R145-01

VOS PL/I User’s Guide

Stratus Computer, Inc.

Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS COMPUTER, INC., STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Computer, Inc., assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents a) is the property of Stratus Computer, Inc., or the third party, b) is furnished only under license, and c) may be copied or used only as expressly permitted under the terms of the license.

Stratus manuals document all of the subroutines and commands of the user interface. Any other operating-system commands and subroutines are intended solely for use by Stratus personnel and are subject to change without warning.

This document is protected by copyright. All rights are reserved. No part of this document may be copied, reproduced, or translated, either mechanically or electronically, without the prior written consent of Stratus Computer, Inc.

Stratus, the Stratus logo, Continuum, StrataNET, FTX, and SINAP are registered trademarks of Stratus Computer, Inc.

XA, XA/R, StrataLINK, RSN, Continuous Processing, Isis, the Isis logo, Isis Distributed, Isis Distributed Systems, RADIO, RADIO Cluster, and the SQL/2000 logo are trademarks of Stratus Computer, Inc.

Apple and Macintosh are registered trademarks of Apple Computer, Inc.
IBM PC is a registered trademark of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
MC68000 and Motorola are registered trademarks of Motorola, Incorporated.
Sun is a registered trademark of Sun Microsystems, Inc.
Hewlett-Packard is a trademark of Hewlett-Packard Company.
i860 is a trademark of Intel Corporation.
All other trademarks are the property of their respective owners.

Manual Name: *VOS PL/I User's Guide*

Part Number: R145
Revision Number: 01
VOS Release Number: 14.0.0
Printing Date: January 1998

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

© 1998 by Stratus Computer, Inc. All rights reserved.

Preface

The *VOS PL/I User's Guide (R145)* documents how to compile, bind, and debug a VOS PL/I program. It also describes how to use VOS PL/I to perform file I/O and to call programs written in other VOS languages.

This manual documents VOS PL/I for VOS Release 14.0.0.

This manual is intended for experienced application programmers who may or may not be knowledgeable PL/I programmers.

Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual. Note, however, that change bars are not used in new chapters or appendixes.

This revision incorporates the following changes related to the new PA-8000 processor.

- new `-processor` values for the `pl1` and `bind` commands
- new predefined preprocessor values

In addition, documentation errors from the previous version of this manual have been corrected, and customer suggestions have been incorporated.

VOS Release 14.0.0 does **not** support XA2000-series modules. However, to maintain cross-development on networked XA2000-series modules and to support firmware running on UCOMM line adapter cards, all VOS compilers still support cross-compiling and cross-binding for XA2000-series modules from XA/R-series modules and Continuum-series modules.

Manual Organization

This manual contains eight chapters and one appendix.

Chapter 1 provides an overview of the steps used to prepare and execute a PL/I program.

Chapter 2 explains how to compile a source module using the `pl1` command and its arguments.

Chapter 3 describes how the compiler optimizes object code. This chapter includes a description of each optimization.

Chapter 4 explains how to preprocess source modules.

Chapter 5 explains how to bind object modules using the `bind` command and its arguments.

Chapter 6 explains how to debug a program module using the `debug` command and its requests. This chapter also explains how to use the `mp_debug` command.

Chapter 7 describes how to perform certain file I/O tasks in the VOS environment.

Chapter 8 explains how to call programs written in languages other than VOS PL/I.

Appendix A provides a table of the VOS internal character code set.

Related Manuals

Refer to the following Stratus manuals for related documentation.

- *VOS PL/I Language Manual (R009)*
- *VOS PL/I Subroutines Manual (R005)*
- *VOS PL/I Transaction Processing Facility Reference Manual (R015)*
- *VOS Transaction Processing Facility Guide (R215)*
- *VOS PL/I Forms Management System (R016)*
- *Introduction to VOS (R001)*
- *VOS Commands Reference Manual (R098)*

Notation Conventions

This manual uses the following notation conventions.

- Italics introduces or defines new terms. For example:

  > The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

  > Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

  ```
  change_current_dir (master_disk)>system>doc
  ```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  ```
  list_users -module module_name
  ```

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

```
display_access_list system_default

%dev#m1>system>acl>system_default

w   *.*
```

Key Mappings for VOS Functions

VOS provides several command-line and display-form functions. Each function is mapped to a particular key or combination of keys on the terminal's keyboard. To perform a function, you press the appropriate key(s) from the command line or display form. For an explanation of the command-line and display-form functions, see the *Introduction to VOS (R001).*

The keys that perform specific VOS functions vary depending on the terminal. For example, on a V103 ASCII terminal, you press the Shift and F20 keys simultaneously to perform the `INTERRUPT` function; on a V105 PC/+ 106 terminal, you press the 1 key on the numeric keypad to perform the `INTERRUPT` function.

> **Note:** Certain applications may define these keys differently. Refer to the documentation for the application for the specific key mappings.

The following table lists several VOS functions and the keys to which they are mapped on commonly used Stratus terminals and on an IBM PC® or compatible PC that is running the Stratus PC/Connect-2 software. (If your PC is running another type of software to connect to a Stratus host computer, the key mappings may be different.) For information about the key mappings for a terminal that is not listed in this table, refer to the documentation for that terminal.

VOS Function	V103 ASCII	V103 EPC	IBM PC or Compatible PC	V105 PC/+ 106	V105 ANSI
`CANCEL`	F18	* †	* †	5 † or * †	F18
`CYCLE`	F17	F12	Alt-C	4 †	F17
`CYCLE BACK`	Shift-F17	Shift-F12	Alt-B	7 †	Shift-F17
`DISPLAY FORM`	F19	- †	- †	6 † or - †	F19 or Shift-Help
`HELP`	Shift-F8	Shift-F2	Shift-F2	Shift-F8	Help
`INSERT DEFAULT`	Shift-F11	Shift-F10	Shift-F10	Shift-F11	F11
`INSERT SAVED`	F11	F10	F10	F11	Insert_Here
`INTERRUPT`	Shift-F20	Shift-Delete	Alt-I	1 †	Shift-F20
`NO PAUSE`	Shift-F18	Shift-* †	Alt-P	8 †	Shift-F18

† Numeric-keypad key

Syntax Notation

A *language format* shows the syntax of a VOS PL/I statement, portion of a statement, declaration, or definition. When VOS PL/I allows more than one format for a language construct, the documentation presents each format consecutively. For complex language constructs, the text may supply additional information about the syntax.

The following table explains the notation used in language formats.

The Notation Used in Language Formats

Notation	Meaning
`element`	Required element.
`element``...`	Required element that can be repeated.
`{element_1 element_2}`	List of required elements.
`{element_1 element_2}...`	List of required elements that can be repeated.
{ *`element_1`* *`element_2`* }	Set of elements that are mutually exclusive; you must specify one of these elements.

Notation	**Meaning**
`[element]`	Optional element.
`[element]...`	Optional element that can be repeated.
`[element_1 element_2]`	List of optional elements.
`[element_1 element_2]...`	List of optional elements that can be repeated.
`[` *`element_1`* *`element_2`* `]`	Set of optional elements that are mutually exclusive; you can specify only one of these elements.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of elements can contain more than two elements. Brackets and braces are sometimes nested.	

In the preceding table, *`element`* represents one of the following VOS PL/I language constructs.

- keywords (which appear in monospace)
- generic terms (which appear in monospace italic) that are to be replaced by items such as expressions, identifiers, literals, constants, or statements
- statements or portions of statements
- elements of a binder control file

The elements in a list of elements must be entered in the order shown, unless the text specifies otherwise. An element or a list of elements followed by a set of three dots indicates that the element(s) can be repeated.

The following example shows a sample language format.

```
(module_term...)[module_attribute]...
```

In examples, a set of three vertically aligned dots indicates that a portion of a language construct or program has been omitted. The following example illustrates this concept.

```
top:
          .
          .
          .
          goto next;
```

Format for Commands and Requests

Stratus manuals use the following format conventions for documenting commands and requests. (A *request* is typically a command used within a subsystem, such as `analyze_system`.) Note that the command or request descriptions do not necessarily include each of the following sections.

`name`

The name of the command or request is at the top of the first page of the description.

Privileged

This notation appears after the name of a command or request that can be issued only from a privileged process. (See the Glossary for the definition of privileged process.)

Purpose

Explains briefly what the command or request does.

Display Form

Shows the form that is displayed when you type the command or request name followed by `-form` or when you press the key that performs the `DISPLAY FORM` function. Each field in the form represents a command or request argument. If an argument has a default value, that value is displayed in the form. (See the Glossary for the definition of default value.)

The following table explains the notation used in display forms.

The Notation Used in Display Forms

Notation	Meaning
█████	Required field with no default value.
▌	The cursor, which indicates the current position on the screen. For example, the cursor may be positioned on the first character of a value, as in `all`.
`current_user` *`current_module`* *`current_system`* *`current_disk`*	The default value is the current user, module, system, or disk. The actual name is displayed in the display form of the command or request.

Command-Line Form

Shows the syntax of the command or request with its arguments. You can display an online version of the command-line form of a command or request by typing the command or request name followed by `-usage`.

The following table explains the notation used in command-line forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Specifying multiple values is **not** the same as specifying a star name. (See the Glossary for the definition of star name.) When you specify multiple values, you must separate each value with a space.

The Notation Used in Command-Line Forms

Notation	Meaning
`argument_1`	Required argument.
`argument_1...`	Required argument for which you can specify multiple values.
{ `argument_1` `argument_2` }	Set of arguments that are mutually exclusive; you must specify one of these arguments.
[`argument_1`]	Optional argument.
[`argument_1`]`...`	Optional argument for which you can specify multiple values.
[`argument_1` `argument_2`]	Set of optional arguments that are mutually exclusive; you can specify only one of these arguments.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of arguments can contain more than two elements. Brackets and braces are sometimes nested.	

Arguments

Describes the command or request arguments. The following table explains the notation used in argument descriptions.

The Notation Used in Argument Descriptions

Notation	Meaning
CYCLE	There are predefined values for this argument. In the display form, you display these values in sequence by pressing the key that performs the `CYCLE` function.
Required	You cannot issue the command or request without specifying a value for this argument. If an argument is required but has a default value, it is not labeled **Required** since you do not need to specify it in the command-line form. However, in the display form, a required field must have a value—either the displayed default value or a value that you specify.
(Privileged)	Only a privileged process can specify a value for this argument.

Explanation

Explains how to use the command or request and provides supplementary information.

Access Requirements

Explains any special access requirements that may affect the operation or output of the command or request.

Examples

Illustrates uses of the command or request.

Related Information

Refers you to related information (in this manual or other manuals), including descriptions of commands, subroutines, and requests that you can use with or in place of this command or request.

Online Documentation

Stratus provides the following types of online documentation.

- The directory `>system>doc` provides supplemental online documentation. It contains the latest information available, including updates and corrections to Stratus manuals and a glossary of terms.
- Stratus offers some of its manuals online, via StrataDOC, an online-documentation product that consists of online manuals and StrataDOC Viewer, delivered on a CD-ROM (note that you must order StrataDOC separately). StrataDOC Viewer allows you to access online manuals from an IBM PC or compatible PC, a Sun® or Hewlett-Packard™ workstation, or an Apple® Macintosh® computer. StrataDOC provides such features as hypertext links and, on the workstations and PCs, text search and retrieval across the manual collection. The online and printed versions of a manual are identical.

If you have StrataDOC, you can view this manual online.

For a complete list of the manuals that are available online as well as more information about StrataDOC, contact your Stratus account representative.

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.
- Customers in North America can call the Stratus Customer Assistance Center (CAC) at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see the file `cac_phones.doc` in the directory `>system>doc` for CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

Commenting on This Manual

You can comment on this manual by using the command `comment_on_manual` or by completing the customer survey that appears at the end of this manual. To use the `comment_on_manual` command, your system must be connected to the RSN. If your system is **not** connected to the RSN, you must use the customer survey to comment on this manual.

The `comment_on_manual` command is documented in the manual *VOS System Administration: Administering and Customizing a System (R281)* and the *VOS Commands Reference Manual (R098)*. There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press Enter or Return, and complete the data-entry form that appears on your screen. When you have completed the form, press Enter.
- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press Enter or Return. Enter this manual's part number, `R145`, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press Enter.

  Note: If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail` request of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Figures

Tables

Chapter 1:
Overview: Programming in VOS PL/I

This chapter presents an overview of the steps used to prepare and execute a PL/I program. The chapter discusses the following topics.

- ‘‘Preparing a Program for Compilation”
- ‘‘Compiling a Source Module”
- ‘‘Binding Object Modules”
- ‘‘Executing and Interrupting a Program Module”

Figure 1-1 shows the sequence of steps you use to produce a program module and the files that these steps produce.

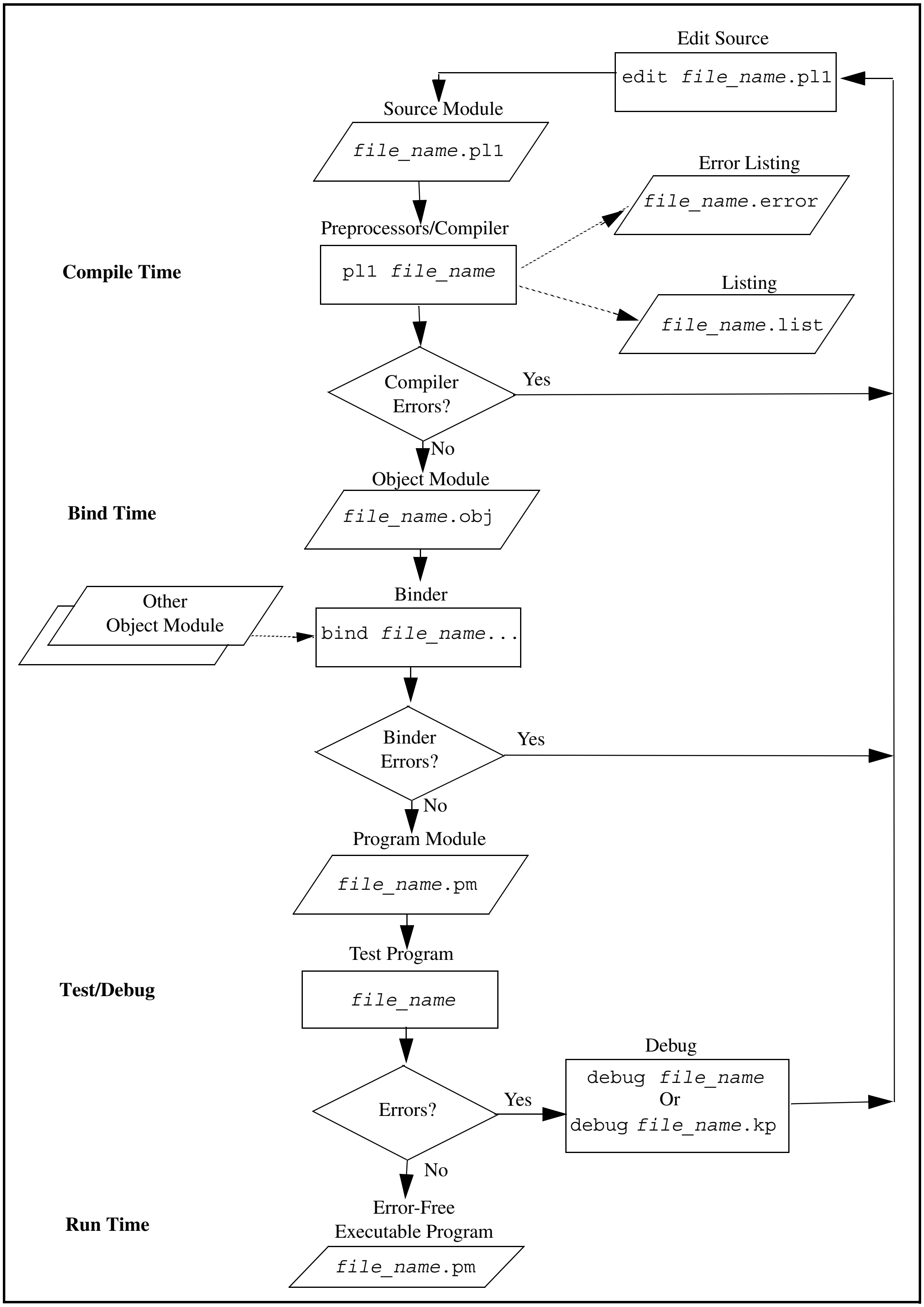


Figure 1-1. Program Development Process

Preparing a Program for Compilation

This section describes how you prepare a program for compilation.

The text of a PL/I program is kept in one or more text files. Each text file is called a *source module*. The file name of a VOS PL/I source module must have the suffix `.pl1`. You write and update source modules with an editor such as the VOS Word Processing Editor or Emacs. For information about the VOS implementation of the PL/I language, see the *VOS PL/I Language Manual (R009)*.

Compiling a Source Module

This section describes how you compile a source module to create an object module.

When you compile a source module, the compiler simultaneously invokes its preprocessors. The preprocessors process the text in the source module while the text passes through the compiler. (See Chapter 4 for more information about the preprocessors.) Next, the compiler translates the code in your program into object code and puts it in your current directory.

The command for compiling VOS PL/I programs is `pl1`. The `pl1` command has the following syntax.

```
pl1 file_name [argument ...]
```

You need not specify the `.pl1` suffix when you enter *`file_name`*.

The `pl1` command produces an object module. An *object module* contains object code and has the name of the source module with the `.obj` suffix in place of the `.pl1` suffix.

If the compilation generates any errors or warnings, the compiler creates an error file with the name of the source module and the suffix `.error`. When you recompile the source module, the compiler overwrites the object module it generated previously, if the file still exists with the same name in the current directory. Likewise, if errors still occur, the compiler overwrites the error file. However, if no errors occur, the compiler deletes the error file.

The following example compiles the source module in the file named `employee_info.pl1`.

```
pl1 employee_info
```

If the compilation is successful, the compiler creates an object module in the file `employee_info.obj`. If the compiler finds any errors, it creates an error file called `employee_info.error` and also displays the error messages on your terminal's screen.

You can enter one or more compiler arguments with the `pl1` command. For example, if you specify the `-list` argument, the compiler creates a listing of the source module in a file that has the name of the source module with the suffix `.list`. For more information about the `pl1` command, see Chapter 2.

Binding Object Modules

This section describes how you bind object modules to create a program module.

After you compile a source module, you must bind the object module or object modules before executing the program. When you use a binder control file, the binder invokes a preprocessor to process the binder control file. (See Chapter 5 for more information about the binder's preprocessor.) After preprocessing, the binder incorporates the following components into one executable program module.

- the object modules you specified
- any object modules required by other object modules being bound
- certain predefined VOS functions and procedures used by the program

The command for binding PL/I programs is `bind`. The `bind` command has the following syntax.

```
bind file_name ... [argument ...]
```

You need not specify the `.obj` suffix when you enter *`file_name`*.

The `bind` command produces a program module. A *program module* contains executable code and has the suffix `.pm`. Program modules are referred to as external commands because you can execute them from command level.

The following example binds the object module in a file named `employee_info.obj`.

```
bind employee_info
```

The preceding command creates a program module called `employee_info.pm`.

You can enter one or more binder arguments with the `bind` command. For example, if you specify the `-map` argument, the binder creates a bind map in a file with the suffix `.map`. For more information about the `bind` command and binder control files, see Chapter 5.

Executing and Interrupting a Program Module

This section describes how you execute a program module and interrupt execution.

To execute a program module named `employee_info.pm`, you type, at command level, the name of the module.

```
employee_info.pm
```

You can omit the `.pm` suffix if no command macro (`.cm`) file with the same name, such as `employee_info.cm`, exists in the search path. (If a command macro file with the same name does exist, it would execute instead of the program module.)

To interrupt the execution of the program module, issue the Ctrl-Break request by holding down the Ctrl key while pressing the Break key. The operating system suspends execution of your program and places the program's process at break level. The following prompt then appears on the terminal's screen.

```
BREAK
Request? (stop, continue, debug, keep, login, re-enter)
```

To stop the execution of a program, press `s` for the `stop` request. To resume the execution of a program, press `c` for the `continue` request. See the *Introduction to VOS (R001)* for information about the break-level requests.

Chapter 2:
Compiling a Source Module

The `pl1` command invokes the VOS PL/I compiler. The compiler simultaneously invokes its preprocessors. After the preprocessors finish processing the source module, the compiler reads the source module, checks it for errors, and produces an object module or an error file or both, depending on the presence and severity of errors in the source module. This chapter explains how to use the `pl1` command and its arguments.

This chapter discusses the following topics.

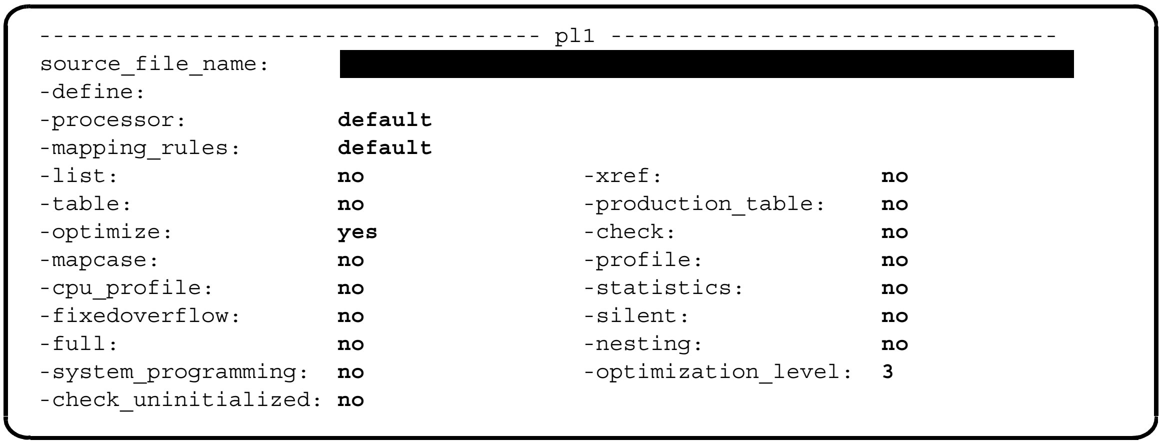
- ‘‘The `pl1` Command”
- ‘‘Summary of VOS PL/I Compiler Arguments”
- ‘‘Creating a Program Listing”
- ‘‘Interpreting Compiler Error Messages”
- ‘‘Checking for Additional Errors at Compile Time and Run Time”
- ‘‘Creating a Symbol Table for Debugging Purposes”
- ‘‘Specifying Alignment Rules and Diagnosing Alignment Padding”
- ‘‘Specifying a Target Processor”
- ‘‘Specifying the Interpretation of Uppercase Letters”
- ‘‘Displaying Compilation Statistics”
- ‘‘Getting Information about Program Execution”

The `pl1` Command

This section describes the `pl1` command’s display form, compares the command-line arguments to the compiler options, and explains the access requirements for compilation.

The `pl1` command has the following display form.

Display Form



```
------------------------------------ pl1 -------------------------------
source_file_name:
-define:
-processor:            default
-mapping_rules:        default
-list:                 no                   -xref:                 no
-table:                no                   -production_table:     no
-optimize:             yes                  -check:                no
-mapcase:              no                   -profile:              no
-cpu_profile:          no                   -statistics:           no
-fixedoverflow:        no                   -silent:               no
-full:                 no                   -nesting:              no
-system_programming:   no                   -optimization_level:   3
-check_uninitialized:  no
```

All of the arguments shown in the display form are discussed in this chapter, except for the `-optimize`, `-optimization_level`, and `-define` arguments. See Chapter 3 for information about the `-optimize` and `-optimization_level` arguments. See Chapter 4 for information about the `-define` argument.

When you invoke the `pl1` command, the source module's name must have the suffix `.pl1`. You can either supply or omit the `.pl1` suffix when you specify the source module's name. The compiler generates an object module, puts it in your current directory, and names it. The name of the object module is the name of the source module with the suffix changed from `.pl1` to `.obj`.

Some of the `pl1` compiler arguments correspond to options of the `%options` PL/I preprocessor statement. In general, compiler-option values specified in a source module using the `%options` statement take precedence over values selected in the display form or on the command line. The arguments `-mapcase`, `-processor`, `-mapping_rules`, and `-system_programming` have corresponding options that can be specified using the `%options` statement in the source module. VOS PL/I also has `%options` compiler options that do **not** have corresponding compiler arguments. For more information about using the `%options` PL/I preprocessor statement, see Chapter 4. See also the *VOS PL/I Language Manual (R009)* for a detailed description of each `%options` compiler option.

Access Requirements

You need read access to a source module to compile it. You need appropriate write and modify access to the directory in which the `.obj` file and any other output files will be created.

Summary of VOS PL/I Compiler Arguments

Table 2-1 briefly describes each argument of the `pl1` command and lists the locations in this manual in which you can find more information about each argument.

Table 2-1. Arguments of the `pl1` Command *(Page 1 of 2)*

Argument	Description	Location of Discussion
`-check`	Checks for out-of-bounds array subscripts and out-of-range substrings	“Detecting Out-of-Bounds Subscripts and Out-of-Range Substrings”
`-check_uninitialized`	Checks for uninitialized variables	“Detecting Uninitialized Variables”
`-cpu_profile`	Inserts code that keeps track of CPU-related information when the program executes	“Getting Information about Program Execution”
`-define`	Predefines a VOS preprocessor variable	Chapter 4
`-fixedoverflow`	Checks for fixed-point arithmetic overflow	“Detecting Fixed-Point Arithmetic Overflow”
`-full`	Creates an assembly language listing	“Creating an Assembly Language Listing”
`-list`	Creates a compilation listing	“Creating a Compilation Listing”
`-mapcase`	Specifies the interpretation of uppercase letters	“Specifying the Interpretation of Uppercase Letters”
`-mapping_rules`	Specifies data-alignment rules and diagnoses alignment padding	“Specifying Alignment Rules and Diagnosing Alignment Padding”
`-nesting`	Creates a compilation listing with nesting levels	“Creating an Assembly Language Listing”
`-optimization_level`	Specifies an optimization level for a program	“Detecting Uninitialized Variables”; see also Chapter 3
`-optimize`	Specifies whether a program should be optimized	Chapter 3
`-processor`	Specifies a target processor	“Specifying a Target Processor”
`-production_table`	Creates a production table	“Using the `-production_table` Argument”
`-profile`	Inserts code that keeps track of the number of times each source statement executes	“Getting Information about Program Execution”

Table 2-1. Arguments of the `pl1` Command *(Page 2 of 2)*

Argument	Description	Location of Discussion
`-silent`	Prevents the display of certain error messages	“Preventing the Display of Certain Error Messages”
`-statistics`	Displays compilation statistics	“Displaying Compilation Statistics”
`-system_programming`	Checks for system-programming errors	“Detecting System Programming Errors” and “Specifying Alignment Rules and Diagnosing Alignment Padding”
`-table`	Creates a symbol table	“Using the `-table` Argument”
`-xref`	Creates a cross-reference listing	“Creating a Compilation Listing with Nesting Levels”

Creating a Program Listing

A *program listing* can be any of the listings that the compiler produces. This section describes the following topics related to program listings.

- “Creating a Compilation Listing”
- “Creating a Compilation Listing with Nesting Levels”
- “Creating a Cross-Reference Listing”
- “Creating an Assembly Language Listing”
- “PL/I Preprocessor Statements That Affect a Listing”

A program listing contains the source module and any include files. Each line of the source code is numbered. The compiler writes any error messages at the end of the program listing.

To create a program listing, you can use any of the arguments shown in Table 2-2. Table 2-2 shows each of the arguments that generate a program listing and the information included in each listing. The `-list` argument creates the simplest form of listing.

Table 2-2. Compiler Arguments That Create Program Listings

Argument	Contents of Listing
`-list`	Compilation listing
`-xref`	Compilation listing and cross-reference listing
`-nesting`	Compilation listing with nesting levels
`-full`	Compilation listing and assembly languag listing

Creating a Compilation Listing

If you select the `-list` argument, the compiler creates a compilation listing. The file containing the compilation listing has the name of the source module with the `.list` suffix.

The example in Figure 2-1 shows part of a compilation listing created with the following command.

```
pl1 employee_info -list
```

```
1.  SOURCE FILE: %hr#d20>HR>Mary_Doe>pl1>employee_info.pl1
    COMPILED ON: 95-01-24 AT: 10:43  by: PL/I Release 13.0
    OPTIONS:     -optimization_level 3

2.           1   employee_info:
             2        procedure options(main);
             3
             4   /* This program accepts employee information, writes it   */
             5   /* to a file, then prints all of the records in the file. */
             6
             7   declare  1  employee_rec,
             8               2  name      char(30),
             9               2  id_num    fixed bin(15),
            10               2  title     char(30);
            11
            12   declare  answer          char(1);
            13   declare  count           fixed bin(15);
            14   declare  emp_file        file;
            15   declare  x               fixed bin(15);
            16
            17        open file(emp_file) title('emp_file -delete') update;
            18        answer = 'y';
3.          19        counts = 0;
            20
            21        do while (answer = 'y');
            22             call enter_info;
            23        end; /* do-while */
            24
            25        call print_info;
            26        close file(emp_file);
            27
            28    enter_info:
            29        procedure;
            30
            31   /* Enter employee information and write it to a file. */
            32
            33        put skip list ('Enter employee''s last name.');
            34        get list (employee_rec.name);
            35        put list ('Enter employee''s ID number.');
            36        get list (employee_rec.id_num);
            37        put list ('Enter employee''s title.');
            38        get list (employee_rec.title);
            39        count = count + 1;
            40        write file(emp_file) from(employee_rec);
            41
            42        put skip list ('Do you want to enter more records?');
            43        put list ('Answer y or n.');
            44        get list (answer);
            45
```

(Continued on next page)

```
46        do while ((answer ^= 'y') & (answer ^= 'n'));
47             put skip list ('The answer must be y or n. Try again.');
48             get list (answer);
49        end; /* do-while */
50
51   end enter_info;
52
53   print_info:
54        procedure;
55
56   /* Display each record on the screen. */
57
58        do x = 1 to count;
59             read file(emp_file) into(employee_rec);
60             put skip list ('Name is ', employee_rec.name);
61             put skip list ('ID number is ', employee_rec.id_num);
62             put skip list ('Title is ', employee_rec.title);
63             put skip list (' ');
64        end; /* do-loop */
65
66        put skip list (count, 'record(s) printed.');
67
68   end print_info;
69
70   end employee_info; /* end of program */
71
```

4.

```
EXTERNAL ENTRY POINTS

NAME             CLASS       SIZE      LOC        ATTRIBUTES

employee_info    constant                         entry external


PROCEDURE employee_info ON LINE 1

NAME             CLASS       SIZE      LOC        ATTRIBUTES

answer           automatic   1         ffffee     char(1)
count            automatic   2         ffffae     fixed bin(15,0)
counts           automatic   2         ffffaa     fixed bin(15,0)
emp_file         constant              000004     file external
employee_rec     automatic   62        ffffb0     structure
   name          member      30        000000     char(30)
   id_num        member      2         00001e     fixed bin(15,0)
   title         member      30        000020     char(30)
enter_info       constant                         entry internal
print_info       constant                         entry internal
x                automatic   2         ffffac     fixed bin(15,0)
```

(Continued on next page)

```
    PROCEDURE enter_info ON LINE 28

    NAME             CLASS       SIZE      LOC        ATTRIBUTES

    sysin            constant              00000c     file external
    sysprint         constant              000008     file external


    PROCEDURE print_info ON LINE 53

    NAME             CLASS       SIZE      LOC        ATTRIBUTES

    sysprint         constant              000010     file external

5.  NO PROGRAMMED OPERATORS USED IN THIS COMPILATION.

6.  CODE GENERATED FOR PROCESSOR: mc68000

7.  STACK FRAME SIZES:
    (fixed length portion only)

    NAME                                         LINE       STACK SIZE

    employee_info                                   1            92
    enter_info                                     28           388
    print_info                                     53           308

8.  WARNING 90 SEVERITY 1 BEGINNING ON LINE 19
    The undeclared name "counts" has been declared as
    a FIXED BIN(15) variable in the current block.
```

Figure 2-1. Compilation Listing

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 2-1.

1. The opening banner contains the path name of the source module, the date and time of compilation, the release number of the VOS PL/I compiler with which the program was compiled, and any arguments that were specified on the command line.

2. The compilation listing contains source code and any include files that have been incorporated into the source module through the use of the `%include` PL/I preprocessor statement.

3. The variable `count` was incorrectly entered as `counts` on this line in order to create the error message shown in **8**.

4. In the symbol listing, the `EXTERNAL ENTRY POINTS` section first shows information about each procedure declared outside of any procedure or begin block. After this, each `PROCEDURE` section lists identifiers declared inside that procedure. In both cases, the symbol listing provides similar information for each identifier.

    - `NAME` specifies the name of the identifier declared in the program. The identifiers listed include the names of all objects, procedures, and labels.

- `CLASS` specifies information about the storage class of the objects and the type of other identifiers: constant or type definition. Labels, internal entries, and external entries are listed as constants.

- `SIZE` specifies the number of bytes or bits, in decimal, allocated for an object. For an array or structure, the size is the total number of bytes allocated for the aggregate. For a procedure or function, the size is not specified.

- `LOC` specifies one of the following offset values.

    - For XA2000-series modules and XA/R-series modules, `LOC` specifies the hexadecimal offset within the stack of an object with automatic storage duration. For Continuum-series modules, `LOC` specifies an offset from the fixed part of the frame (which could begin after a save area) within the stack of an object with automatic storage duration.

    - the offset within the static region of an object with static storage duration

    - the offset of a structure member in relation to the starting address of the immediately containing structure

    - the offset within the static region of a link (pointer) that locates an external data item or procedure entry point

- `ATTRIBUTES` lists additional information about the identifier, including the type of an object, the number of elements in an array, and the return type of a function.

5. The `PROGRAMMED OPERATORS` section lists the programmed operators used in the compilation. A *programmed operator* is a subroutine with a predefined interface that is shared throughout the system and that usually resides in the kernel.

6. The `CODE GENERATED FOR PROCESSOR` section specifies the processor for which the code is generated. See ‘‘Specifying a Target Processor” later in this chapter for more information about specifying a processor.

7. The `STACK FRAME SIZES` section lists the stack frame size, in bytes, of each program-defined procedure or function. For each procedure or function, this section also shows the line number of the first line of executable code.

8. Error messages resulting from the compilation appear at the end of the file.

Creating a Compilation Listing with Nesting Levels

If you select the `-nesting` argument, the compiler creates a compilation listing with the nesting level of each source statement. The example in Figure 2-2 shows part of a compilation listing with nesting levels.

```
1     0     employee_info:
2     0          procedure options(main);
3     1
4     1     /* This program accepts employee information, writes it   */
5     1     /* to a file, then prints all of the records in the file. */
6     1
7     1     declare  1  employee_rec,
8     1                 2  name      char(30),
9     1                 2  id_num    fixed bin(15),
10     1                 2  title     char(30);
11     1
12     1     declare  answer          char(1);
13     1     declare  count           fixed bin(15);
14     1     declare  emp_file        file;
15     1     declare  x               fixed bin(15);
16     1
17     1          open file(emp_file) title('emp_file -delete') update;
18     1          answer = 'y';
19     1          counts = 0;
20     1
21     1          do while (answer = 'y');
22     2               call enter_info;
23     2          end; /* do-while */
24     1
25     1          call print_info;
26     1          close file(emp_file);
```

Figure 2-2. Compilation Listing with Nesting Levels

As shown in Figure 2-2, the nesting levels appear in the column to the right of the line numbers. The first nesting level is 0, the next level down is 1, and so forth.

Creating a Cross-Reference Listing

If you select the `-xref` argument, the compiler creates a compilation listing that contains a cross-reference listing of all of the procedure and object identifiers referenced in the program. (See ‘‘Creating a Compilation Listing’’ earlier in this chapter for more information about compilation listings.) In the `ATTRIBUTES` column, the cross-reference shows the line number in which each procedure or object identifier is defined, the line of each statement in which it is referenced, and the data type for each procedure or object identifier. The example in Figure 2-3 shows a portion of the cross-reference listing for the source module `employee_info.pl1`.

```
EXTERNAL ENTRY POINTS

NAME              CLASS       SIZE      LOC        ATTRIBUTES

employee_info     constant                         entry external
                                                   def 1


PROCEDURE employee_info ON LINE 1

NAME              CLASS       SIZE      LOC        ATTRIBUTES

answer            automatic   1         ffffee     char(1)
                                                   def 12 ref 18 21 23 44 46 46 48
                                                   49 49
count             automatic   2         ffffae     fixed bin(15,0)
                                                   def 13 ref 39 39 58 66
counts            automatic   2         ffffaa     fixed bin(15,0)
                                                   def 19 ref 19
emp_file          constant              000004     file external
                                                   def 14 ref 17 26 40 59
employee_rec      automatic   62        ffffb0     structure
                                                   def 7 ref 40 59
   name           member      30        000000     char(30)
                                                   def 8 ref 34 60
   id_num         member      2         00001e     fixed bin(15,0)
                                                   def 9 ref 36 61
   title          member      30        000020     char(30)
                                                   def 10 ref 38 62
enter_info        constant                         entry internal
                                                   def 28 ref 22
print_info        constant                         entry internal
                                                   def 53 ref 25
x                 automatic   2         ffffac     fixed bin(15,0)
                                                   def 15 ref 58


PROCEDURE enter_info ON LINE 28

NAME              CLASS       SIZE      LOC        ATTRIBUTES

sysin             constant              00000c     file external
                                                   def 34 ref 34 36 38 44 48
sysprint          constant              000008     file external
                                                   def 33 ref 33 35 37 42 43 47


PROCEDURE print_info ON LINE 53

NAME              CLASS       SIZE      LOC        ATTRIBUTES

sysprint          constant              000010     file external
                                                   def 60 ref 60 61 62 63 66
```

Figure 2-3. Cross-Reference Listing

Creating an Assembly Language Listing

If you select the `-full` argument, the compiler creates an assembly language listing in addition to the compilation listing. The example in Figure 2-4 shows the assembly language listing for one statement of the source module `employee_info.pl1`.

```
                                                                   LINE 39
     count = count + 1;

1.                 2.        3.               4.                                    5.

0000062A   286E FFEC     movea.l       -20(a6),a4
0000062E   526C FFAE     addq.w        =1,-82(a4)                                count
```

Figure 2-4. Assembly Language Listing

Before the assembly language listing for each line of executable code, the compiler inserts the line number and the source code from that line. In the following explanation, the numbers in the left margin correspond to the numbers in Figure 2-4.

1. In the assembly language listing, the first column contains the hexadecimal offset, in bytes, of the object code relative to the beginning of this module's code region. If the source module was compiled for an XA2000-series module at optimization level 4, or was compiled for an XA/R-series or Continuum-series module, this is an accurate offset. Otherwise, the binder may compact the code so that, in the resulting program module, the offset from the base of the program's code section is not equal to the offset shown in this listing.
2. The second column contains the object code for the instruction.
3. The third column contains the assembly language instruction.
4. The fourth column contains any operands.
5. The last column contains comments on the code and often identifies the operands of an instruction.

PL/I Preprocessor Statements That Affect a Listing

In addition to the compiler arguments described in the preceding sections, VOS PL/I supports the following PL/I preprocessor statements that affect the compilation listing generated by the compiler.

- The `%page` PL/I preprocessor statement inserts a page break in the listing.
- The `%list` PL/I preprocessor statement indicates that the code following this statement should be included in the `.list` file.
- The `%nolist` PL/I preprocessor statement indicates that the code following this statement should not be included in the `.list` file.

Chapter 4 discusses these statements as well as the other PL/I preprocessor statements that VOS PL/I supports.

Interpreting Compiler Error Messages

This section discusses the following topics.

- ‘‘Error Severity Levels”
- ‘‘Preventing the Display of Certain Error Messages”

At compile time, the VOS PL/I compiler checks for errors, such as syntax errors, undeclared identifiers, and operands that are not compatible with an operator. If the compiler discovers any errors in your source module, it sends an error message to your default output device (usually your terminal).

The compiler also creates an error file in the current directory and writes the error messages to this file. The error file has the same name as the source module with the suffix `.error`. The compiler also appends the error messages to a compilation listing if a listing is produced. The compiler deletes any `.error` file if a subsequent compilation of the same source module contains no errors or warnings.

Error Severity Levels

The VOS PL/I compiler distinguishes five levels of severity in the errors it detects, as shown in the following table.

Severity Level	**Description**
`SEVERITY 0`	Advice
`SEVERITY 1`	Warning
`SEVERITY 2`	Correctable error
`SEVERITY 3`	Uncorrectable error: translation can continue
`SEVERITY 4`	Uncorrectable error: translation cannot continue

The text of the error message usually explains the cause of the error.

A severity-0 error, although valid PL/I, indicates that improvement is possible, usually in the area of performance. Since the source module is syntactically correct, the compiled object module can be bound and executed, but probably with less than optimum efficiency.

A severity-1 error, although possibly valid PL/I, is probably a minor programming error. Since the source module is syntactically and semantically correct at the point of the error or conversion, the compiler continues to compile the source code and produces an object module.

A severity-2 error is invalid PL/I, but the compiler can reinterpret the source code in such a way that it can continue to compile the program. The compiler proceeds as if the faulty code were replaced with the most likely syntactically and semantically correct code and produces an object module.

A severity-3 error is invalid PL/I, and the compiler cannot reinterpret the source in such a way that it can continue to compile the program into a usable object module. Nevertheless, the compiler continues to process the program to detect additional errors.

A severity-4 error is invalid PL/I, and the compiler cannot continue to process the program from the point of the error.

If the compilation results in more than 100 errors, in any combination (excluding severity-0 errors), compilation terminates.

If there are one or more errors but there is no error of severity-3 or greater, the compiler creates an object module that you can bind, but the program may not perform as expected. If a severity-3 or severity-4 error occurs, the object module is not created.

Preventing the Display of Certain Error Messages

If you select the `-silent` argument, the compiler suppresses messages for severity-0 and severity-1 errors from appearing on your default output device. The compiler, nevertheless, puts the error messages in an error file and in any listing it produces. The `-silent` argument does not affect the `(command_status)` command function. (See the *Introduction to VOS (R001)* for more information about `(command_status)`.)

Checking for Additional Errors at Compile Time and Run Time

This section discusses the following topics.

- ‘‘Detecting Out-of-Bounds Subscripts and Out-of-Range Substrings”
- ‘‘Detecting Uninitialized Variables”
- ‘‘Detecting Fixed-Point Arithmetic Overflow”
- ‘‘Detecting System Programming Errors”

The VOS PL/I compiler allows you to check for additional errors at compile time or run time. Table 2-3 shows the VOS PL/I error-checking arguments and briefly describes the errors and usages they report.

Table 2-3. Compiler Arguments That Perform Additional Error Checking

Argument	**Errors or Usages Reported**
`-check`	Checks for out-of-bounds array subscripts at compile time, if possible, and at run time. This argument also checks for out-of-range substrings at run time.
`-check_uninitialized`	Checks, at compile time, that each variable is initialized if it is used in a context where the variable's value is required.
`-fixedoverflow`	Checks, at run time, for fixed-point overflow in arithmetic operations.
`-system_programming`	Checks, at compile time, for a variety of legal but undesirable program constructs, such as references to structure members that do not have the level-one structure name, some cases of implicit data-type conversion, missing members in label arrays if the default case is not specified, and compiler-supplied alignment padding in structures.

Detecting Out-of-Bounds Subscripts and Out-of-Range Substrings

If you select the `-check` argument, the compiler adds extra code that checks for errors at run time. If an error is caused by a constant value, the error-checking code may detect the error at compile time. Otherwise, the error is detected at run time. The following errors can be detected.

- out-of-bounds array subscripts
- out-of-range substring references

An array subscript is out of bounds when it is not within the range of values defined in the array declaration. A substring reference is out of range when a reference to a character string with the `substr` function is not within the range of the declared string. (See the *VOS PL/I Language Manual (R009)* for more information about `substr`.)

The following program, `test_check1.pl1`, illustrates how the `-check` argument returns compile-time error messages.

```
test_check1:
     procedure options(main);

declare  array(1:3) fixed bin(31);
declare  i          fixed bin(31);

     i = 1;
     do i = 1 to 3;
          array(i) = i + 1;
      end;  /* do */

      put list ('Element 4 = ', array(4)); /* Out-of-range subscript */

 end test_check1;
```

If you compile the preceding program with the `-check` argument, you receive the following compile-time error message.

```
WARNING 2166 SEVERITY 1 BEGINNING ON LINE 12
An exception was detected evaluating the constant part of an
expression which reflects an exception that may occur at
runtime if this statement is executed.
```

If you do **not** specify `-check` while compiling the preceding program, the program compiles and executes without any error messages, but the program returns an undefined value for `array(4)`. Therefore, it is a good idea to specify `-check` in programs that contain arrays or substrings. Note, however, that execution is somewhat slower when you specify `-check`.

The following program, `test_check2.pl1`, illustrates how the `-check` argument returns run-time error messages.

```
1   test_check2:
2        procedure options(main);
3
4   declare  array(1:3) fixed bin(31);
5   declare  i          fixed bin(31);
6
7        i = 1;
8        do i = 1 to 3;
9             array(i+1) = i + 1; /* Out-of-bounds subscript */
10        end;  /* do */
11
12        put list ('Element 3 = ', array(3));
13
14   end test_check2;
```

If you compile the preceding program with the `-check` argument, you receive the following run-time error message.

```
One or more subscripts of an array reference are out of bounds.
Error occurred in procedure test_check2, line 9.
Command was test_check2.pm.
Request?  (stop, continue, debug, keep, login, re-enter)
```

Detecting Uninitialized Variables

Two arguments to the `pl1` command affect how the compiler checks for uninitialized variables: `-check_uninitialized` and `-optimization_level`. The compiler diagnoses different categories of uninitialized variables depending on whether you choose an optimization level of at least 3 **and do not** specify `-check_uninitialized`, or whether you choose an optimization level of at least 3 **and** you specify `-check_uninitialized`.

- If you do not select the `-check_uninitialized` argument **but** do select an optimization level of at least 3, the compiler diagnoses instances of automatic and internal static variables within the program that it knows are uninitialized. In this case, the compiler does not issue an error message for a variable that is initialized as part of code executed conditionally.

- If you select the `-check_uninitialized` argument **and** an optimization level of at least 3, the compiler diagnoses instances of uninitialized automatic and internal static variables. In this case, the compiler issues an error message for a variable that is initialized as part of code executed conditionally.

If you select an optimization level of less than 3, the compiler does not diagnose uninitialized variables **even if** you select `-check_uninitialized`.

The compiler diagnoses an uninitialized variable only if the variable has automatic or internal static storage duration. A variable with automatic or internal static storage duration and no explicit initialization has an unpredictable initial value. It is useful to check for uninitialized variables when you want to verify new code or check for possible bugs. The `-check_uninitialized` argument can issue an error message for an occurrence that may not represent a programming error.

The following example illustrates a situation in which variable initialization occurs as part of code executed conditionally.

```
declare a_bit    bit(1);
declare a        fixed bin(15);
declare b        fixed bin(15);

    b = 3;
    if a_bit
        then do;
        .
        .
        .
        end;
    else do;
        a = 4;
        end;
    b = a + b;
        .
        .
        .
```

In the preceding example, if `a_bit` is true, `a` is not initialized. Diagnosis of variables such as these that may be uninitialized is controlled by `-check_uninitialized`.

The `-check_uninitialized` argument does **not** detect uninitialized variables in the following situations.

- if an uninitialized variable appears in any procedure or begin block that contains a label for which **any** of the following applies:
  - The label is assigned to a label variable.
  - The label is used in a nonlocal `goto` statement.
  - The label is passed as an argument.
- if an uninitialized variable appears in any procedure or begin block containing a stream I/O statement (such as `get` or `put`)

- if an uninitialized variable is aliased. (An *aliased* variable denotes the same actual data object as another variable. For example, the `defined` attribute creates an aliased variable.)
- if an uninitialized variable is an argument passed by reference
- if an uninitialized variable is an array or a string of length other than 1
- if an uninitialized variable is an *uplevel reference* (that is, if an uninitialized variable referenced in a nested procedure is declared in an outer procedure)

Detecting Fixed-Point Arithmetic Overflow

If you select the `-fixedoverflow` argument, the compiler generates code to check for fixed-point overflow in arithmetic operations and to signal the `fixedoverflow` condition when the overflow occurs. A fixed-point overflow occurs when the result of a fixed-point operation exceeds the declared precision. The precision result is two, four, or eight bytes. In this case, the high-order bits that caused the overflow are lost, and the remaining bits appear as they normally would in the result. If you do not select `-fixedoverflow`, fixed-point overflow exceptions in arithmetic operations are not detected.

You can use on-units to handle the `fixedoverflow` condition. See the *VOS PL/I Language Manual (R009)* for more information.

Detecting System Programming Errors

If you select the `-system_programming` argument, the compiler performs checks that are useful both in system programming and in application programming. If you specify `-system_programming`, the compiler issues an error message when it detects the following types of situations.

- the following instances of implicit data-type conversion. Note that the following examples assume that a number is of type `fixed bin(15)`, and a bit string is of type `bit(1)`.
  - if a number is assigned to a character string
  - if a string is assigned to a number
  - if a bit string is assigned to a character string
  - if a character string is assigned to a bit string
- alignment padding in structures. For information about this use of `-system_programming`, see ‘‘Specifying Alignment Rules and Diagnosing Alignment Padding” later in this chapter.
- missing members in a label array for which no default case exists
- references to structure members without the level-one structure name

Creating a Symbol Table for Debugging Purposes

This section explains how you create symbol tables for use when debugging a program.

To debug a program in source mode, you must create a symbol table in the object code. A *symbol table* allows you to debug your program at the source language level instead of at the machine language level. You can select one of two arguments to produce this table: `-table` or `-production_table`. Be aware that the inclusion of a symbol table greatly increases the size of the object module. See Chapter 6 for more information on debugging a program in source mode.

If you specify neither `-table` nor `-production_table`, the compiler produces a statement map, which links source file statements to machine language instructions and is useful when stepping through a program in the debugger, though not as useful as a symbol table. However, if you specify `-no_table` in the `bind` command, statement maps are **not** included in the executable program module. See the *VOS Symbolic Debugger User's Guide (R308)* for more information about statement maps.

Using the `-table` Argument

If you select the `-table` argument, the compiler creates a symbol table and statement map, and allocates storage by assigning locations for all identifiers, including any that are not used in the source module. The `-table` argument allows you to use the full functionality of the debugger. However, when you specify this argument, the compiler does not perform certain types of object code optimization, such as interstatement code optimization. The compiler also does not perform inline expansion. As a result, program execution may be slower.

When debugging, you can use the `set` debugger request to modify any value before executing a statement, and you can use the `display` debugger request to show the contents of a variable. With `-table`, a variable that would normally be stored in a register is allocated and kept in memory. Specifying the `-table` argument (as opposed to `-production_table`) ensures that changing the flow of control using the debugger will always work as expected.

If you do not specify the `-table` argument while compiling a program that contains unreferenced variables, the compiler allows some declaration errors to remain undiagnosed. Consider the following example.

```
a:
     procedure options(main);

declare b(10)   fixed bin(15);
declare a(10)   fixed bin(15) based(q);
declare p       pointer;

     p = addr(b);

     p->a(5) = 4;

end a;
```

If you do **not** specify `-table` when you compile the preceding example, the program compiles without any error messages. However, if you **do** specify `-table`, you receive the following error messages and the compilation fails, since `q` was not declared.

```
WARNING 90 SEVERITY 1 BEGINNING ON LINE 5
The undeclared name "q" has been declared as
a FIXED BIN(15) variable in the current block.


ERROR 202 SEVERITY 3 BEGINNING ON LINE 5
A value used in this statement cannot be converted to the
data type required by the context in which it is used.
```

Note that the compiler does not produce initialization information for unreferenced external static variables unless you specify the `-table` argument. This behavior results from the rule that all declarations of an external variable must be identical. See the *VOS PL/I Language Manual (R009)* for more information on external static variables.

The `-table` argument also affects a program's optimization level; see Chapter 3 for more information.

Using the `-production_table` Argument

If you select the `-production_table` argument, the compiler creates a symbol table for identifiers that are used in the source module, produces optimized object code, and creates a statement map. The symbol table created with `-production_table` is smaller than the symbol table created with `-table`. The compiler performs all of the operations that it does with `-table`, except for the following:

- suppress interstatement code optimization
- always store register variables in memory
- generate addresses in the symbol table for unused variables

If you select both `-table` and `-production_table`, `-production_table` takes precedence, and the compiler produces the smaller symbol table and a statement map.

Code produced with the `-production_table` argument may yield unpredictable results if you use the `set` and `continue` debugger requests. Also, with `-production_table`, the `display` and `set` debugger requests might not return the correct value.

Specifying Alignment Rules and Diagnosing Alignment Padding

This section explains how you specify data-alignment rules and diagnose alignment padding in structures.

Two compiler arguments specify data-alignment rules or diagnose alignment padding in structures or do both: `-mapping_rules` and `-system_programming`.

If you select the `-mapping_rules` argument specifying one of the mapping values, the compiler uses the specified data-alignment method when laying out storage for the source module. In addition, when you specify one of the `/check` values (as shown in Table 2-4), the compiler diagnoses alignment padding in structures. The values that can be used in the

`-mapping_rules` argument are shown in Table 2-4. By default, the compiler uses the `default` value, but your system administrator can change the default. To display the current default value for the `-mapping_rules` argument, issue either of the following commands on the command line.

```
display_error m$default_mapping
```

```
display_error 4991
```

In either case, the output shows the current default value for `-mapping_rules` on your module. See the manual *VOS System Administration: Administering and Customizing a System (R281)* for more information about setting the default `-mapping_rules` value.

Table 2-4. Values for the `-mapping_rules` Argument

Value	**Description**
`default`	Specifies the system-wide default data-alignment method. The default method is site-settable.
`default/check`	Same as `default` except, in addition, the compiler diagnoses alignment padding within structures.
`shortmap`	Specifies the shortmap alignment rules.
`shortmap/check`	Same as `shortmap` except, in addition, the compiler diagnoses alignment padding within structures.
`longmap`	Specifies the longmap alignment rules.
`longmap/check`	Same as `longmap` except, in addition, the compiler diagnoses alignment padding within structures.

The following example illustrates the use of `-mapping_rules`.

```
declare  1  bad_rec,
            2  x         char(1),
            2  y         float bin(53),
            2  z         fixed bin(31);
```

If you compile a program containing the preceding structure with the `-mapping_rules longmap/check` argument, you receive the following message.

```
ADVICE 2189 SEVERITY 0 BEGINNING ON LINE 6
A gap of 7 bytes has been found between 'bad_rec.y'
and the previous member in the structure.
```

The compiler diagnoses the padding between `bad_rec.y` and `bad_rec.x`. You can avoid excessive structure padding by rearranging the structure so that the longest elements appear first, as shown in the following example.

```
declare  1  good_rec,
            2  y          float bin(53),
            2  z          fixed bin(31),
            2  x          char(1);
```

Specifying a data-alignment method through the use of an `%options` compiler option (such as `longmap` or `shortmap`) overrides the data-alignment method indicated in `-mapping_rules` or the default data-alignment method (if `-mapping_rules` is not specified), but the compiler still diagnoses alignment padding within structures if you have specified one of the `/check` values on the command line. See Chapter 4 for more information about using the `%options` PL/I preprocessor statement to specify a data-alignment method.

In addition to the `-mapping_rules` argument, you can specify the `-system_programming` argument or the `%options system_programming` compiler option to diagnose alignment padding that appears in structures.

For more information about longmap and shortmap data alignment, see the *VOS PL/I Language Manual (R009)*.

Specifying a Target Processor

This section describes how you specify a target processor. It also describes how you can compile code on a module containing a different processor from that on which the code will execute.

If you select the `-processor` argument or the `%options processor` compiler option, you can explicitly specify the processor on which the code will run (called the *target processor*). Currently, Stratus modules use processors from one of the following processor families.

- MC68000® (XA2000-series modules)
- i860™ (XA/R-series modules)
- PA-RISC (Continuum-series modules)

You can generate code for any XA2000-series module, XA/R-series module, or Continuum-series module from any XA2000-series module, XA/R-series module, or Continuum-series module. Although VOS Release 14.0.0 does not support XA2000-series modules, you can still generate code for an XA2000-series module from an XA/R-series module or Continuum-series module.

If your target processor is located on an XA2000-series module and you have specified `-optimization_level 4`, **or** if your target processor is located on either an XA/R-series or Continuum-series module, the module on which you are compiling must have **at least** 30,000 pages of paging partition available to avoid running out of virtual memory. In addition, you should also have 64 megabytes of physical memory available to achieve optimal compiler performance. See Chapter 3 for more information about virtual memory usage.

You select a value for the `-processor` argument based on the processor type(s) found in the module where the code is to execute. If you do **not** select the `-processor` argument, the compiler generates code that will run on the default system processor.

By default, the compiler uses the `default` value, but your system administrator can change the default. To display the current default value for the `-processor` argument, issue either of the following commands.

```
display_error m$default_processor
```

```
display_error 3932
```

In either case, the output shows the current default value for `-processor` on your module. See the manual *VOS System Administration: Administering and Customizing a System (R281)* for more information about setting the default `-processor` value.

When you use the `-processor` argument and specify a particular processor type in the same processor family as the current module, the compiler creates object code customized for the indicated processor. As a result, the code runs faster on the specified processor, but **cannot**, in some cases, run on alternate processors.

Table 2-5 shows the allowed processor values for the MC68000 processor family. In addition, the table lists the processor indicated by each value and the Stratus models in which the processor is found. Only programs compiled with the `mc68000` value will run on any MC68*xxx* processor. Depending on the processor type(s) found in the module where the code is to execute, you can specify one of the values shown in Table 2-5 in the following situations.

- if the current module uses a processor from the MC68000 family
- if the current module uses a processor from the i860 or PA-RISC family and you are performing a cross-compilation to produce code that will run on a module using a processor from the MC68000 family (see ‘‘Cross-Compilation” later in this chapter for more information about cross-compilation)

Table 2-5. Values for the MC68000 Processor Family *(Page 1 of 2)*

Value	Processor	Model
`default`	Default system processor	Default
`mc68000`	Available for cross-compilation in the UCOMM environment	N/A
`mc68020`	MC68020 with or without the MC68881 co-processor	XA2000 Model 100
`mc68020/mc68881`	MC68020 with the MC68881 co-processor	XA2000 Models 110 to 160
`mc68030`	MC68030 with or without the MC68882 co-processor	XA2000 Model 30 and Models 200 to 260

Table 2-5. Values for the MC68000 Processor Family *(Page 2 of 2)*

Value	Processor	Model
`mc68030/mc68882`	MC68030 with the MC68882 co-processor	XA2000 Model 30 and Models 200 to 260

In general, the processor values in Table 2-5 are forward-compatible. This means, for example, that code compiled with the `mc68020` value will run on modules containing any of the MC68020 or MC68030 processors, but it might not run on a module containing an MC68000 processor.

Table 2-6 shows the allowed processor values for the i860 processor family. In addition, the table lists the processor indicated by each value and the Stratus models in which the processor is found. You can specify one of the values shown in Table 2-6 in the following situations.

- if the current module uses a processor from the i860 family
- if the current module uses a processor from the MC68000 or PA-RISC family and you are performing a cross-compilation to produce code that will run on a module using the i860 processor (see ‘‘Cross-Compilation” later in this chapter for more information about cross-compilation)

Table 2-6. Values for the i860 Processor Family

Value	Processor	Model
`default`	i860XR	XA/R Models 5, 20, 25, and 300
`i80860`	i860XR	XA/R Models 5, 20, 25, and 300
`i80860xp`	i860XP	XA/R Models 35, 45, 305, 310, 320, and 330

> **Note:** If a program contains any object files that were compiled with the `i80860xp` value, the program **will not run** on a module containing the i860XR processor. Code compiled for a module containing the i860XP processor contains assembly instructions that are not available on the i860XR processor.

Table 2-7 shows the allowed processor values for the PA-RISC family. In addition, the table lists the processor indicated by each value and the Stratus models in which the processor is found. You can specify one of the values shown in Table 2-7 in the following situations.

- if the current module uses a processor from the PA-RISC family
- if the current module uses a processor from the MC68000 or i860 family and you are performing a cross-compilation to produce code that will run on a module using the PA-RISC processor (see ‘‘Cross-Compilation” later in this chapter for more information about cross-compilation)

Table 2-7. Values for the PA-RISC Processor Family

Value	Processor	Model
`default`	PA7100	Continuum Models 610S, 610, 620, 1210, 1215, 1225, and 1245
`pa7100`	PA7100	Continuum Models 610S, 610, 620, 1210, 1215, 1225, and 1245
`pa8000`	PA8000	Continuum Models 618, 628, 1218, and 1228

The compiler automatically defines one or more variables for the processor family and one or more variables corresponding to the processor type(s). The variables defined depend on the value of the `-processor` argument. See Chapter 4 for information about how to use these predefined variables during preprocessing.

Depending on the value specified in the `-processor` argument, there are different limits on the maximum number of bytes available for a function's initial stack frame, as described in the following list. Therefore, the value specified in the `-processor` argument affects the amount of nondynamic automatic storage available for a function.

- If the value specified in the `-processor` argument indicates the MC68000 processor, the maximum number of bytes available for each function's initial stack frame is 32,766 bytes.
- If the value specified in the `-processor` argument indicates the i860XR or i860XP processor, the maximum number of bytes available for each function's initial stack frame is 32,752 bytes.
- If the value specified in the `-processor` argument indicates the MC68020 or MC68030 processor, the maximum number of bytes available for each function's initial stack frame is 2,147,483,646 bytes.
- If the value specified in the `-processor` argument indicates the PA7100 or PA8000 processor, the maximum number of bytes available for each function's initial stack frame is 2,147,483,584 bytes.

The amount of automatic storage you can actually declare is somewhat less than these limits because temporary variables generated by the compiler also count toward the limit. Note that although the VOS PL/I compiler supports extremely large values (such as 2,147,483,646), the system does not support them.

Cross-Compilation

You can compile code on a module containing a different processor from that on which the code will run. *Cross-compilation* occurs when a compiler running on a processor from one processor family translates a source module into object code that will execute on a processor of a different processor family. The processor on which the code is compiled is called the *host processor*. The processor on which the code is to run is called the *target processor.*

When you are cross-compiling, use the `-processor` argument to indicate the target processor. For example, if the host processor is from the MC68000 family and the target processor is the i860, enter the following `-processor` argument on the command line.

```
pl1 source_module -processor i80860
```

The display form for the `pl1` command's `-processor` argument restricts the values that you can choose to values for the processor family of the current module **unless** you specify a different processor on the command line before pressing the key that invokes the `DISPLAY FORM` function. If you use the `-processor` argument to specify a different processor on the command line, the display form for `pl1` then restricts your processor choices to values for that family.

An important difference between the `-processor` argument and the `%options processor` compiler option is that you **cannot** use `%options processor` to generate code for a different processor family. For example, if you attempt to compile a program containing the following line on a module using the MC68020 processor, the compiler returns an error.

```
%options processor i80860
```

Specifying the Interpretation of Uppercase Letters

This section describes how you can affect the way in which the compiler interprets uppercase letters.

The compiler, by default, interprets uppercase letters differently from lowercase letters. You can make the compiler appear to be case insensitive by specifying the `-mapcase` argument. When you select the `-mapcase` argument or the `%options mapcase` compiler option, the compiler interprets uppercase letters as their lowercase counterparts, except for uppercase letters in quoted strings.

When you compile a source module with the `-mapcase` argument or `%options mapcase`, and the code contains an external variable name or entry name with one or more uppercase letters, you **may not be able to bind** the resulting object module with another object module that defines the same external variable and that has not been compiled with the `-mapcase` argument. If the binder encounters a reference to the original name (for example, in a binder control file), it will not recognize the original name and its lowercase version as the same name.

Displaying Compilation Statistics

This section discusses how you display compilation statistics.

If you select the `-statistics` argument, the compiler displays compilation statistics on the terminal's screen as the compilation proceeds. If you also specify the `-list` argument, the compiler appends the statistics to the compilation listing. Figure 2-5 shows the compilation statistics of a sample program.

```
COMPILATION STATISTICS FOR PL/I Release 12.0
Source file: %hr#d20>HR>Mary_Doe>pl1>sample_program.pl1

PHASE          DISK  SECONDS    SPACE      PAGING      CPU
init              0        2        1          32        0  15:58:55
pass1             6        2        1          26        0  15:58:57
declare           0        0        1           6        0  15:58:57
pass2            13        2        1          33        1  15:58:59
optimizer         6        3        1          59        1  15:59:02
allocator         0        1        1          40        0  15:59:03
code gen          0        3        1          61        2  15:59:06
cleanup           0        0        1           0        0  15:59:06
total            25       13        1         257        4  15:59:06

NODE PAGES                 3
CODE SIZE               3754
STATIC SIZE              156
SYMTAB SIZE              360
SOURCE LINES              76
LINES PER MIN           1140
```

Figure 2-5. Compilation Statistics

The compilation statistics in Figure 2-5 are divided into seven columns. The information in each column is as follows:

- `PHASE` is the phase of compilation.
- `DISK` is the number of times that the compiler performed disk I/O in each phase of compilation.
- `SECONDS` is the number of seconds that the compiler took to complete each phase.
- `SPACE` is the number of pages that the compiler used in allocating blocks of memory during each phase.
- `PAGING` is the number of page faults that occurred during each phase.
- `CPU` is the amount of time, in seconds, that the central processing unit took to complete each phase.
- The last column indicates the time when the compiler completed each phase.

The compilation statistics also include the number of node pages and the size, in bytes, of the object code, the static data, and the symbol table. (A *node* is a data structure internal to the compiler.) The node pages value represents the total number of pages used by the compiler for symbol nodes, value nodes, constant nodes, and other nodes required to complete the compilation. The number of nodes is an internal count and is roughly proportional to the size and complexity of the program. Finally, the compilation statistics list the number of source code lines and the number of lines compiled per minute.

If your program is very large, the statistics listed in the `SPACE` column may appear in the form `[`*`number`*`, `*`number`*`]`. The first number represents the number of blocks of storage for every 1024 nodes used. The second number represents each 4096 bytes of storage used. For example, `[22, 5]` means that 22 blocks of storage were used and that 20,480 bytes of storage were used. If either number is equal to 32, the program exhausted its storage space.

Getting Information about Program Execution

This section discusses how you get information about program execution.

To get information about the execution of a program module each time you run it, you specify one of two arguments: `-profile` or `-cpu_profile`. If you specify either of these arguments, the compiler inserts, into the object module, code that reports performance information when you run the program.

> **Note:** If you are debugging a program that was compiled for an XA/R-series module or a Continuum-series module, the `step` request could terminate and the program could continue to execute if you specified the `-cpu_profile` argument during compilation. See the *VOS Symbolic Debugger User's Guide (R308)* for more information.

A *profile file* is a non-ASCII file containing performance information about all object modules compiled with the `-profile` or `-cpu_profile` argument and bound together in the program. When program execution is complete, the operating system creates the profile file and puts it in your current directory, overwriting any existing profile file of an earlier execution. The profile file has the same name as the program module, except the suffix is `.profile`.

If you specify the `-profile` argument, the compiler inserts additional code that keeps track of the number of times each source statement is executed.

If you specify the `-cpu_profile` argument, the compiler inserts additional code that keeps track of the following information when the program runs.

- the number of times each source statement is executed
- the amount of CPU time spent executing each statement
- the number of page faults taken executing each statement

You cannot specify the `-profile` and `-cpu_profile` arguments simultaneously.

See Chapter 3 for information about how optimization can affect the analysis of performance information.

To convert the information in the profile file, created with either argument, into readable text, you must create a `.plist` file from it. The following steps explain how to perform the conversion.

1. Compile each source module for which you want performance information with the `-profile` argument or the `-cpu_profile` argument.

2. Bind the object modules with the `bind` command.

3. Execute the program module to create the `.profile` file.

4. Process the `.profile` file with the `profile` command. The operating system creates a file named `program_name.plist`.

5. Display or print the `.plist` file.

When you process the `.profile` file by specifying the following command, the operating system combines the compilation listing and performance information in the `.plist` file. The `.profile` suffix is optional.

```
profile file_name.profile
```

See the *VOS Commands Reference Manual (R098)* for detailed information about the `profile` command.

You can use the `add_profile` command to add the profile information from two profile files, accumulating the sum in the second profile file. See the *VOS Commands Reference Manual (R098)* for detailed information about this command.

Figure 2-6 shows a `.plist` file created by the `profile` command from an execution of a program, `employee_info.pl1`, compiled with the `-cpu_profile` argument.

```
Profile of: employee_info

  Number of statements:     36
  Statements Executed:      33 (91.66% of statements)

STATEMENT      COUNT      CPU(ms)      PAGES CUM %      %

        1          1         0.00          0    0.0  0.00
       17          1        51.89          0   18.4 18.47
       18          1         0.00          0   18.4  0.00
       19          1         0.00          0   18.4  0.00
       21          1         0.00          0   18.4  0.00
       22          2         0.00          0   18.4  0.00
       23          2         0.01          0   18.4  0.00
       25          1         0.02          0   18.4  0.01
       26          1        29.20          0   28.8 10.39
       28          2         0.01          0   28.8  0.00
       33          2        16.76          0   34.8  5.97
       34          2        10.02          0   38.4  3.56
       35          2        11.07          0   42.3  3.94
       36          2         8.65          0   45.4  3.08
       37          2        10.37          0   49.1  3.69
       38          2         8.68          0   52.2  3.09
       39          2         0.02          0   52.2  0.01
       40          2        14.34          0   57.3  5.10
       42          2        15.65          0   62.9  5.57
       43          2         8.03          0   65.8  2.86
       44          2         8.51          0   68.8  3.03
       46          2         0.04          0   68.8  0.01
       51          2         0.01          0   68.8  0.00
       53          1         0.02          0   68.8  0.01
       58          1         0.04          0   68.8  0.01
       59          2        16.55          0   74.7  5.89
       60          2        18.41          0   81.3  6.55
       61          2        18.12          0   87.7  6.45
       62          2        14.93          0   93.1  5.31
       63          2        12.86          0   97.6  4.57
       64          2         0.02          0   97.6  0.01
       66          1         6.45          0   99.9  2.29
       70          1         0.01          0  100.0  0.00

TOTALS:           55       280.83          0

Null statement CPU time: 0.061 milliseconds.
```

Figure 2-6. Run-Time Performance Information for a Program Module

The performance information in the sample `employee_info.plist` file (Figure 2-6) is divided into six columns. The information in each column is as follows:

- `STATEMENT` is the line number of the statement.
- `COUNT` is the number of times that the statement was executed.
- `CPU(ms)` is the number of milliseconds the CPU took to process the statement.
- `PAGES` is the number of page faults caused by the execution of the statement.

- `CUM %` is the percentage of program execution time taken by the statement **and** all preceding statements.

- `%` is the percentage of program execution time taken by the statement.

After the `TOTALS` information, the `Null statement CPU time` section shows the number of milliseconds the CPU took to process statements that contain no instructions. The null time is used in calculating the overhead of the `profile` command. Note that this overhead has already been deducted from the time shown in the `CUM %` column.

Chapter 3:
Optimizing the Object Code

An *optimization* is a code-improving modification that occurs during compilation. An optimization makes a program run faster, take less space, or both. You can choose the level of optimization that the compiler uses to improve the object code generated.

This chapter discusses the following topics related to optimizing the object code.

- ‘‘Optimization Levels’’
- ‘‘Optimizations’’
- ‘‘Program Behavior Changes Caused by Optimization’’
- ‘‘Debugging Optimized Code’’
- ‘‘Analyzing Performance Information in Optimized Code’’
- ‘‘Virtual Memory Usage’’
- ‘‘Paging Partition and Space Requirements’’

Optimization Levels

This section discusses the following topics.

- ‘‘Optimization Levels for an XA2000-Series Module’’
- ‘‘Optimization Levels for an XA/R-Series or Continuum-Series Module’’
- ‘‘Specifying an Optimization-Related Argument’’

You can choose the optimization level that the compiler uses when it optimizes code within the source module. The optimization level dictates which, if any, optimizations are performed. The optimizations that are performed at each level vary depending on the processor family.

Optimization Levels for an XA2000-Series Module

The optimizations listed in this section are described in detail in ‘‘Optimizations’’ later in this chapter.

When you are compiling a source module to run on an XA2000-series module, the levels of optimization are 0, 1, 2, 3, and 4.

If you select optimization level 0, the compiler performs no optimizations. This optimization level is equivalent to specifying `-no_optimize`.

If you select optimization level 1, the compiler performs the following local optimizations.

- local pattern replacement
- short-circuit evaluation of Boolean expressions
- recognition of algebraic identities
- constant folding
- result incorporation
- elimination of unreachable code
- local combination of common subexpressions within a statement

If you select optimization level 2, the compiler performs all level-1 optimizations plus the following global optimizations.

- branch retargeting
- global combination of common subexpressions
- removal of invariant expressions from loops

If you select optimization level 3, the compiler performs all level-2 optimizations plus the following global optimizations.

- constant propagation
- removal of invariant assignments from loops
- strength reduction
- linear test replacement
- elimination of dead assignments
- elimination of useless loops
- detection of uninitialized variables
- inline expansion

If you select optimization level 4, the compiler performs all level-3 optimizations plus the following global optimizations.

- global register allocation
- elimination of dead code and dead stores
- subsumption
- peephole optimization

Although compilation time may be longer, the object code produced when you specify `-optimization_level 4` should be significantly more efficient, and will bind and execute faster than the code produced when you specify a lower optimization level.

While you are developing a program and, thus, repeatedly modifying and recompiling it, you might not want to use optimization level 4 for the following reasons.

- Much more compilation time is required for optimization level 4 than for any of the lower levels. At optimization level 4, compilation time can increase by a factor of two or even more for source modules with procedures having a very large number of lines of code.

- The code generated is often much harder to follow when debugging in machine mode because the values currently in registers may have been set very far away from where they are used. Also, a variable's value is less likely to be in storage for source-mode debugging.

Some optimizations can make it more difficult for you to interpret error messages issued by the compiler and to debug the optimized code. For example, if you compile at optimization level 3, the compiler removes invariant computations from loops. This optimization causes code motion. This means that if the compiler encounters an error during or after the optimization phase, it may issue an error message with a line number that corresponds to a line of the optimized code and not a source code line.

See the ‘‘Program Behavior Changes Caused by Optimization’’ and ‘‘Debugging Optimized Code’’ sections later in this chapter for information about optimization-related changes in program behavior that may cause problems.

Optimization Levels for an XA/R-Series or Continuum-Series Module

The optimizations listed in this section are described in detail in ‘‘Optimizations’’ later in this chapter.

When you are compiling a source module to run on an XA/R-series module or a Continuum-series module, the levels of optimization are 0, 1, 2, 3, and 4.

If you select optimization level 0, registers are only allocated locally. For XA/R-series modules, peephole optimizations are only performed within a single statement at this level. Also, elimination of unreachable code occurs at this level. This optimization level is equivalent to specifying the `-no_optimize` compiler argument.

If you select optimization level 1, the compiler performs the following local optimizations.

- local register allocation
- peephole optimizations within a single statement (for Continuum-series modules)
- local pattern replacement
- short-circuit evaluation of Boolean expressions
- recognition of algebraic identities
- constant folding
- result incorporation
- local combination of common subexpressions within a statement

If you select optimization level 2, the compiler performs all level-1 optimizations plus the following global optimizations.

- branch retargeting
- global combination of common subexpressions
- removal of invariant expressions from loops
- subsumption
- peephole optimizations across statement boundaries
- global register allocation

If you select optimization level 3, the compiler performs all level-2 optimizations plus the following global optimizations.

- constant propagation
- removal of invariant assignments from loops
- strength reduction
- linear test replacement
- elimination of dead assignments
- elimination of useless loops
- detection of uninitialized variables
- elimination of dead code and dead stores
- inline expansion
- instruction scheduling
- no allocation of stack space for automatic variables whose values are kept in registers (for Continuum-series modules only)

If you select optimization level 4, the compiler performs the **same** optimizations that it performs at optimization level 3.

Specifying an Optimization-Related Argument

The compiler optimizes the object code during compilation unless you explicitly specify optimization level 0. (For an XA/R-series module or a Continuum-series module, even if you specify optimization level 0, the compiler still performs local register allocation, and peephole optimizations are performed within a single statement.) The `pl1` command-line arguments shown in Table 3-1 determine the optimization level that the compiler uses for a source module. If you do not specify an optimization level, the default optimization level is 3 unless you select the `-no_optimize` or `-table` argument.

Table 3-1. Optimization-Related Compiler Arguments

Argument	**Description**
`-optimization_level`	Specifies the level of optimization that the compiler uses. Allowed values are 0, 1, 2, 3, and 4. The default is 3.
`-no_optimize`	Specifies optimization level 0. This argument overrides the `-optimization_level` argument as well as the optimization level associated with the `-table` argument if either of these arguments is specified.
`-table`	Specifies optimization level 1. This argument overrides the `-optimization_level` argument if that argument is specified with a value greater than 0. If you specify this argument, the compiler does not perform any global optimizations.

You can determine the optimization level at which a source module has been compiled by looking at the banner (first few lines) of the compilation listing, where the selected compiler options are shown.

Specifying Optimization Levels in a Source Module

You can specify the optimization level for an entire source module or for a procedure in that source module. The optimization level specified for a procedure applies to all contained procedures unless you explicitly override the level for a particular contained procedure. All compilation units are considered to be surrounded by a block that has a maximum optimization level of the value specified in the `-optimization_level` compiler argument or the default value of 3.

To specify the optimization level for a source module, use the `-optimization_level` argument. To specify the optimization level for a procedure, use the `options(max_optimization_level(`*n*`))` clause of the `procedure` statement. If an optimization level is specified for a procedure, and a different value is specified for the source module in the `-optimization_level` argument, the compiler will optimize the procedure at the lower of the two levels specified.

In the following example, the compiler will optimize the `b` procedure within the `a` procedure at optimization level 2, but it will optimize the remainder of `a` at optimization level 3. Assuming that the `-optimization_level` argument was not specified, the compiler will optimize the remainder of the source module at optimization level 3, the default optimization level.

```
a:
     procedure(num_1, num_2) options(max_optimization_level(3));
     .
     .
     .
     b:
          procedure options(max_optimization_level(2));
     .
     .
     .
     end b;
end a;
```

See the *VOS PL/I Language Manual (R009)* for more information about using the `options` clause of the `procedure` statement to specify the optimization level for a procedure.

Optimizations

This section discusses the following topics related to the optimizations that the VOS PL/I compiler can perform.

- ‘‘Local Pattern Replacement”
- ‘‘Short-Circuit Evaluation of Boolean Expressions”
- ‘‘Branch Retargeting”
- ‘‘Eliminating Unreachable Code”
- ‘‘Recognizing Algebraic Identities”
- ‘‘Constant Folding”
- ‘‘Result Incorporation”
- ‘‘Local and Global Combination of Common Subexpressions”
- ‘‘Peephole Optimization”
- ‘‘Constant Propagation”
- ‘‘Removing Invariant Expressions from Loops”
- ‘‘Removing Invariant Assignments from Loops”
- ‘‘Strength Reduction”
- ‘‘Linear Test Replacement”
- ‘‘Eliminating Dead Assignments”
- ‘‘Eliminating Useless Loops”
- ‘‘Detecting Uninitialized Variables”
- ‘‘Global Register Allocation”
- ‘‘Subsumption”
- ‘‘Eliminating Dead Code and Dead Stores”
- ‘‘Inline Expansion”
- ‘‘Instruction Scheduling”
- ‘‘Allocating Stack Space for Automatic Variables”

The optimizations performed depend on the optimization level that you select, as well as the type of machine for which the code is being compiled. See ‘‘Optimization Levels for an XA2000-Series Module” and ‘‘Optimization Levels for an XA/R-Series or Continuum-Series Module” earlier in this chapter for information about which optimizations the compiler performs at each optimization level.

Local Pattern Replacement

During *local pattern replacement*, the compiler replaces various patterns of operators with more efficient sequences of operators or with simpler expressions. Local pattern replacement produces better code for special cases of assignments and branches. For example, the compiler might replace the expression `-(-a)` with `a`.

Short-Circuit Evaluation of Boolean Expressions

The compiler replaces certain patterns of logical operators within Boolean expressions with more efficient sequences of operators. In this optimization, the compiler evaluates only enough of a logical expression to determine the result if the semantics of the operator allow only portions of the expression to be evaluated. For example, consider the following logical expression.

```
expression_1 | expression_2
```

Since the order of evaluation is not guaranteed, the compiler may choose to evaluate either `expression_1` or `expression_2`. If it can be determined at compile time that the evaluated expression is true, the compiler does not need to evaluate the other expression. Therefore, with short-circuit evaluation of Boolean expressions, part of a logical expression may not be evaluated.

PL/I's evaluation of logical AND/OR constructs is very different from C's evaluation of such constructs. In C, the order of evaluation is **guaranteed**. You should be aware of this behavior if your PL/I program calls a C function, or vice versa.

Branch Retargeting

During *branch retargeting*, the compiler replaces branches to branches with branches to the eventual target, thus reducing the working set for programs that contain conditional `goto` statements inside loops. For example, consider the following `goto` statements.

```
top:
   .
   .
   .
   goto next;
   .
   .
   .
next:
   goto top;
```

After branch retargeting occurs, the compiler changes the code in the preceding example as shown in the following example.

```
top:
   .
   .
   .
   goto top;
   .
   .
   .
next:
   goto top;
```

Eliminating Unreachable Code

If a block of code is unreachable or becomes unreachable because of other optimizations, that block is eliminated. For example, in the following fragment, because of the `return` statement, the program never reaches the next line.

```
    .
    .
    .
return;
call zzz;
```

When the unreachable code is removed, the compiler changes the code in the preceding example as shown in the following example.

```
    .
    .
    .
return;
```

On XA/R-series and Continuum-series modules, the compiler always performs this optimization, regardless of the optimization level.

Recognizing Algebraic Identities

If one of an operator's operands is an identity constant, the compiler can often simplify the operation. When an identity constant is used as an operand in an expression, the result of the operation is equal to one of the operands.

The identity constants are as follows:

- 0, for addition and subtraction
- 0 or 1, for multiplication
- 1, for division
- true (`'1'b`) or false (`'0'b`), for the Boolean AND and OR operations

The examples in the following table illustrate how the compiler can simplify expressions containing identity constants.

Expression	**Equivalent Expression**	**Replacement Value**
`x + 0`	`0 + x`	`x`
`x - 0`	None	`x`
`0 - x`	None	`-x`
`x * 1`	`1 * x`	`x`
`x * 0`	`0 * x`	`0`
`x / 1`	None	`x`
`bit1 | '1'b`	`'1'b | bit1`	`'1'b`
`bit1 | '0'b`	`'0'b | bit1`	`bit1`
`bit1 & '1'b`	`'1'b & bit1`	`bit1`
`bit1 & '0'b`	`'0'b & bit1`	`'0'b`

The arguments of the `max` and `min` functions behave similarly to the identity constants. When both arguments of the `max` or `min` function are the same, the result is the value of one of the arguments. Consequently, if both arguments are the same, the compiler replaces the function with the value of one of its arguments, as illustrated in the following example.

```
max(x,x) = x
min(x,x) = x
```

Constant Folding

The compiler replaces, with their results, expressions that consist solely of constant operands and whose results are reliably computable at compile time. This process is known as *constant folding*. For example, the expression `2 + 3` is replaced by the value `5`, and the expression `substr('abcd',2,2)` is replaced by the string `'bc'`.

If your program is compiled on an XA2000-series module, most constant expressions containing floating-point data are **not** folded, since XA2000-series modules support extended-precision intermediate results. If your program is compiled on an XA/R-series module or a Continuum-series module, however, such expressions **are** typically folded.

Result Incorporation

Without optimization, the compiler represents a single assignment involving a string-valued expression with two assignments: the first assignment assigns the value of the string-valued expression to a temporary variable of the correct type and size, and the second assignment assigns the value of the temporary variable to the target variable.

For example, the assignment `(a_str = b_str || c_str)` is internally represented in the following manner.

```
temp = b_str || c_str;
a_str = temp;
```

These temporary variables are necessary to prevent the target variable from overwriting part of the data contained in a source variable.

Some operations, such as long bit-string operations, character-string operations, and string-conversion operations, are more efficient if the compiler assigns the value of the operation directly to the target variable. This direct form of assignment is called *result incorporation*.

The compiler performs result incorporation if all of the following conditions are true.

- The result of the operation is not needed anywhere else.
- The target variable is of the correct data type and size to hold the result of the operation.
- The compiler can determine that none of the source operands overlap the target variable.

If one of the source operands **is** the same variable as the target variable, the compiler may still perform result incorporation.

Local and Global Combination of Common Subexpressions

If you specify an optimization level of 2 or greater, local combination of common subexpressions occurs only within a flow-unit. A *flow-unit* is a sequence of code that may only be entered at its beginning and exited at its end. If the first instruction of a flow-unit is executed, then **all** instructions are executed.

If you specify an optimization level of 1, local combination of common subexpressions is further limited so that it only occurs within a statement.

When performing global combination of common subexpressions, the VOS PL/I compiler takes data flow into account and recognizes opportunities for combining subexpressions even when labels intervene. The following example shows unoptimized code that uses the subexpression `a + b * c` twice.

```
x = a + b * c - d;
y = a + b * c + e;
```

The following example shows the optimized version of the preceding code.

```
t = a + b * c;
x = t - d;
y = t + e;
```

Peephole Optimization

Peephole optimization improves a program's performance by examining a very short sequence of machine language instructions (called the *peephole*) and replacing these instructions with a shorter or faster sequence, if possible. Often, each performance improvement creates opportunities for additional improvements.

For example, consider two machine language instructions: the first moves the contents of register A to register B, and the second moves the contents of register B to register A. If the program code never branches to the second instruction (that is, both instructions are always executed together), and if the second instruction does not have side-effects, the second instruction can be eliminated.

Constant Propagation

If the compiler can detect that a variable has a constant value at compile time, the compiler replaces references to the variable's value with the constant. This process is known as *constant propagation*. When combined with constant folding and removal of dead assignments, this optimization can significantly improve code efficiency. See ''Eliminating Dead Assignments'' later in this chapter for information about dead assignments.

For example, consider the following assignments.

```
a = 3;
b = a * 2;
c = b - 1;
```

If `a` and `b` are not used elsewhere, constant propagation changes the preceding code to the following code.

```
c = 5;
```

In addition, the compiler recognizes a variable that has internal static storage duration, and whose initial value is never changed, as having a constant value. Opportunities for constant propagation can also be introduced by other optimizations, such as strength reduction. See ''Strength Reduction'' later in this chapter for information about this optimization.

Removing Invariant Expressions from Loops

An *invariant expression* is an expression that occurs within a loop, but whose effect would not change if it were removed from the loop. The compiler moves such an expression out of a loop and stores the value in a register or in a temporary variable.

To prevent unnecessary evaluation of an expression, the compiler processes loops from the innermost loop to the outermost loop, and then moves invariant expressions as far as possible out of a nest of loops.

Consider the following example.

```
do i = 1 to 10;
     a(i) = b * c;
end;
```

In the preceding example, `b * c` is an invariant expression whose result is calculated and assigned to `a(i)` 10 times. Because the result does not change, the calculation needs to be performed only once. Therefore, when the code is optimized, the compiler moves the calculation outside of the loop and assigns its result to a temporary variable, `temp`. Each time the loop is executed, the compiler assigns the value stored in `temp` to `a(i)`. The following example illustrates.

```
temp = b * c;
do i = 1 to 10;
     a(i) = temp;
end;
```

This optimization is particularly useful for removing array subscripts from a loop when the subscript is a variable or expression that is not modified in the loop.

Removing Invariant Assignments from Loops

An *invariant assignment* is an assignment that occurs within a loop, but whose effect would not change if it were removed from the loop.

In the following example, the statement `j = 2 * k;` is executed 10 times.

```
do i = 1 to 10;
     j = 2 * k;
     array(i) = i + j;
     end;
```

Since the value of `j` in the preceding example never changes, `j = 2 * k;` does not need to appear inside the loop. Therefore, when the code is optimized, the compiler places the statement before the loop so that it is only executed once. The following example illustrates.

```
j = 2 * k;
do i = 1 to 10;
     array(i) = i + j;
     end;
```

Strength Reduction

Strength reduction is the replacement of operators of greater strength with operators of less strength. Strength, in this context, is the processing time required by an operator. For example, multiplication requires more processing time than addition.

Strength reduction usually involves replacing the multiplication of an induction variable and a loop invariant by the addition of a new induction variable and a loop invariant. An *induction variable* is a variable whose value is updated once per iteration and whose new value is a linear function of an induction variable (usually itself) and a loop invariant. Thus, loop control variables are almost always induction variables.

For example, the following code initializes an array of 4-byte integers, which causes an internal “multiply by 4” that is not obvious to the programmer.

```
declare array(10)      fixed bin(31);

     do i = 1 to 10;
          array(i) = j;
          end;
```

To reference each element of `array`, the compiler must find the base address of `array` and add an offset to it. The first element of `array` is at the base address, the second element has a 4-byte offset, the third element has an 8-byte offset, and so forth. Therefore, to get the offset for an array indexed by `i`, the internal code multiplies `i` by 4, then subtracts 4 (`4 * i - 4`). The code subtracts 4 because the first element has no offset. Consequently, the compiler represents the loop in a form similar to that shown in the following example.

```
     i = 1;
     if i > 10
          then goto exit;
loop:
     addrel(addr(array), 4 * i - 4) -> based_fb31 = j;
     i = i + 1;
     if i <= 10
          then goto loop;
exit:
```

The built-in function `addr` returns the base address of `array`. The built-in function `addrel` uses the base address and the offset (`4 * i - 4`) to calculate the address of `array(i)`. See the *VOS PL/I Language Manual (R009)* for more information about built-in functions.

Since `i` is initialized to 1 before the loop, constant propagation and constant folding cause the compiler to recognize that it is not necessary to include the `exit` label and the condition `if i > 10 then goto exit` that appeared in the previous code. Thus, the compiler changes the loop in the preceding example to that shown in the following example.

```
     i = 1;
loop:
     addrel(addr(array), 4 * i - 4) -> based_fb31 = j;
     i = i + 1;
     if i <= 10
          then goto loop;
```

The strength reduction optimization replaces complex operations with simpler operations. For example, the compiler replaces the expression `4 * i - 4` with the induction variable `i4m4`. The compiler changes the code in the preceding example to the code shown in the following example.

```
          i = 1;
          i4m4 = 4 * i - 4;
     loop:
          addrel(addr(array), i4m4) -> based_fb31 = j;
          i = i + 1;
          i4m4 = i4m4 + 4 * 1;
          if i <= 10
               then goto loop;
```

Since `i` is initialized to 1, the expression `4 * i - 4` equals 0. Also, `4 * 1` is always 4. Constant folding and constant propagation change the code as shown in the following example.

```
          i = 1;
          i4m4 = 0;
     loop:
          addrel(addr(array), i4m4) -> based_fb31 = j;
          i = i + 1;
          i4m4 = i4m4 + 4;
          if i <= 10
               then goto loop;
```

Linear Test Replacement

Linear test replacement involves the replacement of tests involving a linear function of an induction variable and a loop invariant with that of another induction variable and another loop invariant. The compiler performs this optimization if both of the following conditions are true.

- The replacement would eliminate the last use of an induction variable in a loop.
- The value of the induction variable is not needed after exiting from the loop.

In the final two strength reduction examples shown in the preceding section, ‘‘Strength Reduction,” a new induction variable (`i4m4`) was introduced with an increment of 4. Consequently, the compiler could now apply linear test replacement to change the test for `i <= 10` to `i4m4 <= 36` and eliminate the incrementing of `i`. The following example shows the optimized code.

```
          i = 1;
          i4m4 = 0;
     loop:
          addrel(addr(array), i4m4) -> based_fb31 = j;
          i4m4 = i4m4 + 4;
          if i4m4 <= 36
               then goto loop;
```

Eliminating Dead Assignments

Many of the optimizations described in the preceding sections can cause a variable assignment to become “dead.” That is, the target of the assignment is never referenced for the value assigned in that operation. In the linear test replacement example from the previous section, the original induction variable `i` is no longer referenced for its value. The following example shows the result of eliminating a dead assignment.

```
          i4m4 = 0;
     loop:
          addrel(addr(array), i4m4) -> based_fb31 = j;
          i4m4 = i4m4 + 4;
          if i4m4 <= 36
               then goto loop;
```

In the preceding example, the assignment `i = 1` is removed because it is a dead assignment.

Eliminating Useless Loops

A loop is considered useless if it:

- has no effect on program flow
- does not set a variable to a value used after the loop
- makes no procedure calls
- performs no input or output
- references no volatile variables
- has no other language-defined side effects

For example, the compiler can replace the following loop with a simple assignment to `j` if the assignment to `j` can be moved from the loop or if `i` is not used after the loop.

```
     do i = 1 to 100000;
          j = 0;
          end;
```

When the compiler eliminates a useless loop, it issues an error message.

Detecting Uninitialized Variables

The techniques used to perform many of the optimizations described in the preceding sections can also be used to detect uninitialized variables. If you select an optimization level of at least 3, the compiler diagnoses instances of automatic and internal static variables within the program that it knows are uninitialized. In this case, the compiler does **not** issue an error message for a variable that is initialized as part of code executed conditionally.

You can also use the `-check_uninitialized` argument to detect uninitialized variables. If you select the `-check_uninitialized` argument and an optimization level of at least 3, the compiler diagnoses instances of uninitialized automatic variables. In this case, the compiler **does** issue an error message for a variable that is initialized as part of code executed conditionally.

If you select an optimization level of less than 3, the compiler does not diagnose uninitialized variables even if you select `-check_uninitialized`.

See Chapter 2 for more information about `-check_uninitialized` and detecting uninitialized variables.

Global Register Allocation

Global register allocation allows the compiler to allocate registers more efficiently. The compiler keeps the most frequently used variables in registers as much as possible, since accessing a value in a register is much more efficient than accessing one in memory. Local register allocation, which is not an optimization, occurs only within a single statement. Global register allocation occurs within a block.

Subsumption

Subsumption causes the compiler to eliminate unnecessary register copies by combining the logical registers that are the source and target of the copy operation. In register allocation, a logical register represents a value that must be assigned to a machine register. When the compiler can allocate the logical registers that are the source and target to the same machine register, one logical register is said to subsume another.

Consider the following example.

```
x = y;
     .
     .
     .
call add_nums(x);
call calc_totals(y);
     .
     .
     .
```

If the values of `x` and `y` are not changed between the statement `x = y;` and the statement `call add_nums(x);`, the compiler allocates both variables to the same machine register.

Eliminating Dead Code and Dead Stores

The compiler eliminates dead code. *Dead code* consists of expressions whose results are not used or cannot be reached.

Similarly, the compiler eliminates dead stores. *Dead stores* are register variables that are unnecessary because no storage for a variable is needed even though the variable's value may be needed for a later operation. Consider the variable `a` in the following assignments.

```
a = c;
c = 2 * c;
b = a + 4;
```

In the preceding example, if no other references to `a` occur, no storage is needed for `a`. This is true even though the value of `a`, which is kept in a register, is required for the assignment `b = a + 4`.

Inline Expansion

Each time you activate a procedure, there is a cost involved in executing the procedure itself and also in executing the call. When a procedure body is small, the amount of code in the calling sequences may be more than the amount of code in the procedure body. It is, therefore, more efficient to use inline expansion of the procedure into the caller's code. Since inline expansion slows compilation, it is only available with optimization level 3.

During *inline expansion*, the `inline` attribute of the `procedure` statement causes the body of the procedure to be substituted for the call, and the actual arguments are substituted for the formal parameters. Procedures containing the `inline` attribute behave just as they would without the attribute except that they execute more quickly. Because they are part of the stack frame of the enclosing procedure, they have no separate stack frame of their own.

If you specify the `-table` argument, the compiler removes the `inline` attribute so that the debugger will operate correctly.

See the *VOS PL/I Language Manual (R009)* for more information about inline expansion.

Instruction Scheduling

When a source module is compiled at optimization level 3 or 4 for an XA/R-series module or a Continuum-series module, the compiler schedules the execution of instructions to take advantage of the overlapping architecture of reduced instruction set computing (RISC) processors.

By using instruction scheduling, the compiler rearranges the execution of instructions to avoid delays and maximize overlap by, for example, loading data well before it is used and by setting the condition register several instructions before a branch.

Allocating Stack Space for Automatic Variables

On Continuum-series modules, the compiler does **not** allocate stack space for automatic variables whose values are kept in registers rather than being stored. This optimization reduces the size of stack frames, thereby improving the efficiency of data-cache utilization.

Note that this optimization could uncover some types of bugs in your programs. For example, a program that uses an uninitialized variable or that makes an out-of-range reference to an array or string could fail. In this case, the failure could be caused by the fact that out-of-range references are now more likely to touch other variables instead of unused space.

The compiler detects most cases of uninitialized automatic variables. You can specify the `-check_uninitialized` argument if you want additional checking to occur. Note, however, that `-check_uninitialized` still does not detect every instance of uninitialized automatic variables, and that it occasionally returns false hits.

You can detect out-of-range references by specifying the `-check` argument and testing your program. Compiling with `-check` during testing should remove most out-of-range references.

See ‘‘Detecting Uninitialized Variables” earlier in this chapter for more information about detecting uninitialized variables.

Program Behavior Changes Caused by Optimization

This section describes the known changes of program behavior that might cause programming problems and explains the corrective action that is available.

The main compatibility issues related to optimization involve changes of behavior that do not affect the PL/I code itself, but that might cause problems if you attempt to follow the translation of your programs into machine code. For example, optimization makes it difficult to debug code that has been moved from loops.

Elimination of Useless Loops

Depending on the level of optimization that you select, the compiler eliminates loops from a program under the conditions described in ‘‘Eliminating Useless Loops’’ earlier in this chapter. This means that loops whose sole purpose is to introduce delays or wait on low-level locks are eliminated unless some variable referenced in the loop is declared with the `volatile` attribute. The compiler issues a warning when it eliminates a useless loop.

See the *VOS PL/I Language Manual (R009)* for more information about the `volatile` attribute.

Elimination of Dead Assignments and Dead Stores

As described in ‘‘Optimizations’’ earlier in this chapter, many optimizations, such as linear test replacement and constant propagation, could cause some assignments to appear dead that did not appear so before the optimizations took place. The compiler recognizes the potential for references to based, external, or parameter variables in other procedures called from the procedure being optimized, or from other procedures that call the optimized procedure. Because the calling procedure could use the value of such a variable after control returns from the optimized procedure, the compiler must retain the value of the variable to produce expected results.

You may not want dead-store elimination to occur because a variable is referenced in some way by hardware or by another process that is not in the language model. In this case, you must declare the variable with the `volatile` attribute to ensure that the variable continues to be used. See the *VOS PL/I Language Manual (R009)* for more information about the `volatile` attribute.

Changing the Order of Assignments

While the compiler respects the logical relationships between assignments to variables and potential references to the results of such assignments, including the effect of on-units, it may change the order of assignments where it can see no logical effect on the outcome of a program. For example, the compiler may move an assignment in a loop to a position before the loop under the following conditions.

- The expression on the right-hand side is not changed within the loop.
- The assignment is always executed if the loop is executed.
- The target is not referenced in the loop before the assignment.

Be aware that, if a procedure call precedes such an assignment in a loop, the compiler recognizes that the assignment might not be reached. If a program must retain all assignments to a variable because one of the variables in the assignments is referenced in some way by hardware or by another process that is not in a language model, you must declare the variable with the `volatile` attribute.

Elimination of Redundant Assignments

One of the effects of the method used to combine common subexpressions is the elimination of redundant assignments. For example, the compiler eliminates an assignment if the following conditions are true.

- There are two assignments `x = y` in a path.
- All paths that reach the second assignment pass through the first.
- The values of `x` and `y` do not change between the two assignments.

If either the right operand or the left operand of an assignment could be altered in an extra-lingual way, you should declare the variable with the `volatile` attribute. For example, if a variable that is the right or left operand represents a special hardware control register, the variable could change in an extra-lingual way.

Storing the Address of By-Reference Parameters

A procedure must not store the address of one of its parameters for use after the procedure returns. Without this restriction, a compiler must assume that if a variable is passed as an argument to one procedure, regardless of its storage class, then any other procedure call could alter or use the value of that variable, thus eliminating most optimizations in system code where argument passing is a frequent occurrence.

It is an error for a procedure to store the address of one of its parameters for use after the procedure returns, unless one of the following is true.

- The optimization level is less than 3.
- That parameter (and the associated parameter descriptor in the calling procedure) is declared with the `volatile` attribute.

Of course, the generation of storage for the variable must still exist for the address to be valid.

Violations of the Rules for Storage Sharing

Code that violates the rules for storage sharing is more likely to have problems when optimized with the Release 11.0 or later compiler. (These rules are documented in the *VOS PL/I Language Manual (R009)*.) To avoid such problems, you should either follow the documented rules for storage sharing or, for existing programs, insert the `%options untyped_storage_sharing` compiler option. You should avoid using `%options untyped_storage_sharing` in new programs because it degrades optimization quality.

Debugging Optimized Code

Optimized code is sometimes difficult to debug. This section discusses some of the problems you could encounter while debugging optimized code, as well as ways to avoid those problems.

If a fault occurs in code, the line number referred to in the error message and debugger might not be the line at which the fault actually occurred.

When you compile a program with the `-production_table` argument and optimization level 3 or greater, some local scalar variables may not have the values that you expect them to have when you debug the program.

Instruction scheduling and dead-store elimination can cause code movement, which makes it difficult to debug a program. Code produced with these optimizations may yield unpredictable results if you use the `set` or `break` debugger request. For example, if you cannot set a breakpoint on a particular line, the code on that line may have been moved or eliminated. To avoid this problem, you should specify an optimization level of 2. Also, you might want to specify the `-full` argument during compilation if it would be helpful for you to view the program's assembly code. (See Chapter 2 for more information about the `-full` argument.)

As stated previously in this chapter, unreachable code is eliminated at all optimization levels on XA/R-series and Continuum-series modules. Sometimes, however, you might want your program to contain some code that will be executed only during a debugging session, not during normal program execution. To prevent the compiler from eliminating such unreachable code, you might consider changing your program in the following manner.

```
declare always_zero    fixed bin(15) volatile static initial (0);

     if (always_zero ^= 0) then
     /* Code that should not be eliminated goes here */
```

If you delete the `volatile` attribute from the preceding declaration, the compiler will eliminate the unreachable code. See the *VOS PL/I Language Manual (R009)* for more information about `volatile`.

When you compile a program at optimization level 3 or greater but do not create a symbol table, you could experience some of the same problems you encounter when you specify `-production_table` with optimization level 3 or greater, such as code movement.

If, during debugging, it is important that you be able to change the value of a variable at any point in the program and that you be able to change the flow of a program at will, you should compile the program with the `-table` argument. This argument forces the optimization level to 1.

In general, if you experience optimization-related problems during debugging, isolate the object module containing the problem, and recompile that module using a lower optimization level.

For more information about debugging optimized code, see the *VOS Symbolic Debugger User's Guide (R308)*.

Analyzing Performance Information in Optimized Code

This section explains how to analyze performance information in optimized code.

If you compile a program with either the `-profile` or `-cpu_profile` argument to get performance information, using an optimization level greater than 2 can make analysis of the information generated by the `profile` command more difficult.

Generally, the `-profile` argument is used to obtain code coverage information as opposed to performance data. For this reason, you probably should use an optimization level of less than 3 because high optimization levels can cause code to be moved from one statement to another. When analyzing a program in regard to which paths of execution have been taken, you should maintain a clear correspondence between the code unit being executed and the source statement that generated the code. This correspondence can be maintained by using an optimization level of 2 or less.

Using a lower optimization level is also useful when you are trying to get rough performance information by using the `-cpu_profile` argument. In this case, you must be aware that the code generated at higher optimization levels can be very different from the code generated at lower optimization levels. Thus, program execution data obtained with `-cpu_profile` at low optimization levels, while easy to interpret, can be misleading. When you use `-cpu_profile` with optimized production-quality code, it is more difficult to associate the code unit being executed with the source statement that produced the code. If you use the `-cpu_profile` argument, you should consider both the accuracy required and the ease of analyzing the resulting data when choosing an optimization level.

When you use the `-cpu_profile` argument to compile code for an XA/R-series module or a Continuum-series module, the use of `-cpu_profile` can affect the code generated by the compiler. The code is affected because instruction scheduling is inhibited around the instructions that record the elapsed CPU time and, in addition, `-cpu_profile` also affects register allocation.

Virtual Memory Usage

This section discusses the compiler's use of virtual memory.

The compiler uses virtual memory extensively. Virtual memory usage is particularly great at optimization level 4 when compiling for an XA2000-series module, or at any optimization level when compiling for an XA/R-series module or a Continuum-series module. At these optimization levels, one significant use of virtual memory increases as the square of the number of distinct values (that is, potential register occupants) that are used within a given procedure. If a source module contains a procedure with a very large number of lines of code, the compiler may consume more virtual memory than the master disk's paging partition can accommodate.

If there is not enough virtual memory, the compiler issues the following error message.

```
FATAL ERROR 2533 SEVERITY 4
Implementation restriction:  there was insufficient virtual memory
for this compilation.  To eliminate this error you should either break
up this compilation unit, break up this procedure or function, or have
the system administrator increase the size of this module's paging
partition.
```

The preceding error is usually **not** related to the size of a source module, but to the size of the largest procedure in the program. It is unlikely that a procedure in a typical modularly designed program will be large enough to cause this problem.

If the compiler issues the preceding error message when translating a program with a procedure having a very large number of lines of code, you should perform one of the following actions.

- Break up the compilation unit into several smaller compilation units.
- Break up the largest procedure or begin block in the compilation unit into several smaller procedures or begin blocks.
- If it is not easy or desirable to break up the compilation unit or procedure, your system administrator should increase the size of the paging partition. A record of the paging partition usage appears in the `syserr_log.(date)` file if the paging partition shrinks below a system-defined threshold. See the manual *VOS System Administration: Starting Up and Shutting Down a Module or System (R282)* for more information about the `syserr_log.(date)` file. See ''Specifying a Target Processor'' in Chapter 2 for information pertaining to the size of your module's paging partition.

You could also receive the preceding error message if a number of concurrent compilations are performed on one module, and at least one compilation consumes excessive virtual memory. In this case, it is possible that the compilation of a relatively small source module may be affected because of the virtual memory usage of another process. To avoid this problem, make sure that source modules containing very large procedures are compiled one at a time, or that the size of the paging partition is increased.

Paging Partition and Space Requirements

If your target processor is located on an XA2000-series module and you have specified `-optimization_level 4` with your compiler command, or if your target processor is located on either an XA/R-series or Continuum-series module, the module on which you are compiling must have **at least** 30,000 blocks of paging partition available to avoid running out of paging space. In addition, you should have at least 128 megabytes of physical memory available to achieve optimal compiler performance.

Chapter 4:
Using the Preprocessors

When you compile a VOS PL/I source module, the compiler invokes preprocessors to process special statements in the source module before the compiler translates the source code into object code. Likewise, when you bind an object module using a binder control file, the binder invokes a preprocessor to process special statements in the binder control file.

This chapter discusses the following preprocessor-related topics.

- ‘‘Overview”
- ‘‘Using VOS Preprocessor Statements”
- ‘‘Using PL/I Preprocessor Statements”
- ‘‘Using Both Types of Preprocessor Statements”

Overview

This section provides an overview of the three types of preprocessor statements.

The VOS PL/I compiler invokes two preprocessors: the VOS preprocessor and the PL/I preprocessor. The VOS preprocessor is the same preprocessor invoked by the other VOS compilers (except VOS C and VOS Standard C) during compilation. The PL/I preprocessor is specific to the VOS PL/I compiler. This chapter describes the statements used by both preprocessors.

The binder invokes one preprocessor: the binder preprocessor, which is virtually identical to the VOS preprocessor. Chapter 5 discusses the differences between the binder preprocessor and the VOS preprocessor.

There are three types of preprocessor statements.

- VOS preprocessor statements
- PL/I preprocessor statements
- binder-preprocessor statements

VOS preprocessor statements and *PL/I preprocessor statements* allow you to conditionally compile a source module. *Conditional compilation* enables you to switch on or off various statements in a source module. This feature is useful, for example, if you want a program to compile different lines of source code for different processor types.

PL/I preprocessor statements additionally allow you to alter program text, control the generation of a compilation listing, and enable compiler options.

Binder-preprocessor statements allow you to perform conditional inclusion on an object module. *Conditional inclusion* enables you to include certain blocks of text in a binder control file. This is useful, for example, if you want to direct the binder to search different directories for object modules based on the processor family on which the program module will run. Chapter 5 discusses binder control-file preprocessing.

See the *VOS PL/I Language Manual (R009)* for detailed descriptions of the preprocessor statements.

Using VOS Preprocessor Statements

This section discusses the following topics.

- ‘‘Commenting Out Statements”
- ‘‘Using Preprocessor Variables”
- ‘‘Sample Program Using VOS Preprocessor Statements”
- ‘‘Using the Stand-Alone Preprocessor”

As discussed in ‘‘Overview” earlier in this chapter, the VOS preprocessor statements allow you to conditionally compile a source module.

Table 4-1 summarizes the VOS preprocessor statements.

Table 4-1. VOS Preprocessor Statements

Statement	Description
`$define`	Defines a preprocessor variable inside a source module or binder control file
`$else`	Processes the lines up to the next `$endif` statement
`$elseif`	Evaluates an expression as true or false, and conditionally includes the source code up to the next `$elseif` or `$endif` statement
`$endif`	Closes the most recent `$if` or `$elseif` statement
`$if`	Evaluates an expression as true or false, and conditionally includes the source code up to the next `$else`, `$elseif`, or `$endif` statement
`$undefine`	Undefines a preprocessor variable

Preprocessor variables are described in ‘‘Using Preprocessor Variables” later in this chapter.

VOS preprocessor statements **must begin** in the **first column** of the source module. Indentation of nested `$if` statements is, therefore, not allowed.

A VOS preprocessor statement must be contained on a single line. A line containing a VOS preprocessor statement cannot contain comments or parts of the source language. (An exception is the `$endif` statement, which ignores any text following it on the same line, thus allowing you to comment the source code.)

If you specify an `$if` statement, you can use parentheses and the `&` (AND), | (OR), and ^ (NOT) operators to form more complex expressions. The order of operator precedence, from highest to lowest precedence, is NOT, AND, and OR.

See the *VOS PL/I Language Manual (R009)* for a full description of each VOS preprocessor statement, including its syntax.

Commenting Out Statements

You can comment out VOS preprocessor statements with the `/*` and `*/` delimiters. The characters `/*` mark the beginning of a commented-out statement, and the characters `*/` mark the end of a commented-out statement. A statement that appears within the delimiters is ignored by the compiler. The comment delimiters must surround the statement on the same line, or the opening delimiter must appear on the line that precedes the statement, and the closing delimiter must appear on the line that follows the preprocessor statement. In general, you **cannot** put one comment delimiter on the same line as the statement if the corresponding delimiter appears on a different line. Consider the examples in Figure 4-1.

Interpreted as a Comment:

```
/*
$endif
*/
```

Interpreted as a Comment:

```
/*  $endif */
```

Interpreted as a Valid Statement:

```
/*
$endif  */
```

Figure 4-1. Commenting Out VOS Preprocessor Statements

In the third example in Figure 4-1, the preprocessor would interpret the `$endif` as a valid preprocessor statement, **not** as a comment, because the closing comment delimiter appears on the same line as the statement, but the opening delimiter appears on the preceding line.

Using Preprocessor Variables

A *preprocessor variable* is a sequence of 1 to 256 alphabetic characters, digits, and underline (_) characters, in any position, used by the preprocessor. The preprocessor distinguishes between uppercase and lowercase alphabetic characters. No more than 100 preprocessor variables, including predefined preprocessor variables, can be defined for a source module.

You use either the `-define` command-line argument or the `$define` VOS preprocessor statement to predefine a preprocessor variable. If you want to predefine multiple variables, you can either specify them on the command line with `-define`, or you can use multiple `$define` statements in the source module, as shown in the following example.

```
$define REVISION_2
$define REVISION_3
      .
      .
      .
```

You cannot predefine multiple preprocessor variables in a single `$define` statement.

The preprocessor **always** predefines the following preprocessor variables for the processor family.

- `__MC68K__`, when the processor value specified is from the MC68000 family
- `__I860__`, when the processor value specified is from the i860 family
- `__HPPA__`, when the processor value specified is from the PA-RISC family. In addition, when `__HPPA__` is predefined:
  - the compiler's preprocessor also predefines either `__HPPA11__` and `_PA_RISC1_1`, **or** `__HPPA20__` and `_PA_RISC2_0`, depending on the processor's specific family (the PA-7100 family or the PA-8000 family)
  - the binder's preprocessor predefines either `__HPPA11__` or `__HPPA20__`, depending on the processor's specific family (the PA-7100 family or the PA-8000 family)

When you compile a source module, in addition to a preprocessor variable for the processor family, the preprocessor also predefines one or more additional values for the specific processor type within the family. Table 4-2 shows the preprocessor variables that are predefined for each specific processor type, as indicated by the value specified in the `pl1` command's `-processor` argument.

Table 4-2. Predefined Preprocessor Variables

Processor Value	Preprocessor Variable
`default`	Varies, depending on the default system processor
`mc68000`	`__MC68000__` and `__MC68K__`
`mc68020`	`__MC68020__` and `__MC68K__`
`mc68020/mc68881`	`__MC68020__`, `__MC68881__`, and `__MC68K__`
`mc68030`	`__MC68030__` and `__MC68K__`
`mc68030/mc68882`	`__MC68030__`, `__MC68882__`, and `__MC68K__`
`i80860`	`__I80860__` and `__I860__`
`i80860xp`	`__I80860XP__` and `__I860__`
`pa7100`	`__PA7100__`, `__HPPA__`, `__HPPA11__`, and `_PA_RISC1_1`[†]
`pa8000`	`__PA8000__`, `__HPPA__`, `__HPPA20__`, and `_PA_RISC2_0`[†]

[†] The compiler's preprocessor defines the `_PA_RISC1_1` or `_PA_RISC2_0` preprocessor variable; the binder's preprocessor does not define them.

An explanation of Table 4-2 follows.

- If the processor name contains a co-processor, the preprocessor automatically defines both processors. For example, if you specify `-processor mc68030/mc68882`, the preprocessor defines the `__MC68030__`, `__MC68882__`, and `__MC68K__` preprocessor variables.

- For Continuum-series modules, the preprocessor defines the following preprocessor variables.

  - The `__PA7100__` and `__PA8000__` preprocessor variables invoke code that is specific to each processor.

  - The `__HPPA__` preprocessor variable invokes code that will run on all current and future PA-RISC processors.

  - The `__HPPA11__` and `_PA_RISC1_1` preprocessor variables invoke code that will run on all processors in the current and future PA-7100 processor family. The `__HPPA20__` and `_PA_RISC2_0` preprocessor variables invoke code that will run on all processors in the current and future PA-8000 processor family.

  Note: The `_PA_RISC1_1` and `_PA_RISC2_0` preprocessor variables are for Stratus internal use; you should not specify them.

For example, if you specify `-processor i80860` on the command line, the preprocessor automatically defines the `__I80860__` preprocessor variable to indicate the processor type and the `__I860__` preprocessor variable to indicate the processor family.

Chapter 2 describes the `-processor` argument to the `pl1` command. Chapter 5 describes the `-processor` argument to the `bind` command.

Sample Program Using VOS Preprocessor Statements

Figure 4-2 illustrates how to use VOS preprocessor statements in a source module. In the figure, the source module `test.pl1` contains `$if`, `$elseif`, `$else`, and `$endif` VOS preprocessor statements.

```
test:
     procedure options(main);

declare i      fixed bin(15);

$if defined (__MC68020__) | defined (__MC68030__)
     i = 1;

$elseif defined (__I80860__)
     i = 2;

$elseif defined (__PA7100__)
     i = 3;

$else
     i = 0;

$endif

     put skip list (i);
end test;
```

Figure 4-2. Source Module with VOS Preprocessor Statements

Assume that the source module shown in Figure 4-2 is compiled with the following command-line arguments.

```
pl1 test -list -processor i80860
```

The preceding command creates an object module, `test.obj`, as well as the compilation listing (`test.list`) shown in Figure 4-3. The compilation listing illustrates the effects of preprocessing on compilation. Note that only the relevant portion of the compilation listing is shown.

```
 1   test:
 2        procedure options(main);
 3
 4   declare i      fixed bin(15);
 5
 6   +++$if defined (__MC68020__) | defined (__MC68030__)
 7   +++     i = 1;
 8   +++
 9   +++$elseif defined (__I80860__)
10        i = 2;
11
12   +++$elseif defined (__PA7100__)
13   +++     i = 3;
14   +++
15   +++$else
16   +++     i = 0;
17   +++
18   +++$endif
19
20        put skip list (i);
21   end test;
```

Figure 4-3. Compilation Listing with Preprocessed VOS Preprocessor Statements

As shown in Figure 4-3, the compiler inserts three plus signs (+++) in front of each line of the compilation listing that, after preprocessing, will **not** be compiled.

In Figure 4-3, when the `pl1` command is invoked, the compiler automatically defines the `__I860__` and `__I80860__` preprocessor variables to represent the processor family and processor type specified in the `-processor` argument.

With these predefinitions, when the `$if`, `$elseif`, and `$else` statements are processed, only the following controlling expression (from line 9) evaluates to true.

```
defined (__I80860__)
```

Thus, after preprocessing, the only line of conditional code to be compiled is line 10, `i = 2;`. The code on lines 7, 13, and 16 is not compiled.

Using the Stand-Alone Preprocessor

If you want to see the output produced by the VOS preprocessor statements prior to compilation, you can use the `preprocess_file` command to invoke the stand-alone preprocessor. The `preprocess_file` command expands a PL/I source module named *`source_module`*`.pl1` into a source module named *`source_module`*`.pl1.pout`.

The `preprocess_file` command, unlike the `pl1` and `bind` commands, does **not** predefine any preprocessor variables. Also, the `preprocess_file` command ignores comment delimiters, treating commented-out text as text to be compiled.

Typically, you use the `preprocess_file` command to produce an expanded source module so that you can debug complex preprocessor statements.

> **Caution: To compile an expanded source module, you must first rename the `.pout` file to *`file`*`.pl1`. However, do not give the `.pout` file the same name as the original source module, or the original file will be overwritten.**

Figure 4-4 illustrates the use of the `preprocess_file` command to create an expanded source module from the source module shown in Figure 4-2.

```
test:
     procedure options(main);

declare i      fixed bin(15);

     i = 0;


     put skip list (i);
end test;
```

Figure 4-4. Expanded Source Module

In Figure 4-4, because the `-define` argument was not specified on the command line, and because the program did not contain any `$define` statements, no preprocessor variables were defined. Since no preprocessor variables were defined, the only line of conditional code to be compiled is `i = 0;`.

You can also use the `preprocess_file` command to create expanded binder control files. Note, however, that `preprocess_file` does not affect PL/I preprocessor statements in any way.

See the *VOS Commands Reference Manual (R098)* for more information about the `preprocess_file` command.

Using PL/I Preprocessor Statements

This section explains how you use the PL/I preprocessor statements.

The PL/I preprocessor statements allow you to do the following:

- conditionally compile a source module
- control the generation of a compilation listing
- enable compiler options
- alter program text

Table 4-3 summarizes the PL/I preprocessor statements.

Table 4-3. PL/I Preprocessor Statements

Statement	Description
`%do`	Introduces a `%do`-group, which contains a series of PL/I language statements and preprocessor statements.
`%end`	Closes the most recent `%do`-group.
`%if`	Evaluates an expression as true or false and controls subsequent compilation.
`%then` clause (of `%if`)	Enables compilation if the expression in the associated `%if` statement evaluated to true.
`%else` clause (of `%if`)	Enables compilation if the expression in the associated `%if` statement evaluated to false.
`%include`	Inserts a text file into the program text.
`%list`	Re-enables the compiler list facility.
`%nolist`	Disables the compiler list facility.
`%` (null)	Performs no action.
`%options`	Specifies compiler options, most of which correspond to command-line arguments. The options are `default_char_set`, `default_mapping`, `longmap`, `longmap_check`, `mapcase`, `no_mapcase`, `processor`, `shortmap`, `shortmap_check`, `system_programming`, `no_system_programming`, and `untyped_storage_sharing`. See the *VOS PL/I Language Manual (R009)* for more information about each option.
`%page`	Starts a new page in the compilation listing.
`%replace`	Creates a synonym for a literal constant or declared name.

PL/I preprocessor statements are syntactically different from VOS preprocessor statements in the following ways.

- You can indent PL/I preprocessor statements to improve clarity.
- PL/I preprocessor statements need not be contained on a single line.
- A line containing a PL/I preprocessor statement can contain comments.

If you specify an `%if` statement, you can use any arithmetic, relational, bit-string, or concatenation operator, except for the exponentiation operator, to form more complex expressions.

See the *VOS PL/I Language Manual (R009)* for a full description of each PL/I preprocessor statement, including its syntax.

Sample Program Using PL/I Preprocessor Statements

Figure 4-5 shows part of a source module containing PL/I preprocessor statements. This example uses PL/I preprocessor statements to execute different statements depending on the value of `COMPANY`.

```
   main:
        procedure;

        declare number      fixed bin(15);
             .
             .
             .
1.      %replace  COMPANY               by 'Stratus';

2.      %if COMPANY = 'Stratus' %then
             %do;
             %replace FILE_NAME_LENGTH  by 255;
             %replace BUFFER_SIZE       by 8192;
             %replace OPEN              by s$open_file;
             %end;
3.      %else %if COMPANY = 'Tekno' %then
             %do;
             %replace FILE_NAME_LENGTH  by 2;
             %replace BUFFER_SIZE       by 1;
             %replace OPEN              by error_not_available;
             %end;
             .
             .
             .
```

Figure 4-5. Source Module with PL/I Preprocessor Statements

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 4-5.

1. The `%replace` statement replaces the constant value `COMPANY` with the literal value `Stratus` when the source module is compiled.
2. Since `COMPANY` is a synonym for `Stratus`, the `%if` statement evaluates to true. Therefore, the associated `%do`-group is compiled through the corresponding `%end` statement.
3. Since the preceding `%if` statement evaluated to true, the `%else` clause is not evaluated. Therefore, the associated `%do`-group is not compiled.

If you want to see which lines of code controlled by PL/I preprocessor statements will be preprocessed, create a compilation listing. As shown in Figure 4-6, the compiler inserts a number sign (`#`) in front of each line of the compilation listing containing a PL/I preprocessor statement, that, after preprocessing, will not be compiled.

```
1   main:
2   procedure;
3
4   declare number      fixed bin(15);
              .
              .
              .
35
36        %replace  COMPANY               by 'Stratus';
37
38        %if COMPANY = 'Stratus' %then
39             %do;
40             %replace FILE_NAME_LENGTH  by 255;
41             %replace BUFFER_SIZE       by 8192;
42             %replace OPEN              by s$open_file;
43             %end;
44        %else %if COMPANY = 'Tekno' %then
45 #            %do;
46 #            %replace FILE_NAME_LENGTH  by 2;
47 #            %replace BUFFER_SIZE       by 1;
48 #            %replace OPEN              by error_not_available;
49 #            %end;
               .
               .
               .
```

Figure 4-6. Compilation Listing with Preprocessed PL/I Preprocessor Statements

Unexpected results can occur if you do not pay special attention to a program's logic when mixing PL/I **preprocessor** statements with PL/I **language** statements. In the following code fragment, PL/I preprocessor statements are interspersed among PL/I language statements.

```
if i = 34 then
     %if rev = '9.3' %then
          %do;
          i = i + 1;
          %replace SUB by 'ab';
          call process_data;
          %end;
     .
     .
     .
```

In the preceding example, assume that `rev` is equal to `'9.3'`. Since the `%do`-group controls which text is seen by the compiler, the following text is compiled (after preprocessing).

```
if i = 34 then
          i = i + 1;
          %replace SUB by 'ab';
          call process_data;
```

Despite the misleading indentation, the assignment statement is the **only** statement controlled by the `if` statement. The `%replace` statement and the `call` statement are always executed.

See the *VOS PL/I Language Manual (R009)* for more information about the PL/I preprocessor statements.

Using Both Types of Preprocessor Statements

This section explains how you use VOS preprocessor statements and PL/I preprocessor statements in the same program.

Although you can use VOS preprocessor statements and PL/I preprocessor statements in the same source module, neither preprocessor recognizes symbols that are recognized by the other preprocessor. Also, preprocessor statements with similar names cannot be mixed. (For example, you cannot use `%if` interchangeably with `$if`.)

Figure 4-7 shows part of a source module that contains both types of preprocessor statements.

```
   main:
        procedure;

1. %options default_char_set = latin_1, longmap_check;

2. %include 'rev_list';

   $define VOS

   $if defined (VOS)
        %replace  REV_5                 by '1'B;
   $else
        %replace  REV_5                 by '0'B;
   $endif

        %replace REV_6                  by '0'B;
        %replace REV_7                  by '0'B;
        .
        .
        .
        %if REV_5 %then
             %do;
                  call old_rev_routine;
             %end;
        %else
             %if REV_6 | REV_7 %then
                  %do;
                       call new_rev_routine;
                  %end;
        .
        .
        .
   end;
```

Figure 4-7. Source Module Containing Both Types of Preprocessor Statements

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 4-7.

1. The `%options` statement sets the default character set to Latin alphabet No. 1. It also specifies that the longmap alignment rules be used for compilation and instructs the compiler to diagnose alignment padding in structures.
2. The `%include` statement directs the compiler to insert the text file `rev_list.incl.pl1` into the source module. (The compiler automatically adds the

`.incl.pl1` suffix to the include file name, since it was not specified in the source module.)

See the *VOS PL/I Language Manual (R009)* for more information about the `%options` and `%include` PL/I preprocessor statements.

Chapter 5:
Binding Object Modules

The `bind` command invokes the VOS binder. If you specify a binder control file with the `-control` argument, the binder also preprocesses the binder control file. After the preprocessor finishes processing the binder control file, the binder combines a set of one or more object modules into a program module. While building the program module, the binder resolves symbolic references to external programs and variables that are used by the object modules in a set and in the object library.

This chapter discusses the following topics related to binding object modules.

- ‘‘The `bind` Command”
- ‘‘Summary of Binder Arguments”
- ‘‘Using the Binder”
- ‘‘Using a Binder Control File”
- ‘‘Preprocessing a Binder Control File”

The `bind` Command

This section discusses the following topics.

- ‘‘Syntax of Numerical Binder Values”
- ‘‘Access Requirements”

The `bind` command has the following display form.

Display Form

```
------------------------------------- bind -----------------------------------
object_modules:             █
-control:
-search:
-define:
-entry:
-load_point:                default
-max_heap_size:             default
-max_program_size:          default
-max_stack_size:            default
-number_of_tasks:           1
-pm_name:
-processor:                 default
-size:                      default
-stack_fence_size:          default
-stack_size:                32768
-target_module:            current_module
-version:                   Pre-release
-align_mod16:               no          -compact:                   yes
-dynamic_tasking:           yes         -map:                       no
-private_heap:              no          -private_stack:             no
-profile_alignment_faults:  no          -retain_all:                no
-statistics:                no          -subroutines_are_functions: no
-table:                     yes
```

You can provide the binder with the information it needs to perform the binding through the use of command-line arguments, binder control-file directives, or some combination of the two. A *binder control file* is a text file containing directives for the binder. If you do not use a binder control file, you must specify the object modules to bind in the *`object_modules`* argument of the `bind` command.

Syntax of Numerical Binder Values

Many of the binder command-line arguments and control-file directives accept numerical values to represent addresses and sizes. By default, the binder assumes that any specified numerical value is a decimal value. You can change the value's base by specifying one of the suffixes listed in Table 5-1.

Table 5-1. Base Suffixes for Numerical Binder Values

Suffix	Base
`b` or `B`	Binary
`d` or `D`	Decimal
`o` or `O`	Octal
`x` or `X`	Hexadecimal

For example, if you specify `-load_point 80026000x`, the binder handles the number 80026000 as a hexadecimal value and changes the load point to that address.

You can use one of the multiplier suffixes shown in Table 5-2 to change a numerical binder value.

Table 5-2. Multiplier Suffixes for Numerical Binder Values

Suffix	Description
`kb`	The binder multiplies the specified value by 1024.
`mb`	The binder multiplies the specified value by 1,048,576.
`gb`	The binder multiplies the specified value by 1,073,741,824.

For example, if you specify `-size 8mb`, the binder sets the size of the program address space to 8,388,608 bytes.

You **cannot** specify a multiplier suffix and a base suffix for the same numerical value.

Access Requirements

You need read access to all of the object modules you are binding. You need modify access to the directory where the program module is to be created.

Summary of Binder Arguments

Table 5-3 briefly describes each argument of the `bind` command and lists the locations in this manual in which you can find more information about each argument.

Table 5-3. Arguments of the `bind` Command *(Page 1 of 3)*

Argument	Description	Location of Discussion
`-align_mod16`	Aligns the code from each object module on a 16-byte boundary	‘‘Aligning Code on Byte Boundaries’’
`-compact`	Condenses code on an XA2000-series module	‘‘Condensing Code on an XA2000-Series Module’’
`-control`	Specifies a binder control file	‘‘Using a Binder Control File’’
`-define`	Defines variables to be used during binder control-file preprocessing	‘‘Preprocessing a Binder Control File’’
`-dynamic_tasking`	Includes relocation information in the program module	‘‘Including Relocation Information’’
`-entry`	Defines an entry point as the main entry point	‘‘Specifying Object Modules’’
`-load_in_kernel`	Creates a kernel-loadable program	‘‘Creating a Kernel-Loadable Program’’

Table 5-3. Arguments of the `bind` Command *(Page 2 of 3)*

Argument	Description	Location of Discussion
`-load_point`	Specifies the load point for an object module	‘‘Specifying the Load Point for an Object Module’’
`-map`	Creates a bind map	‘‘Generating a Bind Map’’
`-max_heap_size`	Specifies the maximum size of a heap	‘‘Specifying the Maximum Size of a Heap’’
`-max_program_size`	Specifies the maximum size of a program	‘‘Specifying the Maximum Size of a Program’’
`-max_stack_size`	Specifies the total amount of memory that all tasks’ stacks and fence can occupy	‘‘Allocating Memory for Static Tasks’’
`-number_of_tasks`	Specifies the number of static tasks to create in the program module	‘‘Specifying the Number of Static Tasks to Create in the Program Module’’
`-pm_name`	Names a program module	‘‘Naming a Program Module’’
`-private_heap`	Moves the process heap to the process’s private address space	‘‘Moving the Process Heap’’
`-private_stack`	Moves the process stack to the process’s private address space	‘‘Moving the Process Stack’’
`-processor`	Specifies the processor on which the program module is to run	‘‘Locating the Required Object Modules When Cross-Binding’’ and ‘‘Specifying a Processor’’
`-profile_alignment_faults`	Counts the number of alignment faults that occur during program execution	‘‘Counting Alignment Faults’’
`-references_kernel`	Resolves external references in the kernel	‘‘Resolving External References in the Kernel’’
`-relocatable`	Produces a program module that can be loaded at any address	‘‘Creating Program Modules That Can Be Loaded at Any Address’’
`-retain_all`	Places all external entry-point names in the program’s entry map	‘‘Specifying Retained Entry Points’’
`-search`	Specifies a set of search directories	‘‘Specifying the Directories to Search for Object Modules’’

Table 5-3. Arguments of the `bind` Command *(Page 3 of 3)*

Argument	Description	Location of Discussion
`-size`	Specifies the size of a program's address space	``Specifying the Size of a Program's Address Space''
`-stack_fence_size`	Specifies the size of a stack fence	``Specifying the Size of a Stack Fence''
`-stack_size`	Specifies the number of bytes of storage to reserve for the stack	``Specifying the Size of a Stack''
`-statistics`	Displays information about each phase of the binding operation	``Displaying Statistics about the Binding''
`-subroutines_are_functions`	Suppresses certain VOS Standard C messages	``Suppressing Certain VOS Standard C Messages''
`-table`	Includes symbol-table information in the program module	``Including a Symbol Table''
`-target_module`	Checks the spelling of all VOS subroutine names called by the object modules being bound	``Checking VOS Subroutine Names''
`-version`	Specifies the release number to appear in the program module header	``Specifying a Version Number''

Using the Binder

This section discusses the following topics related to the `bind` command's arguments. Note that the `-define` argument is described in ''Preprocessing a Binder Control File'' later in this chapter.

- ''Naming a Program Module''
- ''Specifying Object Modules''
- ''Locating Object Modules and Entry Points''
- ''Resolving External References''
- ''Checking VOS Subroutine Names''
- ''Specifying a Processor''
- ''Generating a Bind Map''
- ''Specifying the Size of a Program's Address Space''
- ''Displaying Statistics about the Binding''
- ''Moving the Process Heap''
- ''Moving the Process Stack''
- ''Aligning Code on Byte Boundaries''
- ''Creating a Kernel-Loadable Program''
- ''Condensing Code on an XA2000-Series Module''
- ''Including Relocation Information''
- ''Including a Symbol Table''
- ''Specifying the Number of Static Tasks to Create in the Program Module''
- ''Creating Program Modules That Can Be Loaded at Any Address''
- ''Resolving External References in the Kernel''
- ''Counting Alignment Faults''
- ''Specifying the Size of a Stack''
- ''Specifying the Size of a Stack Fence''
- ''Specifying the Maximum Size of a Heap''
- ''Specifying the Maximum Size of a Program''
- ''Allocating Memory for Static Tasks''
- ''Suppressing Certain VOS Standard C Messages''
- ''Specifying a Version Number''
- ''Specifying the Load Point for an Object Module''
- ''Initializing External Variables That Have Message Names''

Naming a Program Module

The binder generates a program module (`.pm` file), puts it in your current directory by default, and names it. The binder names the program module depending on which command-line arguments (*`object_modules`*, `-control`, and `-pm_name`) are specified. If more than one of these arguments are specified, the binder names the program module according to the following rules, which are specified in descending order (for example, rule 1 overrides any other rule specified, rule 2 overrides rules 3 and 4 but is overridden by rule 1, and so on).

1. If you specify the `-control` argument, the binder gives the program module the name specified in the `name` binder control-file directive.

2. If you specify the `-pm_name` *`name`* argument, the binder gives the program module the name specified in *`name`*.

3. If you specify the `-control` argument, the binder gives the program module the name of the binder control file.

4. If you specify neither the `-pm_name` argument nor the `-control` argument, the binder gives the program module the **first** name specified in the *`object_modules`* argument.

In all cases, the binder gives the program module the `.pm` suffix in place of the `.bind` or `.obj` suffix.

Specifying Object Modules

You can specify the object module or modules to bind by using the *`object_modules`* command-line argument of the `bind` command, as shown in the following syntax.

```
bind object_module...
```

Each *`object_module`* is the name of a file that contains a compiled source module and has the suffix `.obj`. Specify the object module names by entering a full or relative path name or a star name with or without the `.obj` suffix. Separate object module names with spaces. The following example illustrates.

```
bind add_num init_data
```

Typically, the first procedure in the first source module specified in the `bind` command is the main entry point of the executable program. If you want to define a different entry point as the main entry point, specify the `-entry` argument or the `entry` binder control-file directive. For more information about the program entry point in a VOS PL/I program, see the description of the `entry` binder control-file directive in ‘‘Specifying Directives” later in this chapter.

If an object module contains a call to an entry point that is defined in another object module, you need not specify the name of the latter object module when the object module that defines the called entry point has the same name as the entry point. The binder can locate the object module in the search directories.

See ‘‘Locating Object Modules and Entry Points” later in this chapter for information about the directories that the binder searches for the object modules.

To call an entry point whose name differs from the name of the object module in which it is defined, you must specify the name of the object module to the binder, or use the `add_entry_names` command to create a link to the object module. For example, if a program calls the entries `mul` and `div`, which are separately compiled as `mul.obj` and `div.obj`, and the binder can locate these files, you need **not** list these object modules for the binder. If `mul` and `div` are entry points within `do_arithmetic.obj`, though, you must list `do_arithmetic` as one of the object modules to be bound, or use `add_entry_names`.

For more information about using `add_entry_names`, see ‘‘Using the `add_entry_names` Command” later in this chapter.

Locating Object Modules and Entry Points

The binder links together the object modules that you have specified and additional object modules that are referenced as entry points within the object modules being bound. Each object module can contain calls to additional entry points that are defined within other object modules.

You must know where all object modules and entry points are located, and instruct the binder how to find them. The following sections describe several ways to do this.

- ‘‘Specifying the Directories to Search for Object Modules’’
- ‘‘Locating the Required Object Modules When Cross-Binding’’
- ‘‘Using the `add_entry_names` Command’’
- ‘‘Specifying Retained Entry Points’’

Specifying the Directories to Search for Object Modules

The procedure that the binder uses to search for the `.obj` files differs depending on how you specify the object modules to be bound. If you specify a full or relative path name for an object module in the *`object_modules`* command-line argument or in the `modules` binder control-file directive, the specified path name must locate the object module. The binder does **not** search in any other location for the object module.

In contrast, to find an object module that defines an entry point called within one of the object modules being bound, or that is a simple file name for an object module named in a `modules` binder control-file directive, the binder looks in a set of *search directories* in the following order.

1. any directories specified with the `-search` command-line argument
2. any directories specified in the `search` directive of the binder control file
3. the directories specified in the process’s object library paths list

In each of the preceding sets of directories, the binder searches the directories in the order in which they are listed. See the *Introduction to VOS (R001)* for a complete description of search rules.

The *object library paths list* for a process consists of an ordered sequence of path names for one or more directories in which the binder searches for object modules. Each path name specifies a directory containing one or more object modules. If the binder does not find the object module, it generates a warning indicating the bind error.

Your process’s object library paths list typically contains the following path name.

```
(master_disk)>system>object_library
```

Note: Your process’s object library paths list can also contain other path names.

You can use the `list_library_paths` command to display the directory path names in your process’s object library paths list.

The `-search` argument and the `search` binder control-file directive enable you to specify one or more directories where the binder is to look before it searches the directories specified in the process’s object library paths list. You specify the `-search` argument or the `search` directive when you have stored an object module in a directory other than that listed in your object library paths list. The `-search` argument has the following syntax.

```
bind object_module... -search [search_directory]...
```

For example, if you want to use the object module named `commissions`, which is stored in the directory `>sales>tools`, you could use the following `-search` argument when issuing the `bind` command.

```
bind commissions -search >sales>tools
```

With the preceding `-search` argument, the binder searches for the object module in the `>sales>tools` directory before searching the directories listed in the process's object library paths list.

When you specify the `-search` argument but do not specify a directory, the binder searches the current directory before searching directories specified in a `search` binder control-file directive, if any, and before searching the process's object library directories. If you do not specify the `-search` argument, the binder first searches the directories listed in the `search` binder control-file directive, if any, and then searches the process's object library directories.

The binder allows as many command-line and binder control-file search directories as memory allows. The binder searches a directory only once even if it is specified more than once.

Be aware that if the binder finds an entry point more than once, it issues the following error message.

```
bind: External name entry_point_name is multiply defined.
```

The *`entry_point_name`* value is the name of the entry point.

Locating the Required Object Modules When Cross-Binding

Cross-binding occurs when a binder running on a processor from one processor family binds object modules into a program module that will execute on a processor from a different processor family. The processor on which the code is bound is called the *host processor*. The processor on which the code is to run is called the *target processor*.

For example, cross-binding occurs when you are binding an object module on an XA2000-series module, and the resulting program module will run on an XA/R-series module. Most programmers bind object modules into a program module that will run on the current module, and therefore do not need to be concerned with cross-binding.

> **Note:** Although VOS Release 14.0.0 does not support XA2000-series modules, you can still generate code and cross-bind object modules for an XA2000-series module from an XA/R-series module or a Continuum-series module.

When you are cross-binding, you use the `bind` command's `-processor` argument to indicate the processor on which the program module is to run. See ‘‘Specifying a Processor’’ later in this chapter for information about this argument.

During cross-binding, the binder must be able to locate the object modules that have been compiled for the appropriate processor family. The binder must be able to find the object modules provided by the system and the other program object modules that will be bound together into a program module. The process of setting up a search list so that the binder searches the directories containing the required object modules can be simplified by using a binder control file or a command macro devoted exclusively to cross-binding. The binder will find the appropriate object modules if you perform one of the following actions.

- Specify the required directories using the `bind` command's `-search` argument or the `search` binder control-file directive.
- Add the required library paths and delete any unneeded library paths from the object library paths list using the `add_library_path` and `delete_library_path` commands. You can also use the `set_library_paths` command to change the directories in the object library paths list.

The standard object library paths list specifies the following directory.

```
(master_disk)>system>object_library
```

This standard object library path contains the object modules needed to bind a program that will run on the current module and on other modules that use a processor from the same processor family as is used in the current module.

Table 5-4 lists the object-module directories needed to perform cross-binding.

Table 5-4. Object Library Paths for Cross-Binding

Host Module's Processor Type	Target Module's Processor Type	Directory Needed for Cross-Binding
i860	PA-7100	`(master_disk)>system>object_library.7100`
	PA-8000	`(master_disk)>system>object_library.8000`
	MC68*xxx*	`(master_disk)>system>object_library.68k`
PA-7100	i860	`(master_disk)>system>object_library.860`
	PA-8000	`(master_disk)>system>object_library.8000`
	MC68*xxx*	`(master_disk)>system>object_library.68k`
PA-8000	i860	`(master_disk)>system>object_library.860`
	PA-7100	`(master_disk)>system>object_library.7100`
	MC68*xxx*	`(master_disk)>system>object_library.68k`

For more information about cross-binding, see the manual *Migrating VOS Applications to the Stratus RISC Architecture (R288).*

Using the `add_entry_names` Command

An object module can contain a number of entry points. Other object modules can contain external references to these entry points. To bind object modules into a program module, the binder must find a definition for every external entry point referenced in the program module. The object module containing the definition must be either explicitly specified as an object module to be bound or the binder must be able to find, in its search directories, a name of the form *`entry`*`.obj` for an external entry point. The name can be a link to an object module containing the entry point.

You can use the `add_entry_names` command to create links to one or more object modules that contain the definitions of the entry points. Using these links, the binder will be able to resolve the references. See the *VOS Commands Reference Manual (R098)* for more information about the `add_entry_names` command.

To resolve an external reference in one object module to an external name defined in another object module, the binder must find an object module containing the external name in a directory specified in one of the following:

- the `-search` argument
- the `search` directive
- the object library paths list

Each external name should be identified by a link to an object module or by the object module itself.

If you invoke the `add_entry_names` command, specifying each directory that the binder will search, the `add_entry_names` command looks in one or more object modules for entry points and creates links to the object modules that contain the entry points. For entry points, the link has the name of the entry point and the `.obj` suffix.

Specifying Retained Entry Points

When an external entry point is *retained*, its name is included in the entry map for the program module. If your program uses the `s$find_entry` subroutine or uses tasking, the binder must be directed to place one or more external entry-point names in the entry map for the program module.

- The `bind` command's `-retain_all` argument places all external entry-point names in the program's entry map.

- The `retain` binder control-file directive places one or more external entry-point names in the program's entry map.

For information about retained entry-point names for use with the `s$find_entry` subroutine, see the *VOS PL/I Subroutines Manual (R005)*. For information about retained entry-point names for use in a tasking application, see the *VOS Transaction Processing Facility Guide (R215).*

Resolving External References

The binder resolves symbolic references to entry points that have external scope. In the set of object modules that constitutes a program module, each instance of a particular identifier with external scope denotes the same procedure or variable. See the *VOS PL/I Language Manual (R009)* for information about external scope.

An *entry point* is a location in a program where execution can be transferred to an external routine. You can specify an entry point in a procedure or in a binder control file. In a PL/I program, all procedures with external scope are entry points.

To bind object modules into a program module, the binder must find a definition for every external entry point referenced in the program module. To find these definitions, **either** of the following situations must occur.

- The object module containing the definition must be explicitly specified as an object module to be bound.
- The binder must be able to find, in its search directories, an object module (or a link to an object module) that contains the definition of the external entry point.

The binder is case-sensitive. If you compile a source module using `-mapcase`, you may not be able to bind the resulting object module. In particular, if the source module contains an external entry name or external variable name, and the name has one or more uppercase letters, the binder will not recognize the original name and its lowercase version as the same name. You can use the `synonym` binder control-file directive so that the binder recognizes the two versions of the name as the same name. (See the description of `synonym` later in this chapter for more information.)

See ‘‘Locating Object Modules and Entry Points” earlier in this chapter for more information about how the binder locates the object modules that define the program’s external entry points.

Checking VOS Subroutine Names

When you bind object modules, the binder checks the spelling of all system subroutine names called by the object modules to ensure that they match all known kernel entry points on the host system. If you are binding for a different architecture or a different software release, however, the kernel entry points may not be the same, and the bind may result in undefined system entry points.

To avoid this problem, specify the `-target_module` *`module`* argument, where *`module`* is the name of a module on the current system or on another accessible system. The binder will then look on *`module`* for all VOS subroutine names. If *`module`* is inaccessible for any reason, the binder issues a warning and binds the object module on the current module instead.

For example, if your current module is an XA2000-series module, and you want to bind an object module to run on an XA/R-series module named `#m20`, specify the following command.

```
bind employee_info -target_module #m20 -processor i80860
```

The default value for *`module`* is the current module.

Specifying a Processor

If you select the `bind` command's `-processor` argument or the `processor` binder control-file directive, the specified processor value explicitly tells the binder the processor for which the object modules were compiled. If you do **not** select the `-processor` argument or the `processor` binder control-file directive, the binder expects object modules that were compiled for the default system processor.

The binder uses the value specified in the `-processor` argument for two purposes. First, the value selected determines which preprocessor variable the binder defines. Second, the value selected allows the binder to check that the object modules are compatible with the specified processor type. If an object module is not compatible with the specified processor type, the binder issues a warning.

The processor value that you select for the `bind` command's `-processor` argument **must** be a value that is of the same processor family specified when the object modules were compiled. The `bind` command's `-processor` argument has the following values.

- `default`
- `mc68000`
- `mc68020`
- `mc68020/mc68881`
- `mc68030`
- `mc68030/mc68882`
- `i80860`
- `i80860xp`
- `pa7100`
- `pa8000`

The `default` value specifies the current module's default system processor. This value, as well as some of the other values in the preceding list, can be changed by your system administrator. To display the current default value for the `-processor` argument, issue either of the following commands.

```
display_error m$default_processor
```

```
display_error 3932
```

In either case, the output shows the current default value for `-processor` on your module. See the manual *VOS System Administration: Administering and Customizing a System (R281)* for more information about setting values for the `-processor` argument.

Depending on the value specified in the `-processor` argument or the `processor` binder control-file directive, the binder automatically defines one or two preprocessor variables for the processor family.

- If the value indicates a processor from an XA2000-series module, the binder defines `__MC68K__` as a preprocessor variable.
- If the value indicates a processor from an XA/R-series module, the binder defines `__I860__` as a preprocessor variable.
- If the value indicates a processor from a Continuum-series module, the binder defines `__HPPA__` as a preprocessor variable. In addition, the binder defines `__HPPA11__` or `__HPPA20__`, depending on the processor's specific family (the PA-7100 family or the PA-8000 family).

For information about defining and using preprocessor variables, see ‘‘Preprocessing a Binder Control File” later in this chapter. See also Chapter 4.

Generating a Bind Map

If you select the `-map` argument, the binder creates a bind map in your current directory. The bind map contains information about objects and procedures bound into your program. The bind map has the same name as the program module with a `.map` suffix in place of the `.pm` suffix.

Figure 5-1 shows part of a bind map for the program `employee_info`.

1.

```
 Bound for: Mary_Doe.Human_Resources at hr
 Bound by: bind, Release 12.0
 Bound on: 92-07-24 11:08:36 EDT
```

2.

```
Options:      -compact -table paths -map
Program size: small
```

3.

```
Search directories:

     1    %hr#d20>Human_Resources>Mary_Doe>pl1
     2    %hr#d20>system>object_library
     3    %hr#d20>system>sql_apt_object_library
     4    %hr#d20>system>sql_object_library
     5    %hr#d20>system>c_object_library
```

4.

```
Section      Code            Symtab           Unshared          Shared
         Start    Length   Start    Length   Start    Length   Start    Length
paged   00E00000 000051CA 00E08000 00003538 00E06000 00000EA8 00E07000 000001C2
```

5.

```
Module Map:

Name                            Directory Index               Date Compiled
         Scn        Code              Symtab             Unshared  UERW, SAP
             Start    Length    Start    Length    Start    Length
employee_info                   1                             92-07-24 11:14:37
          p 00E00000 00000C12  00E08000 0000015C  00E06000 0000009C
s$pl1_read                      2                             89-09-08 01:42:49
          p 00E00C18 000006DC  00E0815C 00000358  00E06410 00000064
s$pl1_error                     2                             89-09-08 01:21:10
          p 00E012F8 00000090  00E084B4 0000019C  00E06478 00000026
.
.
.
```

6.

```
External Variable Map:

Name                Scn Address  Length
emp_file             p  00E07000 000001B8 shared
s$pl1_first_time     p  00E071C0 00000002 shared
s$plio_debug         p  00E071B8 00000002 shared
s$plio_fcb_chain     p  00E071BC 00000004 shared
s$u$plio_cursor      p  00E06474 00000002
sysin                p  00E06258 000001B8
sysprint             p  00E060A0 000001B8
```

7.

```
Minimum stack size:           32768
```

(Continued on next page)

```
8.  Main Entrypoint:

    Name                 Code       Unshared
                         Address    Address
    employee_info        00E00124   00E0E000


9.  External Name Definitions:

    Name           Scn Rgn Ref   Address  Unshared   Type     Defining
                           Count          Length              Module
    emp_file        p   s      1 00E07000 000001B8 variable   employee_info
    employee_info   p   c      0 00E00124 00E0E000 entrypoint employee_info
    s$pl1_close     p   c      2 00E03B44 00E0E998 entrypoint s$pl1_close
    s$pl1_error     p   c     13 00E012FC 00E0E478 entrypoint s$pl1_error
    .
    .
    .
10. Undefined External Names:

    Name                  Ref      Use    Referencing
                         Count              Module
    s$add_epilogue_handler
                             1 entrypoint s$pl1_open
    s$add_task_epilogue_handler
                             1 entrypoint s$pl1_open
    s$attach_port            1 entrypoint s$pl1_open
    s$close                  1 entrypoint s$pl1_close
    .
    .
    .
```

Figure 5-1. Sample Bind Map

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 5-1. In the bind map, all addresses and lengths are specified using hexadecimal format unless the description states some other format.

1. The bind map's first section contains the following information.

    - `Bound for` specifies the user name, group name, and system name for the user process that invoked the `bind` command.
    - `Bound by` specifies the name of the command that invoked the binder, and the operating system release number.
    - `Bound on` specifies the date and time that the binder was invoked.

2. The `Options` and `Program size` sections show the options and the size of the program's address space that were in effect for the binding. Program size and most options can be specified with either command-line arguments or binder control-file directives.

3. The `Search directories` section shows the full path names of all directories specified in the following places.

- the `-search` command-line argument
- the `search` binder control-file directive
- the process's object library paths list

For each directory, the `Search directories` section also shows a directory-index number, starting with `1`, that the binder uses to identify the directory in the `Module Map` section. For information about how the binder searches for object modules, see "Specifying the Directories to Search for Object Modules" earlier in this chapter.

4. This section shows the starting address and the length within virtual memory of the following program module elements.

- `Code` specifies executable code.
- `Symtab` specifies the symbol table, if any.
- `Unshared` specifies unshared static data.
- `Shared` specifies shared static data.

In addition, any user-defined sections would appear in this section. See the description of the `section` binder control-file directive later in this chapter for information about defining your own sections.

5. The `Module Map` section shows information about all object modules included in the program module. The `Module Map` information is arranged into the following columns.

- `Name` specifies each object module's name.

- `Scn` specifies the section of the address space in which the binder is to locate the object modules: `p` indicates the paged section, `w` indicates the wired section, and `i` indicates the initialization section.

- `Code` specifies each object module's starting address and length within the program code region.

- `Symtab` specifies each object module's starting address and length within the symbol table. The decimal integer above the starting address is the directory-index number of the directory where the binder located the object module. The binder generates these numbers, and they are listed in the bind map's `Search directories` and `Other directories` sections. (A bind map might not contain an `Other directories` section.)

- `Unshared` specifies the starting address and length of each object module's unshared static data.

- `UERW` and `SAP` specify information that is irrelevant to this discussion.

- `Date Compiled` specifies the date and time that each object module was compiled.

6. The `External Variable Map` section shows information about all variables with external scope that are declared in the program module. The `External Variable Map` information is arranged into the following columns.

   - `Name` specifies the name of the external variable.

   - `Scn` specifies the section of the address space in which the binder is to locate the object modules: `p` indicates the paged section, `w` indicates the wired section, and `i` indicates the initialization section.

   - `Address` specifies the beginning address of the storage that has been allocated for the external variable.

   - `Length` specifies the number of bytes that have been allocated for the external variable.

   In addition, the rightmost column may contain the word `shared` to indicate that an external variable has been allocated in the **shared** static region. If `shared` does not appear, the external variable has been allocated in the **unshared** static region.

7. The `Minimum stack size` section shows the minimum number of bytes, in decimal, that are available for stack data.

8. The `Main Entrypoint` section shows information about the program module's main entry point.

   - `Name` is the name of the main entry point.
   - `Code Address` is the starting address of the main entry point's executable code.
   - `Unshared Address` is an address used for accessing static data.

9. The `External Name Definitions` section shows information about all variables and entry points that are referenced and defined in the program module, and that have external scope. In a PL/I program, entry points are procedures. The `External Name Definitions` information is arranged into the following columns.

   - `Name` specifies the name of the variable or procedure.

   - `Scn` specifies the section of the address space in which the binder has located the object modules: `p` indicates the paged section, `w` indicates the wired section, and `i` indicates the initialization section.

   - `Rgn` specifies the region within the program's address space in which the variable has been allocated or in which the entry point is located: `c` indicates the code region, `s` indicates the shared static region, and `u` indicates the unshared static region.

   - `Ref Count` specifies the number of times that the program module references the variable or entry point.

   - `Address` specifies the beginning address of storage that has been allocated for a variable, or the beginning address where the code for an entry point is located.

  - `Unshared Length` is a static address that is used to address static data.
  - `Type` specifies the type of the name: either variable or entry point.
  - `Defining Module` specifies the name of the object module that defines the variable or entry point.

10. The `Undefined External Names` section shows information about all variables and entry points that are referenced but not defined in the program module, and that have external scope. In most VOS PL/I programs, undefined external names are VOS subroutine names whose code is bound into the operating system kernel. The `Undefined External Names` information is arranged into the following columns.
    - `Name` specifies the name of the variable or entry point.
    - `Ref Count` specifies the number of times that the program module references the variable or entry point.
    - `Use` specifies the use of the name: either variable or entry point.
    - `Referencing Module` specifies the name of the object module that first references the variable or entry point.

The following sections can also appear in the bind map.

- If an error occurs during the binding, the bind map includes a `Linking Errors` section that shows the message associated with the error.
- If a binder control file was used to direct the binding, the bind map includes a `Control File` section that shows the path name and text of the preprocessed binder control file.
- If the binder finds object modules whose directories were not specified as search directories, the bind map includes an `Other directories` section, which shows the full path names of all such directories. For example, this section lists the path names of directories specified with the *`object_modules`* command-line argument and the `modules` binder control-file directive. For each directory, this section also lists a directory-index number that the binder uses to identify each directory in the `Module Map` section.

Specifying the Size of a Program's Address Space

The `-size` argument specifies the size of the address space for which the binder is to bind the object modules. You can specify any numeric value for the `-size` argument, as well as the values `small` (to specify a 2-megabyte address space) and `large` (to specify an 8-megabyte address space).

See ‘‘Syntax of Numerical Binder Values” earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-size`.

On an XA2000-series module, you must specify a `-size` value of `large`, `8mb`, or `64mb` in two situations.

- If the generated code of all object modules needs more than two megabytes, the binder generates an error.
- If a bound program module fits in a smaller space, but there is insufficient dynamic storage space (usually because of stack requirements), the operating system issues an error message and aborts the program when you try to run it.

In either of the preceding situations, you can rebind the object modules, specifying a larger size with the `-size` argument or the `size` binder control-file directive.

The binder uses the following rules to determine a program's address space size.

- If the `size` binder control-file directive is specified, the binder uses the specified value.
- If the `size` binder control-file directive is not specified, the binder uses the value specified in the `-size` argument.
- If no value is specified, the binder determines the default value based on the values shown in the following table.

Processor Type of Target Module	**Size Used for a User Program**	**Size Used for a Kernel Program**
XA2000-series	2 megabytes	8 megabytes
XA/R-series	64 megabytes	64 megabytes
Continuum-series	64 megabytes	4096 megabytes

The program's address space includes areas for the program's code, unshared static data, shared static data, and symbol table, as well as stack space, heap space, maps, and fences.

The *unshared static region* stores variables that have static storage duration but that are not defined with the `external shared` attribute. In a tasking program, each task has its own space in the unshared static region. This region is sometimes called the *internal static region*.

The *shared static region* (also known as the *external static region*) stores shared variables and is used for tasking programs. Shared variables have static storage duration and external scope, and are declared with the `external shared` attribute. If a variable is shared, every task in a program references the same address space for the variable.

A *fence* is an unmapped area of memory; its purpose is to prevent runaway stacks from overwriting other data.

Figure 5-2 shows the layout of a program's address space on XA2000-series modules and XA/R-series modules.

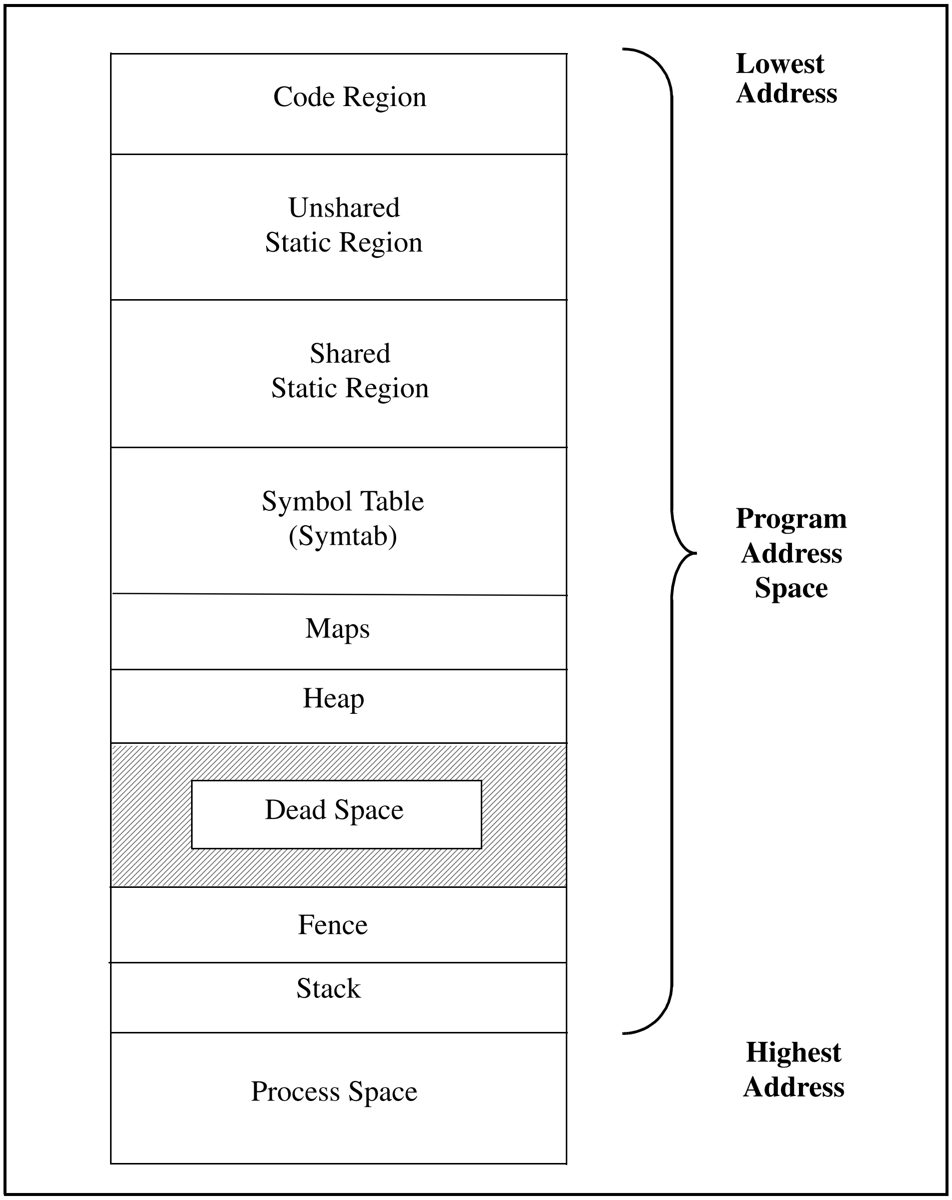


Figure 5-2. Program Address Space on XA2000-Series and XA/R-Series Modules

Figure 5-3 shows the layout of a program's address space on Continuum-series modules.

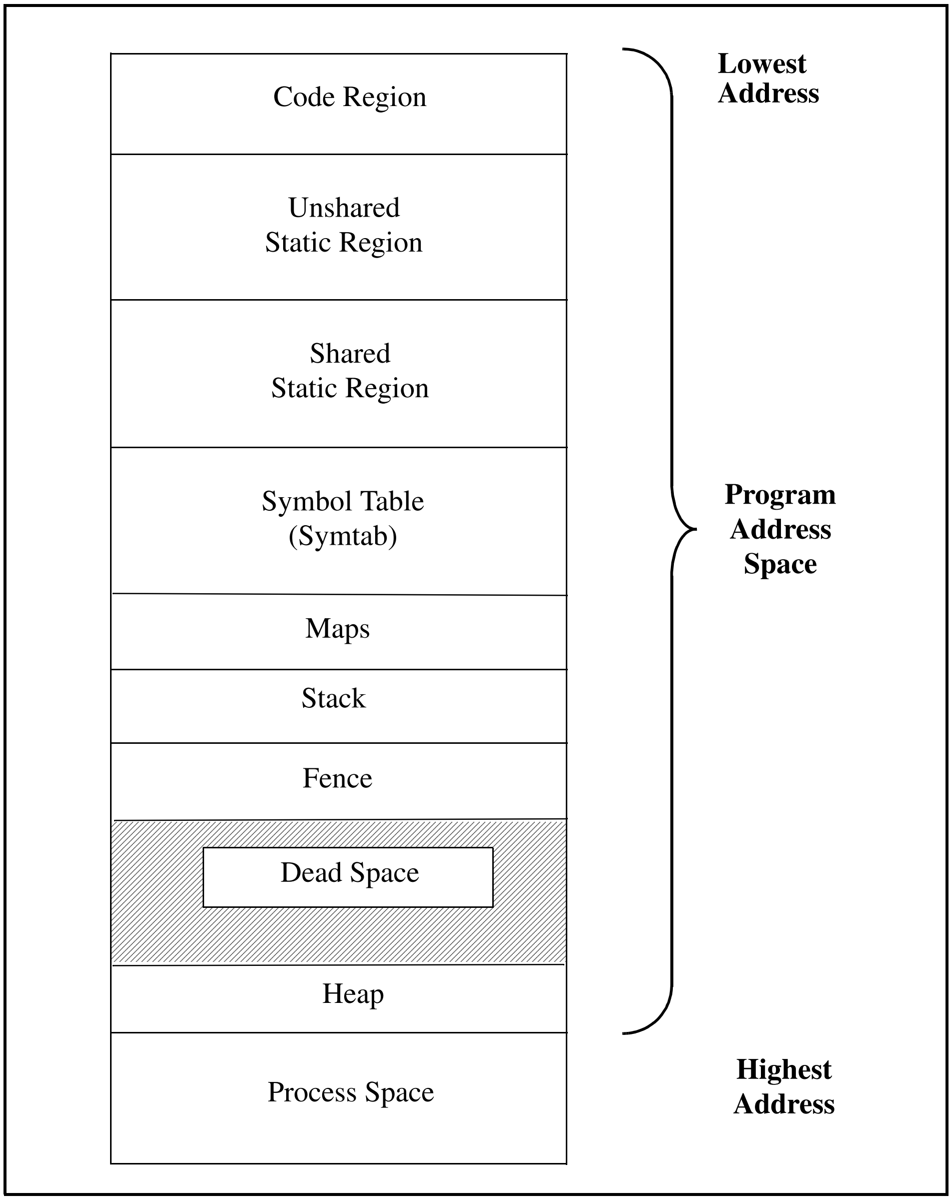


Figure 5-3. Program Address Space on Continuum-Series Modules

Displaying Statistics about the Binding

If you select the `-statistics` argument, the binder displays information about each phase of the binding operation. Figure 5-4 shows the information that the binder displays for a program bound with the `-statistics` argument.

```
Binder Statistics

Phase     Seconds      Paging   Disk I/O        CPU
pass1          12          74        102          5  15:40:32
pass2           0           3          0          0  15:40:32
map             1           3          0          0  15:40:33
TOTAL          13          80        102          5  15:40:33

Total bytes:                  45604
Bytes per minute:            547248
Bytes removed:                 5448
Size of program:                 14
```

Figure 5-4. Binding Statistics

The binding statistics shown in Figure 5-4 are arranged into the following columns.

- `Phase` is the phase of the bind: `parse`, `pass1`, `pass2`, `map`, and the total of all of the phases. If you use a binder control file, the `parse` phase provides information about the parsing of the binder control file. Regardless of whether you specify the `-map` argument, the `map` phase provides information about the creation of the bind map and tables in the program module.
- `Seconds` is the number of seconds the binder took to perform each phase.
- `Paging` is the number of page faults that occurred during each phase.
- `Disk I/O` is the number of times the binder had to access the disk to read the binder control file and the object files, and to read and write to the program module.
- `CPU` is the number of seconds the CPU took to process each phase.
- The last column indicates the time when the binder completed each phase.

Below the preceding information, the binder displays the following statistics.

- `Total bytes` is the total number of bytes in the program module before the binder increased the size of each program section to align with a page of disk space.
- `Bytes per minute` is the number of bytes the binder read and wrote per minute.
- `Bytes removed` is the number of bytes the binder removed by compacting the code.
- `Size of program` is the number of pages in the program module after the binder increased the size of each program section to align with a page of disk space.

Moving the Process Heap

If you specify the `-private_heap` argument, the binder moves the process heap to the process's private address space. This allows the compiler to place the address spaces of otherwise identical processes into different cache locations, often resulting in better system performance by reducing the amount of cache line contention. (See the manual *Migrating VOS Applications to Continuum Systems (R407)* for more information about cache line contention.) Note, however, that this argument does not override the heap limits discussed later in this chapter in ‘‘Specifying the Maximum Size of a Heap.’’ The default value is `-no_private_heap`.

You cannot use this argument if your program uses the `s$connect_vm_region2` subroutine to connect shared virtual memory to the program’s heap area. If your program uses `s$connect_vm_region2` for this purpose and you specify `-private_heap`, the subroutine will return an error message. This argument operates independently of the `-private_stack` argument, which means that you can specify one argument or both arguments simultaneously. This argument has no effect if the program is executed on a Continuum-series module running a release prior to VOS Release 13.1.0 or on an XA2000-series or XA/R-series module.

Moving the Process Stack

If you specify the `-private_stack` argument, the binder moves the process stack to the process’s private address space. This allows the compiler to place the address spaces of otherwise identical processes into different cache locations, often resulting in better system performance by reducing the amount of cache line contention. (See the manual *Migrating VOS Applications to Continuum Systems (R407)* for more information about cache line contention.) Note, however, that this argument does not override the stack limits discussed later in this chapter in ‘‘Allocating Memory for Static Tasks.’’ The default value is `-no_private_stack`.

You cannot use this argument if your program uses the `s$connect_vm_region2` subroutine to connect shared virtual memory to the program’s stack area. If your program uses `s$connect_vm_region2` for this purpose and you specify `-private_stack`, the subroutine will return an error message. This argument operates independently of the `-private_heap` argument, which means that you can specify one argument or both arguments simultaneously. This argument has no effect if the program is executed on a Continuum-series module running a release prior to VOS Release 13.1.0 or on an XA2000-series or XA/R-series module.

Aligning Code on Byte Boundaries

If you specify the `-align_mod16` argument, the binder aligns the code from each object module on a 16-byte boundary. This argument may produce faster code. By default, the binder aligns the code on an 8-byte boundary for XA2000-series modules, on a 16-byte boundary for XA/R-series modules, and on a 64-byte boundary for Continuum-series modules.

Creating a Kernel-Loadable Program

If you specify the `-load_in_kernel` argument, the binder creates a program module that can be loaded with the `load_kernel_program` command. The `load_kernel_program` command loads a program module, such as a library of user programs, separately into the operating system kernel. See the manual *VOS System Administration: Administering and Customizing a System (R281)* for more information about `load_kernel_program`.

Notes:

1. The `-load_in_kernel` argument is used primarily for Stratus internal development. Most users should **not** use this argument.
2. The `-load_in_kernel` argument no longer appears in the `bind` command's display form, but you can still specify it on the command line.

Condensing Code on an XA2000-Series Module

The following discussion of the `-compact` and `-no_compact` binder arguments is relevant **only** when the program module will run on an XA2000-series module. Use of these arguments affects the code resulting from only those source modules that have been compiled at optimization level 3 or less. (The code for source modules compiled at optimization level 4 has already been compacted by the compiler.)

Unless you select the `-no_compact` argument, the binder condenses the code, replacing long branch instructions with short branch instructions wherever possible. If you select `-no_compact`, the binder binds the object modules without condensing the code in the resulting program module. The default argument, `-compact`, produces more efficient bound code than `-no_compact`.

Because shortening branch instructions changes the size of the code in the program module, the assembly language listing produced by the compiler for any of the object modules may become invalid for the corresponding code in the program module. However, the run-time statement map, which the debugger uses, remains correct.

Including Relocation Information

If you specify the `-dynamic_tasking` argument, the binder includes relocation information in the program module. Such information is needed by a program that may change the number of dynamic tasks during execution.

If your program does not use dynamic tasks, you can decrease the size of the program module by specifying `-no_dynamic_tasking`. In addition, if your program is not a tasking program, specifying `-no_dynamic_tasking` also suppresses certain warnings. See the *VOS PL/I Transaction Processing Facility Reference Manual (R015)* for information about dynamic tasks. By default, the binder includes some relocation information.

Including a Symbol Table

If you specify the `-table` argument, the binder includes symbol table information in the program module. If you select `-no_table`, symbol table information is not included in the program module. In addition, `-no_table` strips the statement map from the program module. Thus, with `-no_table`, the resulting program module is smaller in size, but you are restricted to using the debugger in machine mode. With `-no_table`, it is **very difficult** to debug a program, even in machine mode. For example, if you select the `-no_table` binder argument, source-line information, which would otherwise be present, is not available to the debugger for machine-mode debugging.

By default, the binder includes symbol table information in the program module. If you do not use `-no_table` when binding and select either `-table` or `-production_table` when compiling the source module, you can use the debugger in source mode. See the *VOS Symbolic Debugger User's Guide (R308)* for more information about how the `-no_table` argument affects debugging.

> **Note:** Unlike the `pl1` command's `-table` argument, the `bind` command's `-table` argument does not affect the executable code.

Specifying the Number of Static Tasks to Create in the Program Module

The `-number_of_tasks` argument specifies the number of static tasks the binder creates in the program module. The number of static tasks is limited only by the program module's total size. Each static task has its own stack, its own fence (to prevent tasks from overwriting each other), and its own copy of static storage. The last task's stack always has a fence size of at least 32 kilobytes. By default, the binder creates one static task.

See the description of tasking in the *VOS PL/I Transaction Processing Facility Reference Manual (R015)* for more information.

Creating Program Modules That Can Be Loaded at Any Address

The `-relocatable` argument causes the binder to produce a program module that can be loaded at any address. Only **kernel-loadable** programs should be bound with this argument. By default, the binder does not produce a program module that can be loaded at any address.

Notes:

1. The `-relocatable` argument is used primarily for Stratus internal development. Most users should **not** use this argument.
2. The `-relocatable` argument no longer appears in the `bind` command's display form, but you can still specify it on the command line.

Resolving External References in the Kernel

If you specify the `-references_kernel` argument, the binder allows virtually all external references that can be resolved in the kernel to be resolved. By default, the binder does not allow external references to the VOS kernel, other than system calls, to be resolved.

Notes:

1. The `-references_kernel` argument is used primarily for Stratus internal development. Most users should **not** use this argument.
2. The `-references_kernel` argument no longer appears in the `bind` command's display form, but you can still specify it on the command line.

Counting Alignment Faults

The `-profile_alignment_faults` argument instructs the operating system to count the number of alignment faults that occur during program execution. This data is reported by the `profile` command. The alignment fault count replaces the page fault count in program modules that have been compiled with the `-cpu_profile` argument. See the description of the `profile` command in the *VOS Commands Reference Manual (R098)* for information about how this information is logged.

This argument is useful only for programs compiled for XA/R-series modules and Continuum-series modules. The `profile` command **always** reports page-fault data for programs compiled for XA2000-series modules; thus, specifying this argument would have no effect.

By default, the binder does not instruct the operating system to count alignment faults.

For more information about alignment faults, see the manual *Migrating VOS Applications to Continuum Systems (R407)*.

Specifying the Size of a Stack

The `-stack_size` argument specifies the number of bytes of storage to reserve for the stack. The binder uses the value of `-stack_size` to determine whether a program will fit in the defined address space.

This argument accepts any numeric value, but the specified size must be evenly divisible by 4 for an XA2000-series module, evenly divisible by 16 for an XA/R-series module, or evenly divisible by 64 for a Continuum-series module.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-stack_size`.

This argument interacts with the `-number_of_tasks` argument as follows:

- If *`number_of_tasks`* is 1, the stack size is the minimum stack size. The binder guarantees that the size of the program, plus the size of the initial heap (by default, 32 kilobytes), plus the size of the last static task’s stack fence (32 kilobytes), plus the stack size, fits in the specified program size.
- If *`number_of_tasks`* is greater than 1, the stack size is the maximum size of the stack for each static task.
  - If the program is a user program, the binder guarantees that the sum of the sizes of all regions, plus the maximum heap size, plus the maximum stack size, fits in the specified program size. See ‘‘Specifying the Maximum Size of a Heap’’ and ‘‘Allocating Memory for Static Tasks’’ later in this chapter for information about determining the maximum heap size and the maximum stack size, respectively.
  - If the program is a kernel program, the binder guarantees that the sum of the sizes of all regions except `symtab` and `maps` fits in the specified program size.

By default, the binder allocates 32,768 bytes for each static task.

Specifying the Size of a Stack Fence

The `-stack_fence_size` argument specifies the size, in bytes, of the fence to be placed after each static task’s stack.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-stack_fence_size`.

The binder uses the following rules to set the stack fence size.

- If you specify the `stack_fence_size` binder control-file directive, the binder uses the specified value.
- If you do not specify the `stack_fence_size` binder control-file directive, the binder uses the value specified in the `-stack_fence_size` argument.
- If you do not specify any value, the binder sets the stack fence size to 4096 bytes.

The `stack_fence_size` binder control-file directive is described in ‘‘Specifying Directives’’ later in this chapter.

If you do not want a stack fence, specify a value of 0. Note, however, that even if you specify a value of 0, the system still allocates a stack fence of 32,768 bytes for the last static task.

Specifying the Maximum Size of a Heap

The `-max_heap_size` argument specifies the maximum byte size to which the heap can grow.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-max_heap_size`.

The binder uses the following rules to set the maximum heap size for the program.

- If you specify the `max_heap_size` binder control-file directive, the binder uses the specified value.
- If you do not specify the `max_heap_size` binder control-file directive, the binder uses the value specified in the `-max_heap_size` argument.
- If you do not specify any value, the value of `max_heap_size` is 0. Note, however, that a zero value does **not** imply that the heap’s size is 0; instead, the maximum heap size is assumed to be equal to 32,768 bytes for the purpose of checking the size of the address space.

The `max_heap_size` binder control-file directive is described in ‘‘Specifying Directives’’ later in this chapter.

Specifying the Maximum Size of a Program

The `-max_program_size` argument specifies, in bytes, the maximum amount of code and data the program can contain, excluding its symbol tables. The value of `-max_program_size` can be any unsigned 32-bit value.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-max_program_size`.

The binder uses the following rules to set the maximum program size.

- If you specify the `max_program_size` binder control-file directive, the binder uses the specified value.
- If you do not specify the `max_program_size` binder control-file directive, the binder uses the value specified in the `-max_program_size` argument.
- If you do not specify any value, the binder checks the amount of code and data against the address space size. See ‘‘Specifying the Size of a Program’s Address Space” earlier in this chapter for more information about address space size.

The `max_program_size` binder control-file directive is described in ‘‘Specifying Directives” later in this chapter.

Allocating Memory for Static Tasks

The `-max_stack_size` argument specifies the total amount of memory, in bytes, that all static tasks’ stacks and fences can occupy.

See ‘‘Syntax of Numerical Binder Values” earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-max_stack_size`.

The binder uses the following rules to determine how much memory to allocate for the program.

- If you specify the `max_stack_size` binder control-file directive, the binder uses the specified value.
- If you do not specify the `max_stack_size` binder control-file directive, the binder uses the value specified in the `-max_stack_size` argument.
- If you do not specify any value, the binder assumes that the `default` value is equal to the result of the following formula.

  (*`stack_size`* * *`n_tasks`*) +
  (*`stack_fence_size`* * (*`n_tasks`* - 1)) +
  `stack_fence_size_of_last_task`

  In the preceding formula, *`n_tasks`* is the value specified in the `-number_of_tasks` argument. Also, note that the system assigns a value to *`stack_fence_size_of_last_task`*; see ‘‘Specifying the Size of a Stack Fence” earlier in this chapter for more information about the size of the last task’s stack fence.

  Note that the value specified for the `-max_stack_size` argument or the `max_stack_size` binder control-file directive must be greater than 32,767.

  The `max_stack_size` binder control-file directive is described in ‘‘Specifying Directives” later in this chapter.

Suppressing Certain VOS Standard C Messages

The `-subroutines_are_functions` argument suppresses the message that can occur in VOS Standard C programs when a function is being called as a subroutine, or vice versa. This argument is generally not used to bind PL/I programs.

Specifying a Version Number

The `-version` argument specifies the release string that will appear in the program module header. By default, the binder always initializes *`version`* to `Pre-release`.

Specifying the Load Point for an Object Module

The `-load_point` argument assigns a lowest address for the object module. The value of `-load_point` can be any unsigned 32-bit value. For example, if the default load point for the kernel is `80000000x`, specify the following command to change the load point.

```
bind prog1 -load_point 80026000x
```

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of `-load_point`.

The binder uses the following rules to set the load point.

- If you specify the `load_point` binder control-file directive, the binder uses the specified value.
- If you do not specify the `load_point` binder control-file directive, the binder uses the value specified in the `-load_point` argument.
- If you do not specify any value, the binder determines the default value based on the information in the following table.

Processor Type of Target Module	Load Point Set for a User Program	Load Point Set for a Kernel Program
XA2000-series	`00E00000x`, if the address space size is less than or equal to 2 megabytes; otherwise, `00800000x`. See ‘‘Specifying the Size of a Program’s Address Space’’ earlier in this chapter for information about determining address space size.	`00000000x`
XA/R-series	`00008000x`	`80000000x`
Continuum-series	`00002000x`	`00002000x`

Note: Do not attempt to change the load point for programs executing on XA2000-series modules or XA/R-series modules. Such an attempt will result in an unloadable program.

The `load_point` binder control-file directive is described in ‘‘Specifying Directives’’ later in this chapter.

Initializing External Variables That Have Message Names

If your program declares as an external variable a name that is identical to a message name in the current message file, and if the program does not assign an initial value to the variable, the binder initializes the variable to the message code corresponding to the message name. For example, if you declare `e$end_of_file` as a `fixed bin(15)` variable having external scope, and if you are using the standard message file, the binder initializes the variable to the value 1025. In the same way, if you set a nonstandard message file with the `use_message_file` command, the binder can assign the status code number of a message in that message file to an external variable whose name is the same as the status code name of a message in that file.

Tasking programs share external variables whose names begin with `e$`, `m$`, `q$`, or `r$`.

Using a Binder Control File

This section explains how you create a binder control file. Specifically, this section discusses the following topics.

- ‘‘Writing a Binder Control File’’
- ‘‘Specifying Directives’’
- ‘‘Binder Control File Example’’

A sample binder control file appears later in this chapter.

A *binder control file* is a text file that directs the binding of a set of object modules. Using a binder control file, as opposed to entering command-line arguments, facilitates binding when you need to communicate a complex set of specifications to the binder. Also, a binder control file is useful if you intend to bind the same object module or set of object modules more than once and do not want to type the binding specifications each time.

You specify the binder control file with the `-control` argument. The control file must have the suffix `.bind`, although you can omit the suffix when you specify the file’s path name in the `-control` argument. If you use the `-control` argument, the binder control-file directives take the place of most of the arguments you would specify with the `bind` command. You might, however, want to include the `-search` argument (or the `search` binder control-file directive) so that the binder searches your current directory before searching any other directory.

> **Note:** Values specified in a binder control file override values specified on the command line. The one exception to this rule is the `search` binder control-file directive, which adds to (but does not override) the list of directories searched by a `-search` argument, if one is specified.

Every directive, except `end`, has a default value or action, which is provided in the description of the directive. The binder uses this default value when you omit a directive and do not supply different information in the corresponding command-line argument, if any.

See the *VOS Commands Reference Manual (R098)* for more information about binder control files.

Writing a Binder Control File

This section explains some of the syntactic conventions related to writing a binder control file. In a binder control file, the rules for forming an identifier or name are similar to the rules for forming a VOS path name. An identifier or name can contain any printable ASCII character, which includes the 52 uppercase and lowercase alphabetic characters, the 10 decimal digits, and the following 24 graphic characters.

```
" # % * $ + - . / < > @ [ \ ] ^ _ ` { | } ~ , :
```

A number must begin with 0 through 9, +, or -. Subsequent characters in a number can be 0 through 9 and certain letters. For example, identifiers named `2mb` and `0AFFx` are processed as numbers.

See ‘‘Syntax of Numerical Binder Values” earlier in this chapter for more information about specifying a multiplier suffix or a base suffix in certain binder control-file directives.

Adjacent words can be separated by space characters, a colon (`:`), a semicolon (`;`), a comma (`,`), or parentheses (`(` or `)`). You need not enclose a path name or other word in apostrophes **unless** the path name or word contains one of these characters. (Note that, of these special characters, only a colon or comma can be used to form a path name.)

In a binder control file, an empty line is not significant. You can use empty lines to improve the readability of the control file. A comment begins with the characters `/*` and ends with the characters `*/`. The binder disregards comments.

Specifying Directives

This section describes the binder control-file directives that are most useful to VOS PL/I programmers. Other directives are described in the *VOS Commands Reference Manual (R098)*.

- `define:` *`definition_specifier`*`...;`

  Attaches symbolic names to various constant locations in memory. User programs can then use these names as external variables and resolve any references to them to the proper region of memory. You must separate each specifier with a comma. The *`definition_specifier`* argument has the following form.

  `symbol_name` `address (`*`number`*`)`

- `end;`

  Indicates the end of the binder control file. This directive must be the last one in the file. If you specify more than one `end` directive, the first one in the file effectively ends the binder control file.

The following example shows a binder control file that binds four object modules together and is terminated by the required `end` directive.

```
modules:  %s1#d01>Sales>Jones>addition page_aligned,
          %s1#d01>Sales>Jones>subtraction,
          division,
          multiplication no_table;
end;
```

- `entry:` *`identifier`*`;`

  Defines the name of the main entry point of the program. The name *`identifier`* must be the name of an entry point in one of the object modules being bound. If you do not use the `entry` directive, the first entry point the binder finds is used as the main entry point for the program module. If you specify more than one `entry` directive, the binder disregards all but the last one. If the program's main entry point is specified with parameters, the binder issues a warning.

- `high_water_mark:` *`address`*`;`

  Specifies the address of the beginning of heap space for a process on an XA2000-series or XA/R-series module, or the address of the beginning of stack space on a Continuum-series module. Use this directive in place of an object module specified with the `create_data_object` command and bound with the `modules` directive. The `high_water_mark` directive moves the beginning of the heap or stack to a known location in order to create a shared virtual memory space that is not contained in program modules and does not use disk space. For more information about calculating the value for the `high_water_mark` directive and for more information about the `create_data_object` command, see the *VOS Commands Reference Manual (R098)*.

- `load_point:` *`number`*`;`

  Assigns a lowest address for the object module. You can specify any unsigned 32-bit value for *`number`*. For example, if the default load point for the kernel is `80000000x`, specifying the following directive changes the load point.

  ```
  load_point:  80026000x;
  ```

  Note: Do not attempt to change the load point for programs executing on XA2000-series modules or XA/R-series modules. Such an attempt will result in an unloadable program.

  See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`number`*.

  The `load_point` directive overrides the `-load_point` command-line argument. For more information about setting a load point, see ‘‘Specifying the Load Point for an Object Module’’ earlier in this chapter.

- `max_heap_size:` *`size`*`;`

  Specifies the maximum byte size to which the heap can grow.

  See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`size`*.

The `max_heap_size` directive overrides the `-max_heap_size` argument. For more information about specifying the maximum heap size, see ‘‘Specifying the Maximum Size of a Heap’’ earlier in this chapter.

- ▶ `max_program_size:` *`number`*`;`

  Specifies, in bytes, the maximum amount of code and data that the program can contain, excluding its symbol tables. You can specify any unsigned 32-bit value for *`number`*.

  See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`number`*.

  The `max_program_size` directive overrides the `-max_program_size` argument. For more information about specifying the maximum size of a program, see ‘‘Specifying the Maximum Size of a Program’’ earlier in this chapter.

- ▶ `max_stack_size:` *`size`*`;`

  Specifies the total amount of memory, in bytes, that all static tasks’ stacks and fences can occupy.

  See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`size`*.

  The `max_stack_size` directive overrides the `-max_stack_size` argument. For more information about allocating memory for static tasks, see ‘‘Allocating Memory for Static Tasks’’ earlier in this chapter.

- ▶ `modules:` *`module_specifier...`*`;`

  Declares the object modules to be bound. The *`module_specifier`* argument identifies the names of the object modules and how the object modules are to be bound. Each *`module_specifier`* argument has the following syntax.

  ```
  (module_term...)[module_attribute]...
  ```

  The *`module_term`* value is the file name or path name of an object module followed by zero or more module attributes, separated by spaces. The suffix `.obj` is optional for file names supplied with *`module_term`*. Multiple *`module_specifier`* arguments are separated by commas.

  - If *`module_term`* is a file name, the binder looks for the object module first in the directories specified in the `-search` argument, next in the directories specified by the `search` binder control-file directive, and finally in the directories specified in the object library paths list for that process.

  - If *`module_term`* is a full or relative path name, the path name must locate the object module. The binder does not search in any other location for the object module.

You can specify the following values for *`module_attribute`*.

```
compact
no_compact
table
no_table
page_aligned
```

The preceding attributes affect only those object modules with which they are associated. The `compact` and `no_compact` attributes have the same effect as the corresponding `-compact` and `-no_compact` command-line arguments. Likewise, the `table` and `no_table` attributes have the same effect as the `-table` and `-no_table` command-line arguments. For the specified object module or modules, these attributes override any conflicting values specified in command-line arguments.

The `page_aligned` attribute tells the binder to put the first word in the code region of the specified object module on a page boundary. This attribute is useful in connection with shared virtual memory. For information about page alignment and shared virtual memory, see the description of the `s$connect_vm_region` subroutine in the *VOS PL/I Subroutines Manual (R005).*

In the following example, the `modules` directive indicates that the object modules identified by the relative path names `>Sales>Jones>outstanding.obj` and `>Sales>Jones>paid.obj` are to be bound together. This `modules` directive also specifies that `>Sales>Jones>outstanding.obj` will have the `page_aligned` attribute associated with it.

```
modules:  >Sales>Jones>outstanding.obj  page_aligned,
          >Sales>Jones>paid.obj;
```

To indicate that multiple object modules have one or more attributes associated with them, you enclose, within parentheses, the file names or path names of the object modules. You can specify common attributes for multiple object modules by putting the attributes outside the parentheses. In the next example, the `modules` directive indicates that the object modules identified by the file names `outstanding.obj` and `paid.obj` are to be bound together. This `modules` directive also specifies that both object modules will have the `page_aligned` attribute associated with them.

```
modules:  (outstanding.obj, paid.obj)  page_aligned;
```

▶ `name:` *`program_name;`*

Specifies a name for the bound program module. The value of *`program_name`* can be a path name. When you include more than one `name` directive, the binder disregards all but the last one. See ‘‘Naming a Program Module” earlier in this chapter for more information about how the binder names a program module.

In the following example, the `name` directive gives the name `calculator.pm` to the resulting program module.

```
name: calculator;

modules: reading,
         calculating,
         writing;
end;
```

▶ `number_of_tasks:` *`number_of_tasks`*`;`

Specifies the number of static tasks the binder creates in the program module. The number of static tasks is limited only by the program module's total size. Each static task has its own stack and its own copy of static storage. If you do not use the `number_of_tasks` directive, the binder creates one static task.

The `number_of_tasks` directive is used **only** in tasking applications. For more information about the `number_of_tasks` directive, see the *VOS Transaction Processing Facility Guide (R215).*

The `number_of_tasks` directive overrides the `-number_of_tasks` argument. For more information about specifying the number of static tasks created by the binder in the program module, see ''Specifying the Number of Static Tasks to Create in the Program Module'' earlier in this chapter.

▶ `options:` *`option`*`...;`

Specifies one or more binder options. Many of the options have the same effects as the corresponding command-line arguments, which were described earlier in this chapter. Table 5-5 shows the allowed values for *`option`*.

Table 5-5. Values for the `options` Directive *(Page 1 of 3)*

Value	**Description**
`compact`	Same as the `-compact` command-line argument. The default value is `compact`.
`no_compact`	Same as the `-no_compact` command-line argument.
`dynamic_tasking`	Same as the `-dynamic_tasking` command-line argument. The default value is `dynamic_tasking`.
`no_dynamic_tasking`	Same as the `-no_dynamic_tasking` command-line argument.
`kernel`	Tells the binder to create a kernel or, when specified with the `load_in_kernel`, `relocatable`, and `no_library` options, a kernel-loadable program in the same manner as the `bind_kernel` command. The default value is `no_kernel`. For more information, see the description of the `load_kernel_program` command in the manual *VOS System Administration: Administering and Customizing a System (R281)*.
`no_kernel`	Tells the binder not to create a kernel or a kernel-loadable program in the kernel.
`library`	Tells the binder to try to resolve any unresolved references found while processing the `modules` directive. To do this, the binder searches for a module with the same name as the unresolved reference. The default value is `library`.
`no_library`	Tells the binder not to resolve any unresolved references.
`load_in_kernel`	Same as the `-load_in_kernel` command-line argument. The default value is `no_load_in_kernel`.
`no_load_in_kernel`	Same as the `-no_load_in_kernel` command-line argument.

Table 5-5. Values for the `options` Directive *(Page 2 of 3)*

Value	Description
`long_branches`	Tells the binder to generate code that is slightly faster but that can be executed only on the MC68020 processor or later processors in the MC68000 processor family. This option is useful only if branching optimizations have not already been performed on the code. The default value is `no_long_branches`.
`no_long_branches`	Tells the binder not to generate code that can be executed on any processor in the MC68000 processor family.
`mod16`	Same as the `-align_mod16` command-line argument. The default value is `no_mod16`.
`no_mod16`	Same as the `-no_align_mod16` command-line argument.
`private_heap`	Same as the `-private_heap` command-line argument. The default value is `no_private_heap`.
`no_private_heap`	Same as the `-no_private_heap` command-line argument.
`private_stack`	Same as the `-private_stack` command-line argument. The default value is `no_private_stack`.
`no_private_stack`	Same as the `-no_private_stack` command-line argument.
`references_kernel`	Same as the `-references_kernel` command-line argument. The default value is `no_references_kernel`.
`no_references_kernel`	Same as the `-no_references_kernel` command-line argument.
`relocatable`	Same as the `-relocatable` command-line argument. The default value is `no_relocatable`.
`no_relocatable`	Same as the `-no_relocatable` command-line argument.
`require_external_static_def`	Tells the binder to require that external static variables be initialized. The default value is `require_external_static_def`.
`no_require_external_static_def`	Tells the binder not to require that external static variables be initialized.

Table 5-5. Values for the `options` Directive *(Page 3 of 3)*

Value	**Description**
`subroutines_are_functions`	Same as the `-subroutines_are_functions` command-line argument. The default value is `no_subroutines_are_functions`.
`no_subroutines_are_functions`	Same as the `-no_subroutines_are_functions` command-line argument.
`table`	Same as the `-table` command-line argument. The default value is `table`.
`no_table`	Same as the `-no_table` command-line argument.

The values specified in an `options` directive in a binder control file override any conflicting command-line arguments. However, specifying `compact` or `no_compact` for an object module in the `modules` directive overrides a conflicting `options` directive value. The binder reads the entire binder control file before acting on any part. If an option is specified more than once, the last value in the binder control file determines what option will be in effect for the binding operation.

▶ `processor:` *`processor_string`*`;`

Determines the preprocessor variable(s) that the binder defines, and allows the binder to check that the object modules are compatible with the specified processor type. You can specify the following values for *`processor_string`*.

- `default`
- `mc68000`
- `mc68020`
- `mc68020/mc68881`
- `mc68030`
- `mc68030/mc68882`
- `i80860`
- `i80860xp`
- `pa7100`
- `pa8000`

The `default` value specifies the current module's default system processor. If you do not use the `processor` directive or the `-processor` command-line argument, the binder expects object modules that were compiled for the default system processor.

You cannot specify more than one `processor` directive in a binder control file. The `processor` directive overrides the `-processor` command-line argument. For more information about specifying a processor when binding, see ‘‘Specifying a Processor” earlier in this chapter. See also Chapter 4 for information about preprocessing.

▶ `region_load_point:` *`section_name region_name`* `:` *`location`*
`[,` *`section_name region_name`* `:` *`location`*`]...;`

Allows you to place sections and regions at specific addresses. By default, sections are ordered by their first appearance in the binder control file, and regions appear in the following order: code, unshared data, and shared data within a section.

The *`location`* argument can be an absolute address (such as `80000000x`), or it can have the following syntax, which assigns the section or region to a location following another section or region.

```
after section_name region_name
```

See ‘‘Syntax of Numerical Binder Values” earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`location`*.

The binder places any regions without explicit load points **after** all regions with explicit load points. The binder then orders the regions without explicit load points according to the default ordering.

Sections and regions with explicit load points should **not** overlap. Such an overlap causes the binding to fail.

The following example demonstrates one use of `region_load_point`.

```
region_load_point:
     mysect code          : 00000000x,
     mysect static        : 40000000x,
     mysect ext_static    : after mysect static;
```

The preceding example assigns three regions in the user-named section `mysect`. The `code` and `static` regions are assigned absolute addresses, while the `ext_static` region is directed to follow the `static` region. Since no load point was specified for the `symtab` region, the binder will place it after the other three regions. The binder will place the `maps` section after the `symtab` region, by default.

The following example assigns the `wired code` region to an absolute address of `80000000x`, and directs the other regions to follow `wired code`. In addition, the example shows the default ordering of regions for VOS kernels on XA/R-series modules.

```
region_load_point:
     wired code          : 80000000x,
     wired static        : after wired code,
     wired ext_static    : after wired static,
     init code           : after wired ext_static,
     init static         : after init code,
     init ext_static     : after init static,
     paged code          : after init ext_static,
     paged static        : after paged code,
     paged ext_static    : after paged static,
     wired symtab        : after paged ext_static,
     init symtab         : after wired symtab,
     paged symtab        : after init symtab,
     maps                : after paged symtab;
```

Note: The `region_load_point` directive is used primarily for Stratus internal development. Most users should **not** use this directive.

▶ `retain` [`:` *`entry_name`* [`as` *`new_name`*] [`,` *`entry_name`* [`as` *`new_name`*]]] `...;`

Specifies the external entry names for the binder to place in the program module's entry map. This map contains the entry value for each name. You can specify the names of more than one entry point in a `retain` directive if you separate the names with commas. If you do not specify any names, the binder places all entry-point names in the map. If you do not specify any names, omit the colon (`:`) after `retain`.

The `as` *`new_name`* option allows you to specify an alternate name under which the entry point is retained.

See ''Specifying Retained Entry Points'' earlier in this chapter for more information about retaining external entry points.

▶ `search:` *`directory_name`*`...;`

Specifies the names of directories that the binder is to search when it looks for object modules. The specified directories are added to the list of directories specified in the `-search` argument of the `bind` command. Separate multiple *`directory_name`* values with commas.

You can include more than one `search` directive in a binder control file. Each directive adds more directories to the search list. The binder allows as many command-line and binder control-file search directories as memory permits.

In the following example, the `search` directive adds two directories, `>sales>tools>commissions` and `>sales>tools>expenses`, to the list of search directories.

```
search: >sales>tools>commissions, >sales>tools>expenses;
```

For information about the order that the binder uses to search for object modules, see ‘‘Specifying the Directories to Search for Object Modules’’ earlier in this chapter.

- `section:` *`section_name`*`;`
  Specifies the section of the address space in which the binder is to locate the object modules. The values you can specify for *`section_name`* vary by module type.

  On an XA2000-series module or an XA/R-series module, possible values for *`section_name`* are `wired`, `initialization`, and `paged`.

  For kernel programs running on a Continuum-series module, you can specify any section name in *`section_name`* except `maps`, as long as the name follows the naming conventions described in ‘‘Writing a Binder Control File’’ earlier in this chapter. The first four sections of the address space are always the `wired`, `initialization`, `paged`, and `maps` sections, in that order. Any additional section names that you specify will follow these sections in the section map.

  On a Continuum-series module, in addition to the `wired`, `initialization`, `paged`, and `maps` sections, you can specify up to 28 more sections in a binder control file.

  A `wired` or `initialization` module is not subject to page faults. The `load_kernel_program` command deletes the `initialization` section after calling each retained entry point in the `initialization` section. (See the manual *VOS System Administration: Administering and Customizing a System (R281)* for more information about `load_kernel_program`.) A `paged` module will take page faults. A `modules` directive is subject to the most recent `section` directive. By default, all modules for a user program will be `paged`; all modules for a kernel program will be `wired`.

- `size:` *`size`*`;`
  Specifies the size of the address space for which the binder is to bind the object modules. You can specify any numeric value for the `size` directive, as well as the values `small` (to specify a 2-megabyte address space) and `large` (to specify an 8-megabyte address space). This directive has the same effect as the `bind` command’s `-size` argument.

  See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`size`*.

  If you include more than one `size` directive, the binder disregards all but the last one. A size specified in a binder control file overrides a size specified as a command-line argument.

In the following example, the `size` directive specifies a `large` address space of eight megabytes.

```
size: large;
```

See ‘‘Specifying the Size of a Program’s Address Space’’ earlier in this chapter for more information about program address space.

▶ `stack_fence_size:` *`stack_fence_size`*`;`

Specifies the size, in bytes, of the fence to be placed after each static task’s stack.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for the value of *`stack_fence_size`*.

See ‘‘Specifying the Size of a Stack Fence’’ earlier in this chapter for more information about stack fence size.

▶ `stack_size:` *`stack_size`*`;`

Specifies the number of bytes of storage to reserve for the stack. The binder uses *`stack_size`* to determine whether a program will fit in the defined address space.

On XA2000-series modules, the value of *`stack_size`* must be divisible by 4. On XA/R-series modules, the value of *`stack_size`* must be divisible by 16. On Continuum-series modules, the value of *`stack_size`* must be divisible by 64.

This directive interacts with the `number_of_tasks` directive as follows:

- If *`number_of_tasks`* is 1, the stack size is the minimum stack size. The binder guarantees that the size of the program, plus the size of the initial heap (32 kilobytes), plus the size of the fence (32 kilobytes), plus the stack size, fit in the specified program size.

- If *`number_of_tasks`* is greater than 1, the stack size is the maximum size of the stack for each static task. The binder guarantees that the size of the program, plus the size of the initial heap (32 kilobytes), plus the size of the fence (32 kilobytes), plus the total size of all stacks, fit in the specified program size.

The total size of all stacks equals the number of static tasks multiplied by the stack size. By default, the binder allocates 32,768 bytes for each static task.

See ‘‘Syntax of Numerical Binder Values’’ earlier in this chapter for information about specifying a multiplier suffix or a base suffix for *`stack_size`*.

▶ `synonym:` *`synonym_specifier`*`...;`

Specifies an entry name to which one or more names are resolved. Separate multiple *`synonym_specifier`* arguments with commas. Each *`synonym_specifier`* argument has the following syntax.

```
old_name[*] for entry_name
```

All external references matching *`old_name`* are resolved to *`entry_name`*. The *`old_name`* value can have an asterisk as its last character, representing any sequence of zero or more valid identifier characters. Generally, *`old_name`* appears in a `variable_arg_count` directive, where it defines a set of declarations with different numbers or types of arguments.

▶ `variable_arg_count:` *`identifier...;`*

Indicates that the entry name named *`identifier`* can be called with an indefinite number of arguments. Designating the program *`identifier`* value suppresses the warning message the binder normally generates when you call a program with the wrong number of arguments.

You can include more than one `variable_arg_count` directive in a control file. Each directive adds more program names to the list of entry points that accept an indefinite number of arguments.

This directive is generally used to bind C programs, not PL/I programs.

▶ `variables:` *`variable_specifier...;`*

Modifies the attributes of an external variable, and, in some cases, can also be used to define an external variable. An *external variable* has external scope and static storage duration.

The *`variable_specifier`* argument tells the binder the name of an external variable. It can also specify the number of bytes to allocate for the variable and an initial value to give to the variable. In addition, one or more of the following attributes can be associated with the variable: `shared` or `unshared`, `page_aligned`, and `flexible_length`.

The *`variable_specifier`* argument has the following syntax.

```
                                                ⎡  shared  ⎤
name [(size)] [initial (initial_value)]         ⎣ unshared ⎦
[page_aligned] [flexible_length ]
```

The *`name`* value must be the name of an external variable in at least one of the object modules being bound. If it is not, the binder issues a warning. The *`size`* value must be an unsigned integer indicating the number of bytes allocated for the variable. If a size is indicated for a variable in a `variables` directive, that size overrides any length specified in the source module declarations. For a variable to be allocated space, its size must be known.

The *`initial_value`* value specifies an initial value for the variable. The initial value can be either a character-string literal or an integer constant. The binder initializes a `char(`*`n`*`)` variable as a sequence of characters having the length *`size`*.

If the initial value is a character-string literal, it must be enclosed within apostrophes.

The binder initializes an integer variable as a 2-byte or 4-byte signed integer. The *`size`* value for an integer variable must be either 2 or 4. Since a variable declared as `fixed bin(15)` takes 2 bytes of memory and a variable declared as `fixed bin(31)` takes 4 bytes, a `fixed bin(15)` variable's size is 2, and a `fixed bin(31)` variable's size is 4. An initial value that you assign to a variable in a binder control file overrides an initial value specified in a program.

You can abbreviate the word `initial` to `init`. The following example shows a `variables` directive that specifies initial values for two external variables.

```
variables:  num_records (4) init (100000),
           string (10) init ('The Title');
```

The preceding `variables` directive instructs the binder to allocate 4 bytes for an integer variable `num_records`, which is initialized to the value 100,000. It also instructs the binder to allocate 10 bytes for a `char` variable named `string`, which is initialized with the characters `The Title`. Any unused bytes in the string are filled with space characters.

In a tasking program, one or more external variables that are defined as `shared` can be shared among static tasks.

- The `shared` attribute defines a shared variable. The binder allocates storage for each `shared` variable only once, instead of allocating separate storage for each static task.
- The `unshared` attribute defines an unshared variable. For each static task, the binder allocates separate storage for each `unshared` variable. The default attribute is `unshared`.

In a tasking or nontasking program, the binder allocates `shared` external variables in the shared static region. In a nontasking program, the binder allocates `unshared` external variables in the unshared static region. In a tasking program, the binder allocates `unshared` external variables on a per-task basis in that task's unshared static region.

In the following example, the `variables` directive defines two variables as `shared`.

```
variables: global_operand1 shared,
           global_operand2 shared;
```

As shown in the preceding example, when you define more than one *`name`* value to be shared, you specify `shared` after each shared *`name`* value and separate the names with commas.

As an alternative to defining a variable with the `shared` attribute in a binder control file, you can use the `external shared` attribute in the variable's declaration within the source module. When the `shared` or `unshared` attribute is specified for a variable in a binder control file, that attribute overrides any conflicting attribute in the source module declaration.

For more information about how variables can be shared by multiple tasks in a tasking program, see the *VOS Transaction Processing Facility Guide (R215).*

The `page_aligned` attribute tells the binder to allocate storage for a variable on a page boundary. Page-aligned, external variables are sometimes specified for a program that uses the `s$connect_vm_region` subroutine for shared virtual memory. See the *VOS PL/I Subroutines Manual (R005)* for information about the `s$connect_vm_region` subroutine and shared virtual memory.

The `flexible_length` attribute tells the binder to suppress warnings when two object modules declare an external variable with different lengths. In the following example, assume that the `get_record` and `write_record` object modules have declared `variable_1` with different lengths.

```
modules:   get_record, write_record;

variables: variable_1 flexible_length;
```

In the preceding example, the `flexible_length` attribute allows the binder to bind the object modules without issuing a warning that `variable_1` has conflicting lengths. The binder uses the maximum length of `variable_1`, as specified in the source module declarations, to determine the amount of storage to allocate for the variable. However, if a `variables` directive specifies the size of `variable_1`, that size overrides any length specified in the source module declarations.

Binder Control File Example

The sample binder control file in Figure 5-5 produces a program module named `auction_tasks`. The program is a tasking program.

```
name:               auction_tasks;

number_of_tasks:    26;

variables:          xyz    shared;
modules:            auction_tasks,       /* Separately compiled objects       */
                    display_bidder_info,

                    user_form,           /* FMS screen objects, optional      */

                    display_task_info,   /* Monitor requests                  */
                    control_task,
                    create_task,
                    initialize_task,
                    start_task,
                    stop_task;

retain:             user_tasks_entry,    /*  Task entry points                */
                    monitor_task_entry,

                    task_epilogue,       /*  Epilogue handlers                */
                    program_epilogue,

                    display_bidder_info, /*  Programmed monitor request       */

                    display_task_info,   /*  Monitor requests                 */
                    control_task,
                    create_task,
                    initialize_task,
                    start_task,
                    stop_task;

stack_size:         20480;               /*  Maximum stack size for each task */

end;
```

Figure 5-5. Sample Binder Control File

The binder control file in Figure 5-5 contains the following directives.

- The `name` directive specifies that the binder will name the file containing the program module `auction_tasks.pm`.

- The `number_of_tasks` directive specifies that the binder will create 26 static tasks in the program module.

- The `variables` directive specifies that the binder will allocate the external variable `xyz` in the shared static region.

- The `modules` directive specifies the object modules that the binder will bind into the program module.

- The `retain` directive specifies the external entry names that the binder will place in the program module's entry map.

- The `stack_size` directive specifies that the binder will allocate up to 20,480 bytes of the stack for each static task. Note that specifying the value `20kb` would produce the same result.

- The `end` directive terminates the binder control file.

Preprocessing a Binder Control File

This section explains how you preprocess a binder control file.

When you issue the `bind` command and specify a binder control file with the `-control` argument, the binder calls the preprocessor to process binder-preprocessor statements in the binder control file. After preprocessing, the binder uses the directives in the resulting file to direct the binding process.

The binder-preprocessor statements allow you to use conditional inclusion in a binder control file. *Conditional inclusion* occurs when, based on the value of a preprocessor variable, the preprocessor includes or does not include a block of text into a binder control file. For example, using conditional inclusion within a binder control file allows you to direct the binder to search different directories for object modules based on the processor family on which the program module will run.

You use the `-define` command-line argument to define preprocessor variables in the binder.

The discussion of the VOS preprocessor statements in Chapter 4 also applies to this discussion of the binder-preprocessor statements. The binder-preprocessor statements are identical to the VOS preprocessor statements except for the differences summarized in Table 5-6.

Table 5-6. Differences between VOS- and Binder-Preprocessor Statements

VOS Preprocessor Statements	Binder-Preprocessor Statements
These statements are used to perform conditional compilation in a source module.	These statements are used to perform conditional inclusion in a binder control file.
If a listing file is created, the compiler inserts three plus signs (+++) in front of each line of the compilation listing that, after preprocessing, will not be compiled.	If a map file is created, the binder inserts an asterisk (`*`) in front of each line of the bind map that, after preprocessing, will not be read by the binder.
The compiler's preprocessor automatically predefines preprocessor variables for the processor family (either `__MC68K__`, `__I860__`, or `__HPPA__`) **and** for the specific processor type (such as `__MC68030__`). Also, when the compiler's preprocessor automatically defines `__HPPA__`, it also defines both `__HPPA11__` **and** `_PA_RISC1_1`, or both `__HPPA20__` **and** `_PA_RISC2_0`, depending on the specific processor family (the PA-7100 family or the PA-8000 family).	The binder's preprocessor automatically predefines preprocessor variables for the processor family (either `__MC68K__`, `__I860__`, or `__HPPA__`) but **not** for the specific processor type (such as `__MC68030__`). Also, when the binder's preprocessor automatically defines `__HPPA__`, it also defines `__HPPA11__` or `__HPPA20__`, depending on the specific processor family (the PA-7100 family or the PA-8000 family).

See Chapter 4 for more information about the VOS preprocessor statements.

Conditional Inclusion Example

The example in Figure 5-6 illustrates how to use conditional inclusion in a binder control file. In the figure, the binder control file `transact.bind` contains `$if`, `$elseif`, and `$endif` binder-preprocessor statements.

```
name:  transact;

modules:  transact,
          get_record,
          write_record;

search:

$if defined (__MC68K__) & ^defined (SPECIAL)

     >Programming>object_modules>mc68k_objects;

$elseif defined (__MC68K__) & defined (SPECIAL)

     >Programming>object_modules>special_objects;

$elseif defined (__I860__)

     >Programming>object_modules>i860_objects;

$endif

end;
```

Figure 5-6. Binder Control File with Binder-Preprocessor Statements

For the example in Figure 5-6, assume that the `bind` command is invoked with the following command-line arguments.

```
bind -control transact -define SPECIAL -map -processor mc68020
```

The binder-preprocessor's output, contained in the bind map for `transact.pm`, is shown in Figure 5-7. Note that only the relevant portion of the bind map is shown.

```
Control file: %s1#d01>Programming>object_modules>transact.bind

      1   name:  transact;
      2
      3   modules:  transact,
      4             get_record,
      5             write_record;
      6
      7   search:
      8
      9 * $if defined (__MC68K__) & ^defined (SPECIAL)
     10 *
     11 *      >Programming>object_modules>mc68k_objects;
     12 *
     13 * $elseif defined (__MC68K__) & defined (SPECIAL)
     14
     15        >Programming>object_modules>special_objects;
     16
     17 * $elseif defined (__I860__)
     18 *
     19 *      >Programming>object_modules>i860_objects;
     20 *
     21 * $endif
     22
     23   end;
```

Figure 5-7. Bind Map Containing Preprocessor Output

As shown in the bind map fragment (Figure 5-7), the binder inserts an asterisk (`*`) in front of each line of the bind map that, after preprocessing, the binder does **not** read when scanning for binder control file text.

In Figure 5-7, when the `bind` command is invoked, the binder automatically defines two preprocessor variables.

- The binder defines `__MC68K__` because the `-processor` argument specified a processor from the MC68000 family of processors.
- The binder defines `SPECIAL` because the `SPECIAL` preprocessor variable was specified in the `-define` argument.

With these predefinitions, when the `$if` and `$elseif` statements are processed, only the following controlling expression (from line 13) evaluates to true.

```
defined (__MC68K__) & defined (SPECIAL)
```

Thus, after preprocessing, only the path name for the `special_objects` directory is specified as a path name in the `search` directive. The path names for the `mc68k_objects` directory and the `i860_objects` directory are excluded from the resulting binder control file. That is, the binder does not interpret these path names as directories that have been specified in the `search` directive.

Chapter 6:
Debugging a Program Module

The VOS Symbolic Debugger (hereafter called “the debugger”) is a tool that allows you to check the logic of a program systematically while the program is executing. This chapter discusses the following topics.

- ‘‘Summary of Debugger Requests’’
- ‘‘Preparing a Program for Debugging’’
- ‘‘Invoking the Debugger’’
- ‘‘Using Debugger Requests’’
- ‘‘Moving from One Block to Another’’
- ‘‘Displaying Your Current Location’’
- ‘‘Displaying Source Code’’
- ‘‘Positioning Backward and Forward in Source Code’’
- ‘‘Starting Program Execution’’
- ‘‘Listing Frames on the Stack’’
- ‘‘Using Breakpoints’’
- ‘‘Stepping through a Program’’
- ‘‘Using Additional Debugger Requests’’
- ‘‘Using Shortcuts in the Debugger’’
- ‘‘Using the Multiprocess Debugger’’

In this chapter, note that the debugging examples and discussions supplied for Continuum-series modules specifically apply to Continuum-series modules using the PA-7100 processor. See the *VOS Symbolic Debugger User’s Guide (R308)* for examples and discussions related to Continuum-series modules using the PA-8000 processor.

Summary of Debugger Requests

Table 6-1 briefly describes each debugger request and lists the section in this chapter that contains more information about each request. Any requests not described in this chapter are described in the *VOS Symbolic Debugger User’s Guide (R308)*.

Table 6-1. Debugger Requests *(Page 1 of 3)*

Request	**Description**	**Location of Discussion**
`args`	Displays the arguments of the current block	‘‘Displaying Arguments’’
`break`	Sets a breakpoint	‘‘Setting Breakpoints’’

Table 6-1. Debugger Requests *(Page 2 of 3)*

Request	Description	Location of Discussion
`call`	Invokes a procedure	‘‘Calling a Procedure’’
`clear`	Clears a breakpoint	‘‘Clearing Breakpoints’’
`continue`	Continues executing the program	‘‘Continuing Program Execution’’
`disassemble`	Shows the assembly language code for the current line	‘‘Examining a Line’s Assembly Code’’
`display`	Shows an expression, variable, or memory location	‘‘Displaying an Expression’s Value’’
`dump`	Shows a dump of a variable or location	‘‘Displaying a Variable’s Address and Contents’’
`env`	Sets the block environment	‘‘Moving from One Block to Another’’
`help`	Displays online documentation	‘‘Getting Online Help’’
`if`	Executes one or more requests if a condition is met, or executes another request or requests if a condition is not met	‘‘Issuing Conditional Requests’’
`keep`	Saves a keep module of an interrupted program	*VOS Symbolic Debugger User’s Guide (R308)*
`list`	Lists breakpoints	‘‘Listing Breakpoints’’
`machine`	Selects `machine` mode as the source mode	*VOS Symbolic Debugger User’s Guide (R308)*
`pl1`	Selects `pl1` mode as the source mode	*VOS Symbolic Debugger User’s Guide (R308)*
`position`	Sets the current line backward or forward in the source code	‘‘Positioning Backward and Forward in Source Code’’
`quit`	Ends the debugging session	‘‘Ending and Interrupting a Debugging Session’’
`regs`	Displays the contents of all machine registers	‘‘Examining Registers’’
`return`	Returns your process to break level after entering the debugger from break level	*VOS Symbolic Debugger User’s Guide (R308)*
`set`	Assigns a value to a data item	‘‘Changing a Data Item’s Value’’

Table 6-1. Debugger Requests *(Page 3 of 3)*

Request	Description	Location of Discussion
`source`	Displays one or more lines of source code	‘‘Displaying Source Code’’
`source_path`	Specifies or displays the current source path	‘‘Checking for Differences between the Source Module and Program Module’’
`start`	Begins program execution	‘‘Starting Program Execution’’
`step`	Executes the next statement or instruction	‘‘Stepping through a Program’’
`symbol`	Shows a variable’s data type, size, and location	‘‘Displaying Declaration Information’’
`task_status`	Displays information about a task or tasks	‘‘Checking a Task’s Status’’
`trace`	Displays all active procedures on the stack	‘‘Listing Frames on the Stack’’
`where`	Displays the current environment	‘‘Displaying Your Current Location’’

Preparing a Program for Debugging

This section explains how you prepare a program for debugging.

To debug a program with the debugger in `pl1` mode, you first compile the source module using the `pl1` command with one of the following arguments: `-table` or `-production_table`.

Both the `-table` and `-production_table` arguments incorporate a symbol table into the program module. However, the `-table` argument enables you to use the full functionality of the debugger, while the `-production_table` argument enables less functionality but full optimization.

The symbol table produced by `-production_table` is smaller than the table produced by `-table`, and it omits symbols that are declared but never referenced in the source module. As a result of full optimization, a program compiled with `-production_table` may produce unpredictable output if you issue the `set` debugger request or alter the flow of control by specifying a new line number with the `continue` debugger request.

When you compile a source module with the `-table` or `-production_table` argument, you need not specify any additional arguments with the `bind` command. The binder automatically places the symbol table into the program module unless you specify the `bind` command’s `-no_table` argument.

If you specify neither `-table` nor `-production_table`, a default symbol table containing only a statement map is created. See ‘‘Examining a Program’s Assembly Code” later in this chapter for more information about using the default symbol table.

Invoking the Debugger

You can invoke the debugger from command level or from break level. This section discusses the following topics.

- ‘‘Invoking the Debugger from Command Level on a Program Module”
- ‘‘Invoking the Debugger from Command Level on a Keep Module”
- ‘‘Invoking the Debugger from Command Level Using the `mp_debug` Command”
- ‘‘Invoking the Debugger from Break Level on a Program Module”

Invoking the Debugger from Command Level on a Program Module

From command level, you can invoke the debugger to control and examine the execution of a program module. When you invoke the debugger from command level, the operating system loads the specified program module and puts your process at the debugger request level. The `debug` command has the following syntax.

```
debug program_name
```

The following example shows the beginning of a debugging session for the program `employee_info`. Note that the `.pm` suffix is optional.

```
debug employee_info.pm
Entering debug.
New language is pl1.
db?
```

Invoking the Debugger from Command Level on a Keep Module

From command level, you can invoke the debugger to examine a keep module. A *keep module* is an image of a program that was interrupted and stored in its interrupted state in a file with the suffix `.kp`.

Any of the following can interrupt a program.

- a run-time error
- issuing the Ctrl Break request
- issuing the `break_process` command (for background processes)

In all cases, the operating system places your process at break level. If you specify the `keep` request or its abbreviation `k` from break level, the operating system stores the interrupted program in a keep module. The following example illustrates how to create a keep module.

```
BREAK
Request?  (stop, continue, debug, keep, login, re-enter) k
```

If you are running a program in batch mode and an error occurs, the operating system automatically freezes the program in a keep module, unless you have specified other actions in an error handler in the program.

To debug a keep module, you enter the `debug` command and specify the name of the file with the `.kp` suffix. The following example illustrates.

```
debug employee_info.kp
```

You cannot debug a keep module unless the following requirements are met.

- You must debug a keep module on the version of the operating system on which you created the keep module.
- You must debug a keep module on a machine containing the same processor type as the machine on which you created the keep module.
- The path name of the program module that created the keep module must be the same as it was when the keep module was created.
- The program module must not have changed since the keep module was created. You should not rebind the program module until the keep module is analyzed.

Invoking the Debugger from Command Level Using the `mp_debug` Command

The multiprocess debugger allows you to debug one or more processes simultaneously, from a single terminal.

From command level, you invoke the multiprocess debugger by issuing the `mp_debug` command, with no arguments. The following example illustrates.

```
mp_debug
```

See ‘‘Using the Multiprocess Debugger” later in this chapter for more information about `mp_debug`.

Invoking the Debugger from Break Level on a Program Module

From break level, you invoke the debugger on a program module by issuing the `debug` request or its abbreviation `d`. The following example shows the break-level prompt and the debugger’s opening message and prompt.

```
Request?  (stop, continue, debug, keep, login, re-enter) d
Entering debug.
New language is pl1.
db?
```

Debugging from break level allows you to check your program at the point where it halted, whether you stopped it by issuing the Ctrl Break request or an error in the program stopped it.

Using Debugger Requests

This section explains basic debugger requests and concepts. It discusses the following topics.

- ‘‘Terminology”
- ‘‘Specifying Debugger Requests”
- ‘‘Getting Online Help”
- ‘‘Ending and Interrupting a Debugging Session”

Terminology

This section defines some terms that are used in this chapter to describe the debugger requests.

A *procedure* is a sequence of statements, beginning with a procedure name and a `procedure` statement and ending with an `end` statement, that can be activated and executed as a unit.

A *block* is a procedure or a begin block.

Scope is the block of a program in which a particular declared name is known. The *current scope* is the scope of the block that is executing. If execution has not started, the current scope is the block where you are positioned.

A *stack frame* is an area of storage that is associated with an activation of a procedure, begin block, or on-unit. The stack frame holds information that is unique to that activation. This information includes the storage of automatic variables declared within the block and the location to which control returns when the activation is completed.

A *stack* is the area of storage that contains, in an ordered series, all of the stack frames associated with an executing program.

The *current environment* is the unit of executable code or program unit that the debugger is currently accessing. If no blocks are executing and you have not issued any requests, the current environment is the main program block of the source module. Certain requests, particularly `env` and `position`, change the current environment. For more information about changing the current environment, see ‘‘Moving from One Block to Another” and ‘‘Positioning Backward and Forward in Source Code” later in this chapter.

Specifying Debugger Requests

After you invoke the debugger, the `db?` prompt appears on the terminal’s screen indicating that the debugger is ready to process a request. From this prompt, you can specify one or more requests. Separate one request from the next using a semicolon (`;`). The debugger allows as many requests on a line as will fit within the 300-character-per-line limit.

When you issue a debugger request, the debugger carries out the request and then displays another prompt. If you issue a request line with multiple requests, the debugger carries out all of the requests and then displays another prompt. An exception to this rule occurs when a request concerning program execution appears in the middle of a request line. When this happens, the debugger carries out all of the requests up to and including the execution request, but it does not perform the requests appearing after the execution request. Therefore, you should always place execution requests last in a request line. The execution requests are `start`, `continue`, and `step`.

In the following example, the debugger would not perform the `display` request because `start` appears before it.

```
db? break 100; start; display average
```

Getting Online Help

Within the debugger, you can use the `help` request to get online information about all of the debugger requests or more detailed information about one of the requests.

The `help` request has the following syntax.

```
help [request_name]
```

To display brief descriptions of all debugger requests, you specify the `help` request with no arguments. Figure 6-1 shows how to use the `help` request to view the full help screen.

```
db? help
debug:

The following requests are available in debug:

basic, cobol, pl1
fortran, pascal, c,
machine                        Selects language.

args                           Displays the arguments of the current block.
break                          Sets a breakpoint.
call                           Calls a procedure.
clear                          Clears a breakpoint.
continue                       Continues executing the program.
continue <l>                   Continues executing at line number <l>.
disassemble                    Disassembles machine code.
display                        Displays an expression, variable, or location.
dump                           Dumps a variable or location.
env                            Sets the current debugger environment.
help                           Displays online documentation.
if ... then                    Executes a request list if a condition is met.
keep                           Saves a keep module for the program.
list                           Lists breakpoints.
position                       Sets the position in the current module.
quit                           Exits the debugger and terminates the program.
regs                           Displays all registers (entire machine context).
return                         Returns to BREAK level.
set                            Modifies a variable or location.
source                         Displays one or more source lines.
source_path                    Specifies/displays the current source path.
start                          Starts execution of the program.
step                           Executes the next statement.
symbol                         Displays information about a symbol.
task_status                    Displays information about a task or tasks.
trace                          Displays a stack trace.
where                          Displays the current environment.


Internal system commands may be executed if they are preceded by '..'.  For
example, type '..list' to invoke the list command.

For online documentation of a debugger request, type 'help <request_name>'.
```

Figure 6-1. Using the `help` Request

To get more detailed information about one debugger request, specify `help` with the name of the request as an argument. For example, the following `help` request displays information about the `set` request.

```
db? help set
set (in debug):

set <variable> = <expression>
or
set <substring> = <expression>

Assigns the value of <expression> to <variable> or <substring>.
```

Ending and Interrupting a Debugging Session

This section explains how to end and interrupt a debugging session.

To end your debugging session, specify the `quit` request at the debugger prompt. This request has no arguments.

The debugger discards the image of the executing program and all frames on the stack.

To interrupt an executing program, issue the Ctrl Break request. This request produces the following break-level prompt.

```
BREAK
Request?  (stop, continue, debug, keep, login, re-enter)
```

From break level, you can choose one of the six requests shown in the prompt. Table 6-2 lists the requests and describes their functions.

Table 6-2. Break-Level Requests

Request	**Description**
`stop`	Returns your process to command level.
`continue`	Continues the process, if possible, from where you interrupted it.
`debug`	Starts a new debugging session. You may need to use the `env` request to set the current line to a block in your program.
`keep`	Stores the executable image in a file named *`program_module`*`.kp`.
`login`	Creates a subprocess.
`re-enter`	The `reenter` condition is signaled. In VOS PL/I source modules, you can specify an `on reenter` condition handler by using a `reenter` on-unit or the `s$enable_condition` subroutine.

If you select the `continue` request and execution cannot resume, you are returned to break level. In VOS PL/I, you can set up an on-unit in your program to handle this condition. The following example shows an on-unit you could establish for the `break` condition.

```
on break
     put skip list('You cannot break out of this program.');
```

When control reaches the end of a `break` on-unit, control returns to the point of the signal and execution continues. Therefore, the preceding example prevents a user from breaking out of the program.

If you have not set up an on-unit to handle the `break` condition, you can terminate the session with the `stop` request, or you can resume execution from an error-free point in the program (if another entry point exists) by specifying the `re-enter` request. If you specify `re-enter`, however, you **must** have already established an on-unit for the `reenter` condition.

The following program fragment shows a sample on-unit you can use to handle the `reenter` condition.

```
on reenter
     goto REQUEST_LOOP;
     .
     .
     .
REQUEST_LOOP:
     do while(^end_flag);
          call read_next_request;
          call execute_request(end_flag);
     end REQUEST_LOOP;
```

If no on-unit is established for the `reenter` condition, typing `re-enter` at break level returns the process to break level.

Moving from One Block to Another

The `env` request causes the debugger to change the current environment to the environment specified as an argument to the `env` request.

The `env` request has the following syntax.

```
     ⎧       environment_name         ⎫
     ⎪       stack_frame_num          ⎪
env  ⎨ -frameptr pointer_expression   ⎬
     ⎩        -task task_id           ⎭
```

In VOS PL/I, an *environment* is the main program, or a called program, subroutine, or begin block. If the specified block is in a different programming language, the debugger tells you the new language and displays source code in that language if you specify the `source` request. If the debugging mode is `machine`, the debugger displays the language source code, not the assembly code, when you specify the `source` request. (See ‘‘Displaying Source Code” later in this chapter for more information about the `source` request.)

When you issue the `env` request in an active block, the debugger sets the current line to the line that is currently executing in the specified block. In an inactive block, the debugger sets the current line to the first line in the block of the new block environment.

Note that changing environments does not affect program execution. The execution begins at the specified entry point of the program if you have not started the program. If you change environments from a breakpoint, execution starts at the next executable statement after the breakpoint.

> **Note:** If more than one invocation of a block exists on the stack (as in a recursive procedure), the `source` and `position` requests change the current environment in an undefined manner.

The *`environment`* argument can be one of the following:

- a name specifying a procedure
- the name of an object module
- the keyword `-task` with an integer argument
- an unsigned integer specifying the stack frame number of the block
- a signed integer specifying a stack frame before (if the integer is negative) or after (if the integer is positive) the current environment

If *`environment`* is a name, the argument specifies any procedure that is on the stack frame or in the current scope, or that is an external procedure having the same name, as described in the following steps.

1. The new current environment is set to the most recent activation of the block named *`environment`*.
2. If the procedure is inactive, the debugger searches for a procedure known in the current scope.
3. If the debugger finds no procedure known in the current scope, it searches for an external procedure having the same name.
4. If the debugger finds no procedure having the same name, it searches for an object module having the same name.
5. If the debugger finds no object module having the same name, it displays an error message.

The following example specifies a name for *`environment`*. In this case, the new environment is a called procedure named `init_data`.

```
db? env init_data
db? source 5
       1  init_data:
       2       procedure(sum);
       3
       4       declare num1 fixed bin(15);
       5       declare num2 fixed bin(15);
The code address is set to line 1 (the start of the statement).
```

If *`environment`* is an unsigned integer, the argument specifies the stack frame number of an active block. To find the stack frame number of a block, specify the `trace` request. (For more information about the `trace` request, see ‘‘Listing Frames on the Stack” later in this chapter.)

The following example illustrates the use of the `trace` request to find the stack frame number associated with an active block. The `env` request, in this example, brings you to line 7 of the procedure `add_num`. Line 7 is the line of the statement that is presently executing in `add_num`. The code address in that program is set to the start of line 9, the next executable statement.

```
db? trace
#  2:  add_num (line 7 in module add_num)
db? env 2
db? source 3
      7       call init_data(sum);
      8
      9       put skip list ('The value of sum is ', sum);
```

Line 7 of `add_num` calls the `init_data` program. For more information about the `env` request, see the *VOS Symbolic Debugger User's Guide (R308).*

Displaying Your Current Location

If you do not know your location in a program, you can specify the `where` request to display the following information.

- the current line
- the current statement
- the current block environment
- if you are in a tasking program, the current task

This request has no arguments.

In the following example, the `where` request shows that the current environment is the `employee_info` program. The request also shows the debugger mode, the path name of the source module, the source line number, the current statement's line number, and the stack frame number.

```
db? where
In employee_info, pl1 mode (source is
%s1#d01>Sales>employee_info.pl1).
Source line 25 (statement at line 25 in module employee_info), stack
frame #2.
```

Displaying Source Code

To display a program's source code, you issue the `source` debugger request. If you do not specify any arguments, `source` displays the current line of source code. If you specify a *`number`* argument, `source` displays the number of lines of source code specified by *`number`*, starting with the current line. When you issue a `source` request, the debugger resets the current line to the last source line displayed and, if necessary, changes the environment.

The `source` request has the following syntax.

```
source [number] [  -include   ]
                [ -no_include ]
```

The `source` request does not affect program execution. If you have not started the program, the execution begins at the main entry point of the program. If you use the `source` request from a breakpoint, execution starts at the next executable statement after the breakpoint.

In the following example, the `source` request is issued with and without a *`number`* argument.

```
db? source
      1  employee_info:
db? source 5
      1  employee_info:
      2       procedure options(main);
      3
      4  /* This program accepts employee information, writes it   */
      5  /* to a file, then prints all of the records in the file. */
The code address is set to line 1 (the start of the statement).
```

By default, the `source` request displays the source code line that names an include file, but it does not list the contents of the include file. To display the contents of an include file, specify the `-include` argument in the `source` request. For example, the following request displays 10 lines of code, including the contents of any include files encountered.

```
source 10 -include
```

Once you specify the `-include` argument, the debugger displays include files until you specify the `-no_include` argument in the `source` request.

Positioning Backward and Forward in Source Code

When you are viewing source code with the `source` debugger request, you may want to move forward or backward in the code. You can do this using the `position` request.

The `position` request has the following syntax.

```
         ⎧number⎫ ⎡  -include   ⎤
position ⎨      ⎬ ⎢             ⎥
         ⎩string⎭ ⎣-no_include  ⎦
```

To move in either direction, you specify the `position` request with the *`number`* argument. The *`number`* can be an unsigned integer that specifies the line number or a signed integer that specifies the line relative to the current position.

If *`number`* is an unsigned integer, the debugger sets the line with that source line number as the new current line. In the following example, the `position` request sets line 25 of the source module as the current line.

```
db? position 25
db? source
      25       call print_info;
```

If *`number`* is a signed integer, the debugger adds it to the current line number to get the number of the new current line. By specifying a negative number, you can reset the current line to a previous line in the program. The debugger, by default, does not count lines in include files in this calculation. If you want the debugger to count the lines in include files, specify the `-include` argument with the `position` request.

If you specify the *`string`* argument, the debugger searches for the next occurrence of *`string`* in the source code. In this case, you can optionally enclose *`string`* in apostrophes. In the following example, the debugger positions itself at the first occurrence of the string `count`.

```
db? position count
The code address is set to line 1 (the start of the statement).
       13  declare  count           fixed bin(15);
```

The `position` request does not affect program execution.

The example in Figure 6-2 shows how to position the debugger in source code.

```
1.  db? position 1
2.  db? source 6
           1  convert_age:
           2       procedure options(main);
           3
           4       %include 'age_dec.incl.pl1';
           5
           6       put skip list ('What is your age in years?');
3.  db? position +3
    db? source
           9       age_in_days = age_in_years * YEAR;
4.  db? position -2
    db? source
           7       get list (age_in_years);
5.  db? position 1 -include
    db? source 10
           1  convert_age:
           2       procedure options(main);
           3
           4       %include 'age_dec.incl.pl1';
         1-1  %replace YEAR            by 365;
         1-2
         1-3  declare age_in_years     fixed bin(15);
         1-4  declare age_in_days      fixed bin(15);
           5
           6       put skip list ('What is your age in years?');
```

Figure 6-2. Using the `position` Request

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-2.

1. The `position` request positions the debugger to the first line of the source module. In this case, this request is used for demonstration purposes only, since the debugger automatically positions itself at the first line of the source module upon entering the debugging session.

2. The `source` request shows the first six lines of the source module. Note that the data contained in the include file is **not** listed when the `source` request is specified, since the `-include` argument was not specified.

3. The `position` request changes the current line from line 6 to line 9.

4. The `position` request changes the current line from line 9 to line 7.

5. The `position` request changes the current line from line 7 to line 1. The request also specifies that lines in the include file be listed when the `source` request is specified. Note that specifying the `-include` argument with the `source` request also displays lines in an include file.

Starting Program Execution

If you specify the `start` request, the debugger starts executing the program and does not stop until the program terminates or until the debugger finds a breakpoint. This request has no arguments.

Since no breakpoints have been set, the following `start` request executes an entire program.

```
db? start
```

When the program completes execution, the debugger displays the following message.

```
Entering debug after program termination.
New language is machine.
```

For more information about setting breakpoints, see ‘‘Using Breakpoints’’ later in this chapter.

Note: If you debug a program as part of a command macro, be aware that your program must finish executing normally. Otherwise, any subsequent commands in the command macro will not execute. See the *VOS Symbolic Debugger User’s Guide (R308)* for more information.

Listing Frames on the Stack

The `trace` request displays information about environments that have frames on the stack. If the program is a tasking program, each task has its own stack and its own set of environments. (For more information about debugging tasking programs, see ‘‘Checking a Task’s Status’’ later in this chapter.) An environment does not appear on the stack until the execution of that environment has started. Thus, you generally set a breakpoint in the program, specify the `start` request to begin execution, and at the breakpoint, specify the `trace` request. (See ‘‘Using Breakpoints’’ later in this chapter for more information about setting breakpoints.)

The `trace` request has the following syntax.

```
trace [number] [-all] [-args] [-on_units]
```

When you specify the `trace` request without specifying a *`number`* argument, the request displays the entire stack. The following `trace` request shows information about two active environments, `enter_info` and `employee_info`. It also shows the line numbers of the statements that are currently executing and the name of the module that contains the two environments.

```
Break in enter_info (employee_info) -- line 40
db? trace
#  3:  enter_info (line 40 in module employee_info)
#  2:  employee_info (line 22 in module employee_info)
```

You can limit the number of block activations that the debugger displays by specifying an unsigned integer. In the following example, the first `trace` request specifies the value 1 to show the most recent block activation. The second `trace` request specifies the value 2 to show the last two block activations.

```
db? trace 1
#  3:  init_data (line 10 in module init_data)
db? trace 2
#  3:  init_data (line 10 in module init_data)
#  2:  add_num (line 7 in module add_num)
```

The example in Figure 6-3 demonstrates how to display all of the stack frames currently associated with a source module.

```
1.  db? env init_data
2.  db? source 10
           1  init_data:
           2       procedure(sum);
           3
           4       declare num1   fixed bin(15);
           5       declare num2   fixed bin(15);
           6       declare sum    fixed bin(15);
           7
           8       num1= 2;
           9       num2= 3;
          10       sum= num1+ num2;
3.  db? break 10
4.  db? start
    Break in init_data -- line 10
5.  db? trace -all
    #  3:  init_data (line 10 in module init_data)
    #  2:  add_num (line 7 in module add_num)
    #  1:  start_user_program (807ED044x ***)
```

Figure 6-3. Displaying All Current Stack Frames

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-3.

1. The `env` request changes the current environment to the `init_data` source module.
2. The `source` request displays 10 lines of source code in `init_data`.
3. The `break` request sets a breakpoint on line 10 of `init_data`.

4. The `start` request executes the program until it reaches the breakpoint.

5. The `trace` request lists all currently active environments on the stack. In this example, stack frame `#3` shows the `init_data` environment, and stack frame `#2` shows the `add_num` environment.

Occasionally, a program error destroys the information in the stack frame needed by the debugger for some requests. If this information is destroyed, you can find the statement that caused the error by invoking the debugger and stepping through the program or dividing the program with breakpoints until you find the statement causing the error. See ‘‘Stepping through a Program” and ‘‘Setting Breakpoints” later in this chapter for more information.

The example shown in Figure 6-3 was produced on an XA/R-series module. On XA2000-series and Continuum-series modules, the address in stack frame `#1` would be different. The `start_user_program` routine does not have a statement map, so the debugger cannot report a line number for that routine. The address after frame `#1` represents the contents of the program counter register at the time of the call to `init_data`.

You can also display the arguments associated with each block, if any exist, by specifying the `-args` argument. For more information about the `trace` request, see the *VOS Symbolic Debugger User's Guide (R308)*.

Using Breakpoints

This section discusses the following topics.

- ‘‘Setting Breakpoints”
- ‘‘Issuing Requests from Breakpoints”
- ‘‘Clearing Breakpoints”
- ‘‘Listing Breakpoints”

Setting Breakpoints

You can set breakpoints before you run a program, when you encounter a breakpoint in a program, or while you step through a program. For more information about stepping through a program, see ‘‘Stepping through a Program” later in this chapter.

The `break` request has the following syntax.

```
      ┌      ┐
break │ line │ [(request_list)] [-every n]
      │ label│
      └      ┘
```

The `break` request sets a breakpoint on the current statement, on the statement that would be the current statement if *`line`* were the current line, or on the statement represented by *`label`*. The *`request_list`* is a parenthesized sequence of debugger commands separated by semicolons. These requests are executed every time execution stops at the breakpoint. If you specify `-every` *`n`*, execution stops at the breakpoint only every *`n`*th time the statement is reached.

You can use any of the following methods to set a breakpoint.

- Issue the `break` request without an argument to set the breakpoint at the current line.
- Issue the `break` request and specify a line number to set the breakpoint at another line. For example, the request `break 66` sets a breakpoint at line 66.
- Issue the `break` request and specify a label or procedure name to set the breakpoint at a specific point in the program's execution. For example, the request `break print_info` sets a breakpoint at line 53, the line at which the label `print_info` is defined.

To set breakpoints before you run the program, you must perform the following actions.

- Determine where to set breakpoints by scrolling through the source code with the `source` request or by looking at a program listing.
- Set the breakpoints with the `break` request.
- Specify the `start` or `continue` request to start or continue program execution.

If your program contains more than one object module, you must use the `env` request to set breakpoints in environments other than the current environment. For example, if your program consists of two object modules, `add_num` and `init_data`, and the current environment is `add_num`, you cannot set a breakpoint in `init_data` without first using the `env` request to change to `init_data`'s environment. See the *VOS Symbolic Debugger User's Guide (R308)* for more information.

Issuing Requests from Breakpoints

When the debugger stops the program at a breakpoint, you can specify any debugger request. This section discusses the following topics related to the tasks you are most likely to perform.

- ``Displaying an Expression's Value''
- ``Displaying Declaration Information''
- ``Changing a Data Item's Value''
- ``Continuing Program Execution''
- ``Setting Another Breakpoint''
- ``Issuing Conditional Requests''

The only request you **cannot** use from a breakpoint is the `start` request.

The following sections describe the tasks shown in the preceding list.

Displaying an Expression's Value

If you specify the `display` request, the debugger displays the value of the specified expression.

The `display` request has the following syntax.

```
display expression
```

In `pl1` mode, you can specify any of the following types of expressions in the `display` request.

- a constant
- a variable reference
- one of the following VOS PL/I built-in functions: `string`, `substr`, `null`, `byte`, `rank`, `length`, `addrel`, or `addr`
- a function known in the current environment
- an expression combining any of the preceding items using the VOS PL/I arithmetical, relational, and string operators

For example, `100`, `count`, `x >= y`, and `amount >= 100` are valid `pl1`-mode expressions. The permitted relational operators in `pl1` mode are the VOS PL/I operators `<`, `<=`, `=`, `>=`, `>`, `^<`, `^=`, and `^>`.

If *`expression`* is an array, you can display a range of elements in the array with the construct `(`*`low`*`:`*`high`*`)` in the position where a subscript value would be. The specifiers *`low`* and *`high`* are integer values indicating the low and high ends of the range.

You can display all of the elements in an array by specifying the name of the array, and you can use the asterisk (`*`) to specify all of the subscripts in a particular dimension of an array. The example in Figure 6-4 shows how to perform both of these tasks using the `display` request.

```
1. db? position 10; source 19
   The code address is set to line 1 (the start of the statement).
         10       declare two_dim_array(3,4)   fixed bin(15);
         11       declare i                    fixed bin(15);
         12       declare j                    fixed bin(15);
         13       declare counter              fixed bin(15) static initial(0);
         14
         15       call assignment_loop;
         16
         17  assignment_loop:
         18       procedure;
         19
         20       dim1: do i = 1 to 3 by 1;
         21       dim2:    do j = 1 to 3 by 1;
         22                   counter = counter + 1;
         23                   two_dim_array(i, j) = counter;
         24                end dim2;
         25                two_dim_array(i,4) = 999;
         26             end dim1;
         27
         28  end;  /* Assignment_loop */
2. db? break 28; start
   Break in assignment_loop (two_d_array)  -- line 28
3. db? display two_dim_array
   two_dim_array(1,1)  = 1
   two_dim_array(1,2)  = 2
   two_dim_array(1,3)  = 3
   two_dim_array(1,4)  = 999
   two_dim_array(2,1)  = 4
   two_dim_array(2,2)  = 5
   two_dim_array(2,3)  = 6
   two_dim_array(2,4)  = 999
   two_dim_array(3,1)  = 7
   two_dim_array(3,2)  = 8
   two_dim_array(3,3)  = 9
   two_dim_array(3,4)  = 999
4. db? display two_dim_array(*,1:2)
   two_dim_array(1,1)  = 1
   two_dim_array(1,2)  = 2
   two_dim_array(2,1)  = 4
   two_dim_array(2,2)  = 5
   two_dim_array(3,1)  = 7
   two_dim_array(3,2)  = 8
```

Figure 6-4. Displaying Elements in a Two-Dimensional Array

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-4.

1. The `position` request moves the debugger to line 10 of the source module. The `source` request displays 19 lines of source code.

2. The `break` request sets a breakpoint on line 28. The `start` request executes the program until it reaches the breakpoint.

3. The `display` request shows the current values of all elements in the two-dimensional array `two_dim_array`.

4. The `display` request shows the current values of a specified range of elements in `two_dim_array`.

Displaying Declaration Information

If you specify the `symbol` request, the debugger displays the specified symbol's data type, storage class, address, size, offset (if the symbol is a structure member), and the block in which the symbol is declared.

The `symbol` request has the following syntax.

```
symbol symbol_name
```

Figure 6-5 shows how to display declaration information about a data item using the `symbol` request.

```
1. db? break 40
2. db? start

   Enter employee's last name. Chan
   Enter employee's ID number. 4567
   Enter employee's title. Engineer
   Break in enter_info (employee_info)  -- line 40
3. db? symbol count
   count at 04726EE6:
   13 declare count fixed bin(15,0) automatic
                             /* In employee_info, size(count) = 2 bytes  */;
4. db? symbol name
   employee_rec.name at 04726EE8:
   8 declare 2 name char(30)
                             /*   '1 employee_rec' in employee_info
                             , size(name) = 30 bytes, offset = 0 bytes  */;
```

Figure 6-5. Displaying Declaration Information about a Data Name

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-5.

1. The `break` request sets a breakpoint in the program `employee_info` at line 40.

2. The `start` request starts the program.

3. The `symbol` request shows declaration information about the data item `count`. In this case, the address of `count` is `04726EE6`. The data item `count` has the `fixed bin` data type, is two bytes long, and was declared on line 13.

4. The `symbol` request shows declaration information about the data item `name`. The name `employee_rec.name` indicates that `name` is a member of the structure `employee_rec`. The address of `name` is `04726EE8`, and the first element of `name` is located at offset 0. The data item `name` has the `char` data type, and it is 30 bytes long.

Changing a Data Item's Value

The `set` request allows you to change the value of a data item.

The `set` request has the following syntax.

```
set reference = expression
```

The following example changes the value of the data item `x` to 3.

```
db? set x = 3
```

Continuing Program Execution

The `continue` request causes the debugger to restart execution from the line where you interrupted the program.

The `continue` request has the following syntax.

```
         ⎡    line     ⎤
continue ⎢             ⎥
         ⎣ label [:]   ⎦
```

If you select the `continue` request with the *`line`* argument, execution starts from the statement associated with the specified line, if possible. After program termination, the `continue` request does not work. The *`line`* you specify can appear before or after the present line; however, it must be visible within the current scope.

The following example shows how to continue execution from a specified line. In this case, line 44 is specified.

```
db? continue 44
```

If the line is in a containing procedure, the `continue` request unwinds the stack, having the same effect as a nonlocal `goto` statement.

Setting Another Breakpoint

If you select the `break` request with the *`line`* argument, execution stops at the specified line. In the following example, a breakpoint is set for line 59. Because the argument `-every 3` is specified, execution stops every third time execution reaches that statement.

```
db? break 59 -every 3
```

Issuing Conditional Requests

To issue debugger requests conditionally, you can specify an `if` request and specify the other debugger requests to be executed as part of the `if` request. Conditional requests are often used with `break` statements.

The `if` request has the following syntax.

```
if conditional_test then (request_list_1)
[else (request_list_2)]
```

As shown in the preceding syntax, *`request_list_1`* and *`request_list_2`* are both enclosed in parentheses. A request list consists of one or more debugger requests, separated by semicolons.

In the following example, if the variable `z` is less than 1, the value of `x` is set to 3, the value of `y` is displayed, and program execution resumes from the breakpoint. If, however, `z` is not less than 1, the debugger does not change the value of `x` or display `y`, and processing resumes.

```
db? if z < 1 then (set x = 3; display y; continue)
else (continue)
```

The following example demonstrates how to specify an `if` request at a breakpoint.

```
db? break 15 (if z < 1 then (set x = 3; display y)
else (continue))
```

In the preceding example, if the break occurs in a loop, the debugger executes the requests that you specified for each iteration of the loop. If you do not include the `continue` requests in *`request_list_1`* and *`request_list_2`*, the debugger interrupts your program at the specified line each time it goes through the loop.

Figure 6-6 shows how to issue an `if` request at a breakpoint.

```
1. db? source 12
         53  print_info:
         54       procedure;
         55
         56  /* Display each record on the screen. */
         57
         58       do x = 1 to count;
         59            read file(emp_file) into(employee_rec);
         60            put skip list ('Name is ', employee_rec.name);
         61            put skip list ('ID number is ', employee_rec.id_num);
         62            put skip list ('Title is ', employee_rec.title);
         63            put skip list (' ');
         64       end; /* do-loop */
2. db? break 59 (if employee_rec.name = 'Smith' then (display employee_rec.name; continue) else
      (continue))
3. db? start

   Enter employee's last name. Smith
   Enter employee's ID number. 435
   Enter employee's title. Bookkeeper

   Do you want to enter more records? Answer y or n. n
4. employee_rec.name = 'Smith                        '

5. Name is   Smith
   ID number is         435
   Title is  Bookkeeper

           1 record(s) printed.
6. Entering debug after program termination.
```

Figure 6-6. Issuing Conditional Requests

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-6.

1. The `source` request displays 12 lines of source code and allows you to see where breakpoints should be set.

2. The `break` request sets a breakpoint and specifies an `if` request. The breakpoint is set within the loop that reads the record. The `if` request checks the value of `employee_rec.name`. If `employee_rec.name` equals `'Smith'`, `employee_rec.name` is displayed and execution continues. (Note that the request is executed each time the condition is met.) If `employee_rec.name` does not equal `'Smith'`, the debugger continues to execute the program without stopping at the breakpoint.

3. The `start` request starts program execution.

4. The `display` request in the `if` request is executed because `employee_rec.name` equals `'Smith'`. Thus, the information is displayed in the following message.

   ```
   employee_rec.name = 'Smith                              '
   ```

5. The program continues to execute after the breakpoint.

6. The message `Entering debug after program termination` indicates that the debugger has executed the rest of the program.

Clearing Breakpoints

This section describes how to clear breakpoints using the `clear` request.

The `clear` request has the following syntax.

```
      ┌               ┐
      │     -all      │
clear │     line      │
      │  label [ : ]  │
      └               ┘
```

If you specify the `clear` request at a breakpoint, the debugger clears that breakpoint.

In the following example, a breakpoint is set on line 59. If this breakpoint is not cleared, the debugger interrupts program execution at line 59 every time the loop is executed. However, after the third iteration of the loop, the breakpoint is cleared with a `clear` request and a `continue` request is issued so that program execution can progress without interruption.

```
db? start
Break in employee_info -- line 59
db? continue
Break in employee_info -- line 59
db? continue
Break in employee_info -- line 59
db? clear
db? continue
```

To clear a breakpoint before program execution starts or to clear a breakpoint from another point in the program, you issue a `clear` request and specify the breakpoint's line number. In the following example, breakpoints are set on line 21 and line 59. When the debugger interrupts the program at line 21, the `clear` request is issued to clear the second breakpoint at line 59.

```
db? break 21
db? break 59
db? start
Break in employee_info -- line 21
db? clear 59
db? continue
```

To clear all breakpoints set during a debugging session, specify the `clear` request and specify the `-all` argument.

Listing Breakpoints

The `list` request displays all of the breakpoints and any debugger requests executed in the request list. It also displays the number of times the debugger has encountered the breakpoints during the debugging session. This request has no arguments.

For example, the following `list` request shows that breaks have occurred twice at line 59 and once at line 21.

```
db? list
Break at employee_info line 59 , 2 hits
Break at employee_info line 21 , 1 hits
```

Stepping through a Program

The `step` request allows you to execute one or more statements at a time. At each step, you can perform any of the tasks that you can perform at a breakpoint. The `step` request has the following syntax.

```
                ┌  -in   ┐
step [number]   │        │
                └ -no_in ┘
```

If you do not supply the *`number`* argument, the `step` request steps through your program one statement at a time. If you supply a number, the debugger executes that number of statements, starting at the statement following the last statement executed.

The `-in` and `-no_in` arguments are explained later in this section.

The stepping mechanism can be slow. To step through only a part of a program, set a breakpoint on the statement where you want to start stepping, specify the `start` request, and then begin stepping at the breakpoint.

If you are debugging a program that was compiled for an XA/R-series module or a Continuum-series module, the `step` request could terminate and the program could continue to execute in the following situations.

- in a statement that calls an on-unit to handle the `reenter` condition. Set a breakpoint on the statement following this one.
- in a statement that calls a VOS subroutine, such as `s$control_task`, that passes control to another section of user code. You might want to set a breakpoint in the new section of code.
- in a statement that causes a fault or raises a condition. In this situation, control passes to the condition handler for the fault or condition. You might want to set the breakpoint in the condition handler.
- if you specified the `-cpu_profile` argument during compilation

For more information about using the `step` request on XA/R-series modules and Continuum-series modules, see the *VOS Symbolic Debugger User's Guide (R308)*.

Figure 6-7 shows how to step through a program.

```
   db? position 17
   db? source 10
         17      open file(emp_file) title('emp_file -delete') update;
         18      answer = 'y';
         19      count = 0;
         20
         21      do while (answer = 'y');
         22           call enter_info;
         23      end; /* do-while */
         24
         25      call print_info;
         26      close file(emp_file);
   db? position 28
   db? source 13
         28   enter_info:
         29      procedure;
         30
         31  /* Enter employee information and write it to a file. */
         32
         33      put skip list ('Enter employee''s last name.');
         34      get list (employee_rec.name);
         35      put list ('Enter employee''s ID number.');
         36      get list (employee_rec.id_num);
         37      put list ('Enter employee''s title.');
         38      get list (employee_rec.title);
         39      count = count + 1;
         40      write file(emp_file) from(employee_rec);
1. db? break 22
2. db? start
   Break in employee_info -- line 22
3. db? step

   Enter employee's last name. Jones
   Enter employee's ID number. 1234
   Enter employee's title. Engineer

   Do you want to enter more records? Answer y or n. n
   Step complete at employee_info line 23.
4. db? step
   Step complete at employee_info line 25.
   db? step

   Name is   Jones
   ID number is        1234
   Title is  Engineer

           1 record(s) printed. Step complete at employee_info line 26.
```

Figure 6-7. Stepping through a Program

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-7.

1. The `break` request sets a breakpoint on line 22 in the main procedure.
2. The `start` request begins execution and signals a break at line 22 prior to transferring control to the procedure `enter_info`.
3. The `step` request executes the procedure `enter_info`, then returns control to line 23.

4. The `step` request steps through the main procedure and transfers control to the procedure `print_info`.

5. The `step` request executes the procedure `print_info`, then returns control to line 26.

Procedure calls are treated as single statements by the debugger unless you supply the `-in` argument with the `step` request at the statement that calls the procedure. If you select the `-in` argument, the debugger steps into all procedure blocks it encounters. The `-in` argument acts as a switch so that all subsequent `step` requests, with or without the `-in` argument specified, cause the debugger to step into any called procedures encountered.

Figure 6-8 shows how to step into a procedure. Note that `-no_in` is not specified in this example.

```
    db? position 17
    db? source 10
          17       open file(emp_file) title('emp_file -delete') update;
          18       answer = 'y';
          19       count = 0;
          20
          21       do while (answer = 'y');
          22            call enter_info;
          23       end; /* do-while */
          24
          25       call print_info;
          26       close file(emp_file);
1.  db? break 22
2.  db? start
    Break in employee_info -- line 22
3.  db? step -in
    Step complete at employee_info line 28.
4.  db? step
    Step complete at employee_info line 33.
```

Figure 6-8. Stepping into a Procedure

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-8.

1. The `break` request sets a breakpoint on line 22 in the main procedure.

2. The `start` request begins execution and signals a break at line 22, which calls the procedure `enter_info`.

3. The `step -in` request steps into the called procedure block.

4. The `step` request continues stepping through the `enter_info` procedure.

To prevent the debugger from stepping into other called procedures, you must specify the `-no_in` argument with the `step` request before you step into another procedure. The `-no_in` argument acts as a switch that turns off a previously specified `-in` argument. Consequently, the debugger treats subsequent procedure calls as single statements.

Figure 6-9 shows how to avoid stepping into a procedure.

```
   db? position 17
   db? source 10
         17      open file(emp_file) title('emp_file -delete') update;
         18      answer = 'y';
         19      count = 0;
         20
         21      do while (answer = 'y');
         22           call enter_info;
         23      end; /* do-while */
         24
         25      call print_info;
         26      close file(emp_file);
1. db? break 22
2. db? start
   Break in employee_info -- line 22
3. db? step -no_in

   Enter employee's last name. Sanchez
   Enter employee's ID number. 491
   Enter employee's title. Engineer

   Do you want to enter more records? Answer y or n. n
   Step complete at employee_info line 23.
```

Figure 6-9. Stepping Past a Procedure

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-9.

1. The `break` request sets a breakpoint on line 22 in the main procedure.

2. The `start` request begins execution and signals a break at line 22, which calls the procedure `enter_info`.

3. The `step -no_in` request steps past the procedure block. The message `Step complete at employee_info line 23` indicates that the step completes at the next line of `employee_info`. The `enter_info` procedure **does** execute, but the debugger does not step into it.

You can also specify the `-no_in` argument with the `step` request as soon as you step into a procedure block to avoid stepping through procedures unintentionally.

The following list explains how to step into procedure blocks without stepping through lengthy called programs.

1. Set a breakpoint at the procedure call.

2. Start the program.

3. Specify the `step` request with the `-in` argument when the debugger stops the program at the breakpoint.

4. Specify the `step` request with the `-no_in` argument immediately after you enter the procedure block.

5. Specify the `step` or `continue` request from this point on.

Using Additional Debugger Requests

This section describes additional functionality that the debugger provides. It explains and provides examples of the following tasks.

- ‘‘Calling a Procedure”
- ‘‘Displaying Arguments”
- ‘‘Checking for Differences between the Source Module and Program Module”
- ‘‘Checking a Task’s Status”
- ‘‘Examining a Program’s Assembly Code”

Calling a Procedure

The `call` request transfers control to a specified procedure and executes the procedure. The request allows you to pass parameters to the procedure, if parameters are required.

The `call` request has the following syntax.

```
call procedure [ ( parameters . . . )]
```

You can invoke any entry point known to the current environment with the `call` request. You can execute a procedure many times, using different parameter values in each call. You might use the `call` request to execute procedures that are written specifically for debugging purposes. For example, you might write a procedure that receives a pointer to a large structure and returns a formatted version of a few diagnostically useful members of the structure.

If the procedure being called takes parameters, include the parameters in the `call` request. The parameters appear after the procedure name, enclosed in parentheses. Separate multiple parameters with commas. In the following example, the `call` request executes the `adder` procedure, which expects two parameters.

```
db? call adder (a, b)
```

The parameters passed in a `call` request can be any expression that is a valid parameter in PL/I.

You can issue the `call` request before issuing the `start` request, after a break, or after program termination.

To debug the procedure that the `call` request invokes, you probably should set a break in the procedure before calling it. If you set a break at the beginning of the procedure, you can examine the procedure as it executes. Otherwise, the entire procedure executes without interruption, and you can examine only the results of execution, not the actual execution.

To set a break in the procedure, first specify the `env` request to change the current environment to the procedure’s environment. Then issue the `break` request to set the break, and, finally, specify the `env` request again to change the environment back to your original environment. Now you can issue the `call` request to invoke the procedure.

When the called procedure reaches the breakpoint that you set, the debugger changes the current environment to the procedure's environment. You can access variables in the called procedure using the `set` and `display` requests. You can also control the procedure's execution with the `step` and `continue` requests. When execution of the procedure completes, control returns to the calling environment, and the debugger switches the current environment back to the calling environment.

The `call` request invokes a new version of the `debug` command for the called procedure. When you step to the end of a procedure that is executing under control of the `call` request, the `step` request that returns control to the calling environment responds differently from other `step` requests. The different behavior indicates that the additional activation of the debugger associated with the called procedure has ended, and control has returned to the calling program. On XA2000-series modules, the `step` request that returns control to the calling environment displays the following message.

```
Aborting trace upon reentering debug
```

On XA/R-series modules and Continuum-series modules, the `step` request that returns control to the calling environment does not display any message. The debugger immediately displays another debugger prompt.

Displaying Arguments

To display the current values of the arguments associated with a procedure, you specify the `args` debugger request within the block of the particular procedure. This request has no arguments.

> **Note:** The `args` request has unpredictable results on optimized code running on XA/R-series and Continuum-series modules. See the *VOS Symbolic Debugger User's Guide (R308)* for more information.

Figure 6-10 displays the arguments of the called procedure `init_data` using the `args` request.

```
   db? source 11
          1  add_num:
          2       procedure options(main);
          3
          4       declare init_data  entry(fixed bin(15));
          5       declare sum        fixed bin(15);
          6
          7       call init_data(sum);
          8
          9       put skip list ('The value of sum is ', sum);
         10
         11  end add_num;
1. db? env init_data
2. db? source 12
          1  init_data:
          2       procedure(sum);
          3
          4       declare num1   fixed bin(15);
          5       declare num2   fixed bin(15);
          6       declare sum    fixed bin(15);
          7
          8       num1= 2;
          9       num2= 3;
         10       sum= num1+ num2;
         11
         12  end init_data;
3. db? break 12; start
   Break in init_data -- line 12
4. db? args
   Arguments for init_data(sum).
   sum = 5
```

Figure 6-10. Displaying the Arguments Associated with a Procedure

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-10.

1. The `env` request changes the current environment to `init_data`, the environment of a called procedure.
2. The `source` request displays source code in the called procedure.
3. The `break` request sets a breakpoint in the called procedure. The `start` request starts program execution.
4. The `args` request displays the values of all arguments to the called procedure (in this case, the value of `sum`).

For more information about passing arguments, see Chapter 8.

Checking for Differences between the Source Module and Program Module

Two situations cause a source module to differ from a program module.

The first situation occurs when you have edited the source module but have not compiled and bound it: the debugger shows the latest revision of the source code but executes the last version that was compiled and bound.

To determine whether the source module matches the program module, specify the `source` request. If the source module and program module do not match, the debugger issues a message that explains when the source module was last modified and when the program module was last compiled.

The following example shows a message displayed by the debugger for a source module that does not match the program module.

```
db? source
Date-time modified for %hr#d20>HR>Mary_Doe>pl1>employee_info.pl1
was 95-01-16 13:06:40 EST when compiled, but is now 95-02-21 14:23:58
EST.
       1   employee_info:
```

The second situation that causes a source module to differ from a program module occurs when, after compiling the source module and binding the object module, you move the source module to a different directory. The debugger cannot access it because the symbol table directs the debugger to the directory in which the source module was compiled. If there is another source module with the same path name as specified in the symbol table, the debugger uses it. If there is not, the `source` request issues the following error message.

```
Object not found.
```

If this situation occurs, specify the `source_path` request without an argument to determine the path name of the source module.

```
db? source_path
File  # Lines  Path
  0      71  %hr#d20>HR>Mary_Doe>pl1>employee_info.pl1
```

The `source_path` request has the following syntax.

```
source_path [path_name] [-file_number number]
```

To give the debugger the information it needs to find the source module that corresponds to the executing program module, use the `source_path` request with the *`path_name`* argument. The *`path_name`* argument specifies the name of the file or directory where the source module is located. If *`path_name`* is a file name, the debugger searches for the source module in the current directory. If *`path_name`* is a directory name, the debugger searches that directory for a file with the same name as the one with which the source module was originally compiled. If *`path_name`* is a directory name with a file name, the debugger searches that directory for a file with that name.

Checking a Task's Status

If you are debugging a tasking program, specifying the `task_status` request without an argument displays the task ID, the terminal port, and the state of the current task. The current task is the task currently being debugged.

The `task_status` request has the following syntax.

```
task_status [task_id] [-long] [-all]
```

If you specify the *`task_id`* argument, the debugger displays the status of the specified task. The *`task_id`* argument is an integer.

To see the status of all tasks in a tasking program, you specify the `-all` argument with the `task_status` request. To see the task's stack base, stack length, static address, and static length, you specify the `-long` argument.

The following example uses the `task_status` request to show information about the task identified by task ID 1.

```
db? task_status 1
Task id:              1
Terminal Port:        5
State:                Running
```

The following two examples illustrate the use of the `task_status` request using the `-long` argument.

The following example was created on an XA2000-series module.

```
db? task_status -long
Task id:              1
Terminal Port:        5
State:                Running
Stack Base:           04724000x
Stack Length:         32768
Static Address:       00E03000x
Static Length:        2788
```

The following example was created on a Continuum-series module.

```
db? task_status -long
Task id:              1
Terminal Port:        5
State:                Running
Stack Base:           00011000x
Stack Length:         32768
Static Address:       00008000x
Static Length:        2832
```

If you select the `env` request with the `-task` argument and a task's ID number, the debugger sets the current line to the block environment associated with the top stack frame of the specified task. You can then specify the `source` request to see the new task's source code. Note that the specified task must be active. The following example sets the current line to the block environment associated with the top stack frame of the task identified by task ID 4.

```
db? env -task 4
```

Examining a Program's Assembly Code

If you do not compile a source module with the `-table` or `-production_table` argument, you can still use the debugger in *object mode*, also known as *machine mode*. In this mode, you debug assembly code, and you refer to data values and stack frames by address instead of by name.

While in `machine` mode, you can change the mode to `pl1`, but you will not be able to use variable and label names in your debugger requests because no symbol table exists. However, if there is a statement map, you can examine the source code and set breaks on source statement lines.

When a program module contains two or more separately compiled modules, each module might be compiled with different arguments. When this is the case, some environments in the program module might have symbol tables, while others might not. The debugger resets the mode, as appropriate, whenever the environment changes.

The next three sections discuss the following topics.

- "Examining a Line's Assembly Code"
- "Examining Registers"
- "Displaying a Variable's Address and Contents"

For more information about the `machine` request and object-mode debugging, see the *VOS Symbolic Debugger User's Guide (R308).*

Examining a Line's Assembly Code

To see the assembly code translation of a specified source code line, you specify the `disassemble` request and, optionally, a *`line`* argument. If you omit the *`line`* argument, `disassemble` displays the assembly code for the current line.

The `disassemble` request has the following syntax.

```
             ┌   line        ┐
disassemble  │               │
             └  label [ : ]  ┘
```

Figure 6-11 shows the assembly code for the assignment operation on line 19 of the sample program `employee_info`.

```
db? position 19
db? source
      19       count = 0;
db? break 19
db? start
Break in employee_info -- line 19
db? disassemble
/*                                    Line 19

   1.        2.         3.          4.                         5.
   00E0015C  7E00       moveq       =0,d7
   00E0015E  3D47 FFAE  move.w      d7,-82(a6)
```

Figure 6-11. Displaying the Assembly Code Translation of a Line

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-11.

1. The first column shows the hexadecimal representation of the instruction's address.

2. The second column is the object code for line 19. Since some Stratus processors have a varying-length instruction set, instructions can be of different lengths.

3. The third column shows the mnemonic for the assembly language instruction.

4. The fourth column shows the operands.

    - In the first line, the constant `0` is moved into the data register `d7`.
    - In the second line, the contents of `d7` are moved into the address specified by `-82(a6).`

5. The fifth column contains any comments produced by the debugger. In this case, there are no comments.

Examining Registers

To see the contents of the registers at any time during your debugging session, you specify the `regs` debugger request. In addition to the contents of registers, the `regs` request provides machine condition information, which is explained in Figure 6-12, Figure 6-13, and Figure 6-14. This request has no arguments.

XA/R-series modules and Continuum-series modules contain many more registers than XA2000-series modules. The differences between the three processor families are displayed in the next three figures, which debug the same line of code in the program `employee_info`.

Figure 6-12 shows the contents of the registers during a debugging session on an XA2000-series module. Note that the column containing the data registers has as its heading the letter `D`, and the column containing the address registers has as its heading the letter `A`.

```
   db? position 19
   db? source
         19       count = 0;
   db? regs
   MC data at 04726F50, time 00000000

          D         A
     0 00000000  04726F66
     1 00E00001  047A6FE0
     2 00E0E000  04726F1C
     3 00000000  00000001
     4 00000000  00E00124
     5 0000000C  00E0E000
     6 00E00000  04726FE0
     7 0000000C  04726FB8


1. Flags:            interrupt
2. usp:              04726F3C
3. Normal Four Word Stack Frame:
4. fault no:         64 (Interrupt)
5. status:           0000 (ccr=, mask=0)
6. pc:               00E00124 (line 1 in module employee_info)
```

Figure 6-12. Displaying the Contents of Registers for an XA2000-Series Module

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-12.

1. `Flags:` indicates whether a fault or an interrupt occurred. In this case, an interrupt occurred. See a Motorola® processor manual for more information on this section of the output.

2. `usp:` displays the hexadecimal representation of the contents of the user stack pointer. In this case, the user stack pointer contains the address `04726F3C`.

3. This line displays the type of processor being used and the type of exception stack frame associated with the processor. Exception handling is machine-specific; thus, each processor has a different type of exception stack frame. In this case, the processor is an MC68020, and the exception stack frame is the standard one for that processor.

4. `fault no:` displays the number of the fault and the trap instruction that caused the fault. In this case, the operating system executed an interrupt.

5. `status:` displays the hexadecimal representation of the status register, the condition code in the condition code register (`ccr`), and the interrupt priority mask number. The condition codes can be none or any of the following: `X` for extend, `N` for negative, `Z` for zero, `V` for overflow, and `C` for carry. In this case, there is no condition code.

6. `pc:` displays the hexadecimal representation of the contents of the program counter at the time of the fault. In this case, the program counter contains the address `00E00124`.

Figure 6-13 displays the contents of the registers during a debugging session on an XA/R-series module.

```
    db? position 19
    db? source
          19       count = 0;
    db? regs
    r0:  00000000  r1:  802FC19C  r2:  03F25D90  r3:  03F25F90
    r4:  03FA73E4  r5:  03FF1F70  r6:  00000000  r7:  802DD000
    r8:  00000000  r9:  80676000  r10: 03F25F1C  r11: 00000001
    r12: 00000001  r13: 00000000  r14: 03FF186E  r15: 00019AF0
    r16: 03F25F88  r17: 03FAB1FA  r18: 03F25F42  r19: 802426FC
    r20: 00000000  r21: 00000000  r22: 00019220  r23: 00000000
    r24: 00000000  r25: 0000000C  r26: 0000000C  r27: 806767E0
    r28: 80670000  r29: 00000001  r30: 100000A0  r31: 03F25F70

     f2: 00000000   f3: 00000000   f4: 00000000   f5: 00000000
     f6: 00000000   f7: 00000000   f8: 00008000   f9: 800E0000
    f10: 00008000  f11: 00000000  f12: 00000001  f13: FFFFFFFC
    f14: 2BB30466  f15: 0007D1A7  f16: 65726D69  f17: 73735F74
    f18: 63657373  f19: 5F746572  f20: 6D696E61  f21: 6E6F2E50
    f22: 75626C69  f23: 0007D1F4  f24: 858E02AC  f25: 0007D1F4
    f26: D1F48593  f27: 00000007  f28: 0000000A  f29: 00000000
    f30: 14ACCCAA  f31: 00000000

1.  Flags:            busy_interrupt,floating
2.  usp:              03F25D90
3.  fsr:              100000A0  (MRP SI FTE AE:0 RR:0 RM:00)
4.  fault no:         64 (Interrupt)
5.  psr:              001E02A0  (IN PU PIM  PM:0 PS:00 SC:15)
6.  epsr:             00840601  (BE  Proc:1 Step:C1 DCS:8k)
7.  efa:              00000000
8.  pc:               00008214 (line 1 in module employee_info)
```

Figure 6-13. Displaying the Contents of Registers for an XA/R-Series Module

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-13.

1. `Flags:` indicates whether a fault or an interrupt occurred. In this case, the flag indicates what the processor was doing when it was interrupted. See an Intel® processor manual for more information on this section of the output.

2. `usp:` displays the hexadecimal representation of the contents of the user stack pointer. In this case, the user stack pointer contains the address `03F25D90`.

3. `fsr:` displays the status bits of the floating-point status register in hexadecimal representation. The characters in parentheses are the abbreviations defined for the various bit flags.

4. `fault no:` displays the number of the fault and the trap instruction that caused the fault. In this case, the operating system executed an interrupt.

5. `psr:` displays the status bits of the processor status register in hexadecimal representation. The characters in parentheses are the abbreviations defined for the various bit flags.

6. `epsr:` displays the status bits of the extended-processor status register in hexadecimal representation. The characters in parentheses are the abbreviations defined for the various bit flags.

7. `efa:` displays the hexadecimal representation of the *effective fault address*, which is the address of the instruction that caused the program to trap.

8. `pc:` displays the contents of the program counter. The program counter points to the instruction that the debugger is currently executing.

Figure 6-14 shows the contents of the registers during a debugging session on a Continuum-series module using a PA-7100 processor. For a similar example on a Continuum-series module using a PA-8000 processor, see the *VOS Symbolic Debugger User's Guide (R308)*.

```
db? position 19
db? source
       19        count = 0;
db? regs
  r0: 00000000    r1: C03DFF40    rp: C03E05AF    r3: 00007040
  r4: 7FFEC7C6    r5: 7FFAB184    r6: 7FFEC100    r7: 7FFEC938
  r8: 00000000    r9: C052E3C0   r10: 7FFECEF0   r11: 00000000
 r12: 00000001   r13: 00000001   r14: 7FFEC804   r15: 000010A0
 r16: 00008000   r17: 0015D594   r18: 00160C3C    t4: 00000000
  t3: 00000000    t2: 0000000C    t1: 00000001  arg3: 0000B004
arg2: 7FFAC0C0  arg1: 7FFAC0CA  arg0: 00007046    dp: 0000B004
ret0: 00008000  ret1: 00007080    sp: 00007080   mrp: C03E056B

 sr0: 8000       sr1: 8000       sr2: 0000       sr3: 0000
 sr4: 8000       sr5: 0235       sr6: 8000       sr7: 8000

 rctr: 00000000    cr1: 00000000    cr2: 00000000    cr3: 00000000
  cr4: 00000000    cr5: 00000000    cr6: 00000000    cr7: 00000000
pidr1: 00000001  pidr2: 00000000    ccr: 00000000    sar: 0000001C
pidr3: 00000000  pidr4: 00000000    iva: 00000000   eiem: FFFFFFFF
 itmr: 00000000   pcsq: 00008000   pcoq: 00002033    iir: 00000000
  isr: 00000000    ior: 00000000   ipsw: 0204000F   eirr: 00000000
  tr0: 00000000    tr1: 00000000    tr2: 7FFF5400    tr3: 0000000F
  tr4: 00000000    tr5: 00000000    tr6: 00000000    tr7: 00000000
             pcsq_back: 00008000                pcoq_back: 00002037

 fr0: 0000001E00000000   fr1: 0000000000000000   fr2: 0000000000000000
 fr3: 0000000000000000   fr4: 0000000000008000   fr5: 0000000000000000
 fr6: 0000000000000000   fr7: 0000000000000000   fr8: 3FF0000000000000
 fr9: 0000000000000000  fr10: 0000000000000000  fr11: 0000000000000000
fr12: 3FF0000000000000  fr13: 3FF0000000000000  fr14: 3FF0000000000000
fr15: 3FF0000000000000  fr16: 3FF0000000000000  fr17: 3FF0000000000000
fr18: 3FF0000000000000  fr19: 3FF0000000000000  fr20: 3FF0000000000000
fr21: 3FF0000000000000  fr22: 0000000000000000  fr23: 0000000000000000
fr24: 0000000000000000  fr25: 0000000000000000  fr26: 0000000000000000
fr27: 0000000000000000  fr28: 0000000000000000  fr29: 0000000000000000
fr30: 0000000000000000  fr31: 0000000000000000

1. Flags:            fault,floating
2. usp:              00007080
3. fsr:              0000001E  (Flags: none  Enables: VZOU  RM: nearest)
4. fault no:         0 (unassigned vector 0)
5. psw:              0204000F  (SCQPDI  C/B: 00000000)
6. pc:               000021FB (line 1 in module employee_info)
```

Figure 6-14. Displaying the Contents of Registers for a Continuum-Series Module Using a PA-7100 Processor

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-14.

1. `Flags:` indicates whether a fault or an interrupt occurred. In this case, the flag indicates what the processor was doing when it was interrupted. See a Hewlett-Packard® processor manual for more information on this section of the output.

2. `usp:` displays the hexadecimal representation of the contents of the user stack pointer. In this case, the user stack pointer contains the address `00007080`. This value also appears as `sp` in the example's first set of registers.

3. `fsr:` displays the status bits of the floating-point status register in hexadecimal representation. The characters in parentheses are the abbreviations defined for the various bit flags.

4. `fault no:` displays the number of the fault and the trap instruction that caused the fault. In this case, the operating system did not execute an interrupt.

5. `psw:` displays the contents of the Processor Status Word, a 32-bit register containing processor state information.

6. `pc:` displays the contents of the program counter. The program counter points to the instruction that the debugger is currently executing.

Displaying a Variable's Address and Contents

To see the address and hexadecimal representation of a variable or constant, you use the `dump` request.

The `dump` request has the following syntax.

```
dump variable [number]
```

The *`variable`* argument specifies the name of a variable. You cannot specify a constant for *`variable`*. The *`number`* argument specifies an integer that indicates the number of bytes to be displayed.

If you do not specify the *`number`* argument, the debugger attempts to determine how much data to dump based on the type of the variable. A minimum of 16 bytes of data are dumped.

> **Note:** You can use the `dump` request while in `pl1` mode, not just in machine mode. See the *VOS Symbolic Debugger User's Guide (R308)* for more information about using `dump` during source-mode debugging.

The example in Figure 6-15 shows the dump display of the integer variable `count`.

```
db? source
     58       do x = 1 to count;
db? break 58
db? start

Enter employee's last name. Jones
Enter employee's ID number. 1234
Enter employee's title. Engineer

Do you want to enter more records? Answer y or n. n
Break in print_info (employee_info)  -- line 58
db? dump count


   1.          2. 3.        4.                          5.
   04726EE6  0  00014A6F 6E657320 20202020 20202020  |..Jones         |
```

Figure 6-15. Displaying a Dump of a Variable

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-15.

1. The first column shows the hexadecimal representation of the variable’s base address.
2. The second column shows the offset from the address. This information is useful for determining the real address of one element of an array or structure.
3. The third column shows the contents of the first longword (four bytes) of memory, starting from the base address.
4. The fourth column shows the contents of the second, third, and fourth longwords, respectively.
5. The fifth column shows the graphic representation of the bytes contained in each longword. If a byte cannot be represented as a graphic character, the debugger displays a period (.).

   Note: If the variable or constant does not take up four longwords, the unused longwords may contain data for another variable. In this case, the extraneous data has no effect on the debugger.

Using Shortcuts in the Debugger

This section discusses the following topics.

- ‘‘Abbreviating Requests”
- ‘‘Issuing VOS Internal Commands”

Abbreviating Requests

The debugger replaces `first` abbreviation directives in your debugger requests if you have compiled an abbreviations file with the `use_abbreviations` command. For example, if you include the following abbreviation in an `abbreviations` file, and then compile the file with `use_abbreviations`, you can use the abbreviation `t` in place of the `trace` request.

```
first    t    by trace
```

However, the debugger does not replace `subsequent` abbreviation directives. Thus, you can use abbreviations only to abbreviate the names of the debugger requests.

See the *Introduction to VOS (R001)* for more information about using abbreviations.

Issuing VOS Internal Commands

While your process is in the debugger, you can issue internal commands as if you were at command level. To issue an internal command from debugger request level, type the name of the command preceded by two periods.

The following example shows how to specify the internal command `help` with the `-type` argument to list all VOS internal commands.

```
db? ..help -type internal
```

You can use abbreviations for internal commands if you specify the abbreviation in your `abbreviations` file. For example, you can abbreviate the command `..list` to `..l`.

Using the Multiprocess Debugger

This section describes the multiprocess debugger.

As mentioned earlier in this chapter, the multiprocess debugger allows you to debug one or more processes simultaneously, from a single terminal. It is also useful for debugging processes that were started noninteractively.

The multiprocess debugger is used on a debug process set. A *debug process set* is a group of processes that you define with `mp_debug` requests. Once you include a process in the debug process set, you can debug it using the `debug` requests described earlier in this chapter.

You invoke the multiprocess debugger with the `mp_debug` command. This command has no arguments.

Table 6-3 briefly describes each multiprocess debugger request. See the *VOS Symbolic Debugger User's Guide (R308)* for more information about the multiprocess debugger.

Table 6-3. Multiprocess Debugger Requests

Request	Description
`exclude_process`	Removes one or more processes from the debug process set
`help`	Lists all `mp_debug` requests
`include_process`	Includes executing processes in the debug process set
`list_processes`	Lists all processes currently in the debug process set
`mp_login`	Creates and displays a new login process and waits for that process to complete before `mp_debug` continues. Note that `mp_login` performs this behavior only on window terminal devices. If you issue `mp_login` on a device that is not a window terminal device, the request behaves in the same manner as the `..login` command.
`quit`	Removes all processes from the debug process set, terminates the multiprocess debugging session, and returns you to VOS command level
`restart`	Resumes execution of suspended processes
`start_process`	Creates a process, starts it, and immediately suspends it, bringing it into the debug process set
`stop`	Removes all processes from the debug process set
`suspend_process`	Suspends activity of the specified process and enters the debugger
`use_process`	Activates the specified process in the debug process set, allowing you to issue debugger requests to the active process

Figure 6-16 demonstrates some of the `mp_debug` requests in a sample debugging session.

```
1.  batch server
    ready  10:15:41
2.  mp_debug
3.  mp_debug:  include_process server -module %hr#d20
    Including process 'server' with number 1
4.  mp_debug:  start_process requester
    Including process 'requester' with number 2
    Waiting...
    2: Entering debug.
    2: New language is pl1.
    mp_debug:  mp_debug:  1: Entering debug.
    1: New language is machine.
5.  mp_debug:  list_processes
     1  S     Mary_Doe.HR server server.pm
     2 *S     Mary_Doe.HR requester requester.pm
6.  mp_debug:  suspend_process 1
7.  mp_debug:  mp_debug:  use_process 2
    mp_debug:  position 1
    2: The code address is set to line 62 (the start of the statement).
8.  mp_debug:  exclude_process 1
    mp_debug:  list_processes
     2 *S     Mary_Doe.HR requester requester.pm
9.  mp_debug:  exclude_process -all
    mp_debug:  list_processes
    No processes included.
10. mp_debug:  quit
```

Figure 6-16. Sample Debugging Session Using `mp_debug`

In the following explanation, the numbers in the left margin correspond to the numbers in Figure 6-16.

1. From the command line, the `batch` command starts the `server` program as a background process. For more information about the `batch` command, see the *VOS Commands Reference Manual (R098)*.
2. The `mp_debug` command invokes the multiprocess debugger.
3. The `include_process` request includes all processes named `server` started by Mary Doe that are running on module `%hr#d20`. If you specify the `include_process` request with no arguments, **all** processes started by Mary Doe will be included. Note that the multiprocess debugger assigns `server` a process number of 1.
4. The `start_process` request starts the `requester` program and assigns it a process number of 2.
5. The `list_processes` request lists all processes currently in the debug process set. The `list_processes` request displays the following information.
    - the process number
    - the state of the process. A process can be in one of the following states.
        - running, which is designated by the letter `R`
        - active, which is designated by the character `*`

    - suspended, which is designated by the letter `S`
    - dead, which is designated by the letter `D`

- the user name of the person who started the process
- the name of the process
- the name of the program module being executed within the process, if one exists
- the name of the module running the process. If the process is a slave process on the current module, the name does not appear.

6. The `suspend_process` request suspends the `server` program and enters debugging level.

7. The `use_process` request instructs `mp_debug` to use the `requester` program. This means that you can issue any of the debugger requests described in this chapter on the `requester` program. In this case, the `position` request was issued.

8. The `exclude_process` request removes the `server` program from the debug process set.

9. The `exclude_process -all` request removes all processes from the debug process set. In this case, only the `requester` program was excluded, since no other processes remained in the process set.

10. The `quit` request exits you from `mp_debug` and returns you to the command line.

For more information about `mp_debug`, see the *VOS Symbolic Debugger User's Guide (R308)*.

Chapter 7:
VOS PL/I File I/O

You can use VOS PL/I statements to perform certain input and output tasks in the VOS environment. This chapter discusses the following topics related to features of the I/O statements that are unique to the VOS environment.

- ‘‘The VOS I/O System’’
- ‘‘I/O Types’’
- ‘‘VOS Files and File Organizations’’
- ‘‘PL/I File I/O Types’’
- ‘‘I/O Ports’’
- ‘‘Locking Modes’’

In addition, ‘‘File I/O Sample Program’’ at the end of the chapter contains a sample program that illustrates how to perform I/O using VOS PL/I statements.

Using VOS PL/I I/O statements, you can create files, open existing files, perform file and record locking, read and write data, and perform other I/O-related tasks. See the *VOS PL/I Language Manual (R009)* for detailed information about VOS PL/I I/O statements. See the *VOS Reference Manual (R002)* for more information about the VOS file system.

You can also use VOS subroutines to perform I/O. For information about using VOS subroutines for I/O-related tasks, see the *VOS PL/I Subroutines Manual (R005)*.

The VOS I/O System

This section describes the VOS I/O system.

The *VOS I/O system* is the hardware and software that moves data between a processing module’s central processors or primary storage, on one hand, and the module’s peripheral I/O devices or secondary storage, or the network, on the other hand. The I/O system software is constructed so that, from the user’s viewpoint, reading and writing to a disk or tape file appear to be the same operations as reading and writing to an I/O device, such as a terminal.

The I/O system manages several kinds of data structures and I/O devices. Some important kinds of data structures are files, I/O ports, and I/O devices (such as terminals, printers, and tape drives).

I/O Types

This section discusses the I/O types and how you use them.

When a file is opened with the `open` statement, the VOS I/O system requires an *I/O type* to be specified that describes the type of operations to be allowed on the file during this opening. I/O type is not a fixed file characteristic like file organization. A file's I/O type can be changed, for example, from one `open` statement to the next.

The operating system supports the following I/O types.

- append
- dirty input
- input
- output
- truncate
- update

Table 7-1 lists each I/O type and the `open` statement option associated with each type, and provides a general description of the operations allowed with each I/O type.

Table 7-1. I/O Types

I/O Type	Option	Description
Append	`-append`	Opens a file for writing at end-of-file. If the file does not exist, it is created. You can specify `-append` only if the file is opened for keyed sequential output or sequential output. (See ‘‘PL/I File I/O Types’’ later in this chapter for more information about keyed sequential output and sequential output.)
Dirty input	`-dirtyinput`	Opens a file for dirty input. With `-dirtyinput`, normal locking rules are ignored. The program can read from the file even though another program might be modifying the file. See ‘‘Locking Modes’’ later in this chapter for more information about file locking.
Input	`input`	Opens a file for reading only. This is the default I/O type.
Output	`output`	Opens a file for writing only.
Truncate	`-truncate`	Truncates a file before opening it for output.
Update	`update`	Opens a file so that records can be read, written, rewritten, or deleted.

Note: If you specify neither `-append` nor `-truncate` while opening an existing file for output, the file is deleted and a new file is created.

See the *VOS PL/I Language Manual (R009)* for detailed information about each I/O type, including any restrictions that may apply to the use of the I/O type with specific file organizations.

VOS Files and File Organizations

This section describes the four types of file organization and includes examples of how you use each type. It discusses the following topics.

- ‘‘Fixed File Organization”
- ‘‘Relative File Organization”
- ‘‘Sequential File Organization”
- ‘‘Stream File Organization”
- ‘‘File Organization Examples”

A *file* is the fundamental I/O system object. It is the logical unit of storage on a disk pack. A file is a sequence of bytes or a set of records stored as a unit on a disk or tape. In PL/I, a physical device such as a terminal is also considered to be a file.

When a VOS file is created, you can specify the file’s organization. A file’s *organization* is the manner in which data in the file is stored.

A VOS file has one of the following types of file organization.

- fixed
- relative
- sequential
- stream

When a file is created with the `open` statement, the VOS I/O system, as an option, allows you to specify a file organization. Table 7-2 summarizes each file organization.

Table 7-2. File Organizations *(Page 1 of 2)*

File Organization	**Option**	**Description**
Fixed	`-fixed [size]`	Creates and opens a fixed file if no file of the specified name exists. You optionally specify the file’s record size in *`size`*. The default value of *`size`* is `1024`.
Relative	`-relative [size]`	Creates and opens a relative file if no file of the specified name exists. You optionally specify the file’s maximum record size in *`size`*. The default value of *`size`* is `1024`. This is the default file organization for files that are open for direct access.

Table 7-2. File Organizations *(Page 2 of 2)*

File Organization	Option	Description
Sequential	`-sequential`	Creates and opens a sequential file if no file of the specified name exists. This is the default file organization for files that are **not** open for direct access.
Stream	`-stream`	Creates and opens a stream file if no file of the specified name exists.

You access a file by opening it with the `open` statement and then using other I/O statements to read data from or write data to the file. A file's contents can be accessed either as a sequence of one or more bytes (stream I/O) or as a record (record I/O). See ‘‘PL/I File I/O Types” later in this chapter for more information about stream I/O and record I/O.

A record can vary in length as follows:

- With fixed file organization, a record's length must correspond to the file's record size.
- With relative file organization, a record's length can vary from 0 to the number of bytes indicated by the file's maximum record size.
- With sequential or stream file organization, a record's length can vary from 0 to 32,767 bytes.

A fixed or relative file's record size is specified when the file is created.

Fixed File Organization

In a file with *fixed file organization*, records are stored in fixed-length disk or tape regions. A record in a file with fixed file organization contains only data. No record-length information is stored with the records.

Figure 7-1 illustrates fixed file organization. A fixed file is designed to provide the fastest average access to any record and, at the same time, to use disk or tape space most efficiently. Some system files, such as program modules (`.pm` files), have this organization.

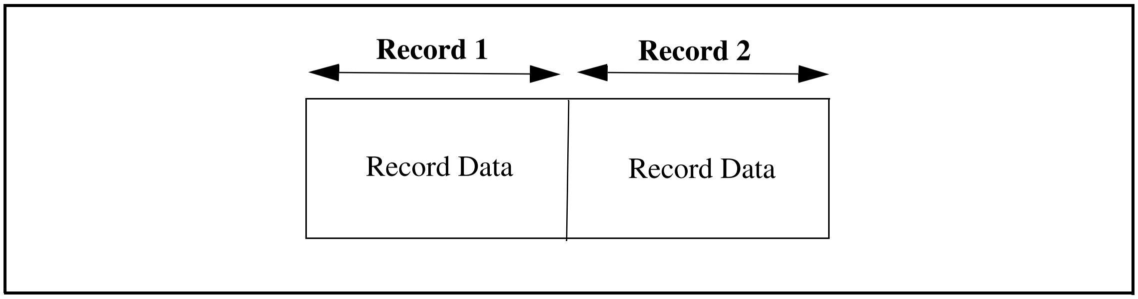


Figure 7-1. Fixed File Organization

With fixed file organization, the fixed length of a record is the *record size* for the file. The record data **must** be equal to the record size or a file I/O error occurs when you attempt to write the record.

In a fixed file, a record must start on an even-numbered byte boundary. Thus, an odd-length record has a pad character after the record data.

Relative File Organization

Relative file organization has characteristics of both sequential file organization and fixed file organization. In a file with *relative file organization*, the records can have various lengths, like those in a sequential file, but the records are stored on the disk or tape in fixed-length regions, like those in a fixed file. In a relative file, a record begins with a 2-byte record length. The physical record size equals the maximum record size plus two bytes containing the record length.

Figure 7-2 illustrates relative file organization.

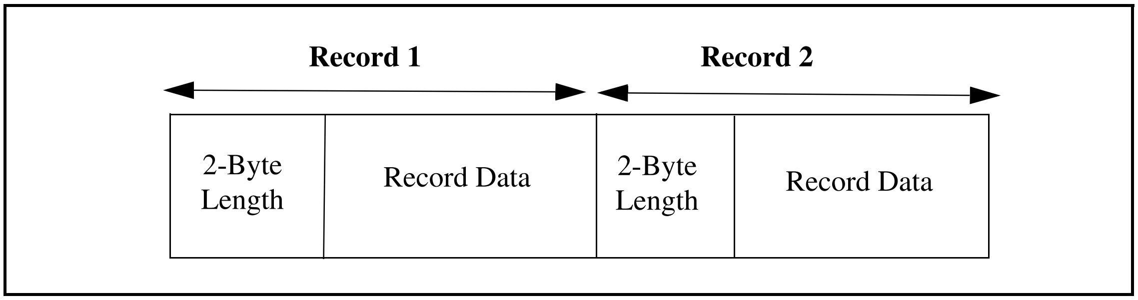


Figure 7-2. Relative File Organization

With relative file organization, the fixed length of a record is the *maximum record size* for the file. The record data can vary in length from 0 bytes to the maximum record size.

In a relative file, a record must start on an even-numbered byte boundary. Thus, an odd-length record has a pad character after the record data.

Sequential File Organization

In a file with *sequential file organization*, each record begins and ends with a 2-byte record length. The physical record size for a sequential file equals the number of bytes containing the record data plus four bytes containing the record length.

Figure 7-3 illustrates sequential file organization. With sequential file organization, records can be stored on the disk or tape with little unused space. Sequential file organization is the default for most VOS commands that create files, such as `emacs` or `create_file`. If you create a file using the `open` statement and do not specify a file organization, sequential file organization is the default, **unless** you are opening a file for direct access. (Relative file organization is the default for files opened for direct access.)

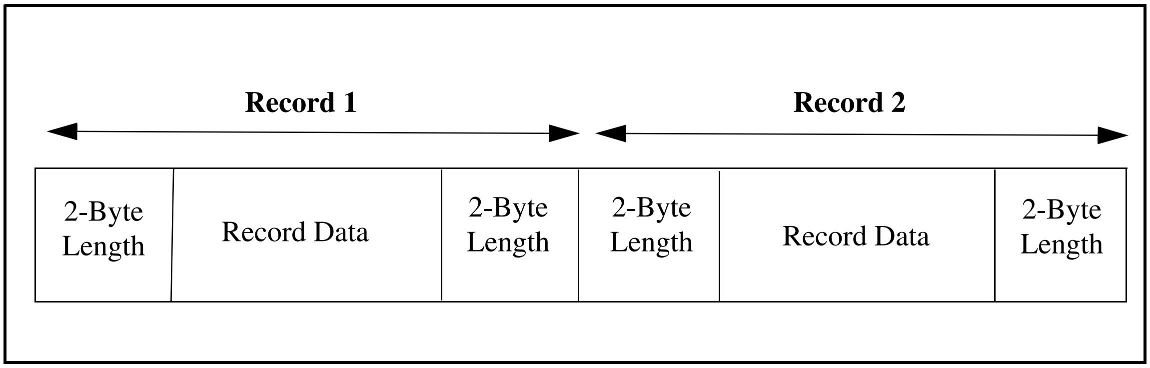


Figure 7-3. Sequential File Organization

The 2-byte length at the beginning of a record in a sequential file always starts on an even-numbered byte boundary. Thus, odd-length records have a pad character after the record data and before the 2-byte length at the end of the record.

Stream File Organization

In a file with *stream file organization*, each record is usually delimited by an ASCII line-feed character (`0A` hexadecimal) in the last byte of each record to mark the end of the record. Bytes after the last line-feed character in a file are also treated as a record, as is a sequence of 32,767 bytes (the maximum record length) without a line-feed character. A record in a stream file begins where the immediately preceding record ends.

Figure 7-4 illustrates stream file organization. With stream file organization, records can be stored on the disk or tape with little or no unused space.

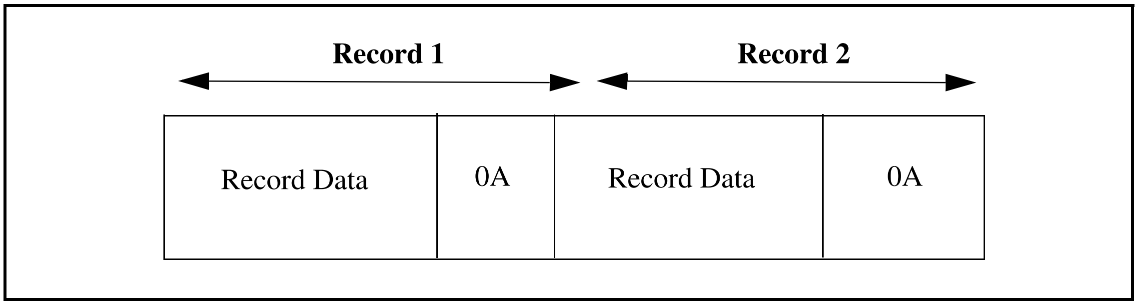


Figure 7-4. Stream File Organization

With stream file organization, the operating system stores the data as a sequence of bytes. You can choose to interpret a line-feed character as the end of a record, or you can choose some other method to determine how data is organized within the file.

File Organization Examples

The examples in the next four sections use variations of the program shown in Figure 7-5 to write data to files with fixed, relative, sequential, and stream file organizations, respectively. In each example, only the file organization of `out_file` changes.

```
file_org:
     procedure options(main);

declare chars_written      char(5);
declare out_file           file;

     open file(out_file) title('out_file') update;

     chars_written = 'abcde';
     write file(out_file) from(chars_written);

     chars_written = 'fghij';
     write file(out_file) from(chars_written);

     close file(out_file);

end file_org;
```

Figure 7-5. File Organization Sample Program

In Figure 7-5, the data that is written to the file contains the following two records.

```
abcde
fghij
```

Fixed File Organization Example

If, using the sample program in Figure 7-5, `out_file` were opened using the following statement, the data would be stored in `out_file` as shown in Figure 7-6. (Note that `-fixed 5` indicates a fixed record length of five bytes.)

```
open file(out_file) title('out_file -fixed 5') update;
```

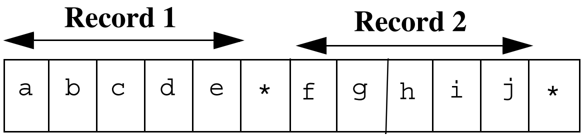


Figure 7-6. Fixed File Organization Example

In Figure 7-6, the `*` character represents unused bytes, filled with pad characters, in each record. In the physical record, these unused bytes actually contain `FF` hexadecimal.

In the example, since the record length is five bytes and the record must end on an even-numbered byte boundary, a single pad character is added after each record. (Note that the pad character is not a part of the record, however.)

The created file has the following contents.

```
abcde
fghij
```

If you attempt (for example, with `write`) to output fewer characters than are specified by the record size, an output error occurs.

Relative File Organization Example

If, using the sample program in Figure 7-5, `out_file` were opened using the following statement, the data would be stored in `out_file` as shown in Figure 7-7. (Note that `-relative 5` indicates a fixed record length of five bytes.)

```
open file(out_file) title('out_file -relative 5') update;
```

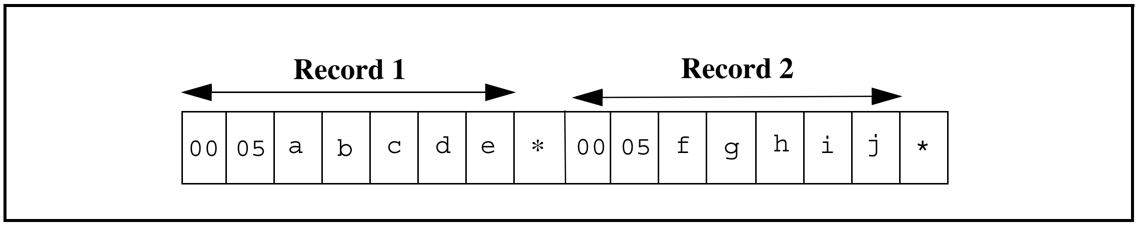


Figure 7-7. Relative File Organization Example

In Figure 7-7, the `*` character represents unused bytes, filled with pad characters, in each record. In the physical record, these unused bytes actually contain `FF` hexadecimal.

In the example, since the record length is five bytes and the record must end on an even-numbered byte boundary, a single pad character is added after each record. (Note that the pad character is not a part of the record, however.) The record's length of five bytes is stored in the first two bytes of each record.

The created file has the following contents.

```
abcde
fghij
```

Sequential File Organization Example

If, using the sample program in Figure 7-5, `out_file` were opened with no file organization specified, or were opened using the following statement, the data would be stored in `out_file` as shown in Figure 7-8. (You need not specify `-sequential`, since sequential file organization is the default.)

```
open file(out_file) title('out_file -sequential') update;
```

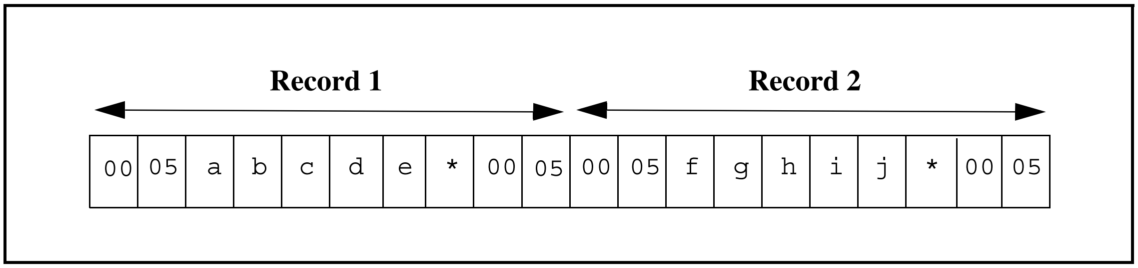


Figure 7-8. Sequential File Organization Example

In Figure 7-8, the `*` character represents unused bytes, filled with pad characters, in each record. In the physical record, these unused bytes actually contain `FF` hexadecimal.

In the example, since the record length is five bytes and the record must end on an even-numbered byte boundary, a single pad character is added to each record. The record's length of five bytes is stored in the first two bytes and the last two bytes of each record.

The created file has the following contents.

```
abcde
fghij
```

Stream File Organization Example

If, using the sample program in Figure 7-5, `out_file` were opened using the following statement, the data would be stored in `out_file` as shown in Figure 7-9.

```
open file(out_file) title('out_file -stream') update;
```

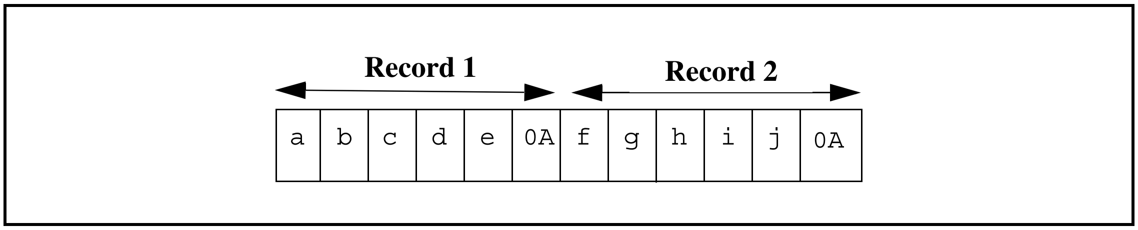


Figure 7-9. Stream File Organization Example

When the data is written to the stream file, the VOS file system inserts line-feed characters (`0A` hexadecimal) in the file.

The created file has the following contents.

```
abcde
fghij
```

In the preceding output, although you cannot see them, both records end with a line-feed character.

PL/I File I/O Types

This section discusses the following topics.

- ‘‘Stream I/O”
- ‘‘Record I/O”

In PL/I, data is written to or read from a file in two ways.

- using stream I/O
- using record I/O

Stream I/O

In *stream I/O*, sequences of characters, up to 256 characters long, are written to or read from a file, using the `put` and `get` statements in PL/I. Remember that in PL/I, a physical device, such as a terminal, is considered a file.

> **Note:** In VOS PL/I, you can also use limited forms of the `read` and `write` statements to operate on a stream file. See the discussion of stream I/O in the *VOS PL/I Language Manual (R009)* for more information.

The following example illustrates stream I/O.

```
put list ('Enter employee''s title.');
```

On input, the PL/I I/O routines scan the input stream and convert the data in the stream into the data type of the matching element in the data list of the `get` statement. On output, you provide a list of data to be written to the stream. The PL/I I/O routines convert these values to characters, and editing (such as leading-zero suppression) occurs. These data lists are called *I/O lists*.

Figure 7-10 illustrates the concept of stream I/O.

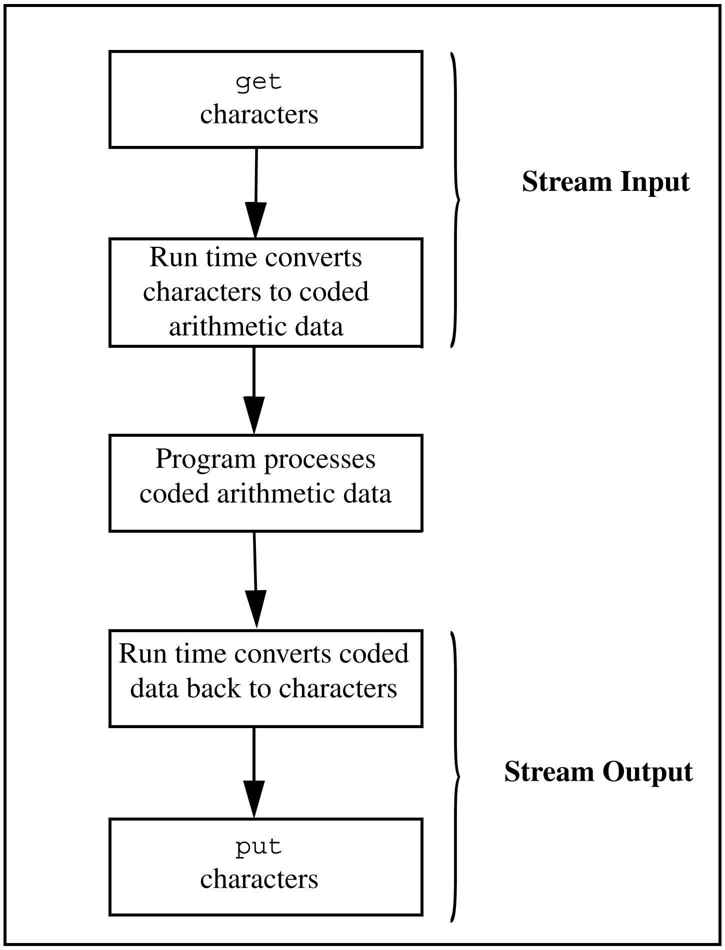


Figure 7-10. Stream I/O

A PL/I stream file does not require the VOS stream organization and, in fact, is not related to stream organization in any way. VOS sequential, fixed, or relative files can also be used as stream files.

Record I/O

In *record I/O*, data in a file is accessed one record at a time, using the `read`, `write`, `rewrite`, and `delete` statements.

The following examples illustrate record I/O.

```
read file(emp_file) into(employee_rec);
write file(emp_file) from(employee_rec);
```

Record I/O statements transmit the storage of a variable to or from a record in a file. The PL/I I/O routines do not perform any conversions and do not check data to ensure that it is of the proper type for storage in a particular variable.

Figure 7-11 illustrates the concept of record I/O.

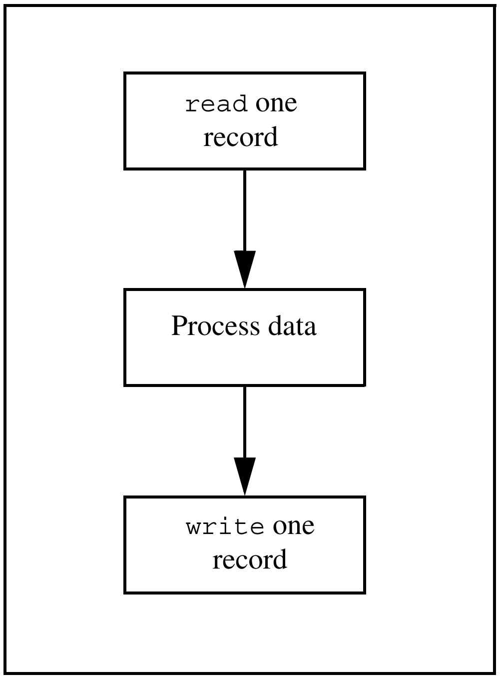


Figure 7-11. Record I/O

Two advantages to using record I/O instead of stream I/O follow.

- Since record I/O has relatively little processing to perform, it is usually faster and requires less storage space than stream I/O.
- Data in noncharacter formats, such as `fixed bin`, can be read or written using record I/O.

The following section discusses how to access records in a file.

Accessing Records

Using record I/O, you can access records in a file in three ways.

- sequentially
- using an index key
- directly by record number

Table 7-3 summarizes the operations you can perform on PL/I record files using the VOS PL/I access modes. The table also shows the VOS file organizations allowed for PL/I record files.

Table 7-3. Accessing Record Files

Access Mode Attribute	Operations Allowed	VOS File Organization
`direct`	`read`, `write`, `rewrite`, or `delete` statement; key is required	Relative or fixed
`keyed sequential`	`read`, `write`, `rewrite`, or `delete` statement; key is optional	Sequential, relative, or fixed (index is required or created)
`sequential`	`read` or `write` statement; key options are not allowed	Sequential, stream, relative, or fixed

The following sections describe the access types.

- ‘‘Direct Access”
- ‘‘Keyed Sequential Access”
- ‘‘Sequential Access”

Direct Access

If you open a file with the `direct` attribute, you access a record by specifying an integer key. An *integer key* is not an index; instead, it refers to the ordinal position of the record in the file. The current position of a direct file is meaningless; records must always be accessed by key values. Records in a direct file can be read, written, rewritten, or deleted.

Specifying the `direct` attribute always implies the `keyed` attribute, since keys are essential to direct access.

Direct access is generally more efficient than keyed sequential access.

See the *VOS PL/I Language Manual (R009)* for more information about direct access.

Keyed Sequential Access

If you open a file with both the `sequential` and `keyed` attributes, you access records either sequentially in index order or by specifying an index key. An *index key* is a file index value associated with a particular record. The file always has a current position. Records in a keyed sequential file can be read, written, rewritten, or deleted.

A keyed sequential file must have an index with character-string keys of up to 64 characters each. By default, a separate-key index named `primary` is used.

When you open a file with the `sequential` and `keyed` attributes for input or update, the current position is initially set to before the first record. The first I/O operation on the file must be a read or involve a `key` or `keyfrom` option to change the current position. If you specify a key value in a `read` statement, the record with that index key value is read and the current position is changed to that record.

The `write` statement, with or without a key value, appends records to the end of the file. If you specify a `write` statement with a non-null value in the `keyfrom` clause, the current position is set to the record being written.

If you specify a `rewrite` or `delete` statement with a non-null value in the `key` clause, the current position is reset to the rewritten or deleted record.

See the *VOS PL/I Language Manual (R009)* for more information about keyed sequential access.

Sequential Access

If you open a file with the `sequential` attribute and without the `keyed` attribute, you access records in the order in which they appear in the file. The file always has a current position. Records in such a file can be read or written, but cannot be rewritten or deleted.

When you open a file with the `sequential` attribute for input or update, the current position is initially set to before the first record. The first `read` statement reads the first record. The next `read` statement advances the current position and reads the second record, and so on.

The `write` statement appends records to the end of the file. It does not change the current position.

See the *VOS PL/I Language Manual (R009)* for more information about sequential access.

I/O Ports

This section discusses how you use I/O ports.

In the VOS environment, an *I/O port* is a data structure, identified by a name or ID, that you can attach to a file or device for the purpose of accessing the file or device. You use ports as follows:

- When you open a file with the `open` statement, the operating system creates a port and attaches the port to the specified file.
- When you close a file with the `close` statement, the operating system detaches and destroys the port.

When you open a file with the `open` statement, the operating system attaches a port to a VOS file or device. Once a port is attached, it remains attached until either you close the file or the program's process terminates (unless you specify that the port be held attached). If you specify a port in the `open` statement and the port is already attached, the statement accesses the file attached to the port and ignores the file path name specified in the statement.

By detaching the port from one file and attaching it to another, the actual source or destination of an I/O operation can be different from the file path name specified in the `open` statement. In this way, a single program can process many different files or read and write data on many different devices without having to be recompiled. You can use the `attach_port` command to attach a port to a file. Similarly, you can use the `detach_port` command to detach a port from a file. See the *VOS Commands Reference Manual (R098)* for information about these commands.

A *port ID* is a 2-byte integer that identifies a port. Port IDs are required for many VOS subroutine calls. For example, the `s$lock_region` and `s$unlock_region` subroutines require a port ID when they are called.

You cannot use PL/I I/O statements to return a port ID. Instead, you must use the VOS subroutine `s$get_port_id`. See the *VOS PL/I Subroutines Manual (R005)* for more information about `s$get_port_id`.

Locking Modes

This section describes how you use locking modes to lock files and records.

Before accessing data in a file, an application program can lock the data: either the entire file or an individual record in the file. When a program calls the `open` statement to open a file, the locking mode is determined for that file and the associated port. By carefully selecting the most appropriate locking mode, multiple programs can get a correct, consistent view of the file data and can access a file without long wait times.

Table 7-4 shows the locking-mode options allowed with the `open` statement, the I/O types you can use with each locking mode, and a description of each locking mode.

Table 7-4. Locking Modes

Locking Mode	Option	I/O Types	Description
Don't-set-lock	`-nolock`	`input`, `update`, `output -append`	The operating system does not lock the file. Before performing I/O on the file, you must lock it using VOS subroutines. See the *VOS PL/I Subroutines Manual (R005)* for more information.
		`output` without `-append`	Same as set-lock-don't-wait.
Implicit-locking	`-implicitlock`	All	Each time you perform I/O on the file, the operating system locks the file for the duration of that I/O and then releases it.
Record-locking	`-recordlock`	`update`	Each individual record is locked when accessed by a record I/O statement.
		`input`	No locking occurs. You can read any records in a file that another process has **not** opened for update in the record-locking mode.
Set-lock-don't-wait	`-nowait`	All	The operating system tries to lock the file. If it cannot, the `undefinedfile` condition is signaled with the oncode set to `e$file_in_use` (`1084`).
Wait-for-lock	`-wait`	`input`, `update`, `output -append`	The operating system tries to lock the file. If it cannot, the program is suspended until the lock becomes available.
		`output` without `-append`	Same as set-lock-don't-wait.

If you use the `-recordlock` option, you must call VOS subroutines to unlock records. Likewise, if you use the `-nolock` option, you must use VOS subroutines to lock and unlock the file. For more information about using subroutines to lock and unlock files or records, see the *VOS PL/I Subroutines Manual (R005).*

If you do not explicitly specify a locking mode in the `open` statement, set-lock-don't-wait mode is the default.

You can use the `who_locked` command or the `s$get_lockers` subroutine to identify the processes that have locked a file and the locking mode used.

The following restrictions apply to the choice of a locking mode when you open a file with `open`.

- If another process has opened a file using implicit-locking mode, you must also specify implicit-locking mode.
- If the implicit-locking attribute of a file has been set with the `set_implicit_locking` command or the `s$set_implicit_locking` subroutine, implicit-locking mode is used for the file regardless of the mode you specify.
- You cannot specify record-locking mode for a stream file or for a file that has been opened for the output or append I/O type.

The operating system's actions for a given locking mode depend on the file organization and I/O type that you specify when opening the file. See the description of the `s$open` subroutine in the *VOS PL/I Subroutines Manual (R005)* for information about these interdependencies.

For information about the commands discussed in this section, see the *VOS Commands Reference Manual (R098)*.

The following sections describe each of the VOS PL/I locking modes.

- ‘‘Explicit File Locking”
- ‘‘Implicit File Locking”
- ‘‘Record Locking”

Explicit File Locking

You can use the `open` statement to lock an entire file explicitly using two locking modes: set-lock-don't-wait and wait-for-lock. Both of these locking modes allow multiple readers **or** one writer at a time. Both modes are in effect from the time the file is opened until the file is closed.

Set-lock-don't-wait Mode

When you open a file using the set-lock-don't-wait mode, the `open` statement attempts to lock the file immediately. The `open` statement locks the file for reading if the I/O type is input, and for writing otherwise. When it cannot lock the file, `open` does not open the file and signals the `undefinedfile` condition with the oncode set to `e$file_in_use` (`1084`).

Wait-for-lock Mode

When you open a file using the wait-for-lock mode, the `open` statement attempts to lock the file immediately. The `open` statement locks the file for reading if the I/O type is input, and for writing otherwise. When it cannot lock the file, `open` waits until it can lock the file (theoretically forever, if necessary), unless the file is being opened for output. In this case, `open` does not open the file and signals the `undefinedfile` condition with the oncode set to `e$file_in_use` (`1084`).

Implicit File Locking

In implicit-locking mode, the entire file is locked, but only for the duration of each I/O operation. For example, if a program is reading data in a file, the file is locked for reading but only during that read operation. Implicit locking allows multiple readers and multiple writers to share the opened file. If you open a file using implicit-locking mode, any other user who attempts to open the file with a different locking mode receives an error message indicating that implicit locking must be used. Conversely, if you try to open a file for implicit locking when the file has already been opened by another process with some other locking mode specified, you receive an error.

If a file is opened for implicit locking, the operating system performs the following actions each time the program performs an I/O operation on the file.

1. locks the file
2. performs the read, write, or file-positioning operation
3. unlocks the file

You can use the `set_implicit_locking` command or the `s$set_implicit_locking` subroutine to set the implicit-locking attribute for a file. Once this attribute is set, a file can be opened only with implicit-locking mode. In this case, regardless of what locking mode is specified in the `open` statement, implicit-locking mode is always used for the file. When the implicit-locking attribute is set on a file and you attempt to open the file with a locking mode other than implicit locking, `open` overrides that locking specification with implicit-locking mode.

Record Locking

In record-locking mode, individual records in a file are locked, not the entire file. This locking mode gives you exclusive use of some records, while allowing other users to access other records in the file. You must unlock the records when you finish with them. With the `open` statement, you cannot specify record-locking mode when you are opening a file for output. Also, you cannot specify record-locking mode when you are opening a file with stream file organization.

When you have specified record-locking for a file, any record that is read or written is locked automatically when the I/O operation begins. To unlock the record, you must explicitly call a VOS subroutine such as `s$seq_unlock_record` or `s$unlock_records`. See the *VOS PL/I Subroutines Manual (R005)* for more information about `s$seq_unlock_record` and `s$unlock_records`.

File I/O Sample Program

The sample program in Figure 7-12 illustrates how you use VOS PL/I statements to do the following:

- open a relative file for direct access
- perform record locking
- use stream I/O to transmit data to and from the terminal
- use record I/O to read and write to a file

The program in Figure 7-12 allows a user to enter a book’s title, author, and price into a file. The user can then read this data from the file or can add more data.

```
   book_info:
        procedure options (main);

   /*  Declare variables.  */

   declare   selection      char(1) static initial (' ');

   declare   1  book,
                2  title    char(30),
                2  author   char(30),
                2  price    picture '$zzzv.99';

   declare   book_file      file;
   declare   number         fixed bin(31);
   declare   rec_number     fixed bin(31);

   /*  Specify error handlers.  */

1.      on undefinedfile (book_file)
             begin;
                  put skip list ('CANNOT OPEN FILE: ERROR ', oncode() );
                  stop;
             end;

2.      on key (book_file)
            begin;
   /*  If an invalid record number is specified, handle error.  */
                  if oncode() = 1269 | oncode() = 1128
                       then begin;
                            put skip list ('INVALID RECORD NUMBER');
                            go to choice_menu;
                       end;
   /*  If the specified record number already exists, handle error.  */
                  else if oncode() = 1275
                       then begin;
                            put skip list ('RECORD NUMBER ALREADY EXISTS');
                            go to choice_menu;
                       end;
                  else
                       begin;
                            put skip list ('ERROR CODE = ', oncode() );
                       end;
                  rec_number = 0;
             end;
```

(Continued on next page)

```
    /*  Open file for direct access.  */

3.       open file (book_file) title ('book_file -relative 68 -recordlock')
                                             record update direct keyed;

    /*  Display a menu asking user to choose an action: add, read, or quit. */
    /*  Call the appropriate procedure to perform the selected action.      */

         do while (selection ^= 'Q' & selection ^= 'q');
    choice_menu:
4.            put skip (1);
              put skip list ('Enter one of the following: ');
              put skip list ('    A = Add a new record');
              put skip list ('    R = Read an existing record');
              put skip list ('    Q = Quit the program');
              put skip list ('Your selection?');
              get edit (selection) (a);

              if (selection = 'A' | selection = 'a')
                   then do;
                        call create_rec_number;
                        call get_book_data;
                   end;
              else if (selection = 'R' | selection = 'r')
                   then call read_and_display;
              else if (selection ^= 'Q' & selection ^= 'q')
                   then put skip list ('INVALID RESPONSE.');
         end;

    /*  Close file and exit program.  */

5.       close file (book_file);
         stop;                          /*  End of main program  */


    /*  Create a new record number.  */

6.  create_rec_number:
         procedure;

         rec_number = 0;

         do while (rec_number = 0);
              put skip list ('Enter the new record number: ');
              get edit (number) (a);
              rec_number = number;
         end;

         return;
    end create_rec_number;
```

(Continued on next page)

```
   /* Accept data from user, then call write_records, which writes  */
   /* the data to book_file.                                         */

7. get_book_data:
        procedure;

        put skip list ('Enter the title of the book.');
        get skip edit (book.title) (a);
        put skip list ('Enter the author''s last name.');
        get skip edit (book.author) (a);
        put skip list ('Enter the price of the book.');
        get skip edit (book.price) (a);

        call write_records;

        return;
   end get_book_data;

   /*  Write the data to book_file.  */

8. write_records:
        procedure;

        write file (book_file) from (book) keyfrom (rec_number);

        return;
   end write_records;

   /*  Read the specified record number in book_file and  */
   /*  display the record information on the screen.      */

9. read_and_display:
        procedure;

        rec_number = 0;

        put skip list ('Enter an existing record number: ');
        get edit (number) (a);
        rec_number = number;
        read file (book_file) key (rec_number) into (book);

        put skip (1);
        put skip list ('Title:   ', book.title);
        put skip list ('Author:  ', book.author);
        put skip list ('Price:   ', book.price);

        return;
   end read_and_display;

   end book_info;
```

Figure 7-12. Sample Program

The following discussion focuses on those aspects of file I/O that relate to record locking, stream I/O, record I/O, and direct access. In the explanation, the numbers in the left margin correspond to the numbers in Figure 7-12.

1. This condition handler returns an error if the program cannot open the file.

2. This condition handler returns errors under the following conditions.

   - if the user specifies a record number for which no record exists
   - if the user attempts to assign an existing record number to a new record
   - if some other situation occurs in which the record number cannot be handled appropriately

3. The `open` statement opens a file, using the following options.

   - `file (book_file)` is the name of the file being opened. If the file does not already exist, the operating system creates one.

   - `title` specifies the path name of the file being opened, as well as certain file characteristics. The program assumes that `book_file` is located in the current directory, since only the file name is specified. The `-relative 68` option specifies that `book_file` is a relative file with a record length of 68 bytes. The `-recordlock` option specifies that `book_file` is being opened for record locking. See ‘‘Relative File Organization’’ and ‘‘Record Locking’’ earlier in this chapter for more information about relative files and record locking, respectively.

   - `record` indicates that `book_file` is being opened for record I/O. See ‘‘Record I/O’’ earlier in this chapter for more information about record I/O.

   - `update` indicates that `book_file` is being opened for update. See ‘‘I/O Types’’ earlier in this chapter for more information about the update I/O type.

   - `direct keyed` indicates that `book_file` is being opened for direct access, which means that you access a record by specifying an integer key. The `keyed` attribute is optional when you specify the `direct` attribute, but is specified here to improve clarity. See ‘‘Direct Access’’ earlier in this chapter for more information about direct access.

4. The `put` statement, which sends data to the terminal’s screen, is an example of stream I/O. See ‘‘Stream I/O’’ earlier in this chapter for more information about stream I/O.

5. The `close` statement closes `book_file`. It also detaches and destroys the port to which `book_file` was attached. See ‘‘I/O Ports’’ earlier in this chapter for more information about ports.

6. The `create_rec_number` procedure is called when the user wants to add a new record. The procedure assigns an integer value representing the new record number to `rec_number`.

7. The `get_book_data` procedure accepts data from the user when the user wants to add a new record. The procedure calls the `write_records` procedure to save the data in the file.

8. The `write_records` procedure writes the data entered by the user in the `get_book_data` procedure to `book_file`. The `write` statement specifies that the contents of the record `book` are to be written to `book_file` at the position specified in `rec_number`. See the *VOS PL/I Language Manual (R009)* for more information about the `write` statement.

9. The `read_and_display` procedure is called when the user wants to read an existing record. The procedure reads the record designated by `rec_number` from `book_file`, then displays that record on the terminal's screen. The `read` statement specifies that the record associated with `rec_number` is to be read from `book_file` and placed into `book`. See the *VOS PL/I Language Manual (R009)* for more information about the `read` statement.

Figure 7-13 shows sample output from the program shown in Figure 7-12.

```
Enter one of the following:
    A = Add a new record
    R = Read an existing record
    Q = Quit the program
Your selection? a

Enter the new record number:  2

Enter the title of the book. The Canterbury Tales

Enter the author's last name. Chaucer

Enter the price of the book. 10.99


Enter one of the following:
    A = Add a new record
    R = Read an existing record
    Q = Quit the program
Your selection? r

Enter an existing record number:  1


Title:    The Hobbit
Author:   Tolkein
Price:    $  3.95

Enter one of the following:
    A = Add a new record
    R = Read an existing record
    Q = Quit the program
Your selection? r

Enter an existing record number:  3

INVALID RECORD NUMBER

Enter one of the following:
    A = Add a new record
    R = Read an existing record
    Q = Quit the program
Your selection? q
```

Figure 7-13. Sample Output

Chapter 8:
Calling Subprograms Written in Other Languages

This chapter explains how a PL/I procedure can call a subprogram written in another language. Specifically, this chapter discusses the following topics.

For information about how a PL/I procedure calls another PL/I procedure, see the *VOS PL/I Language Manual (R009)*.

Overview

This section provides an overview of calling subprograms written in other programming languages.

In VOS PL/I, programs are usually written in a modular fashion. The program is composed of one or more separately compiled source modules, and each source module contains one or more procedures that perform related tasks. Although they are compiled separately, all source modules associated with a program must be bound with the main program. This produces a single executable program module.

You can bind a VOS PL/I object module with object modules written in any of the following languages.

- BASIC. Note, however, that VOS BASIC is available on XA2000-series modules and XA/R-series modules but **is not available** on Continuum-series modules.
- C. Note that the C functionality described in this chapter applies to the VOS Standard C compiler **and** the VOS C compiler.
- COBOL
- FORTRAN
- Pascal
- PL/I

Passing Arguments

You can pass an argument by reference or by value. This section defines these expressions.

When you pass an argument *by reference*, the argument's address is passed to the called subprogram. When the called subprogram modifies the corresponding parameter, it modifies the contents of the storage at that address; therefore, the variable in the calling procedure is also modified. All of the VOS languages except C normally pass arguments in this manner.

You can also pass arguments *by value*, as described in the following list.

- In C, arguments are passed by value. This means that the argument's value, rather than its address, is passed to the called function.
- In other VOS languages, if you need to pass an argument but do not want to change the value of that argument in the calling program, you can pass the argument by value. This means that the calling procedure makes a temporary copy of the argument and passes the address of this copy to the called subprogram. If the called subprogram modifies the argument, the corresponding argument in the calling procedure is not changed. The called subprogram modifies the **temporary** copy of the argument, not the original argument.

A PL/I program can use the `options(c)` attribute to pass certain types of arguments by value to a called C function. See ‘‘Calling VOS Standard C Subprograms” later in this chapter for more information about passing arguments to a C function.

See the *VOS PL/I Language Manual (R009)* for more information about passing arguments by reference or by value.

Declaring Subprograms

This section discusses general issues related to declaring subprograms written in other programming languages.

BASIC, COBOL, FORTRAN, Pascal, and PL/I distinguish between a subprogram that returns a value and a subprogram that does not return a value. For example, in PL/I, a *procedure* is a routine that does **not** return a value to the point where it was invoked, and a *function* is a routine that returns a value to the point of invocation.

Technically, all routines in the C language are functions. However, VOS Standard C functions that return a `void` value are handled like procedures and can be activated by VOS PL/I `call` statements.

If you want a subprogram to return a value, you must specify a `returns` attribute when you declare the subprogram in the calling PL/I procedure. See Figure 8-5 for an example. See the *VOS PL/I Language Manual (R009)* for more information about the `returns` attribute.

VOS PL/I does not allow for aggregate (array or structure) function results.

Using Compatible Data Types

This section discusses data-type compatibility between VOS PL/I and other programming languages. In particular, this section discusses the following topics.

- ‘‘Arrays”
- ‘‘Varying-Length Strings”
- ‘‘Nonconstant Extents”

Data can be accessed by and passed between a VOS PL/I procedure and subprograms written in other high-level languages to the extent that both languages define the particular data type. The VOS PL/I data types, in general, are compatible with the data types defined in each of the other VOS languages.

When a PL/I procedure calls a subprogram written in another VOS language, the arguments and return values passed between the PL/I procedure and the called subprogram must have compatibly defined data types.

When you declare a data item to have external scope, the item can be accessed by two programs written in different languages. For example, if you declare a data structure to have external scope in a PL/I procedure, you can access the data structure in a C function where a compatible data structure is declared with the same name and with the `extern` attribute.

Table 8-1 summarizes the type compatibility of the VOS PL/I data types with the data types in other VOS languages. Note that the variables *`i`* and *`f`* refer to the number of places to the left and right of the decimal point, respectively, in the numbers they define.

Table 8-1. Cross-Language Compatibility of Data Types

BASIC	C	COBOL	FORTRAN	Pascal	PL/I
`name=7`	`float`	`comp-1`	`real*4`	N/A	`float bin(24)`
`name=15`	`double, long double`	`comp-2`	`real*8`	`real`	`float bin(53)`
`name%=15`	`short`	`comp-4`	`integer*2,` `logical*2`	`-32768..32767`	`fixed bin(15)`
`name%=31`	`long, int, enum`	`comp-5`	`integer*4,` `logical*4`	`integer`	`fixed bin(31)`
`name=7`	`float`	`comp-1`	`real*4`	N/A	`float dec(7)`
`name=15`	`double, long double`	`comp-2`	`real*8`	`real`	`float dec(15)`
`name# = (i+f,f)`	N/A	`comp-6,binary,` `pic s9(i)v(f)`	N/A	N/A	`fixed dec(i+f,f)`
`name$=1`	`char`	`pic x display`	`character*1,` `logical*1`	`char`	`char(1)`
`name$=n`	`char id[n]`	`pic x(n) display`	`character*n`	`array [1..n] of char`	`char(n)`
`name$<=n`	`char_varying(n)`	`pic x(n) display-2`	`string*n`	`string (n)`	`char(n) varying`
N/A	N/A	N/A	N/A	`boolean`	`bit(1)`
N/A	N/A	N/A	N/A	`array[1..n] of boolean`	`bit(n)`
N/A	N/A	N/A	N/A	`set`	`bit aligned`
N/A	`(*id)()`	`entry`	`external`	`procedure,` `function`	`entry variable`
N/A	N/A	`label`	N/A	N/A	`label`
N/A	`type *id`	`pointer`	N/A	`^type_name`	`pointer`

Notes:

1. In all VOS high-level languages, data types must be aligned on the same type of boundary to be compatible.

2. Though the C data type `char id[n]` is allocated similarly to the data types shown in its row, it is essentially different from these data types. For example, unlike the other data types listed in this row, an array type in C, such as `char id[n]`, is treated as a pointer in almost all contexts.

3. In VOS PL/I, the `fixed bin(15)` type does **not** support the value -32,768.

4. The precision of decimal data is described in terms of *`i`* (the number of places to the left of the decimal point) and *`f`* (the number of places to the right of the decimal point).

5. There is no equivalent of the VOS COBOL data types `comp-3` (packed decimal) or `pic(9)` in the other high-level languages.

6. The C data type `char` and the PL/I data type `char(1)` are compatible for argument passing, but their function return values are incompatible.

Arrays

The PL/I language's array type is compatible with the array types in languages other than PL/I if the elements of the arrays have compatible types, as shown in Table 8-1. Elements of arrays must have the same alignment padding.

All of the VOS languages except FORTRAN store arrays in row-major order. This means that if array elements are accessed in the order in which they are stored, the rightmost subscript varies most frequently and the leftmost subscript varies least frequently. The following example illustrates.

```
declare c(25,4,2)     pointer;

     c(12,3,1) = null();
```

In the preceding example, the first three elements of the array `c` are `c(1,1,1)`, `c(1,1,2)`, and `c(1,2,1)`. The first element in the third row of the tenth set of `c` is `c(10,3,1)`.

FORTRAN stores arrays in column-major order, which means that if array elements are accessed in the order in which they are stored, the leftmost subscript varies most frequently and the rightmost subscript varies least frequently.

Varying-Length Strings

All VOS languages support strings that can have any length from zero up to a specified upper bound, called *varying-length strings*. Varying-length strings allow a subprogram to receive strings whose length can vary between calls.

You can declare a VOS PL/I procedure to have varying-length string parameters, using the declaration `char(`*`n`*`) varying`. See Table 8-1 for more information about how varying-length strings are declared in other VOS languages.

Nonconstant Extents

All VOS languages except for C and COBOL support strings and arrays that do not have a specified upper bound. Such strings and arrays are said to have *nonconstant extents* (also called *asterisk extents*).

You can declare a VOS PL/I procedure to have string parameters with nonconstant extents, using the declaration `char(*)`. You can also declare VOS PL/I procedures to have arrays with nonconstant extents, using the declaration `(*)` (for example, `a(*) fixed bin(15)`).

See the *VOS PL/I Language Manual (R009)* for more information about using nonconstant extents in VOS PL/I.

Calling VOS BASIC Subprograms

This section explains how you call a VOS BASIC subprogram from a VOS PL/I procedure.

You can call a VOS BASIC subprogram from within a VOS PL/I procedure as long as the data types of the arguments and their corresponding parameters are compatible, as shown in Table 8-1. VOS BASIC does not support external functions.

VOS BASIC passes arguments by value or by reference. It passes arguments that are variables or entire arrays by reference. This means that if the value of the argument is changed within the subprogram, the argument is also changed in the calling program. However, note that VOS BASIC passes arguments that are expressions (such as constants and array elements) by value. This means that the original argument remains unchanged in the calling program, even if the value of the argument changes within the subprogram.

VOS BASIC I/O files use only embedded-key indexes.

As mentioned in ‘‘Overview” earlier in this chapter, VOS BASIC is available on XA2000-series modules and XA/R-series modules but is **not** available on Continuum-series modules.

Calling a BASIC Subprogram

The example in Figure 8-1 shows how to call a BASIC subprogram from within a PL/I procedure.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare basic_proc  external entry(char(10) varying, float bin(53));

declare name        char(10) varying;
declare num         float bin(53);

    name = 'Jones';
    num = 888.8;

    call basic_proc(name, num);

    put skip edit ('name = ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

BASIC Subprogram:

```
sub basic_proc (str$<=10, num=15)

    str$ = 'Smith'
    num = 999.99

subend
```

Output:

```
name = Smith, and num =  999.99
```

Figure 8-1. Calling a BASIC Subprogram

Calling VOS Standard C Subprograms

This section explains how you call a VOS Standard C subprogram from a VOS PL/I procedure.

As discussed in ‘‘Passing Arguments” earlier in this chapter, PL/I normally passes arguments by reference; that is, the address of each argument is passed to the called subprogram. Typically, C functions pass arguments by value, rather than by reference.

If a PL/I procedure passes a C function an address, many possible errors can occur. One possible error message follows.

```
Attempt to modify protected page.
Referencing address 99999999x.
Error occurred in procedure procedure_name, line NUM.
Command was command_name.pm.
```

In the preceding error message, note that *`99999999`*x, *`procedure_name`*, *`NUM`*, and *`command_name`* are terms that the operating system would replace with literal values.

You can prevent errors like the preceding error by using one of the following methods.

- Use the PL/I `options(c)` attribute in the declaration of the entry point to the C function. Figure 8-3 demonstrates this method.
- Write the C function to expect pointers, rather than actual values, as arguments. Figure 8-4 demonstrates this method.

When you specify the `options(c)` attribute for an entry point, all integer, floating-point, varying-length string, and structure arguments are passed by value (that is, the way that C arguments are passed); array and nonvarying string arguments are passed by reference. Furthermore, the compiler converts any single-precision floating-point values to double-precision floating-point values before passing the value to a called C function. Therefore, a PL/I procedure always passes double-precision floating-point data to a C function.

The C `NULL` pointer constant, which is equal to the value 0, is **not** equivalent to the value returned by the VOS PL/I `null` built-in function. This function returns the value `OS_NULL_PTR`, which is equal to the value 1. For compatibility between the two languages, use the VOS Standard C constant `OS_NULL_PTR`, which is equal to the value 1.

The following PL/I code fragment tests whether the null pointer `p` is a PL/I null pointer or a C null pointer. (Note that the use of relational operators other than `=` and `^=` to compare pointer values is a VOS PL/I extension.)

```
if p <= null()
then
.
.
.
```

See the *VOS PL/I Language Manual (R009)* and the *VOS Standard C Reference Manual (R363)* for more information about null pointers.

You can pass an array or structure to a VOS Standard C function only if all members are of the compatible types listed in Table 8-1, and no nonconstant extents are used.

Structures are padded differently in VOS Standard C than in VOS PL/I. In VOS PL/I, the size of a structure is equal to the sum of bytes allocated for all of the structure's members. In VOS Standard C, the size of a structure is equal to the sum of bytes allocated for all of the structure's members **plus** any additional bytes needed for alignment padding. A VOS Standard C structure's size is rounded up so that the structure ends on a boundary that is a multiple of its alignment requirement.

Figure 8-2 illustrates the data-alignment differences between VOS PL/I structures and VOS Standard C structures.

```
declare 1 t     longmap,
          2 i   fixed bin(31),
          2 s   fixed bin(15);
```

PL/I Structure

```
struct  $longmap t
     {
     int i;
     short s;
     } t;
```

C Structure

Figure 8-2. Data-Alignment Differences between VOS PL/I and VOS Standard C Structures

In Figure 8-2, the PL/I structure is six bytes long.

- Four bytes are allocated for `i`.
- Two bytes are allocated for `s`.

The C structure, meanwhile, is eight bytes long.

- Four bytes are allocated for `i`.
- Two bytes are allocated for `s`.
- Since the structure begins on a mod4 boundary, two additional bytes are allocated to pad the structure so that it ends on a mod4 boundary.

Be aware of this difference in data alignment when passing structures to a VOS Standard C function. See the *VOS PL/I Language Manual (R009)* for more information about data alignment in VOS PL/I. See the *VOS C Language Manual (R040)* for more information about data alignment in VOS Standard C.

The next three sections discuss the following topics.

- ‘‘Passing Values to a C Function”
- ‘‘Passing Addresses to a C Function”
- ‘‘Calling a C Function That Returns a Value”

Passing Values to a C Function

Figure 8-3 shows how to call a C function from within a PL/I procedure. In this example, the PL/I calling procedure uses the `options(c)` attribute in the declaration of `c_func1` to pass arguments by value to the C function.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare c_func1  external entry(char(10) varying, float bin(53))
                                                               options(c);

declare name     char(10) varying;
declare num      float bin(53);

    name = 'Jones';
    num = 888.8;

    call c_func1(name, num);
    put skip edit ('name = ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

C Function:

```
#include <stdio.h>

void c_func1 (char_varying(10) name, double num)

{
     name = "Smith";
     num = 999.99;

     printf ("name = %v, and num = %7.2lf\n", &name, num);

     return;
}
```

Output:

```
name = Smith, and num = 999.99
name = Jones, and num =  888.80
```

Figure 8-3. Passing Values to a C Function

As shown in Figure 8-3, `name` and `num` are not modified in `pl1_procedure`. Since the `options(c)` attribute has been specified in the declaration of `c_func1`, in most cases, the **values** of the arguments, not the addresses of the arguments, are passed to `c_func1`. (The exceptions are array and nonvarying string arguments, which are passed by reference.)

Passing Addresses to a C Function

Like Figure 8-3, Figure 8-4 shows how to call a C function from within a PL/I procedure. However, in Figure 8-4, since the PL/I procedure passes arguments by reference, the C function was written to expect pointers, rather than values.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare c_func2   external entry(char(10) varying, float bin(53));
declare name      char(10) varying;
declare num       float bin(53);

    name = 'Jones';
    num = 888.8;

    call c_func2(name, num);
    put skip edit ('name = ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

C Function:

```
#include <stdio.h>

void c_func2(char_varying(10) *ptr_to_name, double *ptr_to_num)

{
     *ptr_to_name = "Smith";
     *ptr_to_num = 999.99;

     printf ("name = %v, and num = %7.2f\n", ptr_to_name, *ptr_to_num);

     return;
}
```

Output:

```
name = Smith, and num = 999.99

name = Smith, and num =  999.99
```

Figure 8-4. Passing Addresses to a C Subprogram

In Figure 8-4, `pl1_procedure` passes the arguments' addresses, rather than their values, to `c_func2`. To accept these arguments, `c_func2` declares the parameters as pointers.

When a PL/I procedure passes an address as an argument, any changes made by the called C function to the data at that address result in changes to the content of the argument passed. Thus, when the program in Figure 8-4 executes, the values of `name` and `num` are modified in `pl1_procedure`, not just in `c_func2`.

Calling a C Function That Returns a Value

Figure 8-5 shows how to call, from within a PL/I procedure, a C function that returns a value. Note that the PL/I calling procedure, `calc_distance`, uses the `options(c)` attribute when declaring `c_func3`.

PL/I Procedure:

```
calc_distance:
     procedure options(main);

declare c_func3    external entry(fixed bin(15), float bin(53))
                                   returns (float bin(53)) options(c);

declare mph        fixed bin(15);
declare time       float bin(53);
declare distance   float bin(53);

    mph = 35;
    time = .55;

    distance = c_func3(mph, time);
    put skip edit('distance = ', distance, ' miles')(a, f(7,2), a);

end calc_distance;
```

C Function:

```
double c_func3(short speed, double hours)

{
     double distance;


     distance = (speed * hours);
     return(distance);
}
```

Output:

```
distance =   19.25 miles
```

Figure 8-5. Calling a C Function

Calling VOS COBOL Subprograms

This section explains how you call a VOS COBOL subprogram from a VOS PL/I procedure. It discusses the following topics.

- ‘‘Calling a COBOL Procedure”
- ‘‘Calling a COBOL Function”

You can call a VOS COBOL procedure or function from within a VOS PL/I procedure as long as the data types of the arguments and their corresponding parameters are compatible.

VOS COBOL, like VOS PL/I, normally passes data by reference.

The following restrictions apply when passing arguments to a VOS COBOL subprogram.

- You can pass an array or structure to a VOS COBOL subprogram only if all elementary items or members are of the compatible types listed in Table 8-1.
- Any argument defined with the `aligned` attribute in PL/I must correspond to a parameter defined with the `synchronized left` phrase in COBOL.
- COBOL I/O files use only embedded-key indexes.

The default alignment padding for all types is identical in the two languages. Also, the allocation for structures containing noncontiguous elements is identical in PL/I and COBOL.

Calling a COBOL Procedure

Figure 8-6 shows how to call a COBOL procedure from within a PL/I procedure.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare cobol_proc  external entry(char(10) varying, float bin(53));

declare name        char(10) varying;
declare num         float bin(53);

    name = 'Jones';
    num = 888.8;

    call cobol_proc(name, num);

    put skip edit ('name =  ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

COBOL Procedure:

```
identification division.
program-id. cobol_proc.

data division.

working-storage section.
linkage section.

01 str display-2 pic x(10).
01 num comp-2.

procedure division using str num.

   move 'Smith' to str.
   move 999.99 to num.

   exit program.
```

Output:

```
name =  Smith, and num =  999.99
```

Figure 8-6. Calling a COBOL Procedure

Calling a COBOL Function

Figure 8-7 shows how to call a COBOL function from within a PL/I procedure.

PL/I Procedure:

```
calc_distance:
     procedure options(main);

declare cobol_func   external entry(fixed bin(15), float bin(53))
                                        returns (float bin(53));

declare mph           fixed bin(15);
declare time          float bin(53);
declare distance      float bin(53);

    mph = 35;
    time = .55;

    distance = cobol_func(mph, time);

    put skip edit('distance = ', distance, ' miles')(a, f(7,2), a);

end calc_distance;
```

COBOL Function:

```
identification division.
program-id. cobol_func.

data division.

working-storage section.

01 distance comp-2.

linkage section.

01 speed comp-4.
01 hours comp-2.

procedure division using speed hours giving distance.

   compute distance = (speed * hours).

   exit program.
```

Output:

```
distance =   19.25 miles
```

Figure 8-7. Calling a COBOL Function

Calling VOS FORTRAN Subprograms

This section explains how you call a VOS FORTRAN subprogram from a VOS PL/I procedure. It discusses the following topics.

- ‘‘Calling a FORTRAN Subroutine”
- ‘‘Calling a FORTRAN Function”

You can call a VOS FORTRAN subroutine or function from within a VOS PL/I procedure as long as the data types of the arguments and their corresponding parameters are compatible, as shown in Table 8-1.

Like PL/I, FORTRAN normally passes arguments by reference. However, note the following limitations.

- The dimensions of FORTRAN arrays are stated in the reverse of the order in which they are stated in PL/I. Therefore, when you declare an entry point in PL/I for a FORTRAN subprogram, you must reverse the subscripts of any array parameters.

- You cannot use a VOS FORTRAN assumed-size dummy array. To achieve the same effect, perform the following actions.
  - Declare a dummy array with dummy bounds.
  - Pass the first element of the PL/I array to the dummy array, and pass the bounds of the array to the dummy bounds (remember to reverse the order of the dimensions).

You can pass PL/I `entry` arguments to FORTRAN `external` parameters.

Calling a FORTRAN Subroutine

Figure 8-8 shows how to call a FORTRAN subroutine from within a PL/I procedure.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare fortran_proc external entry(char(10) varying, float bin(53));

declare name          char(10) varying;
declare num           float bin(53);

    name = 'Jones';
    num = 888.8;

    call fortran_proc(name, num);

    put skip edit ('name = ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

FORTRAN Subroutine:

```
      subroutine fortran_proc (str, num)

      string*10   str
      real*8      num

      str = 'Smith'
      num = 999.99

      end
```

Output:

```
name = Smith, and num =  999.99
```

Figure 8-8. Calling a FORTRAN Subroutine

Calling a FORTRAN Function

Figure 8-9 shows how to call a FORTRAN function from within a PL/I procedure.

PL/I Procedure:

```
calc_distance:
     procedure options(main);

declare fortran_func   external entry(fixed bin(15), float bin(53))
                                  returns (float bin(53));

declare mph             fixed bin(15);
declare time            float bin(53);
declare distance        float bin(53);

    mph = 35;
    time = .55;

    distance = fortran_func(mph, time);

    put skip edit ('distance = ', distance, ' miles')(a, f(7,2), a);

end calc_distance;
```

FORTRAN Function:

```
      real*8 function fortran_func (speed, hours)

      integer*2   speed
      real*8       hours

      fortran_func = (speed * hours)

      end
```

Output:

```
distance =   19.25 miles
```

Figure 8-9. Calling a FORTRAN Function

Calling VOS Pascal Subprograms

This section explains how you call a VOS Pascal subprogram from a VOS PL/I procedure. It discusses the following topics.

- ‘‘Calling a Pascal Procedure”
- ‘‘Calling a Pascal Function”

You can call a VOS Pascal procedure or function from within a VOS PL/I procedure as long as the data types of the arguments and their corresponding parameters are compatible, as shown in Table 8-1.

Both VOS PL/I and VOS Pascal procedures and functions use two kinds of parameters: variable parameters and value parameters. *Variable parameters* can be altered by the called procedure or function; *value parameters* cannot be altered. You should pass PL/I arguments by reference to Pascal variable parameters and by value to Pascal value parameters.

You can pass VOS PL/I `entry` arguments to Pascal `procedure` and `function` parameters.

In VOS PL/I, you can pass strings with nonconstant extents to VOS Pascal conformant arrays. For example, the following declarations are equivalent.

In VOS Pascal: `array [1..u : integer] of char`

In VOS PL/I: `char(*)`

See the *VOS PL/I Language Manual (R009)* for more information about nonconstant extents. See the *VOS Pascal Language Manual (R014)* for more information about conformant arrays.

Calling a Pascal Procedure

Figure 8-10 shows how to call a Pascal procedure from within a PL/I procedure.

PL/I Procedure:

```
pl1_procedure:
    procedure options(main);

declare pascal_proc external entry(char(*) varying, float bin(53));

declare name        char(10) varying;
declare num         float bin(53);

    name = 'Jones';
    num = 888.8;

    call pascal_proc(name, num);

    put skip edit ('name = ', name, ', and num = ', num)(a, a, a, f(7,2));

end pl1_procedure;
```

Pascal Procedure:

```
procedure pascal_proc (var name: string(*); num: real);

begin

     name := 'Smith';
     num := 999.99;

end.
```

Output:

```
name = Smith, and num =  888.80
```

Figure 8-10. Calling a Pascal Procedure

In `pascal_proc` (Figure 8-10), the first parameter, `name`, is declared as a variable parameter, and the second parameter, `num`, is declared as a value parameter. As a result, `pascal_proc` sets the variable `name` in `pl1_procedure` to `Smith`, but does **not** modify the value `999.99`. See ‘‘Passing Arguments’’ earlier in this chapter for more information about passing arguments by reference or by value.

In Figure 8-10, note that `name` (in the PL/I procedure) and `name` (in the Pascal procedure) are declared as varying-length strings with nonconstant extents.

Calling a Pascal Function

Figure 8-11 shows how to call a Pascal function from within a PL/I procedure.

PL/I Procedure:

```
calc_distance:
     procedure options(main);

declare pascal_func   external entry(fixed bin(15), float bin(53))
                                               returns (float bin(53));

declare mph            fixed bin(15);
declare time           float bin(53);
declare distance       float bin(53);

    mph = 35;
    time = .55;

    distance = pascal_func(mph, time);

    put skip edit('distance = ', distance, ' miles')(a, f(7,2), a);

end calc_distance;
```

Pascal Function:

```
type short = -32768..32767;

function pascal_func(speed: short; hours: real) : real;

begin

    pascal_func := speed * hours;

end.
```

Output:

```
distance =   19.25 miles
```

Figure 8-11. Calling a Pascal Function

Appendix A:
VOS Internal Character Code Set

Table A-1 lists the VOS internal character code set.

Table A-1. VOS Internal Character Code Set *(Page 1 of 10)*

Decimal Code	Hex Code	Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of Heading
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tabulation
10	0A	LF	Linefeed
11	0B	VT	Vertical Tabulation
12	0C	FF	Form Feed
13	0D	CR	Carriage Return
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3

Table A-1. VOS Internal Character Code Set *(Page 2 of 10)*

Decimal Code	Hex Code	Symbol	Name
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	EOT Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
32	20	SP	Space
33	21	!	Exclamation Mark
34	22	“	Quotation Marks
35	23	#	Number Sign
36	24	$	Dollar Sign
37	25	%	Percent Sign
38	26	&	Ampersand
39	27	’	Apostrophe
40	28	(	Opening Parenthesis
41	29	)	Closing Parenthesis
42	2A	*	Asterisk
43	2B	+	Plus Sign
44	2C	,	Comma
45	2D	-	Hyphen, Minus Sign
46	2E	.	Period
47	2F	/	Slant

Table A-1. VOS Internal Character Code Set *(Page 3 of 10)*

Decimal Code	**Hex Code**	**Symbol**	**Name**
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-Than Sign
61	3D	=	Equals Sign
62	3E	>	Greater-Than Sign
63	3F	?	Question Mark
64	40	@	Commercial ‘‘at’’ Sign
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K

Table A-1. VOS Internal Character Code Set *(Page 4 of 10)*

Decimal Code	Hex Code	Symbol	Name
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening Bracket
92	5C	\	Reverse Slant
93	5D	]	Closing Bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave Accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g

Table A-1. VOS Internal Character Code Set *(Page 5 of 10)*

Decimal Code	Hex Code	Symbol	Name
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening Brace
124	7C	|	Vertical Line
125	7D	}	Closing Brace
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	SS1	Single-Shift 1
129	81	SS4	Single-Shift 4
130	82	SS5	Single-Shift 5
131	83	SS6	Single-Shift 6

Table A-1. VOS Internal Character Code Set *(Page 6 of 10)*

Decimal Code	Hex Code	Symbol	Name
132	84	SS7	Single-Shift 7
133	85	SS8	Single-Shift 8
134	86	SS9	Single-Shift 9
135	87	SS10	Single-Shift 10
136	88	SS11	Single-Shift 11
137	89	SS12	Single-Shift 12
138	8A	SS13	Single-Shift 13
139	8B	SS14	Single-Shift 14
140	8C	SS15	Single-Shift 15
141	8D		(Not Assigned)
142	8E	SS2	Single-Shift 2
143	8F	SS3	Single-Shift 3
144	90	LSI	Locking-Shift Introducer
145	91	WPI	Word Processing Introducer
146	92	XCI	Extended-Control Introducer
147	93	BDI	Binary-Data Introducer
148	94		(Not Assigned)
149	95		(Not Assigned)
150	96		(Not Assigned)
151	97		(Not Assigned)
152	98		(Not Assigned)
153	99		(Not Assigned)
154	9A		(Not Assigned)
155	9B		(Not Assigned)
156	9C		(Not Assigned)
157	9D		(Not Assigned)
158	9E		(Not Assigned)
159	9F		(Not Assigned)

Table A-1. VOS Internal Character Code Set *(Page 7 of 10)*

Decimal Code	Hex Code	Symbol	Name
160	A0	NBSP	No Break Space
161	A1	¡	Inverted Exclamation Mark
162	A2	¢	Cent Sign
163	A3	£	British Pound Sign
164	A4	¤	Currency Sign
165	A5	¥	Yen Sign
166	A6	¦	Broken Bar
167	A7	§	Paragraph Sign
168	A8	¨	Dieresis
169	A9	©	Copyright Sign
170	AA	ª	Feminine Ordinal Indicator
171	AB	Ç	Left-Angle Quote Mark
172	AC	¨	‘‘Not’’ Sign
173	AD	SHY	Soft Hyphen
174	AE	Æ	Registered Trademark Sign
175	AF	Ø	Macron
176	B0	×°	Degree Sign, Ring Above
177	B1	±	Plus-Minus Sign
178	B2	^{2}	Superscript 2
179	B3	^{3}	Superscript 3
180	B4	´	Acute Accent
181	B5	µ	Micro Sign
182	B6	¶	Pilcrow Sign
183	B7	·	Middle Dot
184	B8	¸	Cedilla
185	B9	^{1}	Superscript 1
186	BA	º	Masculine Ordinal Indicator
187	BB	È	Right-Angle Quote Mark

Table A-1. VOS Internal Character Code Set *(Page 8 of 10)*

Decimal Code	Hex Code	Symbol	Name
188	BC	¼	One-Quarter
189	BD	½	One-Half
190	BE	¾	Three-Quarters
191	BF	¿	Inverted Question Mark
192	C0	À	A with Grave Accent
193	C1	Á	A with Acute Accent
194	C2	Â	A with Circumflex
195	C3	Ã	A with Tilde
196	C4	Ä	A with Dieresis
197	C5	Å	A with Ring Above
198	C6	Æ	Diphthong A with E
199	C7	Ç	C with Cedilla
200	C8	È	E with Grave Accent
201	C9	É	E with Acute Accent
202	CA	Ê	E with Circumflex
203	CB	Ë	E with Dieresis
204	CC	Ì	I with Grave Accent
205	CD	Í	I with Acute Accent
206	CE	Î	I with Circumflex
207	CF	Ï	I with Dieresis
208	D0	Đ	D with Stroke
209	D1	Ñ	N with Tilde
210	D2	Ò	O with Grave Accent
211	D3	Ó	O with Acute Accent
212	D4	Ô	O with Circumflex
213	D5	Õ	O with Tilde
214	D6	Ö	O with Dieresis
215	D7	×	Multiplication Sign

Table A-1. VOS Internal Character Code Set *(Page 9 of 10)*

Decimal Code	Hex Code	Symbol	Name
216	D8	Ø	O with Oblique Stroke
217	D9	Ù	U with Grave Accent
218	DA	Ú	U with Acute Accent
219	DB	Û	U with Circumflex
220	DC	Ü	U with Dieresis
221	DD	Ý	Y with Acute Accent
222	DE	Þ	Uppercase Thorn
223	DF	β	Sharp s
224	E0	à	a with Grave Accent
225	E1	á	a with Acute Accent
226	E2	â	a with Circumflex
227	E3	ã	a with Tilde
228	E4	ä	a with Dieresis
229	E5	å	a with Ring Above
230	E6	æ	Diphthong a with e
231	E7	ç	c with Cedilla
232	E8	è	e with Grave Accent
233	E9	é	e with Acute Accent
234	EA	ê	e with Circumflex
235	EB	ë	e with Dieresis
236	EC	ì	i with Grave Accent
237	ED	í	i with Acute Accent
238	EE	î	i with Circumflex
239	EF	ï	i with Dieresis
240	F0	ð	Lowercase Eth
241	F1	ñ	n with Tilde
242	F2	ò	o with Grave Accent
243	F3	ó	o with Acute Accent

Table A-1. VOS Internal Character Code Set *(Page 10 of 10)*

Decimal Code	Hex Code	Symbol	Name
244	F4	ô	o with Circumflex
245	F5	õ	o with Tilde
246	F6	ö	o with Dieresis
247	F7	÷	Division Sign
248	F8	ø	o with Oblique Stroke
249	F9	ù	u with Grave Accent
250	FA	ú	u with Acute Accent
251	FB	û	u with Circumflex
252	FC	ü	u with Dieresis
253	FD	ý	y with Acute Accent
254	FE	þ	Lowercase Thorn
255	FF	ÿ	y with Dieresis

Glossary

access

1. To read or write from a file or device.

2. The ability to read or write from a file or device.

address

The location of an area of storage. An address is a 4-byte value.

aligned data

Data that is allocated on a particular storage boundary.

allocate

To set aside an area of storage for a particular purpose.

American National Standards Institute (ANSI)

A group that promotes standards for computer languages and devices.

American Standard Code for Information Interchange (ASCII)

A standard 7-bit character representation code that the operating system stores in an 8-bit byte.

ANSI

See **American National Standards Institute**.

argument

1. A variable or value (or, more properly, the address of a variable or value) explicitly passed to a called procedure and corresponding to a parameter of that procedure.

2. A character string that specifies how a command is to be executed.

arithmetic data

Fixed-point, floating-point, and, in some contexts, pictured data.

array

A set of objects of the same data type. In an array, the individual objects, or elements, are stored contiguously at increasing addresses in memory.

ASCII

1. In the VOS internal character coding system, the half of the 8-bit code page with code values in the range `00-7F` (hexadecimal), representing the American Standard Code for Information Interchange.
2. See **American Standard Code for Information Interchange**.

assign

To associate a value with a variable.

asterisk extents

An asterisk, or a comma-list of asterisks, used as an extent expression in the description of a string or array parameter. A parameter with asterisk extents assumes the extents of the corresponding argument. Asterisk extents are also called *nonconstant extents*.

automatic storage

Storage that exists only for the duration of a block of code. An object with automatic storage is temporary. If you do not specify a storage class for a variable that does not appear in a parameter list, automatic is the default.

based storage

Storage that is not allocated unless and until an `allocate` statement specifically requests it. Only variables declared with the `based` attribute have the based storage class. Based data can only be accessed with pointer-qualifiers.

begin block

A block of a program initiated by a `begin` statement and terminated by an `end` statement. A begin block is activated when its `begin` statement is executed.

binary

Base 2; a base designation for arithmetic data.

binary file

A file in which a data item is stored in its internal form, as opposed to its character form. See also **text file**.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library. See also **library**.

bind map

A file, produced by the binder, containing information about a program module. A bind map is only produced if you specify the `-map` argument of the `bind` command.

bind time

The time at which the binder is invoked to combine one or more user object modules and VOS support routines into a program module.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the `bind` command.

binder control file

A text file containing directives for the binder.

binder-preprocessor statement

A statement used by the binder's preprocessor to perform conditional inclusion on an object module. An example is the `$define` statement. See also **PL/I preprocessor statement** and **VOS preprocessor statement**.

bit

The smallest unit of internal computer storage. A bit can have one of two values: `'1'b` or `'0'b`.

bit string

A value consisting of a linear sequence of zero or more bit values. Bit strings are read from left to right.

block

A procedure or begin block; a unit of a program that can be activated.

Boolean value

A bit string of length 1 indicating a truth value. The value `'1'b` means true, and the value `'0'b` means false.

bounds

Those parts of an array extent expression that indicate the maximum and minimum subscript value for each dimension.

break level

A state in which a process is suspended and the user can choose from a limited number of requests.

breakpoint

In a program being debugged, a statement or instruction at which the debugger stops execution to allow you to examine the state of the program. You set breakpoints with the debugger.

built-in function

A function that is predefined within the VOS PL/I language.

by reference

A method of passing arguments in which the called procedure receives the address of the actual storage of a program variable.

by value

A method of passing arguments in which the called procedure receives the address of a temporary storage location rather than the address of a variable's storage.

byte

A unit of storage consisting of eight contiguous bits.

call

To activate a program block, usually by means of a `call` statement.

character

A symbol, such as a letter of the alphabet or a numeral, or a control signal, such as a carriage return or a backspace. Characters are represented in electronic media by character codes.

character string

A sequence of zero or more characters or escape sequences enclosed in quotation marks (for example, `"bcd"`). Character strings are evaluated from left to right.

close

To disconnect from an operating system file or device. For example, the VOS PL/I `close` statement closes a file.

code

1. Machine instructions generated by the compiler.
2. The contents of a source module.
3. To write (in) a source module.

code region

The portion of a standard program module that contains the actual instruction sequences that represent the program.

command

A program invoked from command level, either interactively or as a statement in a command macro.

command macro

A text file having the suffix `.cm` and containing a series of commands to be executed. In addition to invocations of VOS commands and user programs, a command macro can contain command macro statements. The command processor reads and executes the lines from the macro.

comment

Documentary information included in source code that is ignored by the compiler; a comment has no effect on the execution of the program.

compile time

The time at which a compiler is invoked to translate a source module into an object module.

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

compound statement

A language statement, such as `if` or `on`, that contains one or more other language statements.

condition

A state resulting from an exceptional situation, such as division by zero or register overflow. When such a state occurs, a condition might be signaled. When a condition is signaled, an on-unit is executed.

condition handler

A routine that is invoked when a condition is signaled; an on-unit.

constant

A sequence of characters that represents a fixed numerical value. Every constant has an associated data type.

Continuum-series module

A Stratus module that contains a microprocessor from the Hewlett-Packard PA-7100 or PA-8000 precision-architecture reduced instruction set computing (PA-RISC) microprocessor family.

conversion

The process of transforming a value from one data type to another.

cross-binding

Occurs when a binder running on a processor from one processor family binds object modules into a program module that will execute on a processor from a different processor family. See also **cross-compilation**.

cross-compilation

Occurs when a compiler running on a processor from one processor family translates a source module into object code that will execute on a processor from a different processor family. See also **cross-binding**.

current directory

The directory currently associated with your process. The operating system uses your current directory as the default directory when you do not specifically name the directory containing an object that you want the operating system to find. For example, if you supply a relative path name in a command, the operating system uses the current directory as the reference point from which to locate the object in the directory hierarchy.

current position

The location in a file at which the next input or output operation will be performed.

data type

The collective attributes of a value, variable, or object that determine the scheme by which the data item is stored and the operations that can be performed on the data item.

debug

To correct errors (bugs) in a program.

debugger

A tool used to help find program errors.

declaration

A statement that introduces a name and defines its attributes.

default

The value or attribute used when a necessary value or attribute is omitted.

default value

The value that the operating system uses if a specific value is not supplied.

delimiter

One or more white-space characters that separate two adjacent tokens. A delimiter is not part of either of the tokens it delimits.

detach

To disassociate a port from a file or device.

device

Any hardware component that can be referenced and used by the system or users of the system and that is defined in the device configuration table. Terminals, printers, tape drives, and communications lines are devices.

dimension

A section of an array having a certain size, an integral upper bound, and an integral lower bound.

direct access

A method of referring to records by giving their ordinal position within a file. See also **keyed sequential access** and **sequential access**.

directory

A segment of disk storage containing files, links, and subdirectories and having its own access limitations.

element

1. A single object in an array of objects. In an array of structures, each structure is an element.
2. An identifier, statement, or part of a statement.

entry point

A location in a program where execution can begin when the program is activated. An entry point can be specified in a binder control file. In PL/I, each `procedure` and `entry` statement is an entry point.

entry value

The value of an entry point constant or the value assigned to an entry variable. An entry value consists of a display pointer, a code pointer, and a static pointer.

error code

An arithmetic value indicating what, if any, error has occurred. An error code is a 2-byte integer, often representing a VOS status code.

executable statement

A PL/I statement for which the compiler generates machine code.

execute

1. To process an executable statement at run time.
2. To run a program.

expression

A sequence of operators and operands that can be evaluated.

extents

The bounds specification of an array, or the length specification of a string value.

external scope

An identifier with external scope is visible across multiple source modules. Each instance of a particular identifier with external scope denotes the same object or function.

fence

A portion of a user's virtual address space adjacent to either the process or a task's user stack. A fence allows the operating system to detect most references beyond the end of the stack without any corruption of data in adjacent regions.

file

A sequence of bytes or a set of records stored as a unit on a disk or tape. In PL/I, a physical device such as a terminal is also considered to be a file.

file organization

The manner in which data in a VOS file is arranged. The operating system supports four file organizations: stream, sequential, fixed, and relative.

fixed organization

A VOS file organization in which data is stored in records of equal length. See also **relative organization**, **sequential organization**, and **stream organization**.

fixed-point number

An arithmetic value having a predetermined and inflexible number of digits to the right of the radix point.

floating-point number

An arithmetic value consisting of a mantissa and an exponent. In VOS PL/I, floating-point numbers are always written in base 10, but are stored as base-2 values.

full path name

For a file, directory, or link, a name that is composed of the name of the system, the name of the disk, the names of the directories that contain the object, and, finally, the name of the file, directory, or link.

For a device, a name that is composed of the name of the system and the name of the device.

function

A procedure that returns a single value. In PL/I, the `procedure` statement and each `entry` statement of a function must include a `returns` clause, and each `return` statement in the function must include a return value.

heap

A collection of randomly accessible storage associated with a process and available for allocation.

hexadecimal

Base 16; a base designation for arithmetic data.

I/O

Input and output.

identifier

A sequence of 1 to 32 letters, digits, underline characters (_), and dollar-sign characters (`$`). The first character of an identifier must be a letter. An identifier can be a keyword or a declared name.

implicit locking

A VOS locking mode in which the operating system does not lock the file or record for either reading or writing when it opens the file, but rather locks it for the appropriate access type each time a process performs an I/O operation on the file or record.

include file

A text file containing language statements, preprocessor statements, or both, that the compiler inserts into the source module in place of an `%include` PL/I preprocessor statement. PL/I include files have the suffix `.incl.pl1`.

include library

One or more directories that the operating system searches for include files.

index

1. An ordered list of keys associated with the records of a file.
2. A number used to specify an array element.
3. A variable used to control the execution of a do-group.

index key

A file index value associated with a particular record.

initialize

To assign a value to a data object upon its creation.

input file

A file open for reading only; the contents of an input file cannot be altered.

integer

A whole number; an arithmetic value with no fractional part.

internal scope

An identifier with internal scope is visible within only one source module. Within one source module, each instance of an identifier with internal scope denotes the same object or block.

keep module

A file that holds an executable image of a program so that you can debug the program at a later time. Keep modules have the suffix `.kp`.

key

An index key associated with a file record, or an integer indicating the ordinal position of a file record.

keyed file

A file on which an index has been created, or a file of records that can be accessed directly.

keyed sequential access

A method of referring to records in a file by their index-key value. If no key value is specified, records in a keyed sequential file are accessed in index order. See also **direct access** and **sequential access**.

keyword

1. A word that has special meaning to the VOS PL/I compiler. For example, keywords identify data types, storage classes, and statements. You cannot use keywords as identifiers.

2. For VOS commands or requests, an argument label that begins with a hyphen (`-`).

label

See **statement label**.

library

One or more directories in which the operating system looks for objects of a particular type. The operating system defines the following types of libraries.

- include libraries, in which the compilers search for include files
- object libraries, in which the binder searches for object modules
- command libraries, in which the command processor searches for commands
- message libraries, in which the operating system searches for message files associated with individual program modules (`.pm` files)

One of each of these libraries is available in the `>system` directory of each module for all processes running on the module. You can also define your own libraries.

link

1. An object in a directory that directs all references to itself to a file, directory, or another link. Like many other objects, a link has a path name that identifies it as a unique entity in the system directory hierarchy.

2. See also **bind**.

lock

1. A system data structure associated with a file, record, or device that can be set to restrict the use of the object; the restriction remains in effect until the lock is reset.

2. To set a lock on a file, record, or device.

member

An object that is part of a structure. A member can have its own name and a distinct type.

module

A single Stratus computer. A module is the smallest hardware unit of a system capable of executing a user's process.

National Language Support (NLS)

1. The ability of the operating system to represent text in languages other than English.
2. A system of supporting different languages, character sets, date and time formats, and currency formats.

NLS

See **National Language Support**.

nonconstant extents

See **asterisk extents**.

null pointer value

A specific pointer value that does not contain the address of a valid storage location.

object

1. A region of storage, the contents of which can represent values.
2. Any data structure or device in the system that you can refer to by a name or some other identifier. For example, all of the following are objects: directories, files, links, systems, modules, devices, groups, persons, ports, queues, locks, file indexes, and file records.

object library

One or more directories that the operating system searches for object modules.

object module

A file produced by a compiler that contains the machine-code version of one or more procedures; it usually contains symbolic references to external variables and programs. To execute the program, an object module must be processed by the binder to produce a program module, and then loaded by the loader. Object modules have the suffix `.obj`.

on-unit

The statement, begin block, or the token `system` that appears within an `on` statement. An on-unit is executed only if the condition specified in the `on` statement is signaled.

open

To prepare a file or device for a particular type of access. For example, the VOS PL/I `open` statement opens a file.

operand

A subexpression on which an operator acts.

operator

A symbol in an expression that performs an operation (evaluation) on one or more operands. The result of the operation specifies the computation of a value, designates an object or block, generates side effects, or produces a combination of these actions.

optimization

A code-improving modification that makes a program run faster, take less space, or both.

optimizer

Optionally invoked as part of the compilation process, an optimizer modifies executable code so that it runs faster, takes less space, or both.

organization

See **file organization**.

output file

A file opened to receive values from the program.

parameter

A declared name that describes an area of storage on which a called procedure operates; the address of the actual storage must be supplied by the calling procedure. Parameters are sometimes called formal parameters.

path name

A unique name that identifies a device or locates an object in the directory hierarchy. See also **full path name** and **relative path name**.

PL/I preprocessor statement

A statement used by the compiler's preprocessor to conditionally compile a source module, alter program text, control the generation of a compilation listing, or enable compiler options. An example is the `%include` statement. See also **binder-preprocessor statement** and **VOS preprocessor statement**.

pointer

The address of a storage location that contains an object of a particular type. In VOS PL/I, pointers of all types are four bytes long, have the same format, and store an unsigned address.

port

A system data structure that can be attached to a file or device for the purpose of accessing data from the file or device. The operating system automatically creates a port whenever you open a file.

port ID

A 2-byte integer used to identify a port.

port name

The character-string name of a port.

portable

1. Programs that are capable of being moved, without modification, from one machine type and/or operating system to another.
2. Programs that are machine or operating-system independent.

precision

The total number of significant digits in an arithmetic value, and, in the case of fixed-point decimal data, the scaling factor of the value.

preprocessor

Special software that allows you to use certain statements that make it possible to develop programs that are easier to read, easier to change, and easier to port to a different system. Preprocessing can be performed with the compiler, with the binder, or with a stand-alone preprocessor.

preprocessor statement

See **binder-preprocessor statement**, **PL/I preprocessor statement**, and **VOS preprocessor statement**.

preprocessor variable

A sequence of 1 to 32 letters, digits, and underline characters (_), in any position, used with VOS preprocessor or binder-preprocessor statements. You define preprocessor variables with the `-define` argument of the `pl1` command or the `$define` VOS preprocessor or binder-preprocessor statement.

privileged process

A process of a user who is logged in as privileged.

procedure

A sequence of statements, beginning with a `procedure` statement and ending with an `end` statement, that can be activated and executed as a unit.

process

The sequence of states of the hardware and software during the execution of a user's programs. When you log in, the operating system creates a process for you to control the execution of your programs. Your process can create other processes at your request. A process is always in one of three states: running, waiting, or ready.

profile file

A non-ASCII file containing performance information about all object modules compiled with the `-profile` or `-cpu_profile` argument of the `pl1` command and bound together in the program. A profile file has the suffix `.profile`.

program

One or more procedures, from one or more source modules, that together perform a task.

program entry

See **entry point**.

program module

A file containing an executable form of a program. A program module consists of one or more object modules (compiled source programs) bound together. A program module has the suffix `.pm`.

program name

The full or relative path name that identifies a program module.

programmed operator

A subroutine with a predefined interface that is shared throughout the system.

read lock

A lock that allows other tasks or processes to set a read lock on a given object but prevents them from setting a write lock. A read lock allows a reader to ensure that an object will not be modified while it is being read.

record

The data structure that the operating system uses to manage data in a file. For files other than stream files, a record is the smallest unit of data that the operating system I/O routines can access when performing I/O operations on files or devices. For stream files, a record consists of unstructured data.

record I/O

A method of reading from and writing to a file in which data is accessed one record at a time, using the `read`, `write`, `rewrite`, and `delete` statements. See also **stream I/O**.

relative organization

A VOS file organization in which data is stored as varying-length records in fixed-length fields. See also **fixed organization**, **sequential organization**, and **stream organization**.

relative path name

A name that identifies a device or an object in the directory hierarchy without specifying its full path name.

return

1. To terminate a procedure and transfer program control back to the calling block.
2. To terminate an on-unit and transfer control back to the point of the signal.

row-major order

The order in which array elements are stored. When stepping through an array in row-major order, the rightmost subscript varies most frequently, and the leftmost subscript varies least frequently.

run

To execute a program.

run time

The time at which a program module is invoked and executed.

scalar

A single, one-dimensional value or object; not an array or structure, though possibly an array element or structure member.

scope

1. The region of a program's source text in which an identifier is visible (that is, can be used).
2. An attribute of a declared name: `internal` or `external`.

search list

A series of operating system directories to be searched for include files, object modules, or command macros and program modules.

sequential access

A method of accessing records in a file in the order in which they are stored. See also **direct access** and **keyed sequential access**.

sequential organization

A VOS file organization in which data is stored in varying-length records; each record is preceded and followed by two bytes representing its length. See also **fixed organization**, **relative organization**, and **stream organization**.

shared variable

An external variable that can be shared simultaneously by several object modules. Shared variables are allocated in the virtual address space of one or more processes.

sign

An indication of whether an arithmetic value is less than or greater than zero. A sign can be positive or negative.

signal

An asynchronous interrupt for an error condition, or exception, that may be reported during program execution.

source module

A text file (single source program) containing language statements, preprocessor statements, and comments that can be compiled to produce an object module.

stack

An area of storage consisting of an ordered series of stack frames associated with the execution of a program.

stack frame

An area of storage on the stack associated with an activation of a procedure, begin block, or on-unit.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects.

statement

One of several programming constructs that specifies an action or actions to be executed by a program.

statement label

An identifier that is followed by a colon (`:`) and a statement. A labeled statement can be referenced in a `goto` statement.

static region

The region of an object module that contains external references and internal static data.

static storage

Storage allocated within the program module prior to program execution. Variables have the static storage class only if you specifically provide the `static` or `external` attribute in the variable declaration.

storage class

An attribute indicating when and where storage is allocated for an object. VOS PL/I supports five storage classes: `automatic`, `based`, `defined`, `static`, and parameter.

stream I/O

A method of reading from and writing to a file in which sequences of characters, up to 256 characters long, are accessed using the `put` and `get` statements. See also **record I/O**.

stream organization

A VOS file organization in which stream files with varying-length records are stored in a disk or tape region holding approximately the same number of bytes as the record. The record storage regions vary from record to record, and may be accessed on a record or byte basis.

When stream files are used to store text, each record contains one line of text. See also **fixed organization**, **relative organization**, and **sequential organization**.

string

A character string or bit string. See also **bit string** and **character string**.

structure

A sequentially allocated set of objects, grouped under a single name. Within the structure, each object or member can have its own name and a distinct type.

structure member

A variable, array, or structure that is immediately contained in a structure.

subroutine

A procedure that can be invoked by a `call` statement from within another procedure. The `procedure` statement and all `entry` statements of a subroutine must not include the `returns` option. Any `return` statement in a subroutine must not include a return value.

subscript

An arithmetic value used to specify a particular element or elements of an array.

suffix

A character string that begins with a period and is appended to an object name to indicate the type of the object.

symbol table

A construct that the compiler creates in the symtab region of an object module to facilitate symbolic debugging. The symbol table allows the debugger to convert user-defined variable names to locations of data or instructions. The compiler creates a symbol table only if you specify the `-table` or `-production_table` compiler argument.

text file

In general usage, a file of characters. A text file can contain graphic characters and control characters from the VOS internal character set. See also **binary file**.

token

An identifier, literal constant, punctuation symbol, comment, or preprocessor statement.

value

A measurable, describable, storable quantity that is associated with a constant, variable, or expression.

variable

A declared object that can be assigned a value.

varying-length character string

A character string that can have any length from 0 to 32,767 characters.

VOS internal character coding system

The system used internally for encoding character data on Stratus machines. This system, based on the international standard ISO-2022-1986, allows encoding of the multiple character sets needed for National Language Support.

VOS

The virtual operating system used in Stratus computers.

VOS preprocessor statement

A statement used by the compiler's preprocessor to conditionally compile a source module. An example is the `$define` statement. See also **binder-preprocessor statement** and **PL/I preprocessor statement**.

word

A 2-byte (16-bit) area of contiguous storage aligned on a 2-byte boundary.

write lock

A lock that prevents all other tasks and processes from setting either a read or write lock on a given object. A write lock allows a writer to ensure that an object will not be read while being modified and that multiple tasks or processes are not simultaneously modifying the same object.

XA2000-series module

A Stratus module that contains a microprocessor from the Motorola MC68000 microprocessor family.

XA/R-series module

A Stratus module that contains a microprocessor from the Intel i860 reduced instruction set computing (RISC) microprocessor family.

Index