\$XADR32

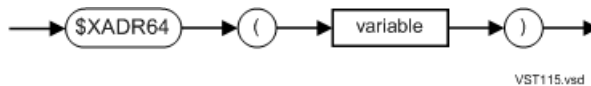


pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$XADR32 \(page 352\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

\$XADR64



pTAL privileged procedure	No
Can be executed only by privileged procedures	No
Sets condition code	No
Sets \$CARRY	No
Sets \$OVERFLOW	No

More information: [\\$XADR64 \(page 353\)](#)

NOTE: 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

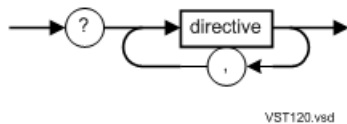
Compiler Directives

- Directive Line (page 495)
- ASSERTION (page 495)
- BASENAME (page 495)
- BEGINCOMPILE (page 496)
- BLOCKGLOBALS (page 496)
- CALL_SHARED (page 496)
- CHECKSHIFTCOUNT (page 497)
- CODECOV (page 497)
- COLUMNS (page 498)
- DEFEXPAND (page 498)
- DEFINETOG (page 499)
- DO_TNS_SYNTAX (page 500)
- ENDIF (page 500)
- ERRORFILE (page 500)
- ERRORS (page 500)
- EXPORT_GLOBALS (page 501)
- __EXT64 (page 501)
- FIELDALIGN (page 502)
- FMAP (page 502)
- GLOBALIZED (page 502)
- GMAP (page 503)
- GP_OK (page 503)
- IF, IFNOT, and ENDIF (page 504)
- INNERLIST (page 505)
- INVALID_FOR_PTAL (page 505)
- LINES (page 506)
- LIST (page 506)
- MAP (page 506)
- OPTIMIZE (page 507)
- OPTIMIZEFILE (page 507)
- OVERFLOW_TRAPS (page 508)
- PAGE (page 508)
- PRINTSYM (page 509)
- PROFDIR (page 509)
- PROFGEN (page 509)
- PROFGEN (page 509)
- REFALIGNED (page 510)
- RESETTOG (page 511)
- ROUND (page 512)
- SAVEGLOBALS (page 512)

I

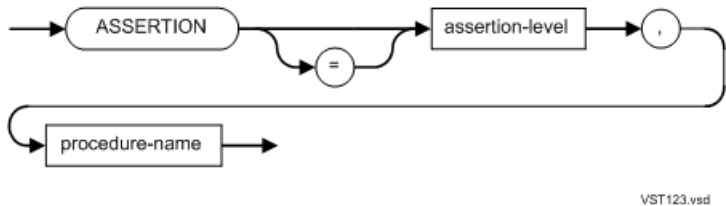
- [SECTION](#) (page 513)
- [SETTOG](#) (page 513)
- [SOURCE](#) (page 514)
- [SRL](#) (page 514)
- [SUPPRESS](#) (page 515)
- [SYMBOLS](#) (page 515)
- [SYNTAX](#) (page 516)
- [TARGET](#) (page 516)
- [USEGLOBALS](#) (page 516)
- [WARN](#) (page 517)

Directive Line



More information: [Directive Line](#) (page 367)

ASSERTION



Default:	None
Placement:	<ul style="list-style-type: none"> • Anywhere in the source file (not in the compilation command) • Must be the last directive on the directive line
Scope:	Applies until another ASSERTION overrides it
Dependencies:	Has no effect without the ASSERT statement
References:	ASSERT (page 456)

More information: [ASSERTION](#) (page 381)

BASENAME

NOTE: This directive can be used only with the EpTAL compiler.



Default:	The raw data file contains the full path name of the source file
Placement:	Only on the command line
Scope:	Applies to the compilation unit

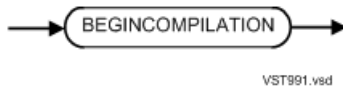
Dependencies:	Use the BASENAME option only with the PROFGEN option
References:	PROFGEN

More information: [BASENAME \(page 381\)](#)

BEGINCOMPILE

NOTE:

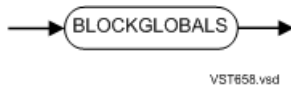
- This directive can appear only in the source file, not in the compilation command.
- The EpTAL compiler ignores this directive.



Default:	None
Placement:	<ul style="list-style-type: none"> • In the source file between the last global data declaration and the first procedure declaration, including any EXTERNAL and FORWARD declarations • Can appear only once in a compilation unit
Scope:	Applies to all source code that follows it in the compilation unit
Dependencies:	<ul style="list-style-type: none"> • Has no effect without the USEGLOBALS directive • If you specify either SAVEGLOBALS or USEGLOBALS, your compilation unit must have exactly one BEGINCOMPILE directive • Interacts with SAVEGLOBALS and USEGLOBALS
References:	<ul style="list-style-type: none"> • SAVEGLOBALS (page 512) • USEGLOBALS (page 516)

More information: [BEGINCOMPILE \(page 382\)](#)

BLOCKGLOBALS



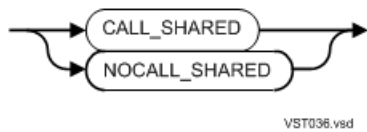
Default:	The compiler allocates data items in the _GLOBAL and \$_GLOBAL data blocks
Placement:	Before the first data declaration in a compilation
Scope:	Applies to the compilation unit
Dependencies:	None

More information: [BLOCKGLOBALS \(page 382\)](#)

CALL_SHARED

NOTE:

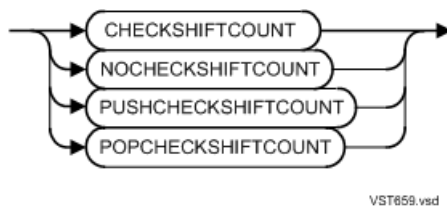
- This directive is useful only for the pTAL compiler. The EpTAL compiler ignores it (and issues a warning).
- You cannot link PIC and non-PIC object files into a single object file.



Default:	pTAL compiler: NOCALL_SHARED EpTAL compiler: CALL_SHARED
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none"> • If both CALL_SHARED and NOCALL_SHARED appear in the same compilation unit, the compiler uses the one that appears last • Do not use CALL_SHARED with GP_OK
References:	GP_OK (page 503)

More information: [CALL_SHARED \(page 383\)](#)

CHECKSHIFTCOUNT



Default:	NOCHECKSHIFTCOUNT
Placement:	Anywhere
Scope:	<ul style="list-style-type: none"> • CHECKSHIFTCOUNT applies to the shift operators that follow it until it is overridden by NOCHECKSHIFTCOUNT • NOCHECKSHIFTCOUNT applies to the shift operators that follow it until it is overridden by CHECKSHIFTCOUNT
Dependencies:	None

CAUTION: If NOCHECKSHIFT is active and a bit-shift operation occurs in which the number of positions in a bit-shift operation is too large, subsequent program behavior is undefined.

More information: [CHECKSHIFTCOUNT \(page 384\)](#)

CODECOV

NOTE: This directive is valid only in the EpTAL command line.



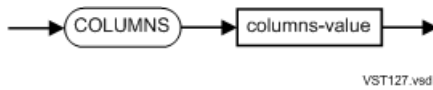
Default:	No code coverage instrumentation in object code
Placement:	Only on the command line

Scope:	Applies to the compilation unit
Dependencies:	None

More information:

- See [Using the Code Profiling Utilities \(page 366\)](#).
- See the *Code Profiling Utilities Manual*.

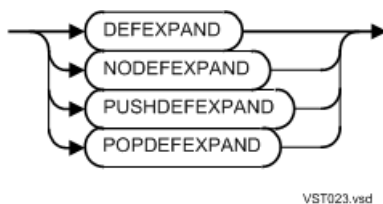
COLUMNS



Default:	COLUMNS 132
Placement:	<ul style="list-style-type: none"> • Anywhere, but if COLUMNS appears in the source code, COLUMNS must be the only directive on the directive line • Typically specified before any SECTION directive
Scope:	<p>Applies to all source code that follows it unless overridden by:</p> <ul style="list-style-type: none"> • Another COLUMNS directive in the same source file (not recommended) • A COLUMNS directive in a source file included by means of a SOURCE directive • A COLUMNS directive in a section identified by a SECTION directive <p>For details, see the explanation that follows this table.</p>
Dependencies:	None
References:	<ul style="list-style-type: none"> • SECTION (page 513) • SOURCE (page 514)

More information: [COLUMNS \(page 385\)](#)

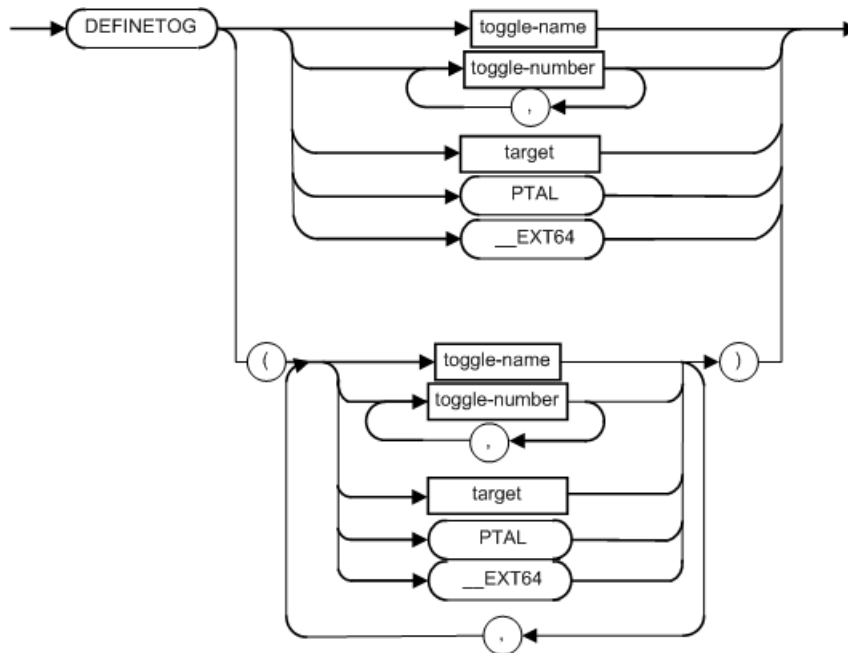
DEFEXPAND



Default:	NODEFEXPAND
Placement:	Anywhere
Scope:	<ul style="list-style-type: none"> • DEFEXPAND applies to subsequent code it until it is overridden by NODEFEXPAND • NODEFEXPAND applies to subsequent code until it is overridden by DEFEXPAND
Dependencies:	DEFEXPAND has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none"> • LIST (page 506) • SUPPRESS (page 515)

More information: [DEFEXPAND \(page 386\)](#)

DEFINETOG

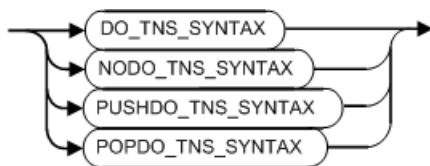


VST212.vsd

Default:	None
Placement:	<ul style="list-style-type: none">• With a parenthesized list, it can appear anywhere• Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none">• SETTOG• RESETTOG• IF and IFNOT• ENDIF• TARGET• __EXT64
References:	<ul style="list-style-type: none">• SETTOG (page 415)• RESETTOG (page 411)• IF and IFNOT (page 398)• ENDIF (page 390)• "TARGET" (page 423)• "__EXT64" (page 394)• Toggles (page 370)

More information: [DEFINETOG \(page 388\)](#)

DO_TNS_SYNTAX



VST663.vsd

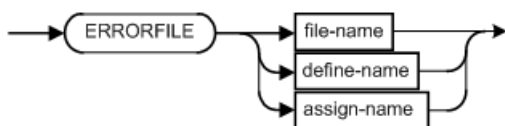
Default:	NODO_TNS_SYNTAX
Placement:	<ul style="list-style-type: none">• Can appear only once in a compilation• Must precede any TARGET directive and any nondirective lines
Scope:	Applies to the compilation unit
Dependencies:	None
References:	TARGET (page 516)

More information: [DO_TNS_SYNTAX \(page 389\)](#)

ENDIF

See [IF](#), [IFNOT](#), and [ENDIF \(page 504\)](#).

ERRORFILE

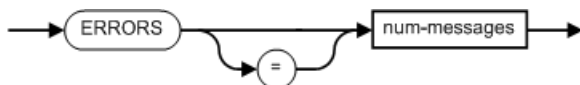


VST137.vsd

Default:	None
Placement:	<ul style="list-style-type: none">• In the compilation command or in the source code before any declarations• Can appear only once in a compilation unit
Scope:	Applies to the compilation unit
Dependencies:	None

More information: [ERRORFILE \(page 391\)](#)

ERRORS



VST138.vsd

Default:	Unlimited number of errors
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

More information: [ERRORS \(page 393\)](#)

EXPORT_GLOBALS



VST662.vsd

Default:	EXPORT_GLOBALS
Placement:	<ul style="list-style-type: none">• Can appear any number of times in a compilation unit• Must appear before the first procedure is compiled• Cannot appear within BLOCK declarations
Scope:	Applies to the compilation unit, except that NOEXPORT_GLOBALS does not affect a compilation's private data block, which is always exported
Dependencies:	<ul style="list-style-type: none">• You must specify NOEXPORT_GLOBALS when declaring a data block that belongs to an SRL• In a compilation that includes USEGLOBALS, the compiler exports the data blocks declared in the USEGLOBALS declarations file only if EXPORT_GLOBALS is active when the compiler encounters the BEGINCOMPILATION directive.
References:	<ul style="list-style-type: none">• BEGINCOMPILATION (page 496)• SRL (page 514)• USEGLOBALS (page 516)

More information: [EXPORT_GLOBALS \(page 393\)](#)

__EXT64



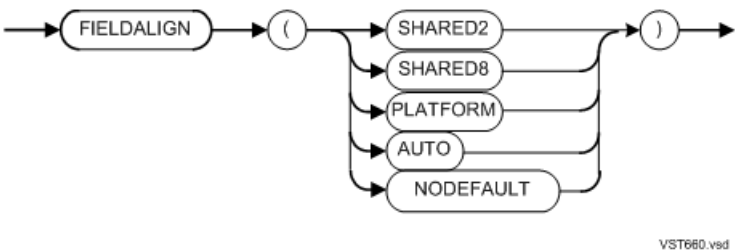
VST834.vsd

Default:	off
Placement:	Must appear either on the compiler command line or in the compiled source code before the first source code token is scanned by the compiler.
Scope:	Affects the entire compilation
Dependencies:	None
References:	<ul style="list-style-type: none">• "DEFINETOG" (page 388)• "ENDIF" (page 390)• "IF and IFNOT" (page 398)• "RESETTOG" (page 511)• "SETTOG" (page 415)• Toggles (page 370)

More information: [__EXT64 \(page 394\)](#)

NOTE: This directive is available in the 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “64-bit Addressing Functionality” (page 531).

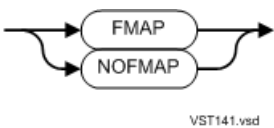
FIELDALIGN



Default:	FIELDALIGN AUTO
Placement:	<ul style="list-style-type: none">• Can appear only once in a compilation unit• Must precede all declarations of data, blocks, and procedures
Scope:	Applies to the compilation unit
Dependencies:	None

More information: [FIELDALIGN \(page 395\)](#)

FMAP



Default:	NOFMAP
Placement:	Anywhere, any number of times. The last FMAP or NOFMAP in the compilation unit determines whether the compiler lists the file map.
Scope:	Applies to the compilation unit
Dependencies:	FMAP has no effect if either NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none">• LIST (page 506)• SUPPRESS (page 515)

More information: [FMAP \(page 396\)](#)

GLOBALIZED

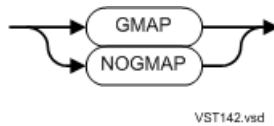
NOTE: This directive is valid only in the eptal command line.



Default:	Generate non-preemptable object code
Placement:	On the command line

Scope:	Applies to the compilation unit
Dependencies:	None

GMAP

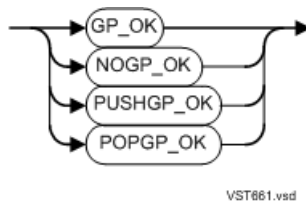


Default:	GMAP
Placement:	Anywhere, any number of times. The last GMAP or NOGMAP in the compilation unit determines whether the compiler lists the global map.
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none"> • GMAP has no effect if NOLIST, NOMAP, or SUPPRESS is active • NOGMAP suppresses the global map even if MAP is active
References:	<ul style="list-style-type: none"> • LIST (page 506) • MAP (page 506) • SUPPRESS (page 515)

More information: [GMAP \(page 397\)](#)

GP_OK

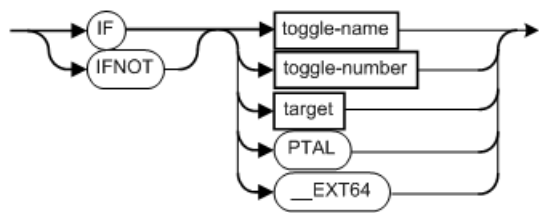
NOTE: The EpTAL compiler ignores this directive.



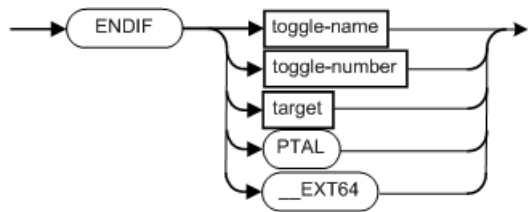
Default:	pTAL compiler:	GP_OK
EpTAL compiler:	NOGP_OK	
Placement:	Anywhere except inside a data block or inside a procedure declaration	
Scope:	<ul style="list-style-type: none"> • GP_OK applies to subsequent code it until it is overridden by NOGP_OK • NOGP_OK applies to subsequent code until it is overridden by GP_OK 	
Dependencies:	Do not use GP_OK with CALL_SHARED	
References:	CALL_SHARED (page 496)	

More information: [GP_OK \(page 397\)](#)

IF, IFNOT, and ENDIF

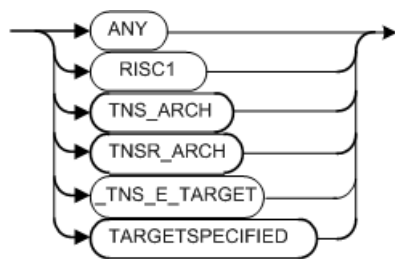


VST694.vsd



VST693.vsd

target



VST695.vsd

Compiler	Implicitly Sets	Implicitly Resets
pTAL	<ul style="list-style-type: none">RISC1TARGETSPECIFIED	_TNS_E_TARGET
EpTAL	<ul style="list-style-type: none">_TNS_E_TARGETTARGETSPECIFIED	RISC1

pTAL

Compiler	IF pTAL	IFNOT pTAL
pTAL or EpTAL	True	False
TAL	False	True

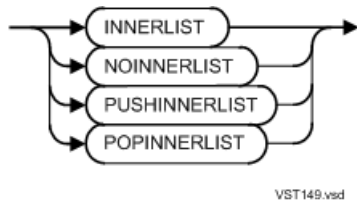
Default:	None
Placement:	<ul style="list-style-type: none">Anywhere in the source file (not in the compilation command)IF or IFNOT must be the last directive on its directive lineENDIF must be the only directive on its directive line
Scope:	Everything between IF or IFNOT and the next ENDIF that specifies the same toggle, target, or keyword

Dependencies:	Interact with: <ul style="list-style-type: none"> • DEFINETO • SETTO • RESETTO • TARGET
References:	<ul style="list-style-type: none"> • DEFINETO (page 499) • RESETTO (page 511) • SETTO (page 513) • TARGET (page 516) • Toggles (page 370)

More information:

- [IF and IFNOT](#) (page 398)
- [ENDIF](#) (page 390)

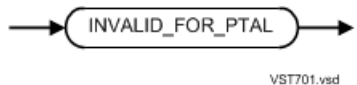
INNERLIST



Default:	NOINNERLIST
Placement:	Anywhere
Scope:	<ul style="list-style-type: none"> • INNERLIST applies to subsequent statements it until it is overridden by NOINNERLIST • NOINNERLIST applies to subsequent statements until it is overridden by INNERLIST
Dependencies:	INNERLIST has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none"> • LIST (page 506) • SUPPRESS (page 515)

More information: [INNERLIST](#) (page 400)

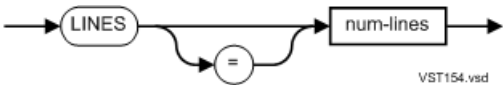
INVALID_FOR_P TAL



Default:	None
Placement:	After IF or IFNOT and before ENDIF
Scope:	Applies to code between itself and ENDIF
Dependencies:	None
References:	IF , IFNOT , and ENDIF (page 504)

More information: [INVALID_FOR_PTAL \(page 401\)](#)

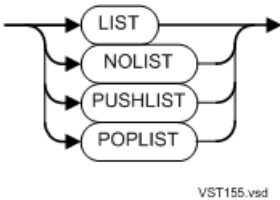
LINES



Default:	LINES 60
Placement:	Anywhere
Scope:	Applies until overridden by another LINES directive
Dependencies:	Has no effect if the list file is a terminal

More information: [LINES \(page 401\)](#)

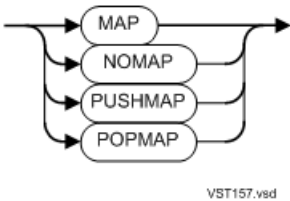
LIST



Default:	LIST
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• LIST applies to subsequent code it until it is overridden by NOLIST• NOLIST applies to subsequent code until it is overridden by LIST
Dependencies:	LIST has no effect if SUPPRESS is active
References:	SUPPRESS (page 515)

More information: [LIST \(page 401\)](#)

MAP

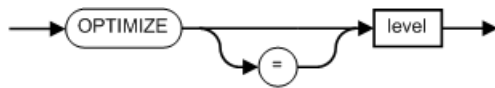


Default:	MAP
Placement:	Anywhere

Scope:	<ul style="list-style-type: none"> • MAP applies to subsequent code it until it is overridden by NOMAP • NOMAP applies to subsequent code until it is overridden by MAP
Dependencies:	MAP has no effect if NOLIST or SUPPRESS is active
References:	<ul style="list-style-type: none"> • LIST (page 506) • SUPPRESS (page 515)

More information: [MAP \(page 402\)](#)

OPTIMIZE



VST685.vsd

Default:	OPTIMIZE 1
Placement:	Outside the boundary of a separately compiled program
Scope:	The optimization level active at the beginning of a separately compiled program determines the level of optimization for that program and any programs it contains
Dependencies:	None

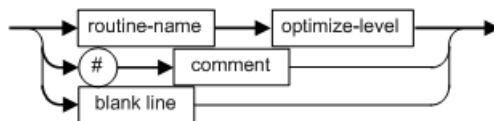
More information: [OPTIMIZE \(page 404\)](#)

OPTIMIZEFILE



VST666.vsd

filename



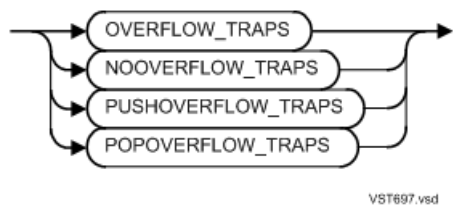
VST664.vsd

Default:	The optimization level that OPTIMIZE specified
Placement:	Only in the compilation command (not in the source file)
Scope:	Applies to the compilation unit
Dependencies:	None
References:	OPTIMIZE (page 507)

NOTE: The pTAL and EpTAL compilers behave differently.

More information: [OPTIMIZEFILE \(page 404\)](#)

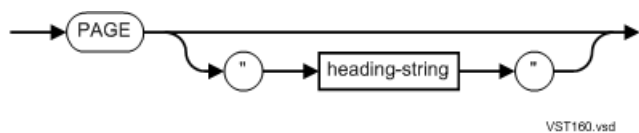
OVERFLOW_TRAPS



Default:	pTAL compiler:	OVERFLOW_TRAPS
	EpTAL compiler:	NOOVERFLOW_TRAPS
Placement:	Before or between procedure declarations	
Scope:	From the point it occurs in the compilation until it is overridden or the compilation ends, whichever occurs first	
Dependencies:	OVERFLOW_TRAPS is overridden by: <ul style="list-style-type: none">• NOOVERFLOW_TRAPS procedure attribute• DISABLE_OVERFLOW_TRAPS block attributes NOOVERFLOW_TRAPS is overridden by: <ul style="list-style-type: none">• OVERFLOW_TRAPS procedure attribute• ENABLE_OVERFLOW_TRAPS block attributes	
References:	See Managing Overflow Traps (page 234)	

More information: [OVERFLOW_TRAPS \(page 406\)](#)

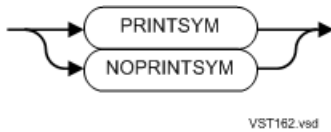
PAGE



Default:	LINES determines page ejects and no heading is printed
Placement:	Only in the source file (not in the compilation command)
Scope:	Applies until overridden by another PAGE directive
Dependencies:	Has no effect if either: <ul style="list-style-type: none">• NOLIST or SUPPRESS is active• The list file is a terminal
References:	<ul style="list-style-type: none">• LINES (page 506)• LIST (page 506)• SUPPRESS (page 515)

More information: [PAGE \(page 407\)](#)

PRINTSYM



Default:	PRINTSYM
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• PRINTSYM applies to subsequent declarations until overridden by NOPRINTSYM• NOPRINTSYM applies to subsequent declarations until overridden by PRINTSYM
Dependencies:	<ul style="list-style-type: none">• PRINTSYM has no effect if NOLIST or SUPPRESS is active• PRINTSYM interacts with SAVEGLOBALS and USEGLOBALS
References:	<ul style="list-style-type: none">• LIST (page 506)• SAVEGLOBALS (page 512)• SUPPRESS (page 515)• USEGLOBALS (page 516)

More information: [PRINTSYM \(page 408\)](#)

PROFDIR

NOTE: This directive can be used only with the EpTAL compiler.



Default:	Default subvolume
Placement:	Only on the command line
Scope:	Applies to the compilation unit
Dependencies:	PROFDIR is ignored if PROFGEN or CODEDOV is not also specified
References:	<ul style="list-style-type: none">• PROFGEN (page 509)• CODECOV (page 385)

More information: [PROFDIR \(page 408\)](#)

PROFGEN

NOTE: This directive can be used only with the EpTAL compiler.



Default:	No instrumentation in object code
Placement:	Only on the command line

Scope:	Applies to the compilation unit
Dependencies:	None

More information: [PROFGEN \(page 409\)](#)

PROFUSE

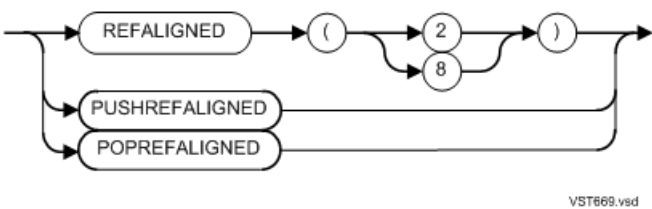
NOTE: This directive can be used only with the EpTAL compiler.



Default:	None
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	Cannot be specified with PROFGEN or CODECOV
References:	<ul style="list-style-type: none"> • PROFGEN (page 509) • CODECOV (page 385)

More information: [PROFUSE \(page 409\)](#)

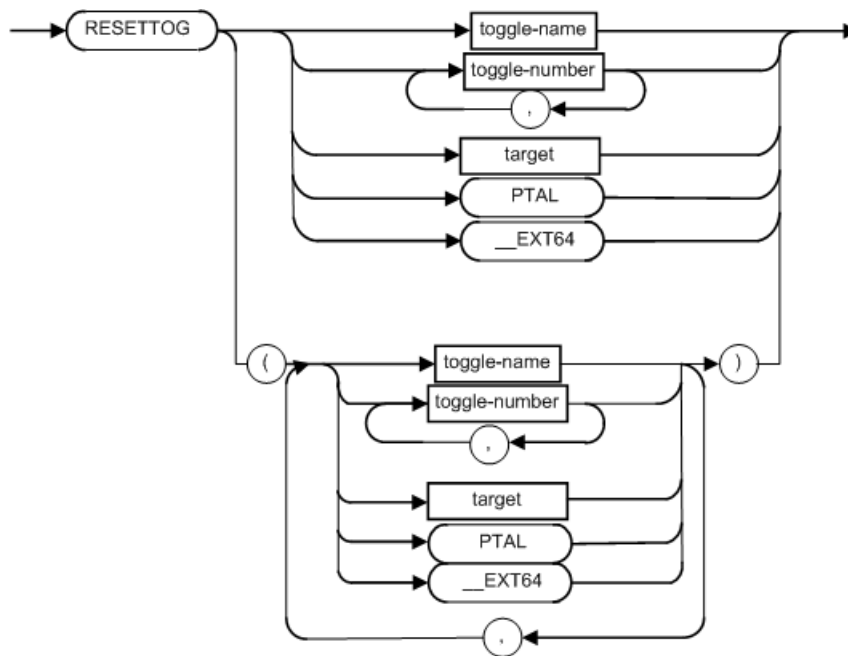
REFALIGNED



Default:	REFALIGNED 8
Placement:	Anywhere
Scope:	Applies to subsequent pointers to nonstructure data items and procedure reference parameters until overridden by another REFALIGN directive
Dependencies:	None

More information: [REFALIGNED \(page 410\)](#)

RESETTOG



VST164.vsd

Default:	None
Placement:	<ul style="list-style-type: none"> • With a parenthesized list, it can appear anywhere • Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none"> • DEFINETOG • ENDIF • __EXT64 • IF and ENDIF • SETTOG • TARGET
References:	<ul style="list-style-type: none"> • DEFINETOG (page 388) • ENDIF (page 390) • “__EXT64” (page 394) • IF and IFNOT (page 398) • SETTOG (page 415) • “TARGET” (page 516) • Toggles (page 370)

More information: [RESETTOG \(page 411\)](#)

ROUND

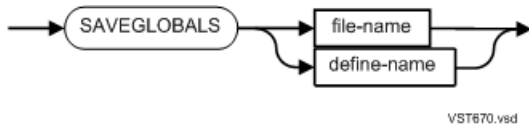


Default:	NOROUND
Placement:	Anywhere
Scope:	<ul style="list-style-type: none">• ROUND applies to subsequent code until overridden by NOROUND• NOROUND applies to subsequent code until overridden by ROUND
Dependencies:	None

More information: [ROUND \(page 412\)](#)

SAVEGLOBALS

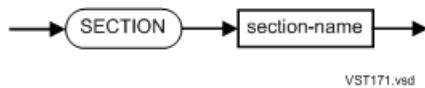
NOTE: The EpTAL compiler does not accept this directive.



Default:	None
Placement:	Either in the compilation command or in the source code before any global data declarations
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none">• If SAVEGLOBALS and USEGLOBALS appear in the same compilation unit, the compiler uses only the one that appears first• The compilation unit must have exactly one BEGINCOMPILATION directive• Interacts with the directives referenced in the next row
References:	<ul style="list-style-type: none">• BEGINCOMPILATION (page 496)• PRINTSYM (page 509)• SYMBOLS (page 515)• SYNTAX (page 516)• USEGLOBALS (page 516)

More information: [SAVEGLOBALS \(page 413\)](#)

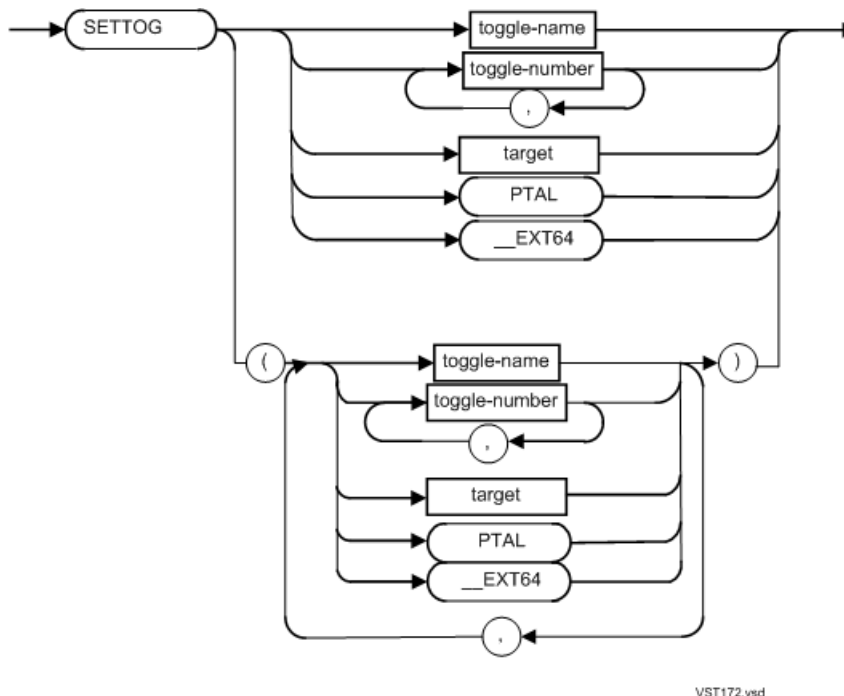
SECTION



Default:	None
Placement:	<ul style="list-style-type: none"> • Only in the source file (not in the compilation command) • Must be the only directive on the directive line
Scope:	Applies to subsequent code until another SECTION directive or the end of the file, whichever is first
Dependencies:	Interacts with SOURCE (see Section Names (page 417))
References:	SOURCE (page 416)

More information: [SECTION \(page 414\)](#)

SETTOG



Default:	None
Placement:	<ul style="list-style-type: none"> • With a parenthesized list, it can appear anywhere • Without a parenthesized list, it must be the last directive on the directive line or compilation command line
Scope:	Applies to the compilation unit
Dependencies:	Interacts with: <ul style="list-style-type: none"> • DEFINETOG • ENDIF • __EXT64 • IF and ENDIF

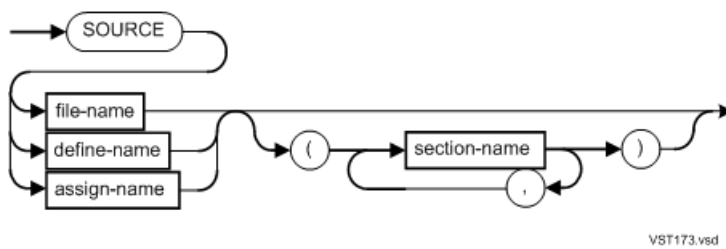
- RESETTOG
- TARGET

References:

- DEFINETOG (page 388)
- ENDIF (page 390)
- “__EXT64” (page 394)
- IF and IFNOT (page 398)
- RESETTOG (page 411)
- “TARGET” (page 516)
- Toggles (page 370)

More information: [SETTOG \(page 415\)](#)

SOURCE



Default:

None

Placement:

- Only in the source file (not in the compilation command)
- Must be the last directive on the directive line

Scope:

Applies to the source file

Dependencies:

- Interacts with COLUMNS
- Interacts with SECTION (see [Section Names \(page 417\)](#))
- Interacts with the directives referenced in the next row

References:

- BEGINCOMPILATION (page 496)
- COLUMNS (page 498)
- LIST (page 506)
- SECTION (page 513)
- SUPPRESS (page 515)
- USEGLOBALS (page 516)

More information: [SOURCE \(page 416\)](#)

SRL

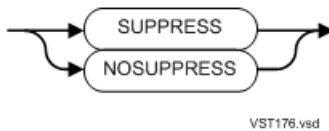
NOTE: The EpTAL compiler ignores this directive.



Default:	None
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	When declaring a data block that belongs to an SRL, you must specify NOEXPORT_GLOBALS and NOGP_OK.
References:	<ul style="list-style-type: none"> • EXPORT_GLOBALS (page 501) • GP_OK (page 503)

More information: [SRL \(page 420\)](#)

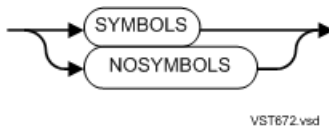
SUPPRESS



Default:	NOSUPPRESS
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	Overrides all the listing directives (see the following row)
References:	<ul style="list-style-type: none"> • DEFEXPAND (page 498) • FMAP (page 502) • GMAP (page 503) • INNERLIST (page 505) • LIST (page 506) • MAP (page 506) • PAGE (page 508) • PRINTSYM (page 509)

More information: [SUPPRESS \(page 420\)](#)

SYMBOLS



Default:	NOSYMBOLS
Placement:	Before the first declaration in the compilation
Scope:	The last legally placed SYMBOLS or NOSYMBOLS applies to the compilation unit

Dependencies:	Interacts with SAVEGLOBALS and USEGLOBALS
References:	<ul style="list-style-type: none"> • SAVEGLOBALS (page 512) • USEGLOBALS (page 516)

NOTE: These linker options discard information that SYMBOLS saves:

- `-x` discards line number information.
- `-s` discards information needed for future linking (use it only in building an executable file).

More information: [SYMBOLS \(page 421\)](#)

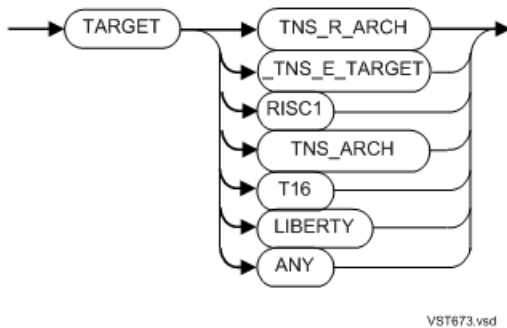
SYNTAX



Default:	The compiler produces an object file
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	Interacts with SAVEGLOBALS and USEGLOBALS
References:	<ul style="list-style-type: none"> • SAVEGLOBALS (page 512) • USEGLOBALS (page 516)

More information: [SYNTAX \(page 422\)](#)

TARGET

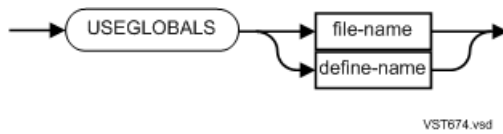


Default:	pTAL compiler:	TNS_R_ARCH
	EpTAL compiler:	_TNS_E_TARGET
Placement:	Anywhere	
Scope:	Applies to the compilation unit	
Dependencies:	None	

More information: [TARGET \(page 423\)](#)

USEGLOBALS

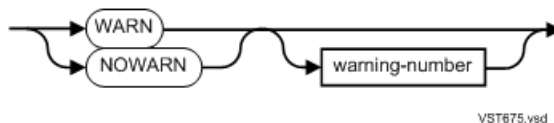
NOTE: The EpTAL compiler does not accept this directive.



Default:	None
Placement:	Either in the compilation command or in the source code before any global data declarations
Scope:	Applies to the compilation unit
Dependencies:	<ul style="list-style-type: none"> • The compilation unit must have exactly one BEGINCOMPILATION directive. • The compiler exports the data blocks declared in the USEGLOBALS declarations file only if EXPORT_GLOBALS is active when the compiler encounters the BEGINCOMPILATION directive. • A module that specifies USEGLOBALS can export a global data block that was declared in the compilation that specified SAVEGLOBALS only if the SAVEGLOBALS compilation exported the data block. Typically, a project that uses SAVEGLOBALS explicitly links globals into the object file and specifies NOEXPORT_GLOBALS (the default) for all individual compilations. • Interacts with the directives referenced in the next row.
References:	<ul style="list-style-type: none"> • BEGINCOMPILATION (page 496) • EXPORT_GLOBALS (page 501) • PRINTSYM (page 509) • SAVEGLOBALS (page 512) • SYMBOLS (page 515) • SYNTAX (page 516)

More information: [USEGLOBALS \(page 423\)](#)

WARN



Default:	WARN
Placement:	Anywhere
Scope:	<ul style="list-style-type: none"> • WARN applies to subsequent code until overridden by NOWARN • NOWARN applies to subsequent code until overridden by WARN; however: To print selected warnings, you must specify WARN before any NOWARN directives. If you specify NOWARN first, subsequent WARN <i>warning-number</i> directives have no effect.
Dependencies:	None

More information: [WARN \(page 424\)](#)

B Disk File Names and HP TACL Commands

NOTE: This appendix applies only to Guardian platforms, not Windows platforms.

- [Disk File Names \(page 518\)](#)
- [HP TACL Commands \(page 520\)](#)

For information about process or device file names, see the *Guardian Programmer's Guide*.

Disk File Names

A disk file name identifies a file that contains data or a program. A disk file name reflects the specified file's location on a NonStop system. The location of a disk file on a NonStop system is analogous to the location of a form in a file cabinet. To find the form, you must know:

- Which file cabinet it is in
- Which drawer it is in
- Which folder it is in
- Which form it is

Analogously, to find a disk file on a NonStop system, you must know:

- Which node (system) it is on
- Which volume it is on
- Which subvolume it is on
- Which disk file it is

In general, disk file names:

- Cannot contain spaces
- Can contain ASCII characters only
- Are not case-sensitive; the following names are equivalent:

myfile
MyFile
MYFILE

- Language functions and system procedures that return file names might return them in uppercase letters (even if the file name was originally in lowercase letters). Check the description of the routine that you are using.

Topics:

- [Parts of a Disk File Name \(page 518\)](#)
- [Partial File Names \(page 519\)](#)
- [Logical File Names \(page 520\)](#)
- [Internal File Names \(page 520\)](#)

Parts of a Disk File Name

A disk file has a unique file name that consists of four parts, with each part separated by a period:

- A D-series node name or a C-series system name
- A volume name
- A subvolume name
- A file ID

Example 365 Disk File Name

`\mynode.$myvol.mysubvol.myfileid`

You can name your own subvolumes and file IDs, but nodes (systems) and volumes are named by the system manager.

All parts of the file name except the file ID are optional except as noted in the following discussion. If you omit any part of the file name, the system uses values as described in [Partial File Names](#) (page 519).

Topics:

- [Node or System Name](#) (page 519)
- [Volume Name](#) (page 519)
- [Subvolume Name](#) (page 519)
- [File ID](#) (page 519)

Node or System Name

The node or system name, such as `\MYNODE`, is the name of the node or system where the file resides. If specified, the node or system name must begin with a backslash (`\`) followed by one to seven alphanumeric characters. The character following the backslash must be an alphabetic character.

Volume Name

The volume name, such as `$MYVOL`, is the name of the disk volume where the file resides. If specified, the volume name must begin with a dollar sign (`$`), followed by one to six or one to seven alphanumeric characters as follows. The character following the dollar sign must be an alphabetic character.

On a D-series system, the volume name can contain one to seven alphanumeric characters.

On a C-series system, the volume name can contain:

- One to six alphanumeric characters if you include the system name
- One to seven alphanumeric characters if you omit the system name

On a C-series system, if you specify the system name, you must also specify the volume name. If you omit the system name, specifying the volume name is optional.

Subvolume Name

The subvolume name, such as `MYSUBVOL`, is the name of the set of files, on the disk volume, within which the file resides. The subvolume name can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

On a D-series system, if you specify the volume name, you must also specify the subvolume name. If you omit the volume name, specifying the subvolume name is optional.

File ID

The file ID, such as `MYFILE`, is the identifier of the file in the subvolume. The file ID can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

The file ID is required.

Partial File Names

A partial file name contains at least the file ID, but does not contain all the file-name parts. When you specify a partial file name, the operating system or other process fills in the missing file-name

parts by using your current default values. Following are the optional file-name parts and their default values:

File-Name Part	Default
node (system)	Node (system) on which your program is executing
volume	Current default volume
subvolume	Current default subvolume

Following are all the partial file names you can specify for a disk file named \BRANCH.\$DIV.DEPT.EMP:

Omitted File-Name Parts	Partial File Name	D-Series System	C-Series System
Node (system)	\$div.dept.emp	Yes	Yes
Node (system), volume	dept.emp	Yes	Yes
Node (system), volume, subvolume	emp	Yes	Yes
Volume	\branch.dept.emp	Yes	No
Volume, subvolume	\branch.emp	Yes	No
Subvolume	\branch.\$div.emp	No	Yes
Node (system), subvolume	\$div.emp	No	Yes

You can change your current default values in various ways:

- You can change the volume and subvolume with the VOLUME command of, for example, the HP TACL products.
- In some cases, you can specify node (system), volume, and subvolume names by issuing HP TACL ASSIGN SSV commands.

Logical File Names

You can use a logical file name in place of the disk file name. A logical file name is an alternate name you specify in an HP TACL DEFINE or ASSIGN command.

Internal File Names

The C-series operating system uses the internal form of a file name when passing it between your program and the operating system. The D-series operating system uses the internal form only if your program has not been converted to use D-series features.

For information about converting external file names to internal file names in a program, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

HP TACL Commands

Before starting the compiler, you can send information to it by using the following HP TACL commands:

- [DEFINE \(page 521\)](#)
- [PARAM SWAPVOL \(page 522\)](#)
- [ASSIGN \(page 522\)](#)

For complete information about these commands, see the following manuals:

- *TACL Reference Manual* (syntactic information)
- *TACL Programmer's Guide* (programmatic information)

- *Guardian User's Guide* (interactive information)
- *Guardian Programmer's Guide* (programmatic information)

DEFINE

- [Substituting File Names for DEFINE Macros \(page 521\)](#)
- [DEFINE Names \(page 521\)](#)

To create a DEFINE message or set its attributes, you must set a CLASS attribute for the DEFINE. The CLASS attributes are:

- [MAP DEFINE \(Guardian Platforms Only\) \(page 521\)](#)
- [TAPE DEFINE \(D-Series Systems Only\) \(page 522\)](#)
- [SPOOL DEFINE \(page 522\)](#)
- [DEFAULTS DEFINE \(page 522\)](#)

Each attribute has an initial setting based on whether the attribute is required, optional, or default.

Substituting File Names for DEFINE Macros

To substitute a file name for a DEFINE name that is being passed by a nonprivileged program to a system procedure, use the following HP TACL commands:

HP TACL Command	Purpose
SET DEFMODE ON	Enable DEFINE processing
SET DEFINE CLASS	Set the initial attribute of a DEFINE command to CLASS MAP*
SET DEFINE	Set the working attributes
ADD DEFINE	Specify a file name to substitute for a DEFINE name
* MAP DEFINES are available only on Guardian platforms.	

DEFINE Names

HP TACL DEFINE names:

- Are not case-sensitive
- Have 2 to 24 characters
- Begin with an equals sign (=) followed by an alphabetic character
- Continue with any combination of letters, digits, hyphens (-), underscores (_), and circumflexes (^)

DEFINE names that begin with an equals sign followed by an underscore (=_) are reserved by HP (for example, =_DEFAULTS).

Example 366 DEFINE Names

```
=A
=The_chosen_file
=Long-but-not-too-long
=The-File-of-The-Week
```

MAP DEFINE (Guardian Platforms Only)

When you log on, the default CLASS attribute is MAP, which requires a file name. A MAP DEFINE substitutes a file name for a DEFINE name used in the source file. For example, suppose that your current CLASS attribute is MAP and your source file includes the DEFINE name =MU<l in a SOURCE directive:

```
?SOURCE =multi
```

Before running the compiler, you can associate file name `\brig.$ullx.cable.port` with `=multi`:

```
ADD DEFINE =multi, FILE \brig.$ullx.cable.port
```

During compilation, the compiler passes the DEFINE name to a system procedure, which makes the file available to the compiler. If the system procedure cannot make the file available, the open operation fails.

TAPE DEFINE (D-Series Systems Only)

The TAPE DEFINE lets you specify attributes for labeled magnetic tapes. For instance, it lets you specify attributes such as block length, recording density, record format and length, number of reels, and labeling.

SPOOL DEFINE

The SPOOL DEFINE lets you specify spooler settings or attributes, such as number of copies, form name, location, owner, report name, and priority.

DEFAULTS DEFINE

In the DEFAULTS class, a permanently built-in DEFINE named `=_DEFAULTS` has the following attributes, which are active regardless of any DEFMODE setting:

Attribute	Required	Purpose
VOLUME	Yes	Contains the default node, volume, and subvolume names for the current process as set by the HP TACL VOLUME, SYSTEM, and LOGON commands
SWAP	No	Contains the node and volume name in which the operating system is to store swap files
CATALOG	No	Contains a substitute name for a catalog as described in the <i>SQL/MP Reference Manual</i> and the <i>SQL/MX Reference Manual</i> .

PARAM SWAPVOL

The PARAM SWAPVOL command lets you specify the volume that the compiler and SYMSERV use for temporary files. For example:

```
PARAM SWAPVOL $myvol
```

The compiler ignores any node specification and allocates temporary files on its own node. If you omit the volume, the compiler uses the default volume for temporary files; SYMSERV uses the volume that is to receive the object file.

Use this command when:

- The volumes normally used for temporary files might not have sufficient space.
- The default volume or the volume to receive the object file is on a different node from the compiler.

ASSIGN

You can issue the HP TACL ASSIGN command before starting the compiler to substitute actual file names for logical file names used in the source file. The HP TACL product stores the file-name mapping until the compiler requests it.

ASSIGN commands fall into two categories:

- [Ordinary ASSIGN Command \(page 523\)](#)
- [ASSIGN SSV \(page 523\)](#)

Ordinary ASSIGN Command

The ordinary ASSIGN command equates a file name with a logical file name used in ERRORFILE, SAVEGLOBALS, SEARCH, SOURCE, and USEGLOBALS directives. The compiler accepts only the first 75 ordinary ASSIGN messages.

NOTE: The EpTAL compiler ignores the SAVEGLOBALS and USEGLOBALS directives.

In each ASSIGN command, specify a logical identifier followed by a comma and the file name or an HP TACL DEFINE name:

```
ASSIGN dog, \a.$b.c.dog
ASSIGN cat, =mycat
```

If the file name is incomplete, the HP TACL product completes it from your current default node, volume, and subvolume. For example, if your current defaults are \X.\$Y.Z, the HP TACL product completes the incomplete file names in ASSIGN commands as follows:

Incomplete File Names	Complete File Names
ASSIGN qq, cat	ASSIGN qq, \x.\$y.z.cat
ASSIGN ss, b.dog	ASSIGN ss, \x.\$y.b.dog
ASSIGN tt, \$a.b.rat	ASSIGN tt, \x.\$a.b.rat.

If you use an HP TACL DEFINE name in place of a file name, the HP TACL product qualifies the file name specified in the ADD DEFINE command when it processes the ASSIGN command. Even if you specify new node, volume, and subvolume defaults between the ADD DEFINE command and the ASSIGN command, the ASSIGN mapping still reflects the ADD DEFINE settings.

If you issue the following commands:

```
ASSIGN aa, $a.b.cat
ASSIGN bb, $a.b.dog
ASSIGN cc, =my_zebra
ADD DEFINE =my_zebra, CLASS MAP, FILE $a.b.zebra
pTAL /IN mysource, OUT $s/ obj
```

the compiler equates SOURCE directives in MYSOURCE to files as follows:

```
?SOURCE aa ! Equivalent to ?SOURCE $a.b.cat
?SOURCE cc ! Equivalent to ?SOURCE $a.b.zebra
?SOURCE bb ! Equivalent to ?SOURCE $a.b.dog
```

You can name new source files at each compilation without changing the contents of the source file.

ASSIGN SSV

The ASSIGN SSV (search subvolume) command lets you specify which node, volume, and subvolume to take files from. The compiler uses ASSIGN SSV information to resolve partial file names in the SEARCH, SOURCE, and USEGLOBALS directives.

NOTE: The EpTAL compiler ignores the USEGLOBALS directive.

For each ASSIGN SSV command, append to the SSV keyword a value in the range 0 through 49. Values in the range 0 through 9 can appear with or without a leading 0.

For example, if you specify:

```
ASSIGN SSV1, oldfiles
```

and the compiler encounters the directive:

```
?SOURCE myutil
```

the compiler looks for oldfiles.myutil.

If you then specify:

```
ASSIGN SSV1, newfiles
```

and run the compiler again, it looks for `newfiles.myutil`.

If you omit the node or volume, the HP TACL product uses the current default node or volume. If you omit the subvolume, the compiler ignores the command. HP TACL DEFINE names are not allowed.

The ASSIGN SSV command also lets you specify the order in which subvolumes are searched. You can specify ASSIGN SSV commands in any order. If the same SSV value appears more than once, the HP TACL product stores only the last command having that value.

For example, if you issue the following commands, the HP TACL product stores only two of the messages:

Assign SSV Command	Stored
ASSIGN SSV28, \$a.b	Yes
ASSIGN SSV7, \$c.d	No
ASSIGN SSV7, \$e.f	No
ASSIGN SSV07, \$g.h	Yes

The compiler stores ASSIGN SSV messages in its SSV table in ascending order.

For each file name the compiler processes, the compiler scans the SSVs in ascending order from SSV0 until it finds a subvolume that holds the file.

For example, if you issue the following ASSIGN commands before running the compiler:

```
ASSIGN SSV7, $aa.b3
ASSIGN SSV10, $aa.grplib
ASSIGN SSV8, mylib
ASSIGN SSV20, $cc.divlib
ASSIGN trig, $sp.math.xtrig
```

and the compiler encounters the following SOURCE directive:

```
?SOURCE unpack
```

the compiler first looks for an ASSIGN message having the logical name `unpack`. If there is none, the compiler looks for the file in subvolumes in the following order:

```
$aa.b3.unpack (SSV7)
$default-volume.mylib.unpack (SSV8)
$aa.grplib.unpack (SSV10)
$cc.divlib.unpack (SSV20)
$default-volume.default-subvolume.unpack
```

The compiler uses the first file it finds. If it finds none named `unpack`, it issues an error message.

When the compiler encounters this directive:

```
?SOURCE trig
```

it tries only `$sp.math.xtrig`; if it does not find that exact file, it issues an error message.

C Differences Between the pTAL and EpTAL Compilers

- General (page 525)
- Data Types and Alignment (page 525)
- Routines (page 525)
- Compiler Directives (page 527)

General

Topic	pTAL Compiler	EpTAL Compiler
RVU	D40 and later	G06.20 and laterH06.01 and later
Compiler command	ptal	eptal
Cross compiler*	NonStop pTAL	NonStop EpTAL
Object code generated	<ul style="list-style-type: none">• TNS/R object code• Non-PIC (default) or PIC• Object files have file code 700 on Guardian platform• Preemptable	<ul style="list-style-type: none">• TNS/E object code• PIC• Object files have file code 800 on Guardian platform• Non-preemptable (default) or preemptable

* For differences between cross compilers, see [NonStop pTAL \(ETK\) \(page 426\)](#)

Data Types and Alignment

Topic	pTAL Compiler	EpTAL Compiler
STRUCTALIGN clause with MAXALIGN attribute	Syntax error	Accepted in template structure declarations (see Declaring Template Structures (page 139))
EXT32ADDR*	Syntax error	Accepted
EXT64ADDR*	Syntax error	Accepted
PROC32ADDR*	Syntax error	Accepted
PROC64ADDR*	Syntax error	Accepted
PROC32PTR*	Syntax error	Accepted
PROC64PTR*	Syntax error	Accepted

* 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01 ^AAP. For more information, see Appendix E, ["64-bit Addressing Functionality" \(page 531\)](#).

Routines

Routine or Attribute	pTAL Compiler	EpTAL Compiler
INTERRUPT attribute		Not recognized
RETURN statement		Issues a warning if a RETURN statement includes both a <i>result-expression</i> and a <i>cc-expression</i> (see Appendix D (page 528))

Routine or Attribute	pTAL Compiler	EpTAL Compiler
\$AXADR routine		Not supported except as a DEFINE name
\$EXECUTEIO routine		Not supported
\$FREEZE routine		Not supported except as a DEFINE name. Use \$TRIGGER instead.
\$HALT routine		Not supported except as a DEFINE name. Use \$TRIGGER instead.
\$INTERROGATEHIO routine		Not supported
\$INTERROGATEIO routine		Not supported
\$LOCATESPTHDR routine		Not supported
\$LOCKPAGE routine		Not supported
\$READBASELIMIT routine		Not supported
\$READSPT routine		Not supported
\$STACKALLOCATE routine	<ul style="list-style-type: none"> If <i>size</i> is not an integral multiple of 8, \$STACK_ALLOCATE rounds <i>size</i> up to the next integral multiple of 8. The returned value is aligned to an 8-byte boundary. 	<ul style="list-style-type: none"> If <i>size</i> is not an integral multiple of 16, \$STACK_ALLOCATE rounds <i>size</i> up to the next integral multiple of 16. The returned value is aligned to a 16-byte boundary.
\$UNLOCKPAGE routine		Not supported
\$TRIGGER routine	Not supported	
\$UNLOCKPAGE routine		Not supported
\$WRITEPTE routine		Not supported
\$EXT64ADDR_TO_EXTADDR*	Not supported	Supported
\$EXTADDR_TO_EXT32ADDR*	Not supported	Supported
\$EXT64ADDR_TO_EXT32ADDR_OV*	Not supported	Supported
\$FIXED0_TO_EXT64DDR*	Not supported	Supported
\$IS_32BIT_ADDR*	Not supported	Supported
\$PROC32ADDR*	Not supported	Supported
\$PROC64ADDR*	Not supported	Supported
\$UFIX*	Not supported	Supported
\$XADR32*	Not supported	Supported
\$XADR64*	Not supported	Supported

* 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, "64-bit Addressing Functionality" (page 531).

In EpTAL compiler starting with SPR T0561H01^AAP, the directive `?__EXT64` expands the functionality of the following routines:

- **\$PROCADDR**: Enables \$PROCADDR to convert PROC32ADDR and PROC64ADDR addresses to PROCADDR addresses.
- **\$XADR**: Enables \$XADR to convert EXT32ADDR and EXT64ADDR typed variable addresses to EXTADDR addresses.

Compiler Directives

Directive	pTAL Compiler	EpTAL Compiler
BEGINCOMPILATION		Ignored
SAVEGLOBALS		Not accepted
USEGLOBALS		Not accepted
CALL_SHARED		Default
NOCALL_SHARED	Default	Not accepted
GP_OK		Not accepted
NOGP_OK		Ignored
PUSHGP_OK		Ignored
POPGP_OK		Ignored
OPTIMIZEFILE	Does not issue warnings for errors in <i>filename</i>	Issues a warning when <i>filename</i> meets one of the “ Compiler Directives ” (page 527)
OVERFLOW_TRAPS	Default	
NOOVERFLOW_TRAPS		Default
SRL		Not accepted
TARGET	Default and only accepted option is TNS_R_ARCH	Default and only accepted option is _TNS_E_TARGET
__EXT64*	Not accepted	Accepted
* 64-bit addressing functionality added to the EpTAL compiler starting with SPR T0561H01^AAP. For more information, see Appendix E, “ 64-bit Addressing Functionality ” (page 531).		

NOTE:

Conditions:

The EpTAL compiler issues a warning when the *filename* in OPTIMIZEFILE:

- Does not exist
- Cannot be opened
- Is not an EDIT file (Guardian operating systems only)
- Has the same *routine-name* on more than one line
- Has a line that:
 - Exceeds 511 characters (Windows operating systems only)
 - Has a *routine-name* that does not match any routine declaration in the source file
 - Has an *optimize-level* other than 0, 1, or 2
 - Has one or more characters other than spaces or tabs:
 - Before *routine-name*
 - After *optimize-level*
 - Between *routine-name* and *optimize-level*

D RETURN, RETURN SCC, and C/C++ on TNS/E

Read this appendix if you write or call pTAL procedures that:

- Return both:
 - A traditional function value by means of the RETURN statement
 - An unrelated condition code value by means of the RETURN SCC attribute
- And are called by C or C++ procedures

On the TNS architecture, a TAL procedure can return both a traditional function value and an unrelated condition code value. Both return values are accessible after the procedure call. pTAL procedures emulate this behavior on both the TNS/R and TNS/E architectures, but C/C++ procedures do not.

On the TNS/R architecture, if a C/C++ procedure calls a pTAL procedure that returns both a traditional function value and a condition code value, the C/C++ compiler issues an error message.

Some programmers work around this C/C++ compile-time error by writing C/C++ prototypes that rely on the knowledge that on the TNS/R architecture, pTAL object code stores the two return values in a single 64-bit value. After the C/C++ procedure calls the pTAL procedure, it extracts from the 64-bit value either both return values (see [Example 367 \(page 528\)](#)) or only the traditional function value (see [Example 368 \(page 529\)](#)).

△ CAUTION: C/C++ prototypes such as these are not guaranteed to work on the TNS/R architecture, and extracting only the traditional function value (as in [Example 368 \(page 529\)](#)) does not work on the TNS/E architecture.

The EpTAL compiler issues a warning whenever a pTAL procedure returns both a traditional function value and a condition code value. To migrate such a procedure to TNS/E, HP recommends that you:

1. Write a pTAL shell procedure that returns the two values in the way that C/C++ returns them (in [Example 369 \(page 529\)](#), this procedure is P_SHELL).
2. Change the alias of the C/C++ prototype to the name of the pTAL shell procedure in [FIX_THIS_LINK](#). (This change eliminates the need to change the calls to this prototype.)
3. Retire the original pTAL procedure linkage name. This allows the `eld` utility to identify any uses of unchanged C/C++ prototypes, instead of producing an executable program that uses the old prototypes (because the `eld` utility does not produce an executable program if there are unresolved procedure references).

Example 367 C Procedure Extracting Two pTAL Return Values from a 64-Bit Value (Works Only on TNS/R Systems—Not Recommended)

pTAL procedure with two return values:

```
int proc p (i, j, k) returnscc;
    int(16)      i;
    int(32) .ext  j;
    int(64)      k;
begin
    ...
    return i, j < k; ! Traditional function value is the value of i.
                   ! Expression j < k sets condition code.
end;
```

C/C++ prototype for accessing pTAL procedure:

```
_tal_alias ("P") long long some_name (short i, int* j, long long k);
```

C/C++ code that captures the 64-bit value:


```
typedef union val_cc_combo
{
    long long combo;
    struct
    {
        long value;
        long condition_code;
    } parts
} val_cc_combo
```

```
...
val_cc_combo.combo = some_name ();
```

C/C++ code that extracts the condition code from the 64-bit value:

```
(short)val_cc_combo.parts.value /* For 16-bit return value */
val_cc_combo.parts.value      /* For 32-bit return value */
```

C/C++ code that extracts the two return values from the 64-bit value:

```
(short)val_cc_combo.parts.condition_code /* Always 16-bits */
```

Example 368 C Procedure Extracting Only the Traditional Function Value from a 64-Bit Value (Works Only on TNS/R Systems)

pTAL procedure with two return values:

```
int proc p (i, j, k) returnscc;
    int(16)      i;
    int(32) .ext j;
    int(64)      k;
begin
    ...

    return i, j < k; ! Traditional function value is the value of i.

                    ! Expression j < k sets condition code.

end;
```

C/C++ prototype for accessing pTAL procedure:

```
_tal_alias ("P") long some_name1 (short i, int* j, long long k);
```

Example 369 Migrating a pTAL Procedure With Two Return Values to TNS/E (Works on TNS/R and TNS/E Systems)

pTAL shell procedure that returns values in the way that C/C++ does:

```
int proc p_shell (result, i, j, k);
    int(32) .      result;
    int(16)      i;
    int(32) .ext j;
    int(64)      k;
begin
    int cc;
    result := p (i, j, k);
    if < then
        cc := -1D
    else
        if > then
            cc := 1D
        else
            cc := 0D;
```

```
    return cc;
end;
```

Declaration of pTAL procedure P in [Example 367 \(page 528\)](#):

```
int proc p (i, j, k) returnscc;
    int(16)      i;
    int(32) .ext j;
    int(64)      k;
begin
    ...
    return i, j < k; ! Traditional function value is the value of i.
                    ! Expression j < k sets condition code.
end;
```

C/C++ prototype for accessing pTAL shell procedure:

```
_tal _alias ("P_SHELL") short xyz
(int* result, short i, int* j, long long k);
```

E 64-bit Addressing Functionality

64-bit addressing functionality has been added to the EpTAL compiler starting with SPR T0561H01 ^AAP. This functionality is accessible only when the directive `__EXT64` is specified.

Code generated for the 64-bit addressing functionality can be executed on all H-series and J-series RVUs. This functionality is not supported by the TAL and pTAL compilers nor can it be executed on D-series and G-series RVUs.

For more detailed information, see:

- [“Address Types” \(page 49\)](#)
- [“Procedure Pointers” \(page 263\)](#)
- [“Built-In Routines” \(page 274\)](#)
- [“DEFINETOG” \(page 388\)](#)
- [“IF and IFNOT” \(page 398\)](#)
- [“__EXT64” \(page 394\)](#)

The following sections describe the address types, procedure pointer types, built-in routines, toggles, and directives for 64-bit addressing functionality:

Address Types

EXT32ADDR

An explicit 32-bit extended address type. The behavior of EXT32ADDR is identical to EXTADDR and implicit conversions to and from EXT32ADDR and EXTADDR are allowed.

```
EXTADDR e32a1;  
EXT32ADDR e32a2;
```

EXT64ADDR

A 64-bit extended address type similar to EXTADDR and EXT32ADDR.

```
EXT64ADDR e64a;
```

PROC32ADDR

An explicit 32-bit procedure address type similar to PROCADDR.

```
PROC32ADDR p32a;
```

PROC64ADDR

A 64-bit procedure address type similar to PROCADDR and PROC32ADDR.

```
PROC64ADDR p64a;
```

Procedure Pointer Types

PROC32PTR

An explicit 32-bit procedure pointer that is similar to PROCPTR.

```
PROC32PTR p (x,y)  
    INT(16) x;  
    INT(16) y;  
END PROCPTR; -- Note keyword PROCPTR here.
```

PROC64PTR

A 64-bit procedure pointer that is similar to PROCPTR and PROC32PTR.

```
PROC64PTR p (x,y)
    INT(16) x;
    INT(16) y;
END PROCPTR; -- Note keyword PROCPTR here.
```

Indirection Symbols

.EXT32

An explicit 32-bit extended address type indirection symbol similar to .EXT.

```
INT .EXT I; ! @I is type EXTADDR
INT .EXT32 J; ! @J is type EXT32ADDR
```

.EXT64

A 64-bit extended address type indirection symbol similar to .EXT and .EXT32.

```
INT .EXT64 J; ! @J is type EXT64ADDR
```

Built-in Routines

\$EXT64ADDR_TO_EXTADDR

```
$EXT64ADDR_TO_EXTADDR ( <EXT64ADDR expression> )
```

Converts 64-bit extended address values to 32-bit extended EXTADDR-typed address values; no check is performed to see if the resulting 32-bit extended address value is valid.

\$EXT64ADDR_TO_EXT32ADDR

```
$EXT64ADDR_TO_EXT32ADDR ( <EXT64ADDR expression> )
```

Converts 64-bit extended address values to 32-bit EXT32ADDR-typed extended address values; no check is performed to see if the 32-bit address value is valid.

\$EXT64ADDR_TO_EXT32ADDR_OV

```
$EXT64ADDR_TO_EXT32ADDR_OV ( <EXT64ADDR expression> )
```

Converts 64-bit extended address values to 32-bit EXT32ADDR-typed extended address values; if the address cannot be represented in 32-bits, an overflow trap occurs. This trap cannot be disabled using the existing overflow trap controlling mechanisms (For example, using NO_OVERFLOW_TRAPS).

\$EXTADDR_TO_EXT64ADDR

```
$EXTADDR_TO_EXT64ADDR ( <EXTADDR or EXT32ADDR expression> )
```

Converts 32-bit extended address values to 64-bit EXT64ADDR-typed extended address values.

\$FIXED0_TO_EXT64ADDR

```
$FIXED0_TO_EXT64ADDR ( <FIXED expression> )
```

Converts value of type FIXED to EXT64ADDR address value.

\$FIX

```
$FIX ( <EXT64ADDR expression> )
```

In addition to the conversions supported by \$FIX, it also converts a value of type EXT64ADDR to integer type FIXED.

\$IS_32BIT_ADDR

```
$IS_32BIT_ADDR ( <address expression> )
```

Returns -1 if the specified address value can be represented as a 32-bit byte address; otherwise, returns 0. Input values can be any of the address types except SGWADDR and SGBADDR, which are 16-bits in length.

\$PROCADDR

`$PROCADDR (<INT(32), PROCADDR, PROC32ADDR, or PROC64ADDR expression>)`

This standard function converts an INT(32), PROCADDR, PROC32ADDR, or PROC64ADDR expression to a PROCADDR. The bit pattern is unchanged.

\$PROC32ADDR

`$PROC32ADDR (<INT(32), PROCADDR, PROC32ADDR, or PROC64ADDR expression>)`

This standard function converts a PROCADDR, PROC32ADDR, or PROC64ADDR expression to a PROC32ADDR. The bit pattern is unchanged.

\$PROC64ADDR

`$PROC64ADDR (<PROCADDR, PROC32ADDR, or PROC64ADDR expression>)`

This standard function converts a PROCADDR, PROC32ADDR, or PROC64ADDR expression to a PROC64ADDR. In pTAL, the bit pattern is unchanged.

\$UFX

`$UFX (<INT(32)-typed expression>)`

\$UFX returns a value of type FIXED.

Returns the FIXED-type zero-extended value of the specified INT(32)-typed expression.

\$XADR

x

`$XADDR (<variable or struct expression>)`

\$XADR returns a value of the extended address type EXTADDR.

When system global addresses are converted to extended addresses, \$XADR returns an absolute extended address in absolute segment 1. Conversions are allowed only if there is an explicit conversion defined to EXTADDR.

\$XADR32

`$XADR32 (<variable or struct expression>)`

Similar in function to \$XADR(), \$XADR32() returns the 32-bit extended address value of type EXTADDR for the specified variable or formal parameter. Conversions are allowed only if there is an explicit conversion defined to EXT32ADDR or to EXTADDR.

When system global addresses are converted to extended addresses, \$XADR32() returns an absolute extended address in absolute segment 1.

\$XADR64

`$XADR64 (<variable or struct expression>)`

Similar in function to \$XADR(), \$XADR64() returns the extended 64-bit address value of type EXT64ADDR for the specified variable or formal parameter. Conversions are allowed only if there is an explicit conversion defined to EXT64ADDR or to EXTADDR.

When system global addresses are converted to extended addresses, \$XADR64() returns an absolute extended address in absolute segment 1.

Implicitly Defined Compilation Toggle `__EXT64`

The state of this toggle reflects whether the `__EXT64` directive (see [__EXT64](#)) has been specified. If the `__EXT64` directive is not specified, the compiler implicitly sets the `__EXT64` toggle off. Likewise, if the `__EXT64` directive is specified, the compiler implicitly sets this toggle on.

This toggle is implicitly defined and maintained by all versions of EpTAL starting with SPR T0561H01^AAP. It is not supported by earlier versions of EpTAL or any version of the pTAL or TAL compiler.

For downward compatibility with earlier versions of EpTAL and with pTAL and TAL, the toggle can be specified in a `DEFINETOG` directive which creates the toggle and implicitly resets it. For more information, see [“DEFINETOG” \(page 388\)](#).

Directives

`__EXT64`

`__EXT64`

Directs the compiler to recognize the keywords and functionality for 64-bit address support. Default is off (the new keywords are not recognized). This directive also sets the implicitly defined toggle `__EXT64` as described in [“Implicitly Defined Compilation Toggle `__EXT64`” \(page 534\)](#).

`__EXT64` must be specified on the command line or before the first token in the source is parsed.

DEFINETOG, RESETTOG, and SETTOG

```
DEFINETOG __EXT64
RESETTOG __EXT64 -- Not recommended
SETTOG __EXT64 -- Not recommended
```

The implicitly defined toggle `__EXT64` reflects the status of the `__EXT64` directive.

This implicitly defined toggle is not supported by the EpTAL compilers prior to SPR T0561H01^AAP nor is it supported by any pTAL or TAL compiler. If you need to compile using earlier versions of EpTAL, pTAL, or TAL compiler, explicitly specify `__EXT64` in a `DEFINETOG` directive which explicitly defines and sets the toggle off in these compilers.

You can specify `DEFINETOG __EXT64` using EpTAL compilers starting with SPR T0561H01^AAP however, doing so has no effect on the implicitly defined `__EXT64` toggle

In TAL, pTAL, and EpTAL prior to T0561H01^AAP, you can `RESETTOG` and `SETTOG` the `__EXT64` toggle, however, this is not recommended. In T0561H01^AAP EpTAL, `RESETTOG` can be applied to the `__EXT64` toggle only if the implicit setting of the toggle is already off; likewise `SETTOG` can be applied to the `__EXT64` toggle only if the implicit setting of the toggle is already on.

IF and IFNOT

```
IF[NOT] { __EXT64 }
```

In addition to the existing functionality of `IF` and `IFNOT`, `IF __EXT64` evaluates to true if and only if the directive `__EXT64` has been specified.

Implicit Address Conversions

Implicit conversions are allowed from smaller extended address, procedure address, and procedure pointer types to larger extended address, procedure address, and procedure pointer types, respectively. In the case of procedure pointers, the prototypes of the two types must also match.

```
EXT64ADDR e64a;
EXTADDR ea;
PROC64ADDR p64a;
PROC32ADDR p32a;
PROC64PTR p64p (x); INT(16) x; END PROCPTR;
PROC64PTR p64p1(x); INT(32) x; END PROCPTR;
PROC32PTR p32p (x); INT(16) x; END PROCPTR;
```

```

PROC p;
BEGIN
    e64a := ea;
    p64a := p32a;
    @p64p := @p32p;
    ea := e64a;      -- Error: conversion must be explicit.
    p32a := p64a;    -- Error: ""      ""  ""  ""
    @p32p := @p64p;  -- Error: ""      ""  ""  ""
    @p64p1 := @p32p; -- Error: mismatched prototypes.
END;

```

Implicit conversions to/from INT(32) and EXT32ADDR are allowed.

Implicit conversions to/from EXTADDR and EXT32ADDR are allowed.

Implicit conversions from FIXED to EXT64ADDR are allowed in assignments only if the FIXED expression yields a constant value known at compile-time; they are interpreted as a byte address value.

Index

Symbols

- " (quotation mark), 39
- \$ (dollar sign), 39
- \$ABS routine, 291
- \$ALPHA routine, 291
- \$ASCIITOFIXED routine, 292
- \$ATOMIC_ routines, 66, 67
- \$ATOMIC_ADD routine, 276
- \$ATOMIC_AND routine, 277
- \$ATOMIC_DEP routine, 278
- \$ATOMIC_GET routine, 279
- \$ATOMIC_OR routine, 280
- \$ATOMIC_PUT routine, 280
- \$AXADDR routine, 293
- \$BADDR_TO_EXTADDR routine, 294
- \$BADDR_TO_WADDR routine, 294
- \$BITLENGTH routine, 295
- \$BITOFFSET routine, 296
- \$CARRY routine
 - description of, 297
 - after assignments, 236
 - atomic operation that can set, 276
 - in nested IF statements, 243
 - nonatomic operations that can set, 289
 - returning its value to calling procedure, 245
- \$CHECKSUM routine, 297
- \$COMP routine, 298
- \$COUNTDUPS routine, 299
- \$DBL routine, 300
- \$DBLL routine, 301
- \$DBLR routine, 301
- \$DFIX routine, 302
- \$EFLT routine, 302
- \$EFLTR routine, 303
- \$EXCHANGE routine, 303
- \$EXECUTEIO routine, 304
- \$EXT64ADDR_TO_EXT32ADDR routine, 307
- \$EXT64ADDR_TO_EXT32ADDR_OV routine, 307
- \$EXT64ADDR_TO_EXTADDR routine, 306
- \$EXTADDR_TO_BADDR routine, 305
- \$EXTADDR_TO_EXT64ADDR routine, 308
- \$EXTADDR_TO_WADDR routine, 306
- \$FILL16 procedure, 308
- \$FILL32 procedure, 308
- \$FILL8 procedure, 308
- \$FIX routine, 309
- \$FIXD routine, 309
- \$FIXED0_TO_EXT64ADDR, 310
- \$FIXEDTOASCII routine, 310
- \$FIXEDTOASCIIRESIDUE routine, 311
- \$FIXI routine, 312
- \$FIXL routine, 312
- \$FIXR routine, 313
- \$FLTR routine, 314
- \$FLTroutine, 314
- \$FREEZE routine, 315
- \$HALT routine, 315
- \$HIGH routine, 315
- \$IFIX routine, 316
- \$INT routine, 317
- \$INT_OV routine, 318
- \$INTERROGATEHIO routine, 319
- \$INTERROGATEIO routine, 320
- \$INTR routine, 321
- \$IS_32BIT_ADDR routine, 321
- \$LEN routine, 322
- \$LFIX routine, 323
- \$LMAX routine, 323
- \$LMIN routine, 324
- \$LOCATESPTHDR routine, 324
- \$LOCKPAGE routine, 325
- \$MAX routine, 326
- \$MIN routine, 327
- \$MOVEANDCXSUMBYTES routine, 327
- \$MOVENONDUP routine, 328
- \$NUMERIC routine, 329
- \$OCCURS routine, 330
- \$OFFSET routine
 - description of, 332
 - structure pointers and, 175
- \$OPTIONAL routine, 333
- \$OVERFLOW routine
 - description of, 335
 - after assignments, 236
 - atomic operation that can set, 276
 - built-in routines and, 276
 - in nested IF statements, 243
 - nonatomic operations that can set, 286
 - returning its value to calling procedure, 245
- \$PARAM routine, 336
- \$POINT routine, 336
- \$PROC32ADDR routine, 337
- \$PROC64ADDR routine, 338
- \$PROCADDR routine, 337
- \$READBASELIMIT routine, 338
- \$READCLOCK routine, 339
- \$READSPT routine, 339
- \$READTIME routine, 340
- \$SCALE routine, 340
- \$SGBADDR_TO_EXTADDR routine, 341
- \$SGBADDR_TO_SGWADDR routine, 342
- \$SGWADDR_TO_EXTADDR routine, 342
- \$SGWADDR_TO_SGBADDR routine, 343
- \$SPECIAL routine, 343
- \$STACK_ALLOCATE routine, 344
- \$TRIGGER routine, 345
- \$TYPE routine, 345
- \$UDBL routine, 346
- \$UDIVREM16 routine, 347
- \$UDIVREM32 routine, 348
- \$UFIX routine, 349

- \$UNLOCKPAGE routine, 349
- \$WADDR_TO_BADDR routine, 350
- \$WADDR_TO_EXTADDR routine, 350
- \$WRITEPTE routine, 351
- \$XADR routine, 352
- \$XADR32 routine, 352
- \$XADR64 routine, 353
- & (concatenation operator), 220
- ' (single quotation mark), 39
- (exclamation mark), 38
- (semicolon)
 - as delimiter, 38
 - in statements, 199
-), 38
 - as delimiter, 38
 - in statements, 199
- *
 - See Asterisk (*), 48
- +
 - See Plus sign (+), 70
- see Hyphen (-)
 - See Minus sign (-), 70
- >
 - in labeled CASE statement, 207
 - in move statement, 218
 - in RSCAN statement, 228
 - in SCAN statement, 228
- .
 - See Period (.), 38
- ... (ellipsis), 207
- .EXT
 - in equivalenced variables, 194
 - in formal parameters, 251
 - in pointers
 - simple, 170
 - structure, 173
 - in referral structures, 141
- .SG
 - in system global pointers, 176
- .tal file extension, 426
- 64-bit addressing functionality, 531
 - Address Types for, 531
 - Built-in Routines for, 532
 - Directives for, 534
 - Implicit Address Conversions for, 534
 - Implicitly Defined Compilation Toggle __EXT64 for, 534
 - Indirection Symbols for, 532
 - Procedure Pointer Types for, 531
- :=
 - assignment operator, 71 see also Assignments
- < see Less than operator, signed (<)
- < > (angle brackets); Brackets
 - angle (< >), 38
- <...> (bit extraction), 70
- <...> := (bit deposit operator), 71
- << (signed left bit shift), 70
- <= see Less than or equal operator, signed (<=)
- <> see Not equal operator, signed (<>)
- > see Greater than operator, signed (>)
- >= see Greater than or equal operator, signed (>=)
- >> (signed right bit shift), 70
- ? (question mark), 39
- @ operator
 - in entry-point identifiers
 - for procedures, 260
 - for subprocedures, 262
 - in pointers, 172
 - in PROC parameters, 256
 - in reference parameters, 254
 - precedence of, 70
- \, 38, 199
 - (colon), 260
- \[\] (square brackets); Brackets
 - square (\, 38
- __EXT64, 534
- __EXT64 directive, 394, 501, 534
- '*' see Multiplication operator, unsigned ('*')
- '+' see Addition operator, unsigned ('+')
- '-' see Subtraction operator, unsigned ('-')
- '<<' (unsigned left bit shift), 70
- '<=' see Less than or equal operator, unsigned ('<=')
- '<>' see Not equal operator, unsigned ('<>')
- '<' see Less than operator, unsigned ('<')
- '=' see Equal sign, as equal operator, unsigned
- '>=' see Greater than or equal operator, unsigned ('>=')
- '>>' (unsigned right bit shift), 70
- '>' see Greater than operator, unsigned ('>')
- '\\' see Remainder operator ('\\')
- 'P' (read-only array symbol) see Read-only arrays
- 'SG'-equivalenced variables see Equivalenced variables
- '/' see Division operator, unsigned ('/')
- / see Division operator, signed (/)

A

- ABS routine, 291
- Absolute value, 291
- Actual parameters
 - description of, 206
 - checking for presence of, 336
 - in CALL statement, 205
 - of DEFINEs, 100
- Addition operator
 - signed (+)
 - in arithmetic expression, 72
 - operand types for, 73
 - precedence of, 71
 - unsigned ('+')
 - in arithmetic expression, 72
 - operand types for, 75
 - precedence of, 71
 - result types for, 75
 - with INT(32) operands, 76
- Address misalignment
 - causes of, 66
 - handling, 67
 - tracing facility for, 66
- Address symbols, base; Symbols, base address, 40
- Address types

- description of, 49
- converting, 52
- stored in pointers, 164
- Address-conversion routines, 283
- Addresses
 - See also Data addresses, 55
 - arrays of, 108
 - as parameters to built-in routines, 275
 - assignment of, 365
 - extended, 77
 - in simple pointers, 172, 174
 - in structure pointers
 - description of, 174
 - within structures, 152
 - nonextended, 78
 - of structures declared in subprocedures, 142
 - types of see Address types
- Aliases for data types, 48
- Alignment
 - base, 119
 - of constant lists, 64
 - of data, 66
 - of structure fields, 117
 - of structures
 - in depth, 123
 - overview, 116
 - of substructures, 124
- ALPHA routine, 291
- AND operator
 - description of, 82
 - condition codes and, 83
 - in conditional expression, 81
 - operand types for, 83
 - precedence of, 71
 - truth table for, 81
- Angle brackets (< >), 38
- ar utility, 430
- Architecture and RVUs, 31
- Arguments, 368
 - See also Parameters, 275
- Arithmetic expressions, 72
- Arithmetic operators
 - signed
 - description of, 73
 - in arithmetic expressions, 72
 - unsigned
 - description of, 75
 - in arithmetic expressions, 72
- Arithmetic overflow testing, 335
- Arrays
 - description of, 108
 - alignment of, in structures, 122
 - as parameters, 254
 - data type of, 345
 - declaring
 - in structures, 143
 - read-only, 111
 - read-write, 108
 - elements of
 - accessing, 54
 - number of, 330
 - length of
 - in bits, 295
 - in bytes, 322
 - nonstring, 113
 - number of elements of, 330
 - of addresses, 108
 - redefining, 154
- ASCII characters
 - set of, 36
 - testing for
 - alphabetic, 291
 - numeric, 329
 - special (nonalphanumeric), 343
- ASCIITOFIXED routine, 292
- ASSERT statement, 200
- ASSERTION directive, 381
- ASSIGN command
 - description of, 522
 - ordinary, 523
 - search subvolume (SSV), 523
- Assignment operator (:=), 71
 - See also Assignments, 71
- Assignments
 - description of, 201
 - bit-deposit, 204
 - character string, 203
 - expressions in, 85
 - FIXED variable, 203
 - hardware indicators after, 236
 - initial, 103
 - move statement, 218
 - number, 203
 - of addresses, 365
 - pointer, 203
 - procedure pointer, 269
- Asterisk (*)
 - as multiplication operator see Multiplication operator
 - in \$ASCIITOFIXED routine, 293
 - in \$FIXEDTOASCII routine, 311
 - in \$FIXEDTOASCIIRESIDUE routine, 311, 312
 - in compiler listing, 400
 - in constant lists, 39
 - in template structures, 140
 - in value parameter, 39
 - to prevent scaling
 - of FIXED initialization value, 48
 - of FIXED parameter, 251
- Atomic operations
 - description of, 276
 - data misalignment and, 66, 67
- ATOMIC_ADD routine, 276
- ATOMIC_AND routine, 277
- ATOMIC_DEP routine, 278
- ATOMIC_GET routine, 279
- ATOMIC_OR routine, 280
- ATOMIC_PUT routine, 280
- Attributes

- block, 235
- procedure, 248
- SCF user interface, 67
- AUTO parameter
 - description of, 118
 - compared to PLATFORM parameter, 119
 - FIELDALIGN clause and, 117
- AXADR routine, 293
- B**
- Backslash (\)
 - See Remainder operator ('\\'), 71
- BADDR address type
 - description of, 165
 - converting, 53
 - parameters of, 251
 - STRING pointers of, 167
- BADDR_TO_EXTADDR routine, 294
- BADDR_TO_WADDR routine, 294
- Base address symbols, 40
- BASENAME directive, 381
- Bases of constants, 44
- BEGIN keyword
 - in compound statement, 200
 - in procedure, 256
 - in structure, 115
 - in subprocedure, 259
- BEGIN-END construct see Compound statements
- BEGINCOMPILE directive
 - description of, 382
 - and global data declarations, 373
 - SOURCE directive and, 419
- Bit fields
 - description of, 46
 - delimiting, 38
- Bit operations
 - description of, 34, 92
 - bit-deposit assignment statement, 204
 - extraction, 93
 - logical, 76
 - precedence of, 70
 - shift, 94
- Bit-deposit assignment statement, 204
- BIT_FILLER declaration, 147
- BITLENGTH routine, 295
- BITOFFSET routine, 296
- Bitwise logical operators, 76
- Block attributes, 235
- BLOCKGLOBALS directive, 382
- Blocks, data see Global data, blocked
- Boolean expressions see Conditional expressions
- Built-in routines, 274
 - See also Atomic operations, 276
- BY keyword in FOR statement, 212
- Bytes, 46

C

- C procedure attribute, 248
- C-series RVU, 31

- C/C++ procedures, 528
- CALL statement, 205
- CALL_SHARED directive, 383
- CALLABLE procedure attribute, 248, 274
- CARRY routine
 - description of, 297
 - after assignments, 236
 - atomic operation that can set, 276
 - in nested IF statements, 243
 - nonatomic operations that can set, 289
 - returning its value to calling procedure, 245
- CASE expressions, 86
- CASE statement
 - description of, 207
 - empty, 207
 - labeled, 207
 - unlabeled, 209
- CBADDR address type
 - description of, 165
 - converting, 53
 - parameters of, 251
 - pointers of, 169
- Character set for pTAL, 36
- Character string constants, 57
- Character-test routines, 284
- CHECKSHIFTCOUNT directive, 384
- CHECKSUM routine, 297
- COBOL procedure attribute, 248
- Code coverage report, 366
- Code Profiling Utilities, 366
- CODECOV directive, 385
- Codes
 - completion, 358
 - condition
 - See Condition codes, 258
- Colon (\), 38
- COLUMNS directive
 - description of, 385
 - SOURCE directive and, 418
- Comma (,), 38
- Commands
 - ASSIGN
 - See ASSIGN command, 522
 - compilation
 - See Compilation command, 357
 - DEFINE see DEFINES
 - Deploy, 431
- Comments, delimiters for, 38
- COMP routine, 298
- Compatibility of pTAL and TAL, 30
- Compilation command
 - description of, 357
 - with compiler directives, 367
- Compilation units, naming, 363
- Compiler directives
 - interpretation and processing of, 367
 - specifying
 - in compilation command line, 367

- in source code, 367
- summary of, 377
- Compiler input directives, 377
- Compiler listing
 - conditionally compiled lines and, 400
- Compiler listing directives, 377
- Compiler listing:asterisk (*) in, 400
- Compilers
 - comparison of EpTAL, pTAL, and TAL, 31
 - differences between pTAL and EpTAL , 527
- Completion codes, 358
- Compound statements
 - syntax of, 200
 - within DEFINE bodies, 99
- Concatenation operator (&), 220
- Condition codes
 - See also Hardware indicators, 234
 - after assignments, 237
 - AND operator and, 83
 - atomic operations that can set, 276
 - C/C++ procedures on TNS/E and, 528
 - group comparisons and, 91
 - nesting, 242
 - nonatomic operations that alter, 286
 - NOT operator and, 83
 - OR operator and, 83
 - returning
 - with RETURN statement, 224
 - with RETURNSCC attribute:in procedure, 248
 - with RETURNSCC attribute:in subprocedure, 257
 - testing after function calls, 224
- Conditional compilation directives, 378
- Conditional expressions
 - description of, 81
 - hardware indicators in, 239
- Constant expressions
 - description of, 81
 - as parameters, 254
 - in data type specifications, 47
- Constant lists
 - description of, 63
 - aligning, 64
 - in array declarations, 113
 - in move statement, 218
- Constants
 - See also LITERALS, 97
 - comparing to data addresses, 56
 - description of, 44
 - lists of see Constant lists
 - numeric bases of, 44
- Constants:in expressions:See Constant expressions, 64
- Continuation lines, 368
- Conventions for syntax diagrams, 20
- Conversion
 - between address types, 52
 - between addresses and numbers, 51
 - implicit, 52
- Copy operation (move statement), 218
- COUNTDUPS routine, 299

- Cross compilers
 - as utility and, 430
 - compiling with, 429
 - debugging and, 429, 431
 - documentation for, 431
 - features of, 426
 - file extension for, 426
 - from PC command line, 427
 - in ETK, 426
 - linking and, 429
 - PC-to-NonStop host transfer tools for, 431
 - platforms for, 426
- CWADDR address type
 - description of, 165
 - converting, 53
 - parameters of, 251
 - pointers of, 169
- D
- D-series RVU, 31
- Data
 - alignment of, 66
 - blocks of:See Global data, blocked, 362
 - misaligned see Address misalignment
 - operations on, 34
 - representation of, 46
 - scanning, 199
 - sets of, 33
 - system global see System global data
 - transferring
 - statements for, 199
 - types of see Data types
- Data addresses
 - arithmetic operations on, 55
 - comparing
 - description of, 77
 - extended addresses, 77
 - nonextended addresses, 78
 - to constants, 56
 - to procedure pointers, 56
 - computing distance between, 55
 - converting to numbers, 51
 - decrementing, 54
 - incrementing, 54
 - storing in variables, 51
- Data allocation statements, 199
- Data types
 - See also Address types, 46
 - aliases for, 48
 - changing, with group comparisons, 90
 - obtaining, 345
 - of expressions, 70
 - pTAL
 - description of, 46
 - compared to TAL, 33
 - specifying, 47
- Data:global:See System global data, 40
- DBL routine, 300
- DBLL routine, 301

- DBLR routine, 301
- Debugging
 - cross compilers and, 429, 431
 - Enterprise Toolkit (ETK) and, 429
 - OPTIMIZE directive and, 404
 - with ASSERTION directive and ASSERT statement, 200
- Decimal point, implied see Implied decimal point
- Declarations
 - description of, 41
 - array
 - See Arrays, declaring, 108
 - BIT_FILLER, 147
 - DEFINE, 98
 - entry point, 260
 - equivalenced see Equivalenced variables, declaring
 - external, 419
 - FILLER, 147
 - function see Procedures, declaring
 - global see Global data
 - LITERAL, 97
 - NAME, 363
 - pointer
 - See Pointers, declaring, 161
 - procedure
 - See Procedures, declaring, 246
 - simple variable see Simple variables, declaring
 - structure see Structures, declaring
 - sublocal, 259
 - subprocedure
 - See Subprocedures, declaring, 257
 - substructure see Substructures, declaring
- Default misalignment handling method, 67
- Default target file;Files
 - OBJECT, 358
- DEFAULTS DEFINE, 522
- DEFEXPAND directive
 - description of, 386
 - output of, 100
 - position of, 99
- DEFINE files, 426
- DEFINE tool, 431
- DEFINES
 - calling, 100
 - CLASS attributes of, 521
 - declaring, 98
 - expansion of, 100
 - how compiler processes, 100
 - LITERAL declarations and, 97
 - names of, 521
 - parameters of
 - actual, 100
 - formal, 98
 - substituting file names for, 521
- DEFINETOG directive, 388
- Definition structures, declaring
 - equivalenced, 194
 - not equivalenced, 138
- Definition substructures
 - declaring, 144
 - redefining, 155
- Delimiters, 38
- Deploy command, 431
- DFIX routine, 302
- Diagnostics directives, 377
- Directive stacks, 369
- DISABLE_OVERFLOW_TRAPS block attribute, 235
- Disk file names
 - description of, 518
 - as compiler directive arguments, 368
 - ASSIGN command and, 522
 - internal, 520
 - logical, 520
 - partial, 519
 - parts of, 518
 - substituting for DEFINE commands, 521
- Division operator
 - signed (/)
 - in arithmetic expression, 72
 - operand types for, 73
 - precedence of, 71
 - unsigned (/)
 - in arithmetic expression, 72
 - operand types for, 75
 - precedence of, 71
 - result types for, 75
 - with INT(32) and FIXED operands, 76
- DLLs (dynamic-link libraries);Libraries
 - dynamic-link (DLLs), 362
- DO keyword
 - in DO-UNTIL statement, 210
 - in FOR statement, 212
 - in WHILE statement, 232
- DO-UNTIL statement
 - description of, 210
 - hardware indicators in, 239
- DO_TNS_SYNTAX directive, 389
- Dollar sign (\$), 39
- Doublewords, 46
- DOWNTO keyword, 212
- DROP statement, 212
- Dynamic-link libraries (DLLs), 362
- Dynamically selected procedure calls, 271

E

- EFLT routine, 302
- EFLTR routine, 303
- eld utility
 - ar utility and, 430
 - migrating to TNS/E and, 528
- Ellipsis (...), 207
- ELSE keyword, 217
- Embedded SQL/MP or SQL/MX, 32
- Empty CASE statement, 207
- EMS (Event Management Service), 67
- ENABLE_OVERFLOW_TRAPS block attribute, 235
- END keyword
 - in compound statement, 200
 - in procedure, 256

- in structure, 115
- in subprocedure, 259
- ENDIF directive, 390
- Enterprise Toolkit (ETK)
 - cross compilers and, 426
 - debugging and, 429
 - DEFINE files and, 426
 - online help for, 431
- Entry points
 - declaring, 260
 - procedure, 168
 - subprocedure, 169
- Equal sign
 - as delimiter, 39
 - as equal operator
 - signed (=):in conditional expression, 83
 - signed (=):operand types for, 84
 - signed (=):precedence of, 71
 - signed (=):without operands, 84
 - unsigned ('='):in conditional expression, 83
 - unsigned ('='):operand types for, 84
 - unsigned ('='):precedence of, 71
 - unsigned ('='):with INT(32) operands, 76
 - unsigned ('='):without operands, 84
- Equivalenced variables
 - description of, 177
 - declaring
 - description of, 178
 - nonstructure, 180
 - system global, 193
 - memory allocation for, 179
- Error messages
 - logging to a file, 391
 - maximum allowed, 393
- ERRORFILE directive, 391
- ERRORS directive, 393
- ETK see Enterprise Toolkit (ETK)
- Event Management Service (EMS), 67
- EXCHANGE routine, 303
- Exclamation mark (, 38
- Executable statements
 - See Statements , 199
- EXECUTEIO routine, 304
- EXPORT_GLOBALS directive, 393
- Exporting program names, 362
- Expressions
 - description of, 69
 - arithmetic, 72
 - as parameters to built-in routines, 275
 - assignment, 85
 - CASE, 86
 - conditional, 81
 - constant
 - description of, 81
 - as parameters, 254
 - in data type specifications, 47
 - data types of, 70
 - group comparison see Group comparison expressions
 - IF, 87
 - special, 85
 - Expressions:Boolean (conditional), 81
 - EXT32ADDR address type
 - description of, 165
 - EXT64ADDR address type
 - description of, 165
 - EXTADDR address type
 - description of, 165
 - comparing, 77
 - converting, 53
 - parameters of, 251
 - pointers of, 169
 - EXTADDR_TO_BADDR routine, 305
 - EXTADDR_TO_WADDR routine, 306
 - EXTDECS file, 419
 - Extended addresses, 77
 - Extended parameters, 255
 - EXTENSIBLE procedure attribute, 248, 250
 - External declarations, 419
 - EXTERNAL keyword
 - in procedure declaration, 246, 247
 - in procedure entry-point declaration, 261
 - Extracting bits, 93
- F
 - FAIL misalignment handling method, 67
 - Feature control, 379
 - FIELDALIGN clause
 - description of, 127
 - role in field alignment, 117
 - FIELDALIGN directive
 - description of, 127, 395
 - FIELDALIGN directive:role in field alignment, 117
 - File IDs, 519
 - File names see Disk file names
 - Files
 - DEFINE, 426
 - EXTDECS, 419
 - input, 356
 - map of, 396
 - object see Object files
 - output, 356
 - source
 - See Source files, 355
 - target, 358
 - temporary, 522
 - FILL16 procedure, 308
 - FILL32 procedure, 308
 - FILL8 procedure, 308
 - FILLER declaration, 147
 - FIX routine, 309
 - FIXD routine, 309
 - FIXED data type
 - See also FIXED variables, 251
 - built-in routines for, 285
 - constants of, 61
 - obtaining
 - with \$DFIX routine, 302
 - with \$FIX routine, 309

- with \$FIXD routine, 309
- with \$FIXED0_TO_EXT64ADDR routine, 310
- with \$FIXR routine, 313
- with \$IFIX routine, 316
- with \$LFIX routine, 323
- parameters of, 251, 252
- FIXED data type:rounding and, 283
- FIXED variables
 - See also FIXED data type, 251
 - rounding, 412
 - scaling
 - description of, 74
 - when assigning numbers to, 203
 - using, 74
- FIXED variables:assigning numbers to, 203
- FIXED(0) data type see FIXED data type
- Fixed-point scaling, 203
- FIXED0_TO_EXT64ADDR, 310
- FIXEDTOASCII routine, 310
- FIXEDTOASCIIRESIDUE routine, 311
- FIXERRS macro, 391
- FIXI routine, 312
- FIXL routine, 312
- FIXR routine, 313
- FLTR routine, 314
- FLTroutine, 314
- FMAP directive, 396
- FOR keyword
 - in FOR statement, 212
 - in move statement, 218
- FOR statement
 - description of, 212
 - nested, 213
 - optimized, 214
 - standard, 214
- Formal parameters
 - indirection symbols and, 41
 - of DEFINES, 98
 - of procedures, 247, 251
 - of subprocedures, 251, 258
 - passing by reference, 41
 - procedure pointers as, 263, 268
 - specifying, 251
- FORTTRAN procedure attribute, 248
- FORWARD keyword
 - in procedure declaration, 246, 247
 - in procedure entry-point declaration, 261
 - in subprocedure declaration, 257, 259
 - in subprocedure entry-point declaration, 262
- fpoint
 - changing, 340
 - obtaining, 336
 - rounding, 412
 - scaling, 203
 - specifying, 323
- FREEZE routine, 315
- Functions
 - See also Procedures, 246
 - atomic

- See Atomic operations, 66
- definition of, 246
- RETURN statement and, 223
- with two return values, 528

G

- Global data
 - See also System global data, 362
 - blocked
 - allocating, 365
 - declaring, 362
 - SECTION directive and, 365
 - sharing, 365
 - SOURCE directive and, 365
 - map of, 397
 - saving and using, 372
 - unblocked, 364
- Global scope, 43
- GLOBALIZED directive, 396
- GMAP directive, 397
- GOTO statement, 215
- GP_OK directive, 397
- Greater than operator
 - signed (>)
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - without operands, 84
 - unsigned ('>')
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - with INT(32) operands, 76
 - without operands, 84
- Greater than or equal operator
 - signed (>=)
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - without operands, 85
 - unsigned ('>=')
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - with INT(32) operands, 76
 - without operands, 85
- Group comparison expressions
 - description of, 88
 - for changing data types, 90
 - testing, 91

H

- HALT routine, 315
- Hardware indicators
 - See also Condition codes, 234
 - across procedures, 244
 - after assignments, 236
 - built-in routines and, 276
 - in conditional expressions, 239

- list of, 234
- Hash mark (#), 39
- HIGH routine, 315
- HP TACL commands
 - description of, 520
 - ASSIGN see ASSIGN command
 - DEFINE see DEFINES
 - RUN, 357
- Hyphen (-)
 - followed by hyphen (-), 38
 - followed by right angle bracket (>), 39

I

- Identifiers
 - description of, 42
 - classes of, 42
 - listing
 - with GMAP directive, 397
 - with MAP directive, 402
 - with PRINTSYM directive, 408
 - saving, 421
- IF and IFNOT directives, 398
- IF expressions, 87
- IF statement
 - See also Conditional expressions, 217
 - description of, 217
 - hardware indicators in, 239
- IFIX routine, 316
- Implicit address conversion, 52
- Implied decimal point
 - ignoring
 - with \$FIXD routine, 310
 - with \$FIXI routine, 312
 - with \$FIXL routine, 313
 - in data type declarations, 48
 - in formal parameters, 253
 - in simple variable declarations, 104
 - moving, 340
 - obtaining
 - with \$DFIX routine, 302
 - with \$IFIX routine, 316
 - with \$LFIX routine, 323
 - parentheses and, 38
- IN file option, 357
- Indexes, accessing array elements with, 54
- Indirection symbols, 41
- Initialization
 - of exported data, 394
 - of read-only arrays, 112
 - of simple pointers, 172
 - of structure pointers, 174
 - scope and, 44
- INNERLIST directive, 400
- Input files, 356
- Instruction codes, listing, 400
- INT data type
 - \$INTR routine and, 321
 - \$IS_32BIT_ADDR routine, 321
 - constants of, 58

- converting, 53
- functions that return values of, 252
- high-order word of, 315
- parameters of, 251, 252
- rounding and, 283
- signed value of, 312
- unsigned value of, 312

- INT routine, 317
- INT(16) data type see INT data type
- INT(32) address type
 - bitwise logical operators and, 76
 - converting, 53
 - obtaining
 - with \$DBLL routine, 301
 - with \$DBLR routine, 301
 - with \$UDBL routine, 346
 - unsigned operators and, 76
- INT(32) data type
 - constants of, 59
 - rounding and, 283
- INT(64) data type see INT data type
- INT_OV routine, 318
- Internal file names, 520
- INTERROGATEHIO routine, 319
- INTERROGATEIO routine, 320
- INTERRUPT procedure attribute, 248, 249
- INTR routine, 321
- INVALID_FOR_PTAL directive, 401
- IS_32BIT_ADDR routine, 321
- Itanium architecture see TNS/E architecture

K

- Keywords
 - description of, 37
 - in syntax diagrams, 21
 - nonreserved, 38
 - reserved, 37

L

- Labeled CASE statement, 207
- Labels
 - address types of, 169
 - dropping, 212
 - in procedures, 273
- LAND operator
 - \$ATOMIC_AND routine and, 277
 - \$ATOMIC_DEP routine and, 279
 - in arithmetic expression, 72
 - operand types for, 76
 - precedence of, 71
 - with INT(32) operands, 76
- LANGUAGE procedure attribute, 248
- ld utility, 430
- Least significant byte, 46
- LEN routine, 322
- Length parameters
 - in CALL statements, 206
 - in declarations
 - procedure, 247

- procedure pointer, 265
 - subprocedure, 258
- passing conditionally, 333
- Less than operator
 - signed (<)
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - without operands, 84
 - unsigned ('<')
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - with INT(32) operands, 76
 - without operands, 84
- Less than or equal operator
 - signed (<=)
 - in conditional expression, 83
 - operand types for, 84
 - precedence of, 71
 - without operands, 85
 - unsigned ('<=')
 - in conditional expression, 83
 - operand types for, 84
 - with INT(32) operands, 76
 - without operands, 85
 - unsigned ('\<=')
 - precedence of, 71
- LFIX routine, 323
- Libraries
 - user, 420
- LINES directive, 401
- Linking
 - description of, 358
- Linking:cross compilers and, 429
- LIST directive
 - description of, 402
 - SOURCE directive and, 418
- LITERAL declarations, 97
- LMAX routine, 323
- LMIN routine, 324
- Local GOTO statement, 215
- Local scope, 43
- LOCATESPTHDR routine, 324
- LOCKPAGE routine, 325
- Logical file names
 - ASSIGN command and, 522
 - compiler directives that accept, 368
 - in place of disk file names, 520
- Logical operators
 - bitwise, 76
 - in arithmetic expressions, 72
 - with INT(32) operands, 76
- longjmp() instruction, 114
- Loops
 - FOR see FOR statement
- Loops:WHILE:See WHILE statement, 214
- LOR operator
 - \$ATOMIC_DEP routine and, 279

- \$ATOMIC_OR routine and, 280
 - in arithmetic expression, 72
 - operand types for, 76
 - precedence of, 71
 - with INT(32) operands, 76

M

- MAIN procedure attribute, 248
- MAP DEFINE, 521
- MAP directive, 402
- MAX routine, 326
- MAXALIGN attribute, 141
- Maximum routines;Routines
 - minimum, 285
- Messages, error see Error messages
- MIN routine, 327
- Minimum routines, 285
- Minus sign (-)
 - as subtraction operator see Subtraction operator
 - as unary operator
 - operand types for, 73
 - precedence of, 70
 - syntax of, 72
- MISALIGNLOG attribute (SCF)
 - misalignment handling and, 67
 - misalignment tracing facility and, 66
- Misalignment see Address misalignment
- Mnemonics, listing, 400
- Modular programming, 34
- Most significant byte, 46
- Move statement, 218
- MOVEANDCXSUMBYTES routine, 327
- MOVENONDUP routine, 328
- Multiplication operator
 - signed (*)
 - description of, 40
 - in arithmetic expression, 72
 - operand types for, 73
 - precedence of, 70
 - unsigned ('*')
 - description of, 40
 - operand types for, 75
 - precedence of, 71
 - with INT(32) operands, 76

N

- NAME declarations, 363
- Named toggles, 370
- Naming compilation units, 363
- NATIVEATOMICMISALIGN attribute (SCF), 67
- Nesting condition codes, 242
- Next address
 - in move statement, 218
 - in RSCAN statement, 228
 - in SCAN statement, 228
- nld utility, 430
- Node names, 519
- NOname directive see name directive
- Nonatomic access, 67

- Nonatomic operations, [281](#)
- Nonextended addresses, [78](#)
- Nonlocal GOTO statement, [215](#)
- Nonreserved keywords, [38](#)
- NonStop EpTAL, [426](#)
- NonStop operating systems, [31](#)
- NonStop pTAL, [426](#)
- NonStop Series see TNS architecture
- NonStop Series/Itanium see TNS/E architecture
- NonStop Series/RISC see TNS/R architecture
- Nonstring arrays, [113](#)
- NOOVERFLOW_TRAPS procedure attribute
 - description of, [234](#)
 - in procedure, [248](#), [250](#)
 - in subprocedure, [257](#), [259](#)
- Not equal operator
 - signed (<>)
 - in conditional expression, [83](#)
 - operand types for, [84](#)
 - precedence of, [71](#)
 - without operands, [84](#)
 - unsigned ('<>')
 - in conditional expression, [83](#)
 - operand types for, [84](#)
 - precedence of, [71](#)
 - with INT(32) operands, [76](#)
 - without operands, [84](#)
- NOT operator
 - description of, [82](#)
 - condition codes and, [83](#)
 - in conditional expression, [81](#)
 - operand types for, [83](#)
 - precedence of, [71](#)
 - truth table for, [81](#)
- Null statement, [199](#)
- Numbers, converting to data addresses, [51](#)
- NUMERIC routine, [329](#)
- Numeric toggles, [370](#)

O

- OBJECT file, [358](#)
- Object files
 - creating, [358](#)
 - generating, [357](#)
 - linking, [358](#)
- Object-file content directives, [377](#)
- OCCURS routine, [330](#)
- Odd-byte references, [172](#), [174](#)
- OF keyword
 - in labeled CASE statement, [207](#)
 - in unlabeled CASE statement, [209](#)
- OFFSET routine
 - description of, [332](#)
 - structure pointers and, [175](#)
- Online help for cross compilers, [431](#)
- Operands
 - in arithmetic expressions, [72](#)
 - scaling FIXED, [74](#)
- Operating systems, [31](#)

- Operations
 - See also Operators, [34](#)
 - atomic
 - See Atomic operations, [276](#)
 - bit see Bit operations
 - data, [34](#)
 - listed by data type, [48](#)
 - nonatomic
 - See Nonatomic operations, [281](#)
- Operators
 - description of, [39](#)
 - AND
 - description of, [82](#)
 - condition codes and, [83](#)
 - arithmetic see Arithmetic operators
 - concatenation (&), [220](#)
 - logical
 - description of, [76](#)
 - in arithmetic expressions, [72](#)
 - NOT
 - description of, [82](#)
 - condition codes and, [83](#)
 - OR
 - description of, [82](#)
 - condition codes and, [83](#)
 - precedence of, [70](#)
 - relational see Relational operators
 - signed
 - See Signed operators, [73](#)
 - unsigned
 - See Unsigned operators, [75](#)
- OPTIMIZE directive, [404](#)
- OPTIMIZEFILE directive, [404](#)
- Optional parameters, [333](#)
- OPTIONAL routine, [333](#)
- OR operator
 - description of, [82](#)
 - condition codes and, [83](#)
 - in conditional expression, [82](#)
 - operand types for, [83](#)
 - precedence of, [71](#)
 - truth table for, [82](#)
- OTHERWISE keyword
 - in labeled CASE statement, [207](#)
 - in unlabeled CASE statement, [209](#)
- OUT file option, [357](#)
- Output files, [356](#)
- Overflow
 - managing
 - generally, [234](#)
 - GOTO statement and, [216](#)
 - testing, [335](#)
- OVERFLOW routine
 - description of, [335](#)
 - after assignments, [236](#)
 - atomic operation that can set, [276](#)
 - in nested IF statements, [243](#)
 - nonatomic operations that can set, [286](#)
 - returning its value to calling procedure, [245](#)

OVERFLOW_TRAPS directive, 406
OVERFLOW_TRAPS procedure attribute
 description of, 234
 in procedure, 248, 250
 in subprocedure, 257, 259

P

P-relative arrays see Read-only arrays

PAGE directive, 407

Page heading, 407

PARAM routine, 336

PARAM SWAPVOL command, 522

Parameters

 See also Arguments, 275

 actual see Actual parameters

 extended, 255

 of built-in routines, 275

 optional, 333

 referencing, 256

Parameters:formal:See Formal parameters, 206

Parentheses

 as delimiters, 38

 implied decimal point and, 38

 operator precedence and, 71

Partial file names

 description of, 519

 ASSIGN SSV command and, 523

PASCAL procedure attribute, 248

PC-to-NonStop host transfer tools, 431

Period (.)

 in bit-deposit assignment statement, 204

 in formal parameters, 251

 in pointers

 simple, 170

 structure, 173

 in structure item identifiers, 38

 in structures

 equivalenced definition, 194

 referral, 141

PIC see Position-independent code (PIC)

PLATFORM parameter

 description of, 118

 compared to AUTO parameter, 119

 FIELDALIGN clause and, 117

Plus sign (+)

 as addition operator

 See Addition operator, 71

 as unary operator

 in arithmetic expression, 72

 operand types for, 73

 precedence of, 70

POINT routine, 336

Pointers

 description of, 34

 address types stored in, 164

 allocation of, 161

 assignment statements with, 203

 declaring

 overview, 161

 procedure see Procedure pointers, declaring
 system global:See System global data, declaring,
 pointers, 176

 VOLATILE, 163

 procedure see Procedure pointers

 simple see Simple pointers

 structure see Structure pointers

 testing for nonzero values, 56

Pointers:declaring:simple:See Simple pointers, declaring,
196

Pointers:declaring:structure:See Structure pointers,
declaring, 197

Pointers:stepping;Stepping pointers, 54

POPname directive see name directive

Position-independent code (PIC), 383

Pound sign (#);# (hash mark or pound sign), 39

Precedence of operators, 70

PRINTSYM directive, 408

PRIV procedure attribute, 248

Private data area, 32

PRIVATE keyword, 364

Privileged mode, 274

Privileged routines, 281

PROC address type, 251

PROC keyword, 246

PROC(32) address type, 251

PROC32ADDR address type
 description of, 165

PROC32ADDR routine, 337

PROC64ADDR routine, 338

PROCADDR address type
 description of, 165
 comparing to PROCPTR, 56
 converting, 53
 parameters of, 251
 pointers of, 168

PROCADDR routine, 337

Procedure calls (CALL statement), 205

Procedure entry points, 168

Procedure pointers

 description of, 263

 address types of, 168

 assignments to, 269

 comparing to data addresses, 56

 declaring

 as formal parameters, 268

 as variables, 266

 in structures, 267

 for dynamically selected procedure calls, 271

Procedure-parameter routines, 286

Procedures

 description of, 32, 246

 address types of, 168

 as parameters, 254

 attributes of, 248

 bodies of, 256

 C/C++, 528

 callable, 274

 converting from variable to extensible, 250

- declaring, [246](#)
- dynamically selected calls to, [271](#)
- extensible, [250](#)
- EXTERNAL declaration of, [246](#), [247](#)
- formal parameter specification in, [251](#)
- FORWARD declaration of, [246](#), [247](#)
- labels in, [273](#)
- languages of, [250](#)
- main, [248](#)
- resident, [249](#)
- scope of, [43](#)
- system, [34](#)
- that return condition codes, [225](#), [250](#)
- typed
 - See Functions, [246](#)
- using hardware indicators across, [244](#)
- variable, [249](#)
- with RETURN statements, [223](#)
- with two return values, [528](#)

PROCPTRs see Procedure pointers

PROFDIR directive, [408](#)

PROFGEN directive, [409](#)

Profile-guided optimization, [366](#)

PROFUSE directive, [409](#)

Program control statements, [199](#)

pTAL language

- applications, [31](#)
- character set for, [36](#)
- compatibility with TAL, [30](#)
- elements of, [36](#)
- features of, [32](#)
- services for, [34](#)
- syntax of see Syntax

Punctuation characters in syntax diagrams, [21](#)

PUSHname directive:See name directive, [367](#)

Q

- Quadruplewords, [46](#)
- Question mark (?), [39](#)
- Quotation mark ("), [39](#)
- See also Single quotation mark ('), [39](#)

R

- Read-only arrays
 - address types of, [169](#)
 - constant lists in, [113](#)
 - declaring, [111](#)
- READBASELIMIT routine, [338](#)
- READCLOCK routine, [339](#)
- READSPT routine, [339](#)
- READTIME routine, [340](#)
- REAL data type
 - numeric constants of, [62](#)
 - obtaining
 - with \$FLTR routine, [314](#)
 - with \$FLTroutine, [314](#)
 - parameters of, [251](#)
- REAL data type:functions that return values of;UNSIGNED data type:functions that return values of;FIXED data type:functions that return values of, [252](#)
- REAL(32) data type see REAL data type
- REAL(64) data type
 - numeric constants of, [62](#)
 - obtaining
 - with \$EFLT routine, [302](#)
 - with \$EFLTR routine, [303](#)
- Records see Structures
- Recursion, [33](#)
- Redefinitions
 - array, [154](#)
 - pointer
 - simple, [158](#)
 - structure, [159](#)
 - rules for, [153](#)
 - simple variable, [153](#)
 - substructure
 - definition, [155](#)
 - referral , [157](#)
- REALIGNED clause
 - with simple equivalenced pointers, [187](#)
 - with structure pointers, [134](#)
- REALIGNED directive, [410](#)
- Referral structures, declaring
 - equivalenced, [195](#)
 - not equivalenced, [141](#)
- Referral substructures
 - declaring, [146](#)
 - redefining, [157](#)
- Relational operators
 - in conditional expressions, [83](#)
 - signed
 - in address comparisons, [77](#)
 - operand types for, [83](#)
 - precedence of, [71](#)
 - unsigned
 - in address comparisons, [77](#)
 - operand types for, [84](#)
 - precedence of, [71](#)
 - with INT(32) operands, [76](#)
 - with extended addresses, [77](#)
 - with nonextended addresses, [78](#)
- Relocatable data blocks:See Global data, blocked, [362](#)
- Remainder operator ('\\')
 - in arithmetic expression, [72](#)
 - operand types for, [75](#)
 - precedence of, [71](#)
 - result types for, [75](#)
 - with INT(32) and FIXED operands, [76](#)
- Reserved keywords, [37](#)
- RESETTOG directive, [411](#)
- RESIDENT procedure attribute, [248](#), [249](#)
- RETURN statement, [223](#)
- RETURNSCC procedure attribute
 - for procedures, [248](#), [250](#)
 - for subprocedures, [257](#), [258](#)
- RISC see TNS/R architecture

ROUND (default) misalignment handling method, 67

ROUND directive, 412

Rounding

expressions unaffected by, 283

ROUND directive and, 412

type-conversion routines and, 283

Routines

See also Functions, 274

address-conversion, 283

arithmetic, 285

built-in, 274

character-test, 284

maximum, 285

miscellaneous built-in, 286

procedure-parameter, 286

pTAL privileged, 281

type-conversion, 282

variable-characteristic, 285

RSCAN statement, 228

Run-time environment directives, 379

S

SAVEGLOBALS directive, 372, 413

SCALE routine, 340

Scaling FIXED values

by specifying fpoint, 74

in assignment statements, 203

with \$SCALE routine, 340

SCAN statement, 228

SCF user interface

attributes of, 67

misalignment handling and, 67

misalignment tracing facility and, 66

Scope of declared items, 43

Search subvolume (SSV) command, 523

SECTION directive

description of, 414

global data blocks and, 365

SOURCE directive and, 417

Section names, 414

Segment Page Table (SPT)

address of, 324

copying an entry from, 339

Selector

in labeled CASE statement, 207

in unlabeled CASE statement, 209

Semicolon (\, 38, 199

Services

pTAL, 34

system, 34

Services:CRE;CRE services;Common run-time environment

(CRE) services, 34

setjmp() instruction, 114

SETTOG directive, 415

SGBADDR address type

description of, 165

converting, 53

parameters of, 251

pointers of, 167

SGBADDR_TO_EXTADDR routine, 341

SGBADDR_TO_SGWADDR routine, 342

SGWADDR address type

description of, 165

converting, 53

parameters of, 251

pointers of, 167

SGWADDR_TO_EXTADDR routine, 342

SGWADDR_TO_SGBADDR routine, 343

SGXBADDR address type

description of, 165

converting, 53

parameters of, 251

pointers of, 167

SGXWADDR address type

description of, 165

converting, 53

parameters of, 251

pointers of, 167

Shared code

See Position-independent code (PIC), 383

SHARED2 parameter

description of, 117, 128

FIELDALIGN clause and, 117

SHARED8 parameter

description of, 118, 129

FIELDALIGN clause and, 117

Shifting bits

description of, 94

precedence of operators for, 70

Short-circuit expression evaluation, 83

SIGILL signal (signal #4), 67

Signed operators

arithmetic, 73

bit shift, 70

relational, 83

Simple pointers

description of, 161

addresses in, 174

as parameters, 254

declaring

equivalenced, 183

not equivalenced, 170

equivalenced, 183

initializing, 172

redefining, 158

using, 149

VOLATILE, 163

within structures, 148

Simple variables

as parameters, 254

data type of, 345

declaring

equivalenced, 182

not equivalenced, 103

equivalenced, 193

length of

in bits, 295

in bytes, 322

- redefining, 153
 - within structures, 142
- Single quotation mark ('), 39
- sINT, 274
- Slash (/) see Division operator
- Smear operation, 222, 308
- Source code listing, 402
- SOURCE directive
 - description of, 416
 - global data blocks and, 365
 - NOLIST directive and, 418
 - system procedure declarations and, 419
- Source files
 - checking syntax of, 422
 - compiling, 355
 - listing, 402
- Spacing rules in syntax diagrams, 21
- Special expressions, 85
- SPECIAL routine, 343
- SPT, 324
- SPT (Segment Page Table)
 - address of, 324
 - copying an entry from, 339
- SQL/MP or SQL/MX in pTAL, 32
- Square brackets ([\]), 38
- SRL directive, 420
- STACK_ALLOCATE routine, 344
- Stacks, directive
 - See Directive stacks, 369
- Standard functions see Built-in routines
- Statements
 - categories of, 199
 - compound see Compound statements
 - null, 199
 - role in program, 45
- Static T flag, 234
- Storage units, 46
- STRING data type
 - functions that return values of, 252
 - numeric constants of, 58
 - parameters of
 - actual:passed conditionally, 334
 - actual:passed unconditionally, 206
 - formal:for procedure pointers, 265
 - formal:for procedures, 247, 251, 252
 - formal:for subprocedures, 251, 252, 258
- STRUCT data type, 251, 252, 255
- STRUCT keyword
 - in structures
 - definition, 138
 - referral, 141
 - template, 139
 - in substructures
 - definition: redefined, 155
 - definition: not redefined, 144
 - referral: not redefined, 146
 - referral: redefined, 157
- STRUCTALIGN (MAXALIGN) attribute, 137, 140
- STRUCTALIGN clause, 141

- Structure items
 - arrays, 143
 - filler bits or bytes, 147
 - offsets of
 - in bits, 296
 - in bytes, 332
 - pointers
 - simple, 148
 - structure, 151
 - procedure pointers as, 263
 - simple variables, 142
 - substructures
 - definition, 144
 - referral, 146
- Structure pointers
 - description of, 161
 - addresses in, 174
 - as parameters, 254
 - declaring, 173
 - initializing, 174
 - redefining, 159
 - reference alignment with, 134
 - VOLATILE, 164
 - within structures, 151
- Structures
 - description of, 114
 - alignment of
 - description of, 116
 - arrays in, 122
 - base, 119
 - fields of, 117
 - in depth, 123
 - as parameters, 254, 255
 - data type of, 345
 - declaring
 - definition: equivalenced, 194
 - definition: not equivalenced, 138
 - referral: equivalenced, 195
 - referral: not equivalenced, 141
 - template, 139
 - items within see Structure items
 - layout of, 115
 - length of
 - in bits, 295
 - in bytes, 322
 - maximum nesting levels in, 115
 - number of occurrences of, 330
 - redefining, 153
- Sublocal declarations, 259
- Sublocal scope, 43
- SUBPROC keyword, 257
- Subprocedure entry points, 169
- Subprocedures
 - See also Functions, 246
 - description of, 32
 - address types of, 169
 - bodies of, 259
 - declaring, 257
 - formal parameter specification in, 251

- FORWARD declaration of, 259
- sublocal declarations in, 259
- that return condition codes, 258
- variable, 258
- with RETURN statements, 223
- Substructures
 - alignment of, 124
 - data type of, 345
 - declaring
 - definition, 144
 - referral, 146
 - length of
 - in bits, 295
 - in bytes, 322
 - number of elements of, 330
 - redefining
 - definition, 155
 - referral, 157
- Subsystem Control Facility *see* SCF user interface
- Subtraction operator
 - signed (-)
 - in arithmetic expression, 72
 - operand types for, 73
 - precedence of, 71
 - unsigned ('-')
 - in arithmetic expression, 72
 - operand types for, 75
 - precedence of, 71
 - result types for, 75
 - with INT(32) operands, 76
- Subvolume names, 519
- SUPPRESS directive, 420
- Swap volume, 522
- SWAPVOL command, 522
- SYMBOLS directive, 421
- Syntax
 - checking, 422
 - conventions for, 20
 - summary of, 432
- SYNTAX directive, 422
- System clock setting, 339
- System global data
 - See also* Global data, 362
 - declaring
 - equivalenced, 193
 - pointers, 176
 - pointers to, 167
- System names, 519
- System procedures
 - description of, 34
 - SOURCE directive and, 419
- System services, 34

T

- TACL commands *see* HP TACL commands
- TACL DEFINE tool, 431
- TAL
 - compatibility with pTAL, 30
 - procedures that return two values, 528

- TARGET directive, 423
- Target file option, 358
- Template structures, declaring, 139
- Temporary files, 522
- Temporary variables
 - creating, 232
 - dropping, 212
- THEN keyword, 217
- TNS architecture RVUs;TNS/R architecture RVUs;TNS/E architecture RVUs, 31
- TNS/R native mode, 67
- TNSMISALIGN attribute (SCF), 67
- TO keyword, 212
- Toggles
 - description of, 370
 - turning off, 411
 - turning on, 415
- Tracing facility, 66
- Transfer Tool, 431
- Traps, managing
 - generally, 234
 - GOTO statement and, 216
- TRIGGER routine, 345
- TYPE routine, 345
- Type-conversion routines, 282
- Typed procedures *see* Functions

U

- UDBL routine, 346
- UDIVREM16 routine, 347
- UDIVREM32 routine, 348
- UFX routine, 349
- uINT, 274
- Unlabeled CASE statement, 209
- UNLOCKPAGE routine, 349
- UNSIGNED data type
 - parameters of, 251
- Unsigned operators
 - arithmetic, 75
 - bit shift, 70
 - relational, 84
- UNSPECIFIED procedure attribute, 248
- UNTIL keyword
 - in DO statement, 210
 - in RSCAN statement, 228
 - in SCAN statement, 228
- USE statement, 232
- USEGLOBALS directive
 - description of, 423
 - SAVEGLOBALS and BEGINCOMPILE and, 373
 - SOURCE directive and, 419
- User library, 420

V

- VARIABLE procedure attribute
 - for procedures, 248, 249
 - for subprocedures, 257, 258
- Variable-characteristic routines, 285
- VARIABLE-to-EXTENSIBLE procedure conversions, 250

Variables

- description of, [43](#)
- equivalenced *see* Equivalenced variables
- FIXED, [74](#)
- procedure pointers as, [263](#)
- scope of, [43](#)
- simple *see* Simple variables
- storing data addresses in, [51](#)
- temporary
 - creating, [232](#)
 - dropping, [212](#)
- types of, [43](#)

Visual Studio .NET, [426](#)

VOLATILE pointers

- simple, [163](#)
- structure, [164](#)

VOLATILE procedure attribute, [138](#)

Volume names, [519](#)

W

WADDR address type

- description of, [165](#)
- converting, [53](#)
- parameters of, [251](#)
- pointers of, [167](#)

WADDR_TO_BADDR routine, [350](#)

WADDR_TO_EXTADDR routine, [350](#)

WARN directive, [424](#)

Warning messages, [424](#)

WHILE keyword

- in RSCAN statement, [228](#)
- in SCAN statement, [228](#)
- in WHILE statement, [232](#)

WHILE statement

- description of, [232](#)
- hardware indicators in, [240](#)

Words, [46](#)

WRITEPTE routine, [351](#)

X

XADR routine, [352](#)

XADR32 routine, [352](#)

XADR64 routine, [353](#)

XOR operator

- in arithmetic expression, [72](#)
- operand types for, [76](#)
- precedence of, [71](#)
- with INT(32) operands, [76](#)

Z

ZZBlnnnn target file, [358](#)

