SPARCompiler Ada User's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 802-3639-10
Revision A November, 1995

Contents

"The old order changeth, yielding place to new."

Tennyson

Preface

SPARCompiler Ada is the Sun Microsystems, Inc. Ada language compiler and toolset. It is one of the two software components in the Sun Ada Language Development Environment. The other component is SPARCworkst/Ada, which consists of a suite of OPEN LOOKR programming and development tools: AdaVision, dbtool (a visual debugger), LRMTool, and EditTool. (SPARCworks/Ada is documented separately in the *SPARCworks/Ada User's Guide*.)

SPARCompiler Ada and VADS

SPARCompiler Ada is based on the Rational Software Corporation's Verdix Ada Development System (VADS). Most of the SPARCompiler Ada documentation consists of an edited and amended version of the VADS documentation from Rational Software Corporation. Please note that for this first release of SPARCompiler Ada we have changed all references in the manuals from VADS to SPARCompiler Ada with the following exception, the directory or file names containing the words VADS or Verdix. For example, the directory `verdixlib` remains unchanged.

Note – This documentation set refers to SPARCompiler Ada as SC Ada..

Organization of the SPARCompiler Ada Manuals

The SPARCompiler Ada documentation set consists of the following:

SPARCompiler Ada User's Guide

Contains conceptual and tutorial information for how to use SPARCompiler Ada.

SPARCompiler Ada Reference Guide

Contains reference information for the file and directory system, commands and debugger commands and topics.

SPARCompiler Ada Programmer's Guide

Provides detailed information on various components of the programming environment.

SPARCompiler Ada Runtime System Guide

Contains information concerning the SPARCompiler Ada runtime system.

SPARCompiler Ada Multi-threaded Ada

Contains information concerning the SPARCompiler Ada multi-threaded runtime system.

Documentation Contents

The following paragraphs provide a brief description of the contents of each manual in the documentation set.

Contents: User's Guide

Chapter 1 Introduction to SPARCompiler Ada

This chapter describes the SC Ada components; what each component is used for; and how the components relate to each other. Brief descriptions of each tool are contained in this chapter. The directory structure associated with the SC Ada release is also described.

Chapter 2 Getting Started

Most users like to quickly get the feel of their software, even if they are not familiar with all the intricacies of the system. This chapter provides step by step directions for compiling and executing existing Ada software on SC Ada as well as directions for creating new Ada programs and compiling and executing them.

Chapter 3 Compiling Ada Programs

This chapter introduces concepts and pertinent information about compiling Ada language applications with SC Ada. Invocation examples, showing some of the more commonly used options are presented here.

Chapter 4 Linking and Executing Ada Programs

This chapter presents information necessary to use the linker and pre-linker as well as information about executing the program.

Chapter 5 Debugging Ada Programs

Debugging an application is an integral part of the development cycle. This chapter introduces the debugger and discusses how to use the SC Ada debugger in the development cycle.

Chapter 6 X-Window Debugging

This chapter introduces the SC Ada X-Window debugger and provides instructions on debugging Ada programs in the X-Window environment.

Chapter 7 Tutorial Development Session

Although *Getting Started* presented a brief example program, it was simple and only intended as a demonstration. The session included in this chapter, however, is a larger application. It takes you through all phases of the development cycle: source code generation, source code formatting, compiling, linking, debugging, and executing.

Contents: Reference Guide

Contents: Programmer's Guide

Chapter 4 Machine Code Insertions

This chapter discusses techniques for inserting machine code into applications being developed on SC Ada, and presents several examples.

Chapter 5 Interface Programming

This chapter deals with information that you will need to know to successfully use programs and libraries written in C with your Ada language programs. Here you will also find a discussion of a modular approach to converting C programs to Ada.

Appendix A User Library Configuration

This appendix contains complete instructions for optionally configuring self-hosted SC Ada systems.

Appendix B POSIX Conformance Document

This POSIX.5-1990 Conformance Document describes those items specified in the POSIX.5.1990 standard as implementation-defined that must be documented in order for SC Ada to claim conformance to it.

Appendix C XVIEW Interface and Runtime System

This appendix contains documentation on XViewt (X Window-System-based Visual/Integrated Environment for Workstations), also a Sun Microsystems toolkit that provides a windowing interface through which users can support interactive, graphics-based applications.

Appendix F Implementation-Dependent Characteristics

This appendix presents information that deals with how certain aspects of SC Ada are implemented. This appendix meets the requirements put forth in Appendix F of the Ada Language Reference Manual.

Contents: Runtime System Guide

Contents: SPARCompiler Ada Multithreading Ada Application

Chapter 3: Solaris MT and the Threaded Ada Runtime

This chapter contains information specific to the SUN Threads implementation.

Chapter 4: Debugging in the Multithreaded Environment

This chapter contains information about operation of the debugger in the multi-threaded environment.

Chapter 5 Examples

This chapter contains a series of examples illustrating different aspects of Multithreaded SC Ada.

How To Use This Manual Set

This manual set provide's users with a task-oriented aid to producing SC Ada language software with SPARCompiler Ada. Although the documentation contains separate chapters that discuss various SC Ada components, most chapters revolve around the concept of using SC Ada to perform specific tasks.

Because the documentation is task-oriented, it fills a dual role: as a tutorial and as a reference. We suggest that you begin by using the documentation as a learning aid. After you understand the mechanics of SPARCompiler Ada and how to produce an Ada language application by using SC Ada, you can use the manuals as a reference.   Reference entries are in alphabetical order, enabling you to find an entry without consulting the index. When using the documentation as reference resources, you can look in either the body of the text or in the *Reference Guide*.

The index is comprehensive and thoroughly cross-referenced, making the task of locating a topic within the documentation relatively quick and simple. If the topic you are looking for is not listed in the index, check the *Table of Contents*. It may be that the topic is broad enough to encompass a whole section or chapter, or that your selection is not sufficiently defined to be found in the index.

Reporting Problems To Product Support

Sun Microsystems welcomes your input and encourages you to report problems or make suggestions for improving our software, documentation, or service. To submit suggestions or bug reports, contact your local Sun Answer Center. Please ask your sales representative for more information: contract price, nearest Sun Answer Center, and so forth.

“Things are always at their best in their beginning.“

Blaise Pascal

Introduction to SC Ada 1

SPARCompiler Ada provides a complete software environment for developing Ada language applications. Hereafter, SPARCompiler Ada is referred to as SC Ada. SC Ada consists of a compiler, a runtime system, a debugger and a toolset. SC Ada provides interactive debugging including full batch-mode support and screen-oriented debugging.

The toolset is a collection of utilities that developers can use to help with library management, program generation, program analysis and other development tasks. SC Ada tools include a source code formatter or “pretty printer” to standardize source code printouts.

SPARCompiler Ada is currently validated for the SPARCstation] operating under Solaris] 2.1.

The SC Ada user interface provides an extended set of programming tools. The SC Ada toolset includes:

Name of Tool	Function
`ada`	Invoke the SC Ada compiler
`a.app`	Invoke the SC Ada preprocessor
`a.ar`	Create an archive library of SC Ada object files
`a.cleanlib`	Reinitialize library directory
`a.cp`	Copy unit and library information
`a.das`	Disassemble object files

Name of Tool	Function
`a.db`	Debug SC Ada and C source code programs
`a.du`	Summarize disk usage for SC Ada libraries
`a.error`	Analyze and disperse error messages
`a.header`	Print the information stored in a unit's net header
`a.help`	Invoke an interactive help utility for SC Ada
`a.info`	List or change SC Ada library directives
`a.ld`	Build an executable program from compiled units
`a.list`	Produce a source code listing
`a.ls`	List compiled units
`a.make`	Recompile source files in dependency order
`a.mklib`	Create an SC Ada library directory
`a.mv`	Move unit and library information
`a.path`	Report or change an SC Ada library search list
`a.pr`	Format source code
`a.prof`	Analyze and display profile data
`a.report`	Report SC Ada deficiencies
`a.rm`	Remove an SC Ada unit from a library
`a.rmlib`	Remove a compilation SC Ada library
`a.symtab`	Display symbol information for all static package variables and constants
`a.tags`	Create a source file cross reference of units
`a.vadsrc`	Display versions and create a library configuration file
`a.version`	Display if licensed for Multithreaded SC Ada
`a.view`	Provide aliases and history for a C shell user
`a.which`	Determine which project library contains a unit
`a.xdb`	Invoke the X-Window debugger
`a.xref`	Print cross-reference information for a given SC Ada unit or library

1.1 Overview of System Components

The remaining sections of this chapter provide a more in-depth look at each SC Ada component.

1.1.1 The SC Ada Compiler

At the heart of SC Ada is an optimizing compiler that incorporates several extensions over standard tools. Most notable of these are the error recovery and diagnostic messages that enhance the compiler. The SC Ada compiler complies fully with ANSI/MIL-STD-1815A-1983 (Ada LRM). Both the compiler and debugger adhere strictly to the standards put forth in the *SC Ada* LRM regarding syntax, semantics, runtime errors and runtime exceptions.

An SC Ada compiler is different from most other language compilers because SC Ada compilation uses information collected from previous compilations. This information is called “separate compilation information”. SC Ada maintains separate compilation information in ordinary directories and files. SC Ada stores separate compilation information, object files and intermediate files in binary format.

Unlike compilers for other languages that deal with files, SC Ada compilers work with *compilation units*, which are the smallest pieces of code that can compile successfully. An ASCII source file can contain several units, only some of which can be used in a specific program. Typically, any discussions of SC Ada center around units rather than files, although files are an important piece of the SC Ada environment.

With SC Ada, many functions typically performed by the compiler, such as error processing and disassembly, are handled by the toolset, leaving the compiler free to handle compilation. Figure 1-1 displays a compilation excerpt showing error messages from the compiler.

```
PROCEDURE test IS

    GENERIC
        TYPE T IS PRIVATE;
        INP :IN T;
    PACKAGE P1 IS END P1;

    PACKAGE PKG IS
        TYPE PRIV IS PRIVATE;
        DC: CONSTANT PRIV;

        PACKAGE I1 IS NEW P1 (PRIV,DC);
----------------------------^A                        ###
---------------------------------^B                   ###
--### A:error:RM 7.4.1(4):type is not yet fully defined
--### B:error:RM 7.4.3(2):illegal use of deferred constant

    private
        type priv is (x);

        package I2 is new p1 (PRIV,DC);
----------------------------------^A                  ###
--### A:error:RM 7.4.3(2):illegal use of deferred constant
```

Figure 1-1 Compiler Output

1.1.2 The SC Ada Runtime System

The SC Ada Runtime System (RTS) is a multi-tasking runtime system that supplies a wide variety of services.

Part of the RTS is always linked with the user program and is called the user library. The other part of the runtime system is the microkernel.

1.1.2.1 The SC Ada User Library

The User library has two major components. One provides services for SC Ada applications such as memory allocation and deallocation, integer to string conversion, SC Ada exception handling, and starting and exiting user programs. The other part of the user library provides SC Ada tasking and VADS EXEC services.

1.1.2.2 The SC Ada Microkernel

The Microkernel is the software layer that provides an interface between the user application and the hardware. Many services such as interrupt handling, subprogram call stack management, and POSIX thread management are handled by the microkernel. The microkernel is small and delivers excellent real-time performance.

1.1.2.3 How It All Fits Together

Figure 1-2 is taken from the Runtime System Overview and shows the different software layers of an SC Ada program. At the top is the SC Ada application, which is the software that you write. The application is linked with the user library, which is shown as all the layers encompassed by the dark curly brace on the left. All this is layered on top of the microkernel, shown as the box at the very bottom.

The microkernel is user configurable and written totally in SC Ada.

References

Runtime System Overview, SPARCompiler Ada Runtime System Guide

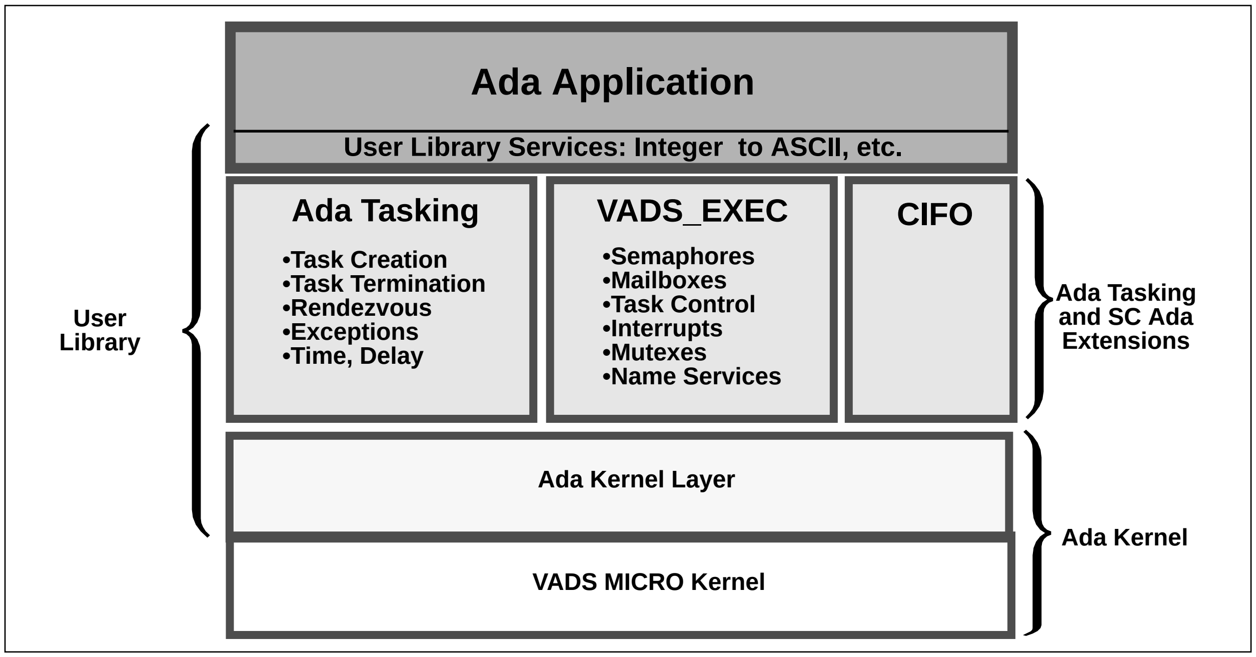


Figure 1-2 Runtime System Partitions

1.1.3 The SC Ada Debugger

The SC Ada debugger, `a.db`, is a symbolic debugger for SC Ada and C programs. The debugger supports both a traditional line-oriented mode and a screen-oriented mode.

```
68       configuration (col) = row
69    end
70
71    Procedure remove_queen (row, col: in integer) is
72       vacant: constant integer := 0
73    begin
74*      safe_row (row) := true;
75       safe_up_diag (row + col) := true;
76       safe_down_diag (rwo - col) := true;
77       configuration (col) := vacant;
78    end;
79
80    function is_safe (row, col:in integer) return boolean is
81    begin
-*-------------------------------------------------queens.a--
**MAIN PROGRAM ABANDONED - EXCEPTION ”constraint error” RAISED
(program is now stopped on line where exception was raised)
stopped 8 instructions after ”/usr2/vads/examples/queens.a”:74
in remove_queen
74       safe_row (row) := true;
 => RM 4.1.1(4): index value greater than upper bound of index
subtype
```

Figure 1-3 Debugger Output

1.1.4 The SC Ada Toolset

In addition to the compiler and debugger, SC Ada includes a full complement of tools to aid programmers in developing their SC Ada language applications. The tools are grouped into four classes:

- Program Generation Tools
- Analysis Tools
- Library Management Tools
- Miscellaneous Tools

The following sections describe each class of tools, list the tools included in each group and briefly describe each tool. For a complete description of each of the following tools, see the *Reference Guide,* SC Ada *Command Reference.*

1.1.4.1 Program Generation Tools

Program generation tools are utilities that make the generation of source code, object files and executable files less taxing.

a.ld

The utility `a.ld` is a prelinker that gathers information for each unit that is dependent on the main unit. `a.ld` verifies that all units are up to date, gathers elaboration and exception information into an executable image file and creates a list of commands, object names and options with which it invokes the linker.

References

`a.ld`, *SPARCompiler Ada Reference Guide*

Chapter 4, “Linking and Executing Ada Programs”

a.make

When compiling a program that consists of multiple units, changing a unit upon which other units depend requires that the dependent units, as well as the changed unit, be recompiled in a specific order. Without `a.make` you must manually maintain a compilation order table, which enables you to determine the order of recompilation and to accurately reconstruct the latest version of a program. However, building, maintaining and using a compilation order table by hand is tedious, difficult and most importantly, error-prone. `a.make` automates the reconstruction process. It performs the minimal number of recompilations necessary to update a unit, program, library or group of libraries after the source code is changed.

References

`a.make`, *SPARCompiler Ada Reference Guide*

1.1.4.2 Analysis Tools

SC Ada includes these tools to aid in the analysis of programs under development.

a.das

The utility `a.das` is a stand-alone disassembler. Use `a.das` to disassemble SC Ada object code into machine-code instructions. `a.das` interleaves the object code with the source code.

Normally, `a.das` is used as part of the edit-compile cycle to examine compiler output during development, particularly for machine code procedures.

References

`a.das`, *SPARCompiler Ada Reference Guide*

a.db

The SC Ada symbolic debugger debugs Ada programs at the source level. Also, use it to analyze code written in C.

References

`a.db`, *SPARCompiler Ada Reference Guide*

Chapter 5, “Debugging Ada Programs”

a.error

Although generally called from the compiler, you can use the `a.error` utility separately. `a.error` analyzes error messages produced by the SC Ada compiler. As an option you can direct that the error messages be inserted into the source code file. In this way, you can automatically invoke the editor and make corrections while reading the error message, which indicates both the position and nature of the error.

References

Automatic invocation of `a.error`:

`ada`, *SPARCompiler Ada Reference Guide*

`a.error`, *SPARCompiler Ada Reference Guide*

a.header

This utility enables the viewing of the information stored in the net file corresponding to a unit. Each distinct piece of information has its own option that can be used by itself, or in conjunction with other options. Unless stated otherwise, each piece of information is printed on its own line. The information is always printed in a predefined order and cannot be changed.

References

`a.header`, *SPARCompiler Ada Reference Guide*

a.prof

`a.prof` is a Statistical Profiler that provides an accurate description of the CPU usage in all parts of an Ada program, including time spent in the SC Ada runtime system.

References

`a.prof`, *SPARCompiler Ada Reference Guide*

`a.prof`, *Statistical Profiler, Chapter 3, SPARCompiler Ada Programmer's Guide*

a.symtab

`a.symtab` generates a listing containing symbol information for one or more Ada units. This symbol information is generated for all static variables and constants declared at the package level.

References

`a.symtab`, *SPARCompiler Ada Reference Guide*

a.xdb

The SC Ada X-Window symbolic debugger debugs Ada programs at the source level in the X-Window environment.

References

`a.xdb`, "Running a.xdb" on page 6-3

a.xref

This tool prints cross-reference information for a given Ada unit, several specified Ada units or an entire Ada library.

References

`a.xref`, *SPARCompiler Ada Reference Guide*

1.1.4.3 Library Management Tools

An SC Ada library is a directory that is initialized to enable the SC Ada compiler to operate in that directory. SC Ada libraries contain the files `ada.lib`, `gnrx.lib` and `GVAS_table` and the directories `.imports`, `.lines`, `.nets` and `.objects`. All work using SC Ada must occur in an SC Ada library. Ada libraries can reference other libraries, enabling a program to access non-local units during compilation. This organization enables programmers to work on local version of individual program units while retrieving the remainder of the program from previously-developed libraries.

We supply a number of tools to help you create, manage and maintain SC Ada libraries. This section describes each of the SC Ada Library Management Tools.

a.cleanlib

This utility “cleans” a specified SC Ada library, emptying the subdirectories and the files `ada.lib`, `gnrx.lib` and `GVAS_table` of all separate compilation information but preserving all non-compilation information in `ada.lib`. `a.cleanlib` has no effect on non-SC Ada files.

References

`a.cleanlib`, *SPARCompiler Ada Reference Guide*

a.cp

This utility copies all information associated with the named unit(s) or file(s). When a unit is specified, the corresponding `.nets` and `.objects` files are copied and the `ada.lib` entries are copied for the affected unit(s).

References

`a.cp`, *SPARCompiler Ada Reference Guide*

a.du

`a.du` is useful for determining disk usage. When invoked, this command lists the size, in bytes, of files in each library unit in the specified or current directory.

References

`a.du`, *SPARCompiler Ada Reference Guide*

a.info

Use `a.info` to review or alter the directives that affect the current library. Use `a.info` either interactively or by passing it the necessary parameters from the command line.

References

`a.info`, *SPARCompiler Ada Reference Guide*

a.ls

List the units in the current directory that are compiled, by using the `a.ls` utility. `a.ls` provides options to vary the amount of information that it displays.

References

`a.ls`, *SPARCompiler Ada Reference Guide*

a.mklib

`a.mklib` transforms a specified directory into an SC Ada library. As explained previously, all Ada compilation must occur in an SC Adalibrary, which contains required files and directories. When you invoke the SC Ada tool `a.mklib`, if the specified directory does not exist, it is created. The necessary files and directories are placed in the directory and the directory is converted to an SC Ada library.

References

`a.mklib`, *SPARCompiler Ada Reference Guide*

a.mv

This utility moves all information in the named unit(s) or file(s). When a unit is specified, the corresponding `.nets` and `.objects` files are moved and the `ada.lib` entries are deleted from the source library and created in the target library for the affected units.

References

`a.mv`, *SPARCompiler Ada Reference Guide*

a.path

When compiling Ada programs with SC Ada you must search libraries other than the current library for files or units that must be included in the compilation. Use `a.path` to either review which paths are searched or to change the list of libraries to be searched.

References

a.path, *SPARCompiler Ada Reference Guide*

a.rm

a.rm removes all the entries in ada.lib and compiler-generated files of the specified unit(s) or file(s). Any deleted information affects all dependent units of the specified unit(s) or file(s).

References

a.rm, *SPARCompiler Ada Reference Guide*

a.rmlib

A directory that is made into an SC Ada Ada library can be returned to non-library status by using the tool a.rmlib. This tool removes all the library components from the specified library, but leaves everything else intact. The library becomes a normal directory and you can no longer execute the Ada compiler or SC Ada tools in that directory.

References

a.rmlib, *SPARCompiler Ada Reference Guide*

a.tags

a.tags is similar to the ctags(1) utility for C programs. a.tags indexes the location of units in a file called tags. Some editors use the tags file to access source directly by unit name rather than by directory and filename.

References

a.tags, *SPARCompiler Ada Reference Guide*

a.vadsrc

Determine which versions of SC Ada are installed on a system by using the tool a.vadsrc. When invoked, a.vadsrc displays a list of all versions of SC Ada that are available on a system, whether a .vadsrc file exists in the home directory or current directory and the contents of that file. Also, use a.vadsrc in an interactive mode to create the file .vadsrc.

References

`a.vadsrc`, *SPARCompiler Ada Reference Guide*

a.which

Use `a.which` to list the name of the source file of a unit that is visible in the current SC Ada library.

References

`a.which`, *SPARCompiler Ada Reference Guide*

1.1.4.4 Miscellaneous Tools

a.app

`a.app` is an Ada source pre-processor that is invoked either during an Ada compile or separately. It supports macro substitution, conditional compilation and the inclusion of normal source files.

References

`a.app`, *SPARCompiler Ada Reference Guide*

Ada Preprocessor, Chapter 2, SPARCompiler Ada Programmer's Guide

a.ar

`a.ar` enables the creation of an archive library of all objects corresponding to the units in the closure of a specified unit or of all objects in the current SC Ada library. Error messages are generated and the archive is not built if an Ada unit requiring elaboration or an out-of-date unit are to be included in the archive. Options are provided to force these units into the archive if so desired.

References

`a.ar`, *SPARCompiler Ada Reference Guide*

a.help

`a.help` includes information for all of the SC Ada commands as well as for a number of development concepts that are discussed in the SC Ada documentation. In essence, if it is in the SC Ada Reference Manual, on-line help is available using `a.help`.

References

`a.help`, *SPARCompiler Ada Reference Guide*

a.list

After you produce a source file that contains no errors, list the file by using `a.list`. This tool includes line numbers in the source code listing and sends the listing to standard output. You can suppress line numbers and redirect output.

References

`a.list`, *SPARCompiler Ada Reference Guide*

a.pr

`a.pr` is a source code formatter included with SC Ada. The utility includes options for customizing output to meet individual Ada coding standards.

References

`a.pr`, *Ada Formatter, SPARCompiler Ada Programmer's Guide*

a.report

Use `a.report` to generate problem reports concerning errors in the SC Ada product or documentation.

References

`a.report`, *SPARCompiler Ada Reference Guide*

a.version

`a.version` displays a message indicating if you are licensed for Multithreaded Ada. If you are not licensed for MT Ada, an error message is displayed.

References

`a.version`, *SPARCompiler Ada Reference Guide*

a.view

`a.view` is a shell script that enables you to define aliases for commonly used SC Ada commands. You must use the C shell in order to use `a.view`.

1.2 What does SC Ada look like on your System?

A thorough understanding of the various SC Ada components and how they interrelate enables you to use SC Ada more efficiently and effectively. In order to perform a number of tasks related to using SC Ada or to the development process, you must know about the SC Ada directory structures.

Figure 1-4 illustrates the release directory structure.

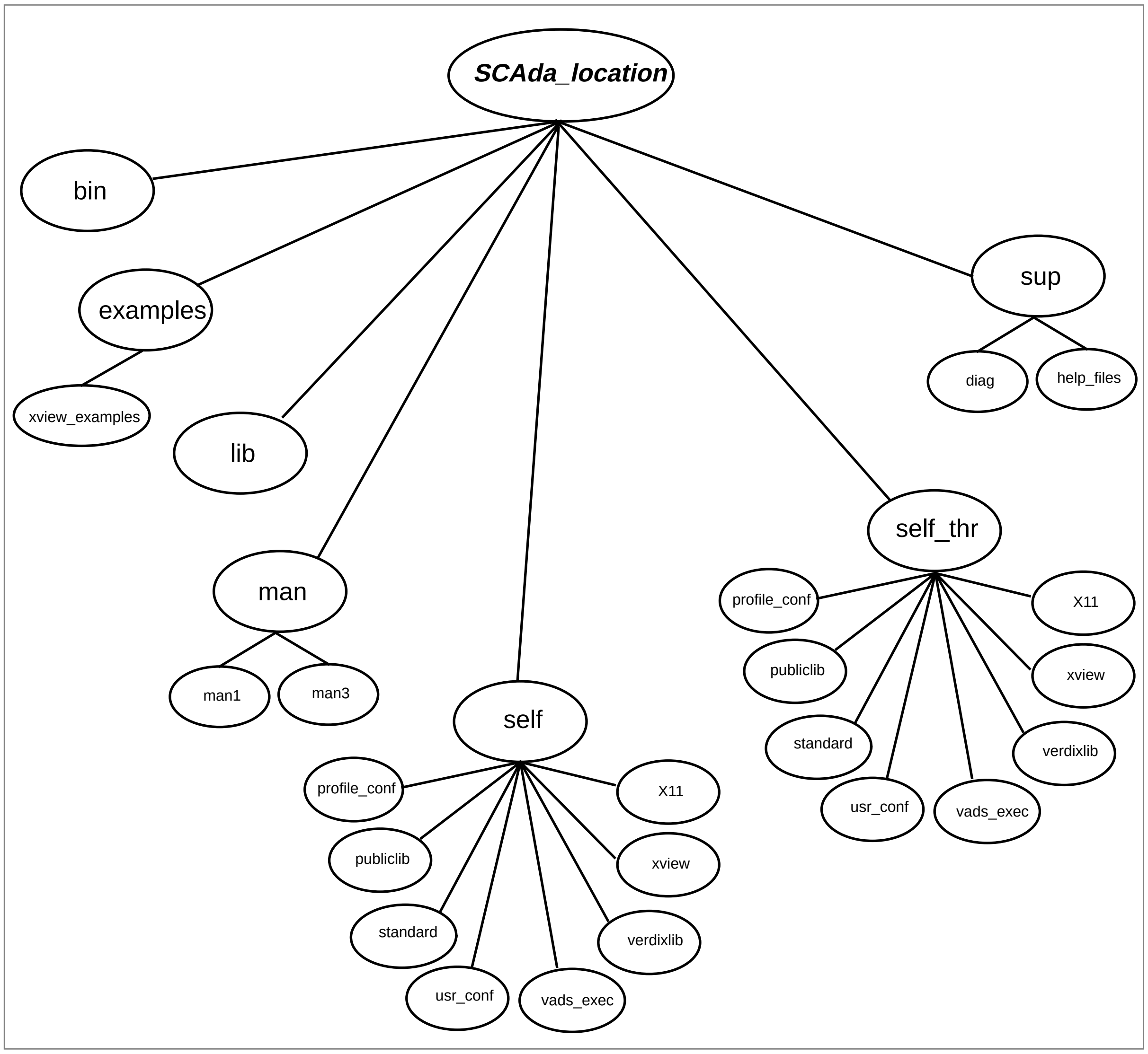


Figure 1-4 Directory Structure

1.2.1 The SC Ada Release Directory Structure

When installing SC Ada on a hardware platform you first select where you want the SC Ada release directory structure to reside. Place the SC Ada software in an already-existing empty directory or create a separate directory to hold each version of SC Ada that you want to install. We often refer to the directory *`SCAda_location`*, which is a user-defined name and always refers to the directory that you create for the current SC Ada installation.

After you decide where to install the SC Ada software, install SC Ada by following the installation instructions found in the Installation Guide. As SC Ada is installed, the installation procedure creates a release directory structure under your *`SCAda_location`* directory. The installation procedure then copies numerous files into the appropriate directories. When you are ready to configure or use SC Ada, many of the files that you need are in this release directory structure.

The subdirectories where the SC Ada files are installed are created under *`SCAda_location`*. The following paragraphs briefly describe each of these subdirectories.

`bin` contains SC Ada executables.

`examples` contains sample programs illustrating Ada language use and demonstrating the capabilities of the language. (The `examples` directory is neither warranted nor supported by Sun Microsystems.)

- `xview_examples` contains example programs illustrating the use of XView.

`lib` contains a thread-safe math library.

`man` contains manpages for SC Ada tools, which can be installed. Access them with the system `man` command.

- `man1` contains the manpages for all the SC Ada tools.
- `man3` contains the manpages on the SC Ada libraries.

`self` contains the libraries based on `standard`.

- `posix` contains the SC Ada implementation of the POSIX5 bindings.
- `profile_conf` is an enhancement of the user library configuration library, `usr_conf`. To capture and write profiling data, `v_usr_conf_b.a` is modified and two files are added, `profile.a` and `profile_b.a`.

`publiclib` contains public domain packages written in Ada. (The `publiclib` directory is neither warranted nor supported by Sun Microsystems.)

`standard` contains the SC Ada implementation of `package STANDARD` and all other predefined packages and library units. Packages used to support the predefined environment are included.

`usr_conf` contains configuration files for the runtime system.

`vads_exec` contains the user interface to the SC Ada runtime services consisting of interrupt handling, Ada tasking extensions, semaphores, mailboxes, pool allocation, memory management, mutex/condition variables, name services and stack operations. These services augment the predefined Ada tasking capabilities.

`verdixlib` contains mathematical functions, operating system calls, command line arguments and other programs.

`X11` contains bindings to X11.

`self_thr` contains the libraries based on `standard`. This directory is used with Multithreaded Ada.

`profile_conf` is an enhancement of the user library configuration library, `usr_conf`. To capture and write profiling data, `v_usr_conf_b.a` is modified and two files are added, `profile.a` and `profile_b.a`.

`publiclib` contains public domain packages written in Ada. (The `publiclib` directory is neither warranted nor supported by Sun Microsystems.)

`standard` contains the SC Ada implementation of `package STANDARD` and all other predefined packages and library units. Packages used to support the predefined environment are included.

`usr_conf` contains configuration files for the runtime system.

`vads_exec` contains the user interface to the SC Ada runtime services consisting of interrupt handling, Ada tasking extensions, semaphores, mailboxes, pool allocation, memory management, mutex/condition variables, name services and stack operations. These services augment the predefined Ada tasking capabilities.

`verdixlib` contains mathematical functions, operating system calls, command line arguments and other programs.

`xview` contains Ada package specifications and bodies that parallel the similarly named C header files for XView in the include directories `/usr/openwin/include/xview` and `/usr/include`.

`X11` contains bindings to X11.

`sup` contains additional executable programs called by other SC Ada tools, installation tools and help files accessed with `a.help` and `a.db`. It contains the following directories:

`diag` contains diagnostic utilities.

`help_files` contains the help files for SC Ada and `a.db`.

1.3 How to Use SC Ada

In this section we look at how to use SC Ada and its tools to develop an Ada application. We do not develop an application yet, that comes in *Getting Started*. But we walk through a typical development cycle to get a feel for how, when and why to use the different SC Ada components.

1.3.1 Self-host Development Overview

Development of an application begins with the creation of a directory, called an SC Ada library, in which you compile and link your application. To create this directory, use the tool `a.mklib` either interactively or non-interactively. `a.mklib` creates the specified directory, along with subdirectories and files that you need to compile and link your program.

After creating an SC Ada library, you must use `a.path` to let SC Ada know which libraries to search while compiling and linking. This enables your application to access the routines you need from other libraries. Adding libraries to the search list (and deleting them) is easy and can be done at any time.

In addition to specifying a library search list, you may need to specify a number of directives for the compiler and linker. To do this, use `a.info`. Like `a.path`, `a.info` adds, deletes, modifies and lists directives.

Use any ASCII-output editor to create your source code. If your company has coding standards that you must follow, use the `a.pr` source code formatter to format your code. `a.pr` has a number of parameters that adjust the appearance of your source code, from varying margin sizes to pagination and line length.

With source code completed, you are ready to try compiling your application, using the command `ada`. Because source code seldom compiles the first time, you usually go through the edit-compile cycle several times. Your task is aided by invoking the compiler with the `-ev` option, which inserts error messages into the source code and then calls the editor so you can make corrections.

If you are using machine code in your application, `a.das` provides stand-alone disassembly capabilities. Disassembling your machine code enables you to examine compiler output during the development cycle.

Recompiling the entire application when you are editing a small part of it is time consuming and error prone. Use `a.make` to ease this task. It performs the minimum recompilations necessary to update a changed part of the program.

Use the `-l "linker"` option to `a.make` or call the SC Ada linker, `a.ld`, directly the SCAda linker, SCAda LD, to produce the executable file.

When your application compiles properly, the compiler produces an executable file that you run by typing the name of the file (for example, `a.out`). However, you may have runtime errors, which require additional iterations of the edit-compile cycle.

SC Ada supplies several tools that help you manage and maintain the SC Ada libraries. Some of these, like `a.mklib` and `a.path`, we examined already. Others perform tasks, that are essential to keeping your development overhead to a minimum. `a.cp` and `a.mv` copy and move library information during development. After finishing a development project, remove unnecessary SC Ada components and libraries with `a.cleanlib` or `a.rmlib`. `a.cleanlib` "cleans" a library by removing information related to the compilations that you have performed (this information is called separate compilation information). When you finish the development cycle and have an executable application, the separate compilation information is no longer required. Remove it to reduce disk space usage and overhead.

If a library is no longer required, remove the library with `a.rmlib`. `a.rmlib` does not physically remove the library from the file system; it simply removes the files and subdirectories that are created by `a.mklib`. This only affects files and directories created by `a.mklib`.

Figure 1-5 is a flowchart depicting the typical development process for a self-host applications.

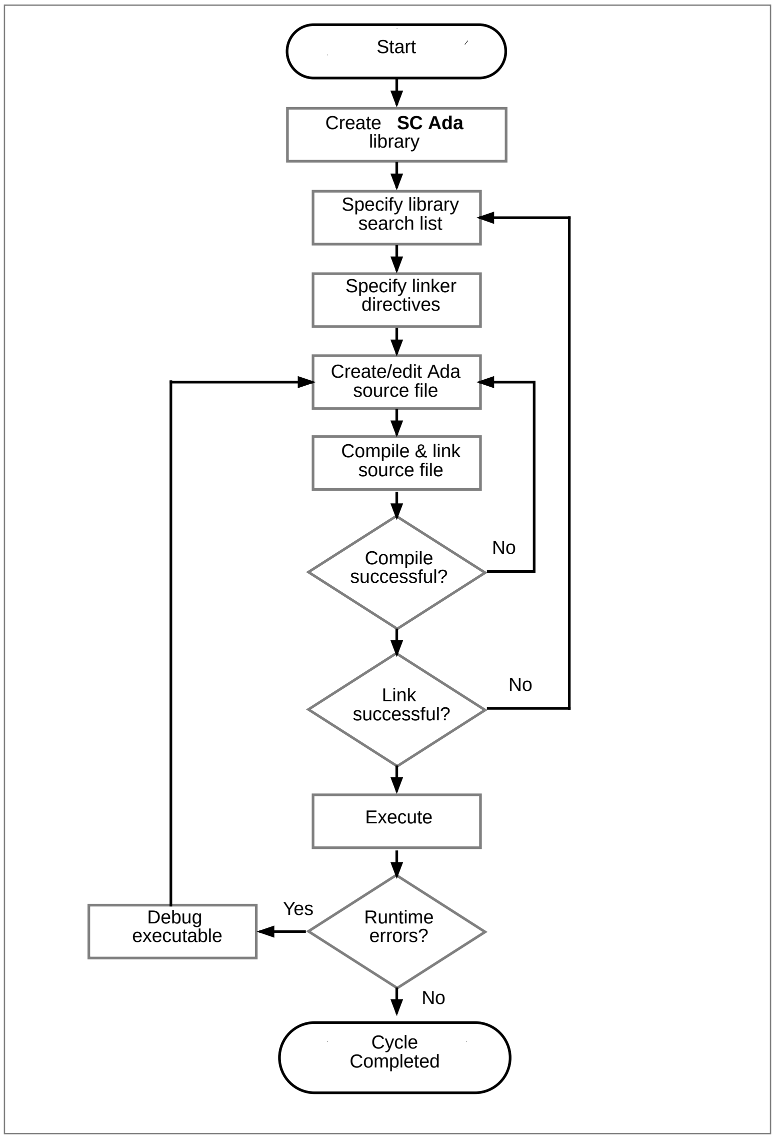


Figure 1-5 Development Cycle for Self-host Applications

References

SC Ada libraries created by `a.mklib`

"Reeling and Writhing of course to begin with," the Mock Turtle replied, "and the different branches of arithmetic - Ambition, Distraction, Uglification and Derision."
Lewis Carroll

Getting Started 2

In this chapter, you learn how to get started with SC Ada. Two basic situations are covered. The first situation is where Ada source code must be recompiled and executed using SC Ada. This is the case with benchmarks and existing software. The second situation is generating new Ada programs using SC Ada. For this case, you develop a small application to demonstrate many of the features of the system.

In either case, tasks must be completed before either compiling existing software or generating new software. Completion of these tasks enables you to use your system and SC Ada most effectively.

2.1 Before Using SC Ada the First Time

You must complete several tasks before starting with SC Ada.

2.1.1 Where are the SC Ada Tools

To access SC Ada tools, add the path for the directory where the SC Ada tools reside to the host OS PATH variable, making the complete range of SC Ada commands available.

Usually, the path is set in one of the shell configuration files (`.profile`, `.cshrc`, `.login`) so that commands are available on subsequent logins.

For example, if the SC Ada tools reside in `/usr2/ada_2.1/bin`, add this directory to your OS path variable defined in `set path =`. To make the path visible during the current session, remember to source the file. In the following example, the OS path variable is initialized in the `.login` file.

```
% vi .login
  set path = (. /usr2/ada_2.1/bin ......)
  :wq
% source .login
```

2.1.2 Configuring the User Library

On SC Ada, configuration is limited to the user library. The user library contains support routines for Ada programs. Many user library functions are implicitly called by compiler-generated code. Configuration of the user library focuses on the memory management support underlying the Ada allocators. The supplied default configuration is sufficient for most applications so typically no configuration is required for the user library.

If you must override the default configuration, modify the user library configuration `package V_USR_CONF`. *Appendix A* of the *Programmer's Guide* has detailed directions for doing this.

2.2 Help!!

Sun Microsystems provides online help for each of the SC Ada utilities and for debugger commands and concepts through the `a.help` tool. This tool is invoked by entering `a.help` followed by the name of the utility to review. Entering `a.help` alone provides information on the help utility itself and a list of subjects online help is available for. For example, to view the online help information on `a.make`, enter:

```
% a.help a.make
```

To exit the online help utility, enter `q`.

On some systems, reference manual entries are available for the compiler and tools by using the `man` command. If they are installed, a listing of available topics is obtained by typing:

```
% man ada
```

To view an entry for a specific tool or utility, enter the name of the utility on the command line. For example,

```
% man a.make
```

References

a.help, *SPARCompiler Ada Reference Guide*

Debugger help, *SPARCompiler Ada Reference Guide*

2.3 Resolving References/Finding Declarations

SC Ada resolves references and finds declarations by searching for a compilation unit in the following manner:

1. Looks for the unit in the current library. This includes examining all the directives in the `ada.lib` file in that library.
2. Looks in each library on the ADAPATH of the current library. These libraries are searched in the order in which they appear on the ADAPATH line in the current library's `ada.lib` file.

For example, consider the situation illustrated by . There is an Ada library with a single package specification called PACK. There are three sublibraries: `main`, `sub1` and `sub2`. `sub1` and `sub2` each contain a different body for `package PACK`. One of them prints out "one" while the other prints out "two". The sublibrary `main` has a main program that `with`'s `PACK`. The sublibrary `main` has both `sub1` and `sub2` on its ADAPATH.

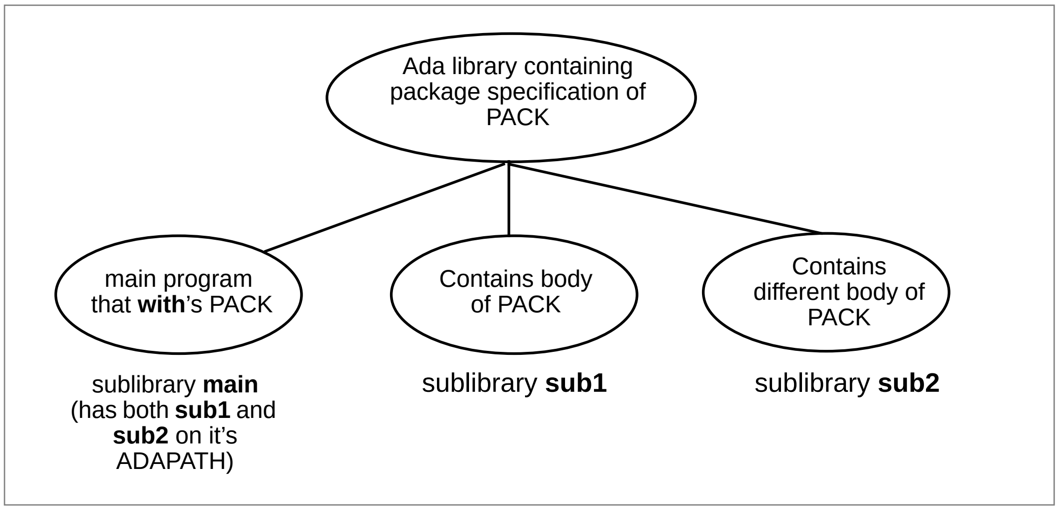

Ada library containing package specification of PACK
main program that **with**'s PACK
Contains body of PACK
Contains different body of PACK
sublibrary **main** (has both **sub1** and **sub2** on it's ADAPATH)
sublibrary **sub1**
sublibrary **sub2**

Figure 2-1 Finding Declarations

The question is, which version of the body of PACK is linked into the main program? The answer is, the one that appears first in the `ada.lib` file found in sublibrary `main`. Its appearance hides all subsequent bodies of PACK. Therefore, if the contents of the `ada.lib` file in `main` are as follows, the body of PACK found in sublibrary `sub1` is linked and the resulting executable file prints "one".

```
!ada library
ADAPATH= /usr2/mylib/sub1 /usr2/mylib/sub2 /usr2/mylib
ADAPATH= /usr2/ada_2.1/self/verdixlib
/usr2/ada_2.1/self/standard
main:MHNLSB 2E936CE6:main01:
main:MHNLSS 2E936CE6:main01:
```

If you reverse the order of the libraries `sub1` and `sub2` in the `ada.lib` file and relink, you get different results when you execute the program (i.e., "two" is printed).

The libary search path (ADAPATH) indicates which libraries are searched and in what order to resolve unit references. Your current Ada library is implicitly at the head of the path. By default, if `a.mklib` is invoked with the interactive option, the libraries `standard` and `verdixlib` are included on your library search path.

Sun Microsystems supplies a tool, `a.path`, to display and modify the library search path.

To display the libraries currently on the search path, use `a.path` with no options:

```
% a.path
/usr2/mylib/sub1
/usr2/mylib/sub2
/usr2/mylib
/usr2/ada_2.1/self/verdixlib
/usr2/ada_2.1/self/standard
```

Figure 2-2 a.path Output

If units in other SC Ada libraries are referenced, include these libraries on the ADAPATH. Use `a.path` to add these to your ADAPATH. If you do not do this, you get error messages similar to the following when you attempt to compile your files:

```
/usr2/mylib/test.a:1, line 1, char 6:error: spec of missile not
found in searched libraries
/usr2/mylib/test.a:1, line 4, char 1:error rm 8.3: identifier
undefined
```

For example, suppose you reference files in the library `/usr2/yourlib`. You must add it to your path.

```
% a.path -i /usr2/yourlib
```

References

`a.path`, *SPARCompiler Ada Reference Guide*

2.4 Compiling and Executing Existing Software

Often, pre-existing software must be recompiled and executed using SC Ada. This is true when running benchmarks or converting to SC Ada from another system. In these cases, take the following steps.

2.4.1 Creating an SC Ada library

Since all Ada source code must be compiled in an SC Ada library, you must create an SC Ada library to hold the source code or you must convert the directory in which the source code resides to an SC Ada library. An SC Ada library is simply a directory on which `a.mklib` has been run. When `a.mklib` is executed on a directory, it converts it to an SC Ada library by creating the `ada.lib`, `gnrx.lib` and `GVAS_table` files and the `.imports`, `.lines`, `.nets` and `.objects` directories in that directory.

To convert a directory containing source code into an SC Ada library move into the directory where the code resides. Use the **-i** (interactive) option to `a.mklib`. In the following example, assume that the code to be recompiled is in the directory `/usr2/mylib`.

```
% cd /usr2/mylib
% a.mklib -i
```

The interactive option to `a.mklib` displays all the versions of SC Ada resident on your system and prompts you for which version of SC Ada you want to use. Select the version desired.

Figure 2-3 shows the interactive output from **a.mklib**.

```
2 versions of Ada are available on this machine:
    Target Name    Version     Ada Location
1   SELF_TARGET    3.0         /usr2/ada_3.0/self
                               host_name, host_os, version_number
2   SELF_TARGET    3.0         /usr2/ada_3.0/self_thr
                               host_name, host_os, version_number
 Selection (q to quit):
2 versions of SCAda are available on this machine:

    Target Name    Version     SCAda Location
1   SELF_TARGET    3.0         /usr2/vads/self
                               host_name, host_os, version_number
2   SPARC                      3.0/usr2/vads_sparc
                               host_name, host_os, version_number

Selection (q to quit):
```

Figure 2-3 Interactive a.mklib

In response to the `Selection:` prompt, enter the number of the compiler you want to use.

Two versions of the supplied libraries are provided in the directories `self` and `self_thr`. Your application must be built with the supplied Ada libraries exclusively from one of these two directories. You chose which runtime system you wish to use when you make your application libraries with `a.mklib`. If you are going to be using the *SCAda Threaded* runtime (VADS MICRO microkernel), use the libraries in *`SCAda_location`*`/self.` If you are going to be using the *Solaris MT* runtime (Solaris Threads microkernel, use the libraries in *`SCAda_location`*`/self_thr`.

Since we are using a self-host system and since the interactive `a.mklib` shows us that two SC Ada versions are available and we wish to use the *SCAda Threaded* runtime, the response to the `Selection (q to quit):` prompt is `1`.

For the purpose of this discussion, the *`SCAda_location`* is indicated as `/usr2/target`.

References

`a.mklib`, *SPARCompiler Ada Reference Guide*

SPARCompiler Ada Multithreading Ada Application

2.4.2 Recompiling Your Source Code

Sun Microsystems supplies a powerful automatic recompilation tool which determines all unit dependencies and compiles your files in the correct order automatically. In this case, your files reside in a clean Ada library, that is, an Ada library in which no compilations have yet occurred. Automatically compile all files in this library in the correct dependency order with a single command.

```
% a.make -v -f *.a
```

If all your source files do not have the same extension, list them after the `a.make -f` option.

When referencing units in other user libraries, you must first compile them before compiling the dependent units.

References

`a.make`, *SPARCompiler Ada Reference Guide*

2.4.3 Creating a New Executable Program

To create a new executable file, invoke the SC Ada linker (`a.ld`). The only required argument to this tool is the name of the main unit of the executable file. This unit must be a parameterless procedure or integer function. For example, we assume the main unit is named `test`.

```
% a.ld test
```

This generates an executable file with the name `a.out`. To generate an output file with a different name, use the **-o** option.

```
% a.ld test -o test.out
```

References

`a.ld`, *SPARCompiler Ada Reference Guide*

Chapter 4, “Linking and Executing Ada Programs”

2.4.4 Running the Executable Program

To run the executable program you just generated, enter the name of the executable file.

```
% a.out
```

2.4.5 Possible Problems

Problems can occur during this process. The following is a list of some of the more common problems, their causes and solutions.

Problem: Files do not compile successfully and return errors indicating required information is not found in the searched libraries or an identifier is unidentified.

Cause: The required information is not visible in the Ada library in which the compilations are taking place.

Solution: Place the Ada libraries containing the required information on your library search list. See *Resolving References/Finding Declarations* on page 2.

Problem: When you attempt to use `a.mklib`, you get the following message:

```
a.mklib: command not found
```

Cause: You did not make the SC Ada tools visible to your operating system path variable.

Solution: Add the directory where the SC Ada tools are held to the OS path variable. See *Where are the SC Ada Tools* on page 1.

Problem: Your program does not execute correctly.

Cause: Any of a number of reasons.

Solution: Use the SC Ada debugger to figure out what is wrong with your program. See *Running the Debugger* on page 9.

2.5 Creating and Executing New Ada Programs

The first step in writing and compiling Ada programs is creating an SC Ada library. This is simply a directory in which to compile programs. The compiler creates and maintains several sub-directories of information in an SC Ada library directory. The example in this section illustrates how to create an SC Ada library, create Ada source code and compile, link and execute a program. The same basic process is used to develop any program. The Tutorial in the *User's Guide* provides a more detailed example .

2.5.1 Creating an SC Ada Library

Because SC Ada requires that all compilation occur in an SC Ada library, the first step in your development cycle is to create an SC Ada library. For this example, we create a library in `/usr2` with the name `mylib`.

```
% mkdir /usr2/mylib
% cd /usr2/mylib
```

Use the `a.mklib` tool with the **-i** (interactive) option to initialize `mylib` as a SC Ada library:

```
% a.mklib -i
```

The system responds with output similar to this:

```
2 versions of Ada are available on this machine:
    Target Name   Version   Ada Location
1   SELF_TARGET   3.0       /usr2/ada_3.0/self
                            host_name, host_os, version_number
2   SELF_TARGET   3.0       /usr2/ada_3.0/self_thr
                            host_name, host_os, version_number
 Selection (q to quit):
```

Two versions of the Sun Microsystems-supplied libraries are provided in the directories `self` and `self_thr`. Your application must be built with Sun Microsystems-supplied Ada libraries exclusively from one of these two directories. You chose which runtime system you wish to use when you make your application libraries with `a.mklib`. If you are going to be using the *SCAda Threaded* runtime (VADS MICRO microkernel), use the libraries in

SCAda_location/self. If you are going to be using the *Solaris MT* runtime (Solaris Threads microkernel, use the libraries in *SCAda_location*/self_thr.

Since we are using a self-host system and since the interactive a.mklib shows us that two SC Ada versions are available and we wish to use the *SCAda Threaded* runtime, the response to the Selection (q to quit): prompt is **1**.

a.mklib converts mylib to an SC Ada library.

References

a.mklib, *SPARCompiler Ada Reference Guide*

SPARCompiler Ada Multithreading Ada Application

2.5.2 Creating Your Ada Source file

The source file for this example is short, so type it in from the keyboard. When compiled and executed, the example displays the text:

```
Hello, world.
```

From your keyboard, enter the following source code and give the file the name hello.a:

```
with text_io;
procedure hello is
begin
    text_io.put_line(“Hello, world.”);
end hello;
```

Figure 2-4 ”Hello, world” Source

2.5.3 Compiling and Linking the Source File

Invoke the compiler with the `-M` option, which produces an executable file using the named unit as the main program. The `-o` (output) option names the resulting executable file as specified (`hello.outhello` in this case) instead of giving it the default name of `a.out`. Use the `-v` (verbose) option, to display compilation information as the unit compiles.

```
% ada -M hello.a -o hello.out
```

Provided your source file contained no typing errors, the compiler produces an executable file named `hello.out.`

Note – Compiling with the `-O0` (no optimization) option may simplify debugging, but isn't required. For example,

```
ada -O0 -M hello.a -o hello.outada -O0 -M hello.a -o hello
```

References

`ada`, *SPARCompiler Ada Reference Guide*

Chapter 3, "Compiling Ada Programs"

2.5.4 Running the Executable Program

Now that you have an executable program, run it by entering the name of the executable file:

```
% hello.out
```

`hello.out` produces the following result:

```
Hello, world.
```

References

Chapter 4, "Linking and Executing Ada Programs"

2.6 Running the Debugger

Here we show what it looks like to run the debugger on this same program. Invoke the debugger:

```
% a.db hello.out
```

The debugger responds:

```
Debugging: /usr2/mylib/hello.out
SCAda_library: /usr2/mylib
library search list:
        /usr2/mylib
        /usr2//self/verdixlib
        /usr2/ada_2.1/self/standard
```

We can display the program and execute it. To display the program, enter the `l` (lower-case L) debugger command at the > prompt:

```
> l
```

The debugger responds by displaying the program in this format:

```
2* procedure hello is
3  begin
4    text_io.put_line("Hello, world.");
5< end hello;
>
```

Execute the program from within the debugger by entering the **r** debugger command at the > prompt:

```
> r
```

The debugger responds by displaying the executable name, the output from the program and the program exit status, as follows:

```
hello.out

Hello, world.
process has exited with status 0
reloading /usr2/mylib/hello.out
```

Finally, exit the debugger by entering the debugger command `quit`, as shown here:

```
> quit
```

References

`a.db`, *SPARCompiler Ada Reference Guide*

Chapter 5, “Debugging Ada Programs”

Chapter 6, “X Window Debugging”

”Well begun is half done.”

Horace

Compiling Ada Programs 3

This chapter describes the use of the compiler and compiler processing. It describes how to use `a.error` to examine different types of error messages. Lock files and assembler output are discussed at the end of the chapter.

3.1 Invoking the Compiler

Ada source files must compile in an SC Ada library directory.

The basic syntax for invoking the SC Ada compiler is:

```
ada [options] ada_source... [linker_options] [object_file.o] ...
```

Separate each option (and possible arguments) with spaces. For example:

```
ada -v -w myfile.a
```

References

“Creating an SC Ada library” on page 2-6

`a.mklib`, *SPARCompiler Ada Reference Guide*

3.1.1 Invocation Examples

By default, compilation occurs in the current working directory, which must be an SC Ada library. To compile a source file that resides in the current working directory, use the command:

```
ada myfile.a
```

From the current directory, compile a source file that resides in another directory by including the full path for the source file, as shown in the following example. The source file is not moved.

```
ada /path/myfile.a
```

By using the `-L` *`library_name`* option you can compile in an SC Ada library other than the current working directory. The command

```
ada -L SCAda_library myfile.a
```

compiles the source file `myfile.a` (located in the current working directory) in *`SCAda_library`*.

By providing a full path for the source file and using the `-L` *`library_name`* option, compile a source file from a directory that neither contains the source nor is an SC Ada library. The command

```
ada -L SCAda_library /path/myfile.a
```

compiles `myfile.a` (located in the directory `/path`) in *`SCAda_library`*.

3.2 Compiler Processing

For each compilation unit, the compiler first analyzes the source code to ensure that Ada lexical and syntax rules are observed. The compiler determines which additional units are needed from the current library or other SC Ada libraries to supply any required separate compilation information.

For instance, in the `fact.a` and `prob.a` examples used in *Unit Dependencies* on page , the file `fact.a` containing `FACTORIAL` is compiled. A record is made in the containing library that a function named `FACTORIAL` can be called and that `FACTORIAL` expects an `INTEGER` argument and returns an `INTEGER` result. A compilation of the file `prob.a` uses FACTORIAL. Because the Ada source file `prob.a` references a compiled Ada unit `FACTORIAL`, the command

```
ada prob.a
```

requires the compiler to find the unit `FACTORIAL` before compilation of `PROBABILITY`.

The compiler searches the current Ada library for an entry for `FACTORIAL` in `ada.lib`. If the entry is not found, the compiler searches, in turn, each of the additional Ada libraries identified in the library search list. If it cannot find FACTORIAL, the compiler reports an error. If an entry is found, the compiler identifies the file in the `.nets` subdirectory containing the corresponding separate compilation information for `FACTORIAL`. If this file does not exist or is damaged, the compiler reports an error.

The compiler then proceeds with a semantic analysis of the current unit and reports semantic errors as appropriate. If no errors exist, the compiler generates code and the library directory is updated. Note that the library is updated both with the separate compilation information and the linkable object code on a unit-by-unit basis. Thus, as Ada requires, even if some units in a multi-unit compilation have errors, those units without errors are compiled and placed in the library. Most users prefer to keep one unit in each file, to avoid such inconsistencies.

The compiler generates object files compatible with the host linker.

3.2.1 Separate Compilation Information

SC Ada maximizes the economy that separate compilation brings to a large programming effort. With SC Ada , subdivide a large program, develop it as discrete units and compile each unit separately. Compile individual units before the project is complete. Recompilation of individual units is done without affecting non-dependent units. Separate compilation reduces software development time significantly.

Ada encourages separate compilation and defines library mechanisms to deal with unit dependencies and subprogram specifications and bodies in separate files. SC Ada provides simple tools for referencing compiled units in other libraries and enables large programs to be designed with a modular approach. Programmers can work on one facet of the project using units from libraries other than their own. When the local portion of the project is completed, other users can access the current library with SC Ada library tools.

SC Ada complements these Ada features by providing a program library system that enables programmers to access units that do not reside in their local library. Ada rules governing unit dependencies still apply. Each SC Ada library has a list of other SC Ada libraries to search when a unit cannot be found locally. This information is in the library search list of the `ada.lib` file. The specified libraries are searched in the order listed in the library search list.

Compilation units in an Ada source file are compiled in the order in which they appear in the file.

3.3 Automatic Recompilation - a.make

When compiling a program that consists of multiple units, changing a unit upon which other units depend requires that the dependent units, as well as the changed unit, be recompiled in a specific order. You can manually maintain a compilation order table to determine the order of recompilation and to reconstruct the latest version of a program. But, building and maintaining a compilation order table by hand is difficult and error-prone. `a.make` automates this reconstruction by performing the minimal number of recompilations necessary to bring a unit, program, library or group of libraries up to date after source changes.

The most general form of the command is shown in this example:

```
a.make [options] unit_name
```

`a.make` determines which units must be recompiled to produce a current version of *`unit_name`*. It calls `a.ld` to create the appropriate executable file if *`unit_name`* is capable of being a 'main' unit; otherwise, it just ensures that the named unit is up-to-date, recompiling any dependencies if necessary.

`a.make` improves upon the `make(1)` utility. Both process dependency information that allows the recompilation of only those files containing out-of-date objects but `a.make` does not require a user-maintained "make file." Instead, `a.make` automatically uses the dependency information in the nets generated by the compiler to determine which units must be recompiled. For example, if unit `IMAGE` depends on `PIXEL`, `IMAGE` cannot be compiled before `PIXEL`. `IMAGE` must be recompiled if `PIXEL` is changed. As long as the dependency information is recorded by a previous compilation of each unit, the command

```
a.make image
```

determines and executes the minimal sequence of compilations necessary to build a new version of `IMAGE` that is up-to-date with the Ada units it depends on.

`a.make` uses the files in the `.nets` directory to determine the correct order of compilation. Unless the `-f` option is used, the utility has no knowledge of any source file until that file is compiled in a way that changes the program library. Use the **-f** option to compile groups of files not previously entered in the library.

Options unknown to `a.make` are passed to `a.ld`.

References

`a.make`, *SPARCompiler Ada Reference Guide*

3.4 Compiler Optimizations

The SC Ada optimizer performs most classical code optimizations and several that are specific to Ada:

- Code straightening
- Constant folding, copy propagation and strength reduction
- Redundant branch and range check elimination
- Common subexpression elimination
- Hoisting of loop invariant computations and range checks
- Strength reduction on induction expressions within a loop
- Range propagation for elimination of constraint checking
- Elimination of assignment to dead local variables
- Address optimization exploits target addressing modes
- Subsumption of moves
- Hoisting subexpressions common to the `then` and the `else` parts of `if` statements.

In addition, the following SC Ada compiler features relate to the runtime performance of the generated code:

- Local scalar and access variables automatically allocated in registers
- Loop variables allocated in registers
- Parameters passed in registers
- Graph coloring register allocation scheme
- Code generation for math coprocessors
- Target specific peephole optimization

Insert the following pragmas in your source code to enhance performance:

`pragma OPTIMIZE_CODE(OFF|ON)` suppresses or re-enables optimization for a specific subprogram or package.

`pragma VOLATILE(`*`object_name`*`)` guarantees that references to the named object are not optimized away.

Invoke the SC Ada optimizer using the `-O` command line option to `ada` and `a.make`. The default level of optimization provides for hoisting from loops (`-O4`). Different levels of optimization are available and are discussed under the `ada` and `a.make` commands in the SC Ada *Reference Guide.*

If the `-O0` (prevent optimization) option is specified, the optimizer is not run, even if `pragma OPTIMIZE_CODE(ON)` is specified.

Note that `pragma OPTIMIZE_CODE(OFF)` overrides any `-O` directive and suppresses use of the optimizer on the enclosing subprogram. It inhibits some of the optimizations performed by the code generator. This is to prevent any optimizations on machine code procedures (the original purpose of `OPTIMIZE_CODE`).

Using the INFO directive `REGISTER_VARIABLES` may also improve performance. When this directive is set to `TRUE`, the compiler places subprogram local variables into registers whenever possible. This is independent of the SC Ada optimizer. Note that on some systems, including the SPARC, this directive cannot be "turned off" or set to `FALSE`.

Using the `-O0` option can alleviate some problems when debugging. For example, using a higher level of optimization, you may receive a message that a variable is no longer active or is not yet active. If you experience these problems, set the optimization level to 0 using the `-O0` option.

Caution – The SC Ada optimizer is enhanced to do aggressive optimizations, using range information supplied by the user. This can cause poorly-written user code to fail. This problem, caused by variables declared with specific ranges getting values outside those ranges, occurs when code is compiled with checks suppressed. If you are using `pragma SUPPRESS`, be sure that range declarations reflect the true ranges of variables.

References

`a.make`, *SPARCompiler Ada Reference Guide*

Optimization (`ada -O`), *SPARCompiler Ada Reference Guide*

`a.info`, *SPARCompiler Ada Reference Guide*

Register Variables, *SPARCompiler Ada Reference Guide*

3.5 Compiler Error Messages

The compiler writes all error messages to the standard error stream, `stderr`. Error messages are grouped into these categories:

Fatal Errors

Errors of such severity that meaningful recovery is impossible and compilation of the file stops. After checking that the fatal error is not due to accidental misuse of SC Ada tools (e.g., attempting a compilation in a non-SC Ada library directory), report fatal errors toyour local Sun Answer Center as described in `.a.report` in the *SPARCompiler Reference Guide*.

General Errors

Semantic in nature but do not fall within a specific Ada LRM reference. The compiler does generate code and update the library for a unit that is error-free, even if other units in the file have errors.

Internal Errors

Occasionally produced by Ada programs containing unusual semantic errors but occasionally occur because of faults within the compiler. Internal errors are most often of the form:

```
internal assertion error at file my.c, line 394
```

or

```
internal case error at file my.c, line 546
```

When an internal error occurs, report its complete text and the program construct where it occurs to your local Sun Answer Center .

Lexical Errors

Errors in the formation of literals, identifiers and delimiters. The compiler performs no semantic analysis on a unit containing lexical errors and does not update the library. The compiler attempts to correct the error internally to minimize its impact on the discovery of further lexical and syntax errors.

Semantic Errors

Errors made in the usage of Ada language constructs. The compiler generates no code for units with semantic errors and does not update the library. The compiler does generate code and update the library for a unit that is error-free, even if other units in the file have errors. All semantic error messages refer to the specific section, subsection or paragraph within the Ada LRM.

Syntax Errors

Errors in the formation of grammatical constructs. The compiler performs no semantic analysis on a unit containing syntax errors and does not update the library. The compiler does attempt to correct an error to minimize its impact on the discovery of further lexical and syntax errors.

Warnings

Errors not serious enough to prevent code generation or that indicate questionable use of a construct or dangerous programming practice.

All error messages include information that shows the context of the error message. The additional context lines trace back the nested inline subprogram expansions and/or generic instantiations. The first additional message is the "highest" level and the last additional message is the place where the error occurs.

For example, in the following, the error messages that begin with **at line** indicate the context of the error. The first line traces the error back to the nested inline subprogram. The final line indicates the place where this error occurs in the program:

```
  63:      z(y);
A ---------^
A:error: should be general purpose register
A:error: at line: 26 column: 12 within subprogram z in
         file:       /vc/test/inline/bug.a
A:error: at line: 23 column: 3 within subprogram bit_and in
         file: /vc/ada_2.1/self/standard/v_bits_b.a
A:error: at line: 14 column: 23 within subprogram b_and in
         file: /vc/ada_2.1/self/standard/v_bits_b.a
```

References

reporting problems, *SPARCompiler Ada Reference Guide*

3.6 Compiler Error Message Processing

Usually, `a.error` is called directly from the compiler using the `ada` options. It reads the specified file or standard input, determines the source file(s) containing errors and processes the errors according to the options specified in the following command.

```
a.error [option] [error_file]
```

The `-e` and `-ev` options of the `ada` command automatically call `a.error` to process any compiler error messages resulting from the current compilation.

Note – `a.list` produces a program listing that closely resembles the output of `a.error` with the `-l` option for programs containing no errors.

Redirect compiler output to a file and examine it with the `a.error` utility or pipe it directly to `a.error`.

To illustrate, a file containing errors, `badtry.a`, is compiled:

```
-- file: badtry.a --
with TEXT_IO;
procedure BADTRY is
   subtype T is range 1..1f;
   COUNT : T;
   SUM : INTEGER;
   type REAL is digits 6;
   AVG : REAL
begin
   for COUNT in T loop
   SUM := SUM + I;
   end loop;
   AVG := SUM / COUNT;
   TEXT_IO.PUT(INTEGER'IMAGE(SUM));
   TEXT_IO.PUT(REAL'IMAGE(AVG));
end MAIN
```

Figure 3-1 a.error Example

The output from the compilation of `badtry.a` is redirected, using the following command from `csh(1)`:

```
ada badtry.a  >& badtry.errors
```

or the following from `sh(1)`:

```
ada badtry.a > badtry.errors 2>&1
```

The file `badtry.errors` contains the following messages. Each line that contains an error is listed and followed with a description of the error.

```
/usr2/babbage/badtry.a, line 4, char 25: lexical error:
deleted "f"
/usr2/babbage/badtry.a, line 4, char 15: syntax error:
"identifier" inserted
/usr2/babbage/badtry.a, line 9, char 1: syntax error:     ";"
inserted
/usr2/babbage/badtry.a, line 16, char 5: syntax error: "main"
replaced by ";"
```

Figure 3-2 a.error Messages

If you invoke ada using the -ev option, ada calls a.error with the **-v** option. a.error writes the error messages directly in the original source file as comments and calls the vi text editor. All error message lines are prefixed with two hyphens (--) that denote an Ada comment. Line numbers are suppressed, error messages are marked with the pattern ### and the editor is positioned in the file with the cursor at the point of the first error. The error description includes one or more error messages that begin with a capital letter and point to the place in the program where the error is detected. Subsequent lines beginning with corresponding letters provide brief synopses of the errors encountered. The following code segment illustrates the file badtry.a with error messages embedded.

```
  -- file: badtry.a --
     with TEXT_IO;
  procedure BADTRY is
     subtype T is range 1..1f;
----------------------^A                    ###
----------------------------^B              ###
--### A:syntax error: "identifier" inserted
--### B:lexical error: deleted "f"
     COUNT : T;
     SUM : INTEGER;
     type REAL is digits 6;
     AVG : REAL
  begin
--<<A                               ###
--### A:syntax error: ";" inserted
     for COUNT in T loop
     SUM := SUM + I;
     end loop;
     AVG: = SUM / COUNT;
     TEXT_IO.PUT(INTEGER'IMAGE(SUM));
     TEXT_IO.PUT(REAL'IMAGE(AVG));
  end MAIN;
------^A                                                     ###
--### A:syntax error: "main" replaced by ";"
```

Figure 3-3 a.error Output

The file contains four lexical and syntax errors. First, an identifier naming a type is omitted before the keyword `range`. The compiler continues as though this identifier is inserted but does not edit the original source file. The next error is a lexical error, resulting from the malformed integer literal '1f'. The compiler continues as though the 'f' is deleted. The remaining error messages show that a semicolon must precede `begin` and that the designator after `end` has a different name than is given to the subprogram.

Error messages are marked with ### so that errors are located easily and subsequently deleted. Even if `a.error -v` or `a.error -e` intersperses error messages into a file, the compiler can process that file without deleting the error messages. Because **-v** places the error messages directly in the source file, if **a.**`error -v` is called again before the messages are deleted or the error corrected, a second copy of the same messages appears.

Since all error lines are flagged with ###, you can use the `vi` command

```
:g/###/d
```

to delete them. Any source lines with ### in them are also deleted; consequently, it is advisable not to use ### in your source code.

Edit the file `badtry.a` to repair the lexical and syntax errors and resubmit it to the compiler. If those errors are fixed correctly, semantic analysis can proceed.

3.7 Lock Files

Because the compiler updates libraries as part of the compilation process, it must prevent other compilations from accessing a library during certain portions of the procedure. Usually, this feature is invisible to the user, although it causes longer compilation times if several compilations are sharing libraries.

System failures or other operational disruptions cause the locking mechanism to remain in effect after compilation is aborted. The mechanism consists of advisory files whose presence prevents some part of the compiler from writing to a specific library file. If the compiler finds a lock file (files ending in `.lock`) in the SC Ada library, it waits and looks again. If after several such tries the lock file remains, the compiler issues a message such as this:

```
/usr2/babbage/termspec.a, line 170, char 4: fatal: timeout.
Library ada.lib is locked (lock file is /usr2/babbage/ada.lock)
```

The compiler message names the specific lock file at the end of the message. To proceed, remove the lock file and restart the compilation:

```
% rm ada.lock
% ada termspec.a
```

Note – The lock file can be there correctly if other users are compiling files in the same SC Ada library. Before removing the lock file, check the active processes to ensure that this is not the case.

3.8 Assembler Output

Although the compiler does not produce assembly-code listings, `a.das` produces an assembler listing of a compiled unit. A similar listing is obtained interactively, using the debugger `a.db`.

References

`a.das`, *SPARCompiler Ada Reference Guide*

Section 5.16.3, “Instruction and Source Modes,” on page 5-49

3.9 Compiling Generics

Ada generics are program templates. The template looks like a normal Ada package or subprogram, except the specification is preceded by a list of the generic formal parameters. These parameters can be types, objects or subprograms.

An executable instance of a generic, an instantiation, is created by providing values for the formal parameters. Within the instance, the actuals are substituted in place of the formal parameters.

Most compilers generate a new body of code for each instantiation of a generic. While this allows for maximum optimization of the generated code (since optimizations can be performed based on the values of the generic actual parameters), it is potentially very expensive in terms of the space consumed.

SC Ada provides an alternative approach in which a single body of code is shared by several instances of a generic. Sharing generic bodies reduces code size dramatically when multiple instances occur in a single program. It reduces compilation time for units instantiating the generic and reduces recompilation time when the generic body is changed. The trade-off is that sharing generic bodies in this way results in a slight time increase for the execution of the generic.

The SC Ada compiler does not attempt to share all generics. The runtime cost of shareable code for generics with private type parameters is unreasonably high. In general, code can be shared for generics that have discrete type parameters, object parameters and subprogram parameters.

Recompilation time is reduced by separately compiling generic specifications and bodies. Under most circumstances, the support provided by SC Ada for separate compilation and sharing is transparent to the end user.

The remainder of this section discusses how SC Ada manages generics. Some of the visible effects of shared generics and how to deal with circumstances that require user intervention are explained.

Every SC Ada library contains a file named `gnrx.lib` which is used by SC Ada to manage generics and instantiations. Entries in this file contain information about generic specifications, generic bodies and instantiation requests.

A shared generic body is created automatically by a compilation when a shareable generic is instantiated and a shareable body cannot be found in any library on the current library search list. During a verbose compilation, the shared body is identified by a symbol formed by appending a dollar sign and a number onto the end of the generic name. The command `a.ls` lists the shared generic by the name of the generic. Note that no association exists between the shared generic and the instantiation that created it, except that they reside in the same library.

Changing the generic body creates a new “generic template”. Thus, all instantiations of the generic must be updated.

As part of its support for separate generic compilation, SC Ada updates the generic instantiations instead of requiring recompilation of the units containing the instantiations. Furthermore, in the case of shared generics, only the single shared body is updated, rather than each instantiation. Thus, SC Ada support for shared and separately compiled generics eliminate hours of unnecessary recompilations.

SC Ada automatically updates unshared instantiations and shared bodies that are in the same library as the generic body (and lists these updates when the generic body recompiles). All instantiations sharing a body are implicitly updated when a shared body is updated, including instantiations in other libraries.

However, if a shared body is in a library other than that of the generic body, it is not visible when the generic is recompiled and the shared body becomes out-of-date. A similar out-of-date condition occurs if non-sharing instantiations of the generic exist. If an attempt is made to link a program that uses an out-of-date instantiation, the link fails due to unresolved references within the shared body or the body of the unshared instantiation.

If `a.make` is used to build the program, it detects the out-of-date body and updates the shared body automatically. Alternatively, use the `ada` command directly; the `-R` option instructs the compiler to examine the shared generics in the library and update any that are out-of-date with respect to their associated generic body.

Be aware of this potential condition and its tell-tale symptom, undefined references in the body. Generally, it occurs when generic bodies are kept in a separate library from instantiations. Use of `a.make` or the compiler recompile library option to correct this situation can save hours of unnecessary recompilations.

References

`pragma SHARE_CODE`, *SPARCompiler Ada Programmer's Guide*

Generic Declarations, *SPARCompiler Ada Programmer's Guide*

"We gotta go and never stop going 'til we get there.
Where we going man? I don't know but we gotta go."

Jack Kerouac

Linking and Executing Ada Programs 4

This chapter provides information about linking and executing Ada programs including the use of LINK directives, WITH*n* directives, declarations and other linker features. Detailed descriptions of the tools to link and execute programs are in the *Reference Guide*.

4.1 Linking and Executing - A Quick Overview

This section provides a brief overview of the process to link and execute an Ada program. A more detailed discussion of many of the aspects of linking and executing programs in SC Ada follows this section.

4.1.1 Invoking the linker

Invoke the linker, `a.ld` using the following syntax:

```
a.ld [options] unit_name [ld_options]
```

`options` (before *`unit_name`*) are options to the linker itself. *`unit_name`* is the name of a parameterless main subprogram and must name a non-generic subprogram. If *`unit_name`* is a function, it returns a value of `type STANDARD.INTEGER`.

`ld_options` are options to the host operating system linker.

Place options for the `ld(1)` linker after *`unit_name`*. The options can refer to archive libraries, option switches, object files or library abbreviations. For example, the `a.ld` command for a compiled Ada unit that makes use of `pragma INTERFACE` and references C functions can look like this:

```
a.ld my_unit c_function.o -o my_unit
```

References

`a.ld`, *SPARCompiler Ada Reference Guide*

SPARCompiler Ada User's Guide, Chapter "Debugging Ada Programs, page-1

4.1.2 Steps in the Linking Process

Producing an executable Ada program from object code is a two step process: prelinking and linking. The command `a.ld` performs the prelinking and then invokes the linker. `a.ld` computes a list of units on which the main unit depends and then verifies that all units are up to date. Finally, `a.ld` gathers elaboration and exception handler information into an object file, builds the command to invoke the linker and invokes the linker.

4.1.2.1 Prelinking

Prelinking involves first building a list of all the Ada units that must be linked into the executable program. `a.ld` begins by putting the name of the main Ada unit (the only `a.ld` invocation parameter required) on the list. `a.ld` then adds the names of all the units required by the main unit to the list. An Ada unit depends on all units that it `with`s. In this code segment for example:

```
with text_io;
with global_types;
with sort;
procedure main_prog is
begin
    ...
end;
```

The main Ada unit is called main_prog and depends on the units text_io, global_types and sort. If any of these units is a package specification, then a.ld adds the names of both the package specification and the package body to the list. If a unit contains subunits, a.ld adds the name of each subunit to the list.

After a.ld builds the list of units on which the main unit directly depends, it reads the name of the next unit on the list and determines which units it requires. a.ld adds to the list the names of those units not on the list. This process continues until every unit on the list is examined and the names of all withed units are added to the list. This process is called computing the *transitive closure* of the units on which the main unit depends.

As a.ld builds the transitive closure list, it verifies that all units are up-to-date. In the preceding example, this means that text_io, global_types and sort must be compiled prior to compiling main_prog.

a.ld now constructs two tables and creates an object file in which to place the two tables. The object file has the same name as the main Ada unit but with a .o extension and resides in the .objects directory.

For example, if you invoke a.ld with this command

```
a.ld my_main
```

the prelinker creates the file .objects/my_main.o to hold the two tables.

One of the tables contains a list of all the library units that must be elaborated. When constructing this table a.ld determines the correct elaboration order of the library units. The second table contains the exception information for each unit, which is sorted and compacted by a.ld to speed up exception processing during program execution.

For each Ada unit on the transitive closure list, a.ld adds an object file (if required) to the list of files that must be linked to produce an executable file. a.ld scans the Ada library to find active LINK directives and to incorporate the corresponding library files and linker options into the linker invocation. Any parameter option that a.ld does not recognize is passed to the linker.

The order of object files in the list created by `a.ld` is:

- Object file named in the `STARTUP` LINK directive
- Object file containing elaboration and exception tables (created by `a.ld`)
- Object files corresponding to the transitive closure of the main unit
- Options and objects from `pragma LINK_WITH`
- Options and objects from WITH*n* directives
- Options and objects from the command line
- Runtime library archive

If any of the units being linked require tasking, the runtime library archive is specified by either the `MIN_TASKING` or the `TASKING` LINK directive. In all other instances, the archive is specified by the LIBRARY LINK directive. The `MIN_TASKING` LINK directive is selected when the program does tasking but does not have any `abort` or `select with terminate` statements.

References

Subunits of Compilation Units, *SPARCompiler Ada Reference Guide*

Section 4.2.2, "Directives that Affect Linking," on page 4-8

4.1.2.2 Linking

As mentioned in the preceding discussion of the prelinker, the last task that `a.ld` performs is invoking the host OS linker, `ld(1)`, to perform the link. Normally, you need not look at the invocation command for the linker but if you want, use the **-v** option to `a.ld`. After you save the output from the **-v** option to a file, edit the file to create a shell script that invokes the host OS linker.

To provide the host OS linker with a list of the required object files, `a.ld` must list them on the invocation line. host OS systems limit both the number of arguments and the number of characters on a command line. When `a.ld` invokes the linker, if an invocation violates either of these limits `a.ld` creates an archive, using the host OS archiver `ar(1)` and `ranlib(1)` to contain all of the object files. (Because of the previously mentioned host OS limitations, `a.ld` probably must invoke `ar` more than once to create the archive.)

After the archive is created, `a.ld` runs `ranlib` to convert the archive to a library and invokes the host OS linker, using the name of the archive file as one of the parameters. If you use the `-v` option to `a.ld`, all of the commands used to create the executable (`ar`, `ranlib` and `ld`) are printed on standard output.

One reason that you may want to use a shell script to invoke the linker directly is that certain errors that occur during linking, undefined symbols, for example, are sometimes easier to pinpoint if you exercise more control over the linking process.

4.1.3 Executing a Program

Execute a compiled program in Solaris by using the program name. If the executable image is not in the current directory (which must be on the user path), invoke the executable by either including the directory that contains the executable in the shell PATH variable or by typing the full path name of the executable:

```
/usr2/ada_progs/a.out
```

4.1.4 Display Command Line Options and Environment Variables

Sun Microsystems, Inc., provides an Ada interface to the command line arguments and to the environment variables. This interface is provided in `package U_ENV` in `standard`. Using this package, it is possible to display the command line arguments to a program and the current values of any or all of the environment variables.

The following short program illustrates the use of this package. It displays the value of the environment variable PWD, the command line options and all the environment variables.

Sun Microsystems, Inc., provides an Ada interface to the command line arguments. This interface is provided in `package U_ENV` in `standard`. Using this package, it is possible to display the command line arguments to a program.

The following short program illustrates the use of this package. It displays the command line options.

```
with text_io; use text_io;
with u_env;
with c_strings; use c_strings;
procedure test_u_env is
    arg:c_string;
    value:c_string;
begin
    arg := to_c ("PWDHOME");
    value := u_env.getenv (arg);
    put_line (to_string (value));
    put_line("Command line arguments --");
    for count in 0 .. u_env.argc -1
    loop
    put_line(integer'image (count) & ": " &
            """" & u_env.argv(count).s & """");
    end loop;
    new_line;
    put_line("Environment variables --");
    for count in 0 .. u_env.envc -1
    loop
    put_line(integer'image (count) & ": " &
            """" & u_env.envp(count).s & """");
    end loop;

end test_u_env;
```

Figure 4-1 Use of package U_ENV

References

`package U_ENV`, *SPARCompiler Ada Reference Guide*

4.2 Linking in Detail

4.2.1 Library Unit Elaboration

Elaboration occurs when an initialization of a static variable cannot be done as constants at compile time, for tasks and when the user gives explicit package initializations. Supplies pragmas to allow some control over elaboration:

```
pragma INITIALIZE(STATIC | DYNAMIC);
```

The pragma can appear only in a library package specification or body. It indicates that all objects in the package specification should be statically / dynamically initialized. If static initialization is desired and an object cannot be initialized statically, code is generated to initialize it, and a warning message is displayed.

```
pragma NOT_ELABORATED;
```

Can only appear in a library package specification. It indicates the package will not be elaborated. It is used for the RTS, for the configuration package or when the main program is written in another language. It may be used carefully in other circumstances. It suppresses the generation of elaboration code and issues warnings if elaboration is required.

Elaboration is done by a subprogram created for this purpose. A list of all the elaboration subprograms is compiled and sorted into an acceptable elaboration order by the prelinker (`a.ld`). The elaboration order calculation is an overhead for the prelinker, but results in relatively little runtime overhead. Note that each elaboration routine is called just once.

The elaboration order is not guaranteed to be totally accurate. It is based on information about which units `with` which other units, rather than which units actually reference potentially unelaborated entities during their own elaboration.

In some cases elaboration cannot be known at compile time. In these cases, a bit is used for each entity; initially unset and then set when it is elaborated. These elaboration checks will trigger `PROGRAM_ERROR` at runtime if they fail. They introduce an overhead of a bit test and conditional branch.

4.2.2 Directives that Affect Linking

To generate executable files `a.ld` reads directives from the Ada library, which includes each of the `ada.lib` files on the library search list. LINK directives control items that are linked into the executable file. Add and remove directives from the library by using the tool `a.info`. The following subsections describe the directives that are specific to the prelinker and provide syntax examples for each of the directives.

4.2.2.1 LIBRARY, TASKING and MIN_TASKING Directives

The LINK directive `LIBRARY` provides the name of the file that contains the Runtime System library. The syntax for the LINK directive `LIBRARY` is

```
LIBRARY:LINK:filename:
```

where *`filename`* is the name of the runtime system library.

The LINK directive `TASKING` provides the name of the file that contains the Runtime System library for programs that use tasking with either an `abort` or `select with terminate` statement. The syntax for the LINK directive TASKING is

```
TASKING:LINK:filename:
```

where *filename* is the name of the runtime system library for programs with tasking.

The LINK directive MIN_TASKING provides the name of the file that contains the Runtime System library for programs that use tasking but don't have any `abort` or `select with terminate` statements. The syntax for the LINK directive `MIN_TASKING` is

```
MIN_TASKING:LINK:filename:
```

where *`filename`* is the name of the runtime system library for programs using minimal tasking.

`a.ld` automatically chooses the correct Ada Runtime System library, `LIBRARY`, `TASKING` or `MIN_TASKING` and passes this choice to the linker. All three of these directives are supplied in the `ada.lib` for `standard`, as part of this release.

Normally, modifying this directive is not necessary; the RTS archive files supplied with the release are suitable for building the majority of programs. You may need to change the directives if you purchase the Runtime System source code and are modifying that source. When this occurs, you must build new archives and consequently, must change the directives to point to the new archive files.

To add or modify LINK directives, use the `a.info` command.

References

`a.info`, *SPARCompiler Ada Reference Guide*

4.2.2.2 WITHn Directives

The `WITH`*n* directive enables you to include additional object files and archives to pass to the linker. However, we discourage the use of WITH*n* directives because the process is error prone. `pragma LINK_WITH` provides the same functionality as `WITH`*n* directives but in a cleaner way.

WITH*n* directives enable the automatic linking of object modules compiled from other languages or Ada object modules for units that are not named in the context clauses (with statements) of the Ada source code. WITH*n* directives appear in the `ada.lib` file and have this syntax:

```
WITHn:LINK:object_file:
WITHn:LINK:archive_file:
WITHn:LINK:linker_option:
```

where *n* is a number.

Place any number of `WITH`*n* directives into a library but they must be numbered contiguously beginning at `WITH1`. Any break in the sequence `WITH1`, `WITH2`, `WITH3`, ...stops the prelinker from processing `WITH`*n* directives. If the directives have the names `WITH1`, `WITH2`, `WITH4` and `WITH5`, only `WITH1` and `WITH2` are processed because `WITH3` is missing; `WITH4` and `WITH5` are not found.

To add or modify the `WITH`*n* directives, use the `a.info` command.

Place `WITH`*n* directives in the local library or in any library on the search list. A `WITH`*n* directive in a local library or earlier on the library search list hides the same numbered WITH*n* directive in a library later in the library search list. This is reported as a warning by `a.ld`.

References

a.info, *SPARCompiler Ada Reference Guide*

4.2.3 Linking Foreign Object Files

A foreign object file is any object file that is not automatically included in the linking process by `a.ld`. This includes object files created by non-Ada compilers, such as C, Fortran, assemblers or even object files created by the compiler.

To link foreign object files with an Ada program, introduce the files into the linking process by using either `pragma LINK_WITH` or `WITH`*n* directives (`pragma LINK_WITH` is preferred).

`a.ld` does not automatically link object files containing subprograms or objects referenced by `pragma INTERFACE`. You must place appropriate `pragma LINK_WITH` calls in the source with the `pragma INTERFACE` to tell where to get the foreign object file. `pragma LINK_WITH` documents the name of the foreign object file for anyone who reads the source code.

Include foreign object files in the linking process by using either pragma `LINK_WITH` or `WITH`*n* directives. Ada modules included in this way must contain `pragma NOT_ELABORATED`, because these modules are not elaborated and must not depend on elaboration code. `pragma NOT_ELABORATED` suppresses the generation of elaboration code and causes a warning if elaboration is required.

References

Pragmas and Their Effects, *SPARCompiler Ada Programmer’s Guide*

Interface Programming, *SPARCompiler Ada Programmer’s Guide*

”How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth.”
Conan Doyle

Debugging Ada Programs 5

The SC Ada symbolic debugger debugs Ada programs at the source level. It can aid in understanding code written by others. Basic `a.db` features, concepts and commands are presented.

`a.db` includes a line-mode and a screen-mode interface. The screen-mode debugger provides a more convenient interface for most program debugging at the source level. All commands for the line-mode debugger are available in screen mode by prefacing them with a colon. The colon serves as a line-mode escape from screen mode. Also, many line-mode commands are directly available in screen mode.

This chapter provides a functional introduction and an explanation of most commands; the *SCAda Reference Guide* provides detailed information.

Commands are discussed in this order:

- display commands
- execution commands
- breakpoint commands
- positioning commands
- screen-mode commands

Note – Documentation for `a.db` refers frequently to the `INTERRUPT` key (`<INTR>`). The system command `stty` enables this function to be assigned to any convenient key (often `<CONTROL-C>`).

References

Debugger Reference, *SPARCompiler Ada Reference Guide*

Operating system documentation, *SPARCompiler Ada Reference Guide*

`stty`, *Solaris Developer Documentation*

5.1 Using the Debugger As a Learning Tool

The SCAda symbolic debugger is a tool for finding runtime errors in programs but its ability to relate the source code to the flow of execution makes it a useful tool when modifying or debugging code that someone else has written. The following paragraphs outline an approach to using the debugger in screen mode that provides a 'window' through which to view the operation of another programmer's code.

If the Ada libraries containing the source code are on the library search list, use the debugger `e` (enter) command to view routines without needing to determine the location of their source files. Use the `s` (step) and `a` (advance) commands to walk through the code as it executes. This lets the programmer read through the source code in execution order and limits the code to be read. If a subprogram is stepped into accidentally, use the `bd` (break down) command to set a breakpoint immediately after the call to the subprogram, followed by the `g` (go) command to reach the breakpoint.

As the programmer begins to see how the code works, some portions require more careful examination. The `b` (set break) and `d` (delete break) commands allow selection of areas that can be reached quickly using either the `g` (go) or `r` (run) command.

When a breakpoint is reached, the current state of the program can be examined. The print commands `p` and `P` print the values of entire variables or selected components of variables. Use the `cs` (call stack) command to display the current call stack and the values of the arguments passed to each routine. The `cd` (call down) and `cu` (call up) commands can move the entire debugging context to another routine on the call stack so that the source code and variables in those routines can be examined. After moving along the call stack, use the `ct` (call top) command to return to the top of the stack.

5.2 Invoking the Debugger

Invoke the debugger by issuing this command:

```
a.db [a.db_options] [executable_file
[executable_file_options]]
```

If no options are specified, `a.db` begins in line mode. If no *`executable_file`* is specified, the debugger searches for the executable file `a.out` searches for the executable file `a.out`.

In addition to the invocation line, supply parameters to `a.db` with `set` commands in an invocation file.

`a.db` is a 'wrapper' program that executes the correct debugger executable based upon directives visible in the `ada.lib` file. This enables multiple SCAda compilers to exist on the same host. Only one SCAda `bin` directory need be listed in the user PATH variable. To change to another SCAda version on the same host, execute `a.db` in a library created for a particular release.

Note – Compile with the `-O0` (no optimization) option for full symbolic debugging.

References

Section 5.11, "Modify Debugger Configuration: set," on page 5-17

5.2.1 Command File Input: -i

The `-i` option tells the debugger that its initial input is not from the keyboard but is instead from a file. When the debugger commands in this file are all read and executed, the debugger switches to reading from the keyboard unless the file contains a `quit` or `exit` command. The commands can be any line-mode debugger commands.

References

debugger invocation, *SPARCompiler Ada Reference Guide*

5.2.2 Ada Library Directory: -L

If the executable program is an Ada program, the debugger needs the name of the Ada library directory where the program is built. Normally programs are debugged from within the Ada library where they are built, so the debugger checks the current working directory to see if it is an Ada library directory. If the current working directory is not an Ada library directory or if it is not the Ada library directory where the program is built, supply the debugger with that information with the **-L** option.

If the current working directory is not an Ada library and no **-L** option is used, the debugger searches the directories in a set source command, obtained by reading the `.dbrc` file, for the first directory that contains an `ada.lib` file.

References

debugger invocation,*SPARCompiler Ada Reference Guide*

5.2.3 Executable File

If an executable file is given, the debugger uses the host OS `PATH` environment variable to find its full path name if the full path is not specified. Note that if the current directory is not in the host OS `PATH` variable, executables in it cannot be found without specifying a full or relative pathname.

If no executable filename is given at invocation, `a.db` debugs `a.out` by default.

References

debugger invocation, *SPARCompiler Ada Reference Guide*

5.2.4 Display Debugger Executable: -sh

The `-sh` option prints the name of the actual debugger executable file. Based on directives in the `ada.lib` file, the wrapper program, *`SCAda_location`*`/bin/a.db` finds the correct debugger for the Ada library.

References

debugger invocation, *SPARCompiler Ada Reference Guide*

5.2.5 Screen-mode Invocation: -v

Use the `-v` option to invoke the debugger in screen mode. When in screen mode, `Q` returns to the line mode.

The screen-mode debugger provides a more convenient interface than the line-mode interface for most program debugging at the source level. All commands discussed for the line-mode debugger are available in screen mode as well by prefacing them with a colon.

The screen-mode interface includes additional commands that do not require a preceding colon.

References

Section 5.16.4, “Screen-mode Window Commands,” on page 5-50

screen mode, *SPARCompiler Ada Reference Guide*

5.2.6 Redirecting Debugger Input and Output

Normally the debugger reads from the terminal. By using the following redirection options to `a.db`, redirect standard input, standard output and standard error to a file.

`<` *`filename`*	`Direct input to the debugger from` *`filename`*`.`
`>` *`filename`*	*`Direct output from the debugger to filename.`*
`>& filename`	`Direct debugger output and error messages to` *`filename`*`.`

Two restrictions to using redirection are:

- You cannot use screen mode when debugging input is a file.
- Your program cannot share the debugger input file. Use `set input` *`filename`* or `set run <` *`filename`* to set the input file for your program. A sample `debug.in` file might be:

```
set input my_prog.in
r
quit
```

Run the debugger in the background by appending "&" to the invocation line. For example:

```
    a.db my_prog < debug.in >& debug.out &
```

Or, if `my_prog` has input parameters, use the debugger `-r` option:

```
a.db -r "my_prog my_prog_options" < debug.in >& debug.out &
```

The `-r` option provides a means to disambiguate options to the debugger and options to the executable file as they are interpreted by the shell on subsequent invocations of the debugger. Enclosing the shell commands that pertain to the executable file within the quotes ensures that they are not interpreted by the shell to apply to the debugger.

References

redirecting debugger input and output, *SPARCompiler Ada Reference Guide*

screen mode, *SPARCompiler Ada Reference Guide*.

5.3 Halting the Program”, ”stopping the program being debugged”

Stop the program being debugged by setting a breakpoint, or if the program is running, press the key mapped to `<INTR>`.

References

stop the program, *SPARCompiler Ada Reference Guide*

terminate debugger session (`exit`), *SPARCompiler Ada Reference Guide*

(`quit`), *SPARCompiler Ada Reference Guide*

5.4 Terminating the Debugger Session

Type `quit` or `exit` in line mode or `:quit` or `:exit` in screen mode to leave the debugger.

References

terminate debugger session (`exit`), *SPARCompiler Ada Reference Guide*

terminate debugger session (`quit`), *SPARCompiler Ada Reference Guide*

5.5 On-line Debugger Help

Access on-line help for `a.db` and SCAda utilities during a debugging session by entering the following command:

```
>help [subject]
```

If the subject is omitted, a list of debugger commands for which help is available is displayed. Obtain this overview by typing `intro` after a help prompt. Output is automatically paged.

References

help command, *SPARCompiler Ada Reference Guide*

5.6 Command Syntax

Most debugger commands are of the form

```
keyword [parameters]
```

In line mode, place a list of commands on a single line by separating them with semicolons as illustrated here:

```
bd; g
```

In screen mode, this command line must begin with a colon.

Commands are executed when `<RETURN>` is pressed. An exception is a breakpoint command followed by a block of commands where the breakpoint is set after the final `<RETURN>`.

The debugger uses Ada syntax for comments, that is, characters between the double hyphen (`--`) and `<RETURN>` are ignored.

While in line mode, use `<RETURN>` to repeat the most recent of several commands (`a`, `ai`, `s`, `si`, `l`, `li`, `/` or, `?`). Debugging a program with a `r` (run) or `g` (go) command clears `<RETURN>` until one of the repeatable commands is used again. Each command that is repeated this way is marked with the phrase "<RETURN> repeats" in the documentation. In screen mode, type a period to repeat the previous command.

The debugger expects data addresses and instruction addresses to be hexadecimal numbers. When a hexadecimal number might be interpreted as a decimal number or as an identifier, use a leading zero.

Print or modify register values. To specify a register, precede the register name with a dollar sign. For example: `$g1$GR0`.

Filenames used with the debugger commands `e`, `vi`, `edit`, `r` and `set` can be enclosed in double quotes (`"/usr2/src/foo.a"`). Quotes are required around simple filenames that contain characters having some special meaning in the shell. The debugger interprets `csh(1)` and `ksh(1)` tilde notation and shell environment (exported) variables for filenames.

When a name can refer either to a debugger keyword or a variable name, the debugger interprets it as a keyword. To force interpretation as a variable, precede the name with a backslash.

Control characters have their usual meaning. In addition, the interrupt signal `<INTR>` interrupts the program being debugged if it is running or it interrupts the current debugger operation. The debugger immediately responds to interruption with a prompt for the next command.

References

command syntax, *SPARCompiler Ada Reference Guide*

Section 5.12.6, "Display Register Contents: reg," on page 5-28

5.7 Debugger Concepts

Using a symbolic debugger effectively depends on understanding several language and implementation concepts, including the notion of a program call stack, name overload resolution and visibility rules for language objects.

5.7.1 Call Stack, Home and Current Positions

To represent the current state of the program being debugged, `a.db` uses a model known as the *call stack*. The call stack represents all currently active subprograms in the program being debugged. These are subprograms that are called but have not returned to their caller. When the program is executing, the subprogram that is currently executing is at the top of the call stack. A subprogram call 'pushes' the called subprogram on top of the stack. When a subprogram returns to its caller, the returning subprogram is 'popped' from the top of the stack.

When the program being debugged halts at a breakpoint or after a single step, the subprogram containing the point of execution where the program stops is the top of the call stack. The debugger provides a command `cd` (call down) that lets the user 'move' down the call stack, that is, from the current subprogram to the subprogram that called the current one. The following commands are for moving up (`cu`), to the top (`ct`), to the bottom (`cb`) and displaying the call stack (`cs`). As is explained, changing the current level on the call stack changes the variables that are directly visible.

When the program stops, the debugger initializes two 'positions' — the `home position` (execution position) and the `current position` (viewing position). The `home position` represents where the program executes next (or where the program is executing) at the current level of the call stack. The `current position` represents that part of the source code seen on the terminal at this moment. The viewing position changes as debugger commands display different source files or disassemble part of the program. The execution position only changes when moving up and down on the call stack. Whenever the execution position changes, the viewing position changes to match it.

The purpose of the `home position` is to show exactly where execution is at the current level of the call stack. When displaying a line of source code or machine instructions containing the home position, the debugger shows a ***** in the left margin next to the line. In the top subprogram of the call stack, the home position is the next instruction to execute. In all other subprograms on the call stack, the home position is the source line (or machine instruction) that called the next higher subprogram.

The purpose of the current position concept is to make commands that operate on line numbers more convenient to use. Certain listing commands (`l`, `li`, `w` and, `wi`) use the current position as their default starting point. In addition, the `l` and `li` commands change the current position to be the last item listed. In this way, successive `l` or `li` commands provide a way of displaying successive sections of text or disassembled instructions. Another command using a line number is the `b` command, set breakpoint. If no explicit line number is given, the current position is used.

References

call stack,*SPARCompiler Ada Reference Guide*

current position, *SPARCompiler Ada Reference Guide*

home position, *SPARCompiler Ada Reference Guide*

5.7.2 Overload Resolution (Disambiguation)

`a.db` supports a subprogram, task or package name as the object of the **e** (enter subprogram or file), `vi` (enter screen-oriented mode) and `b` (set breakpoint) commands. When debugging Ada programs, the name supplied to these commands need not be fully qualified (the simple name `sin` can replace `math.sin`).

When a simple name is used for one of these commands, the debugger searches ‘program-wide’ to find the name. This search enables the user to set a breakpoint or enter any subprogram regardless of the current visibility rules.

If the search finds multiple definitions for the same name, the debugger issues a message displaying each of the visible alternatives. The notation is `simple_name'n`, where *n* is a number (e.g., `sin'2`). An example of the debugger message follows. A breakpoint is being set at a subprogram with the simple name `add` which is overloaded, as shown here:

```
>b add
=> add is overloaded. Use 'n to select one:
add'1 (integer, integer) return integer
add'2 (integer, integer) return real_a
add'3 (integer, integer) return float
add'4 (integer)
add'5 (float)
>b add'3
```

Figure 5-1 Overload Resolution

Append the suffix *'n* to the subprogram name to indicate precisely the name to be referenced, where *n* is one of the numbers listed in the overloading message (*b add'3*).

The debugger assigns a number to each occurrence of a simple name, starting with 1. These numbers remain constant throughout the debugging session. When the debugger finds overloading, all of the overloadings are listed.

References

overloading, *SPARCompiler Ada Reference Guide*

5.7.3 Names, Variables and Visibility Rules

`a.db` uses scope and visibility rules similar to Ada for looking up variable names. The debugger takes the current home position as the basis for looking up a variable name for the `p` or other commands. If an Ada subprogram, package or task contains the current home position, the debugger uses Ada visibility rules in looking up the name. The debugger finds the same definition for the name as the Ada compiler finds if the name is part of the program text corresponding to the current home position.

In addition to names defined in the program being debugged, the debugger recognizes the names of hardware registers when prefixed by a dollar sign. Use these names anywhere that you can use a program variable name, including as the left side of the assignment operator (`:=`).

References

names, *Ada Reference Manual*

assignment statement, *SPARCompiler Ada Reference Guide*

home position, *SPARCompiler Ada Reference Guide*

visibility rules, *SPARCompiler Ada Reference Guide*

5.7.4 User Procedure Calls

It is possible in `a.db` to call a procedure that is part of the program being debugged. This facility enables you to build customized displays for key objects, verify the status and contents of data structures and to execute interactive procedures from inside the debugger. Display the output of the procedure(s) on the screen or capture it in a log file.

User procedure calling is implemented as part of the **p** (display) command. For example:

```
>p factor(5.0)
 120
```

While the **p** command enables the display of most Ada objects and constructs, user procedure calling provides the capability to customize the display of objects specific and/or critical to the user program.

References

procedure calls, *SPARCompiler Ada Reference Guide*

5.8 Asynchronous Debugging

It is possible to operate the debugger in asynchronous mode. This means that the debugger is able to accept and execute commands while the program is running.

Note that the debugger is asynchronous only in the sense that the debugger continues to accept commands after the program has started or resumed execution. The debugger itself is still synchronous. It accepts and executes one command at a time. It does not prompt for another command until it completes the last one. It cannot start and stop individual tasks independent of the rest of the program.

There are two ways to enter asynchronous mode: through the `-A` command line option or the `async` option to the debugger's `set` command.

In asynchronous mode, however, all keystrokes are assumed to be debugger commands whether or not the program is running. Three commands have been added to the debugger, `put` and `put_line`, to allow characters to be sent to the program's input and `x` which allows you to monitor user-specified memory locations. `put` and `put_line` are identical, except that `put_line` puts a new line character at the end of the string. Both commands take either a quoted string or a series of characters and write them down to the program's standard input.

References

`async` reference, *SPARCompiler Ada Reference Guide*

`put` command, *SPARCompiler Ada Reference Guide*

`put_line` command, *SPARCompiler Ada Reference Guide*

monitor memory location (x command), *SPARCompiler Ada Reference Guide*

5.9 Macro Preprocessing

The SCAda debugger now supports macro preprocessing as a mechanism to package a series of debugger commands and to parameterize those commands. The goal is to let the user create new commands by packaging debugger commands together and make that package appear like a single debugger command.

Preprocessor commands have been created to start and end macro definitions (`Macro <`*`name`* and `End Macro`) and to expand those definitions (by the appearance of a macro name).

New debugger commands have been added to implement macro processing:

`dm`	`delete macros.`
`em`	`edit macro`
`lm`	`list macros.`
`pm`	`print macro (do not execute)`

All of these commands are case insensitive.

A macro is defined as follows:

```
Macro name
  debugger command
  debugger command
    ...
End Macro
```

Macros are invoked as:

```
name [param1 [param2 [...]]]
```

The name of the macro is only recognized at the beginning of the line.

References

delete macros (`dm`), *SPARCompiler Ada Reference Guide*

`em` (edit macro), *SPARCompiler Ada Reference Guide* list macros (`lm`), *SPARCompiler Ada Reference Guide*

macros, *SPARCompiler Ada Reference Guide*

5.10 Loading the Program

Self-target debuggers automatically load the executable and prepare it for execution.

5.11 Modify Debugger Configuration: set

Use `set` commands to modify the current debugger parameters. Use them at any time during normal operation of `a.db`.

The `set` command uses this format:

```
set [option [value]]
```

Current settings of all the parameters are available in line mode by typing

```
set
set all
```

or in screen mode

```
:set
:set all
```

Use the commands on a command line or supply them to the debugger at invocation. In addition to normal filenames, `set` interprets environment variables and understands the `csh(1)` and `ksh(1)` tilde (~) conventions.

When debugging in line mode or in screen mode with `set output tty`, characters written to `stdout` by the program are not processed. When debugging in screen mode with `set output pty`, lines of characters written to `stdout` are partially processed, to avoid having a blank line at the bottom of the screen.

When the response to a debugger command is longer than the current size of the lower window, some lines are written and the prompt `--More--` appears at the bottom of the window. No further output occurs until you respond to the prompt by pressing `<SPACE>`.

This behavior can be undesirable when debugging programs that use cursor-positioning or other special character sequences to affect output to the video terminal. For debugging programs of this type, toggling between screen mode and line mode or between `set output pty` and `set output tty` is often helpful.

References

`set` command, *SPARCompiler Ada Reference Guide*

5.12 Display Commands

The following debugger commands are available to display variables, source code, tasks and memory.

5.12.1 Display Lines: l, w

Use `l` (list) and `w` (window) to view source code and use similar formats.

```
l [line] [, number]
w [line] [, number]
```

`l` lists *number* of lines of the current source file starting at the specified *line* or current line. Repeat it by pressing <RETURN>. For example, the command

```
l 5, 15
```

lists 15 lines of code starting with line 5.

Similarly, `w` lists a window of source text surrounding the current or specified line. For example, the command

```
w 15, 20
```

lists the 20 lines surrounding line 15.

The `w` command sets the <RETURN> key to repeat the `l` command; typing `l` continues listing where the last `l` or `w` command stopped.

The default for number of lines to be listed is 10. Change the default with the `set lines` command. The default specification for *line* is the current line, which is marked with a < character to the left. The possible forms of *line* are:

number	move to the specified line
+number	move number lines forward
-number	move number lines backward
*	move to the home position

If `*` appears after the line number, the line is the home line; = after the line number indicates that a breakpoint is set there. When - appears after the line number, a breakpoint is set within the code generated for this line but not on the first instruction. If both = and - apply to the same line, the last one set is displayed.

References

`l` command, *SPARCompiler Ada Reference Guide*

`w` command, *SPARCompiler Ada Reference Guide*

5.12.2 Display Breakpoints: lb

The command `lb` lists all the currently set breakpoints. A number in brackets is displayed to the left of each breakpoint and is used with `d` to delete individual breakpoints. Sample output for `lb` is shown here:

```
   20   task p2 is
   21      pragma priority (7);
   22   end p2;
   23   task body p2 is
   24=     i:integer;
   25   begin
   26      for i in 1..50 loop
   27         put ("Task p2 prints this");
   28         new_line;
   29      end loop;
   30   end p2;
   31
   32   begin
   33      put ("The main subprogram prints this");
   34=     new_line;
*----------------------------------------------taskpr1.a--
:lb
[1]  "/vc/ada_ex/taskpr1.a":14 in p1
[2]  "/vc/ada_ex/taskpr1.a":24 in p2
[3]  "/vc/ada_ex/taskpr1.a":34 in taskpr1
```

Figure 5-2 Output from lb Command

References

`lb` command, *SPARCompiler Ada Reference Guide*

5.12.3 Display Instructions: li, wi

`li` and `wi` display disassembled machine instructions and use the following formats, where *`number`* is a set of lines of the current source file starting at the specified *`line`* or current line.

```
li [line|instruction] [, number]
wi [line|instruction] [, number]
```

`li` (list instructions) lists a specified number of lines of disassembled instructions (source code with corresponding assembly language code). Repeat it by pressing `<RETURN>`. Similarly, `wi` (window instructions) prints a window of disassembled code surrounding a specified *`line`* or instruction address (hexadecimal number).

Here is sample output:

```
  21      pragma priority (7);
  22  end p2;
  23  task body p2 is
  24=      i:integer;
  25  begin
  26*      for i in 1..50 loop
  27          put ("Task p2 prints this");
  28          new_line;
  29      end loop;
  30  end p2;
*--------------------------------------------taskpr1.a--
:li
   26          for i in 1..50 loop
 015ba4:*  or      %g0, +01, %i1
   27                   put("Task p2 prints this");
 015ba8:   sethi   %hi(+015c00), %g2
 015bac:   add     %g2, +020, %o0
 015bb0:   sethi   %hi(+015c00), %g3
 015bb4:   add     %g3, +010, %o1
 015bb8:   call    0ff68     -> _A_put.118S12.text_io
 015bbc:  #nop
   28                   new_line;

:li
   28                   new_line;
 015bc0:   or      %g0, +01, %o0
 015bc4:   call    0ec78     -> _A_new_line.70S12.text_io
 015bc8:  #nop
   29          end loop;
 015bcc:   addcc   %i1, +01, %i1
 015bd0:   subcc   %i1, +032, %g0
 015bd4:   ble     015ba8
 015bd8:  #nop
  30  end p2;
:quit
```

Figure 5-3 Output from li Command

The listing starts with the specified position (*line* or *instruction*). If the *line*|*instruction* parameter is *, listing begins with the home position. If the position is not specified, the listing begins with the current position.

The display contains program machine instructions interspersed among the source lines. The first instruction is preceded by the source line that generated it except when no source is available or disassembly begins mid statement.

In addition to the `li` and `wi` commands for displaying instructions, the debugger can be put into *`instruction`* submode of screen mode. The **I** command toggles in and out of instruction mode. In *`instruction`* submode, the source window contains disassembled machine instructions, interspersed with source code, if available. Although the source window contains machine instructions, control it as usual. In this mode, the `s` and `a` debugger commands are interpreted as their machine instruction counterparts, the `si` and `ai` commands, respectively. The `b` command is interpreted as `bi`, setting a breakpoint at the machine instruction under the cursor. All searching and window commands are available, including the `p` command. Use the `p` command with registers.

Use `a.das` for disassembling object files outside the debugger.

References

disassembler (`a.das`), *SPARCompiler Ada Reference Guide*

`li` command, *SPARCompiler Ada Reference Guide*

`wi` command, *SPARCompiler Ada Reference Guide*

5.12.4 Display Tasks: lt

`lt` (list task) lists the state of all tasks that were created but not reclaimed after termination. `lt` produces output such as that shown in the following abbreviated example.

```
Q  TASK             NUM  STATUS
   rand_delay       14   suspended at accept for entry rand
   T philosopher    13   in rendezvous T output[2].put_cursor
R  T philosopher    12   ready
R  T philosopher    11   ready
R  T philosopher    10   ready
R  T philosopher    9    ready
   T fork         8    suspended at fast accept for entry pick_up
   T fork         7    suspended at fast accept for entry pick_up
   T fork         6    suspended at fast accept for entry pick_up
   T fork         5    suspended at fast accept for entry pick_up
   T fork         4    suspended at fast accept for entry pick_up
   T dining_room    3    suspended at select
        open entries:  allocate_seat  leave
*  T output         2    executing
        in rendezvous with T philosopher[13] at entry put_cursor
   <main program>   1    awaiting terminations
```

Figure 5-4 Output from `lt` Command

The simple `lt` command displays four columns of information for each task. The columns are labeled `Q#, TASK, NUM` and `STATUS`.

When tasks are dynamically created (i.e., anonymous tasks), only their addresses are listed. Use the task number with the `task` and `cs` (call stack) commands.

The `lt` *`task`* form of this command provides additional information about a single task.

The command

```
lt dining_room
```

produces this output:

```
>lt dining_room
=> a task type has no address (a task object does): dining_room
Q  TASK           NUM  STATUS
   T dining_room  3    suspended at select
        ENTRY          STATUS   TASKS WAITING
        allocate_seat  open     - no tasks waiting -
        enter                   - no tasks waiting -
        leave          open     - no tasks waiting -
        waiting to execute fast rendezvous in a calling task
        thread id = 01007d3a0
        thread status = PT_CWAIT
        tcb address  = 01007d1c6
        static priority  = 0
        current priority = 0
        parent task:    <main program>[1]
```

Figure 5-5 Output from `lt` task Command

The first lines are the same as the brief display. The added information includes a table of entry queue status giving the entry name, whether the entry is open for a select and the ordered lists of the tasks waiting at that entry.

The static priority line gives the task priority. Following that is current priority. A task executes at the higher of its own static priority and the current priority of any task with which it is in rendezvous. The static priority line displays the priority of a task prior to the task attempting to rendezvous with another task. The current priority is a dynamic priority that is used by the RTS schedule. The current priority can be elevated above the static priority while in rendezvous with a higher priority task or as a result of priority inheritance.

The parent task line describes the master of this task. The task that is executing this master is 045068. With this information it is possible to construct a tree of tasks, linked by their masters to their parent tasks. Note that the task executing the master is not necessarily the task that creates it.

Dynamic tasks list both addresses and the corresponding task type in the **lt** display.

The `lt use` form of this command displays the location and usage of the task stacks.

The command

```
lt use
```

produces this abbreviated output:

```
>lt use
Q  TASK            NUM  STATUS
   rand_delay      14   suspended at accept for entry rand
        wait stack: 0100b7720 .. 0100b86bf, used 499 out of 4000 [12%]
        stack: 0100b4f50 .. 0100b86ef, used 4368 out of 14240 [30%]
        exception stack: 0100b3bc8 .. 0100b4f4f, used 4962 out of 5000
[99%]
   T philosopher   13   in rendezvous T output[2].put_cursor
        wait stack: 0100b27f0 .. 0100b378f, used 443 out of 4000 [11%]
        stack: 0100b0020 .. 0100b37bf, used 4699 out of 14240 [32%]
        exception stack: 0100aec98 .. 0100b001f, used 4962 out of 5000
[99%]
R  T philosopher   12   ready
        wait stack: 0100ad8c0 .. 0100ae85f, used 1120 out of 4000 [28%]
        stack: 0100ab0f0 .. 0100ae88f, used 4643 out of 14240 [32%]
        exception stack: 0100a9d68 .. 0100ab0ef, used 4962 out of 5000
[99%]
R  T philosopher   11   ready
        wait stack: 0100a8990 .. 0100a992f, used 443 out of 4000 [11%]
        stack: 0100a61c0 .. 0100a995f, used 5472 out of 14240 [38%]
        exception stack: 0100a4e38 .. 0100a61bf, used 4962 out of 5000
[99%]
R  T philosopher   10   ready
         wait stack: 0100a3a60 .. 0100a49ff, used 443 out of 4000 [11%]
         stack: 0100a1290 .. 0100a4a2f, used 9776 out of 14240 [68%]
         exception stack: 01009ff08 .. 0100a128f, used 4962 out of 5000
[99%]
R  T philosopher   9    ready
        wait stack: 01009eb30 .. 01009facf, used 443 out of 4000 [11%]
        stack: 01009c360 .. 01009faff, used 14080 out of 14240 [98%]
        exception stack: 01009afd8 .. 01009c35f, used 4962 out of 5000
[99%]
  T  fork          8    suspended at fast accept for entry pick_up
        wait stack: 010099c00 .. 01009ab9f, used 443 out of 4000 [11%]
        stack: 010097430 .. 01009abcf, used 4611 out of 14240 [32%]
        exception stack: 0100960a8 .. 01009742f, used 4962 out of 5000
[99%]
```

```
(Continued)
   T fork          7    suspended at fast accept for entry pick_up
        wait stack: 010094cd0 .. 010095c6f, used 443 out of 4000 [11%]
        stack: 010092500 .. 010095c9f, used 4611 out of 14240 [32%]
        exception stack: 010091178 .. 0100924ff, used 4962 out of 5000
[99%]
   T fork          6    suspended at fast accept for entry pick_up
        wait stack: 01008fda0 .. 010090d3f, used 2368 out of 4000 [59%]
        stack: 01008d5d0 .. 010090d6f, used 4611 out of 14240 [32%]
        exception stack: 01008c248 .. 01008d5cf, used 4962 out of 5000
[99%]
   T fork          5    suspended at fast accept for entry pick_up
        wait stack: 01008ae70 .. 01008be0f, used 443 out of 4000 [11%]
        stack: 0100886a0 .. 01008be3f, used 6720 out of 14240 [47%]
        exception stack: 010087318 .. 01008869f, used 4962 out of 5000
[99%]
   T fork          4    suspended at fast accept for entry pick_up
        wait stack: 010085f40 .. 010086edf, used 443 out of 4000 [11%]
        stack: 010083770 .. 010086f0f, used 11024 out of 14240 [77%]
        exception stack: 0100823e8 .. 01008376f, used 4962 out of 5000
[99%]
   T dining_room   3    suspended at select
        wait stack: 010081010 .. 010081faf, used 499 out of 4000 [12%]
        stack: 01007e840 .. 010081fdf, used 4699 out of 14240 [32%]
        exception stack: 01007d4b8 .. 01007e83f, used 4962 out of 5000
[99%]
*  T output        2    executing
        wait stack: 01007c0e0 .. 01007d07f, used 3200 out of 4000 [80%]
        stack: 010079910 .. 01007d0af, used 6973 out of 14240 [48%]
        exception stack: 010078588 .. 01007990f, used 4962 out of 5000
[99%]
   <main program>  1    awaiting terminations
        wait stack: 07fffb680 .. 07fffc61f, used 3986 out of 4000 [99%]
        stack: 07fdfb690 .. 07fffc62f, used 5295 out of 2101152 [0%]
       exception stack: 07fdfa308 .. 07fdfb68f, used 0 out of 5000 [0%]
```

References

`lt` command, *SPARCompiler Ada Reference Guide*

table of Task State Conditions, *SPARCompiler Ada Reference Guide*

master task, *Ada Reference Manual*

termination of tasks, *Ada Reference Manual*

5.12.5 Display Variables: p

The command

```
p expression
```

displays Ada variables. The debugger supports simple variable names (e.g., `MARK`, `R_A`) and selected components and expanded names (e.g., `MARK.LINE`) and indexed components (e.g., `X(Y)`, `Z(1,2)`). Use them in combination (e.g., `X(M.Z)`, `B.X(1)`).

Ada subprograms and functions can be called (entry calls are not yet supported) and the results of a function can be further used in an expression. Only parameters of mode `in` are supported; `in out` or `out` parameters are not. Default parameter values are not yet supported. The debugger does not call a function that returns an array or a record; however, functions returning access values are supported. A parameterless function or subprogram must be called using empty parentheses,

```
p display_board( )
```

References

p command, *SPARCompiler Ada Reference Guide*

5.12.6 Display Register Contents: reg

The simple command

```
reg
```

lists the contents of the processor registers as they are when the program stops. Sample output is shown in the following example.

```
   1  -- taskprl.a
   2
   3  with TEXT_IO; use TEXT_IO;
   4
   5
*-----------------------------------------------------------
taskpr1.a---
:reg
g0:       0  o0: f7fff158  l0: f7fff428  i0: f7fff158   pc:    7d1c
g1:       8  o1:       44  l1: f7fff428  i1:        0  npc:    7d20
g2: f7fff528  o2:       8  l2: f7fff428  i2:        1    y: 16800000
g3:       b  o3:       0  l3:       8  i3: f7fff2c0  psr: 00001080
g4: f7fc1150  o4:     168  l4:       8  i4:       44       impl
ver nzvc ec
g5: 54595045  o5:       0  l5: f7fff2c0  i5: f7fff158         0
0 0000  0
g6: f7fff568  o6: f7fff118  l6:       8  i6: f7fff568      ef pil
s ps et cwp
g7:       8  o7:     7cfc  l7: f7fff428  i7:    211ec       1   0
1  0  0   0
```

Figure 5-6 Output from `reg` Command

The command

```
reg f
```

lists the floating point coprocessor registers if a coprocessor is supported in the implementation.

```
:reg f

fsr:     rd  rp  tem  ftt  qne  fcc  aexc  cexc
5060421  0   0    a    0    0    1    1     1
 f0: 0.000475                 f1: 518.113708              d0: 0.000000
 f2: 0.000100                 f3: 0.000009                d2: 0.000000
 f4: 0.000100                 f5: 0.000009                d4: 0.000000
 f6: 1.000000                 f7: -NaN                    d6: 0.007813
 f8: -NaN                     f9: -NaN                    d8: -NaN
 F6: -NaN                    f11: -NaN                   d10: -NaN
f12: -NaN                    f13: -NaN                   d12: -NaN
f14: -NaN                    f15: -NaN                   d14: -NaN
f16: -NaN                    f17: -NaN                   d16: -NaN
f18: -NaN                    f19: -NaN                   d18: -NaN
f20: -NaN                    f21: -NaN                   d20: -NaN
f22: -NaN                    f23: -NaN                   d22: -NaN
f24: -NaN                    f25: -NaN                   d24: -NaN
f26: -NaN                    f27: -NaN                   d26: -NaN
f28: -NaN                    f29: -NaN                   d28: -NaN
f30: -NaN                    f31: 2104.600098            d30: -NaN
```

Figure 5-7 Output from `reg f` Command

Note that individual registers may be displayed in a variety of formats using the `p` command.

References

`p` command, *SPARCompiler Ada Reference Guide*

`reg` command, *SPARCompiler Ada Reference Guide*

5.12.7 Display Call Stack for a Task: cs

The `cs` command displays the call stack, starting with the current frame and has the following format:

```
cs [reg] [number] [in task]
```

`reg` provides an extra 3-line hexadecimal dump per frame. *`number`* specifies the topmost number of frames to be displayed. The `in` *`task`* clause displays the call stack for a specific task. Use the task name or find the task index with the `lt` command.

References

cs command, *SPARCompiler Ada Reference Guide*

5.12.8 Display Task Number: task

The command

```
task
```

displays the address of the current task.

References

task command, *SPARCompiler Ada Reference Guide*

5.12.9 Display Exceptions

Use the `p` command to display exceptions. Entering the following command prints the current exception name and a PC value very near where the exception was raised:

```
% p $exception
```

This command is useful if you set a breakpoint in an exception handler and there are many possible places where the exception could be raised. In addition, if the handler is `when others =>`, this command helps by displaying the actual name of the exception.

References

exception handling, *SPARCompiler Ada Reference Guide*

5.12.10 Value Assignment

Use the **:=** operator in the following formats to assign a value to a variable, register or memory location(s). No type checking is performed between the name and the expression.

```
name := expression
address [, number] := expression
```

name is any Ada expression that identifies a scalar or a register name (preceded by a dollar sign). *expression* is any scalar expression. *address* is either hexadecimal or decimal. *number* is an optional parameter indicating how many bytes are modified. If *expression* evaluates to an integer, *number* must be 1, 2, 3 or 4. If expression evaluates to a floating point number, *number* must be either 4 or 8. If absent, *number* defaults to 4.

On compilers that support floating point coprocessors, assign the coprocessor registers values directly. The number of bits in the coprocessor registers differ between implementations. Values assigned in this way are approximations and the conversion routines are slow at present, particularly for wide coprocessor registers.

After the memory is modified, a line of the form

```
address: new/old
```

is printed. *address* is the altered memory location, *new* is the current value. *old* is its previous value. For Ada names, *address* is not printed.

References

:= (assignment) command, *SPARCompiler Ada Reference Guide*

Ada names, *Ada Reference Manual*

5.12.11 Display Raw Memory

Command formats for displaying raw memory are:

```
hexadecimal_address[:display][number]
decimal_address:display[number]
name:display [number]
```

display consists of a length character alone, a length character and a format character or a format character alone. The default is shown in brackets.

LENGTH	
B	8-bit [default number base established by set obase]
D	64-bit floating point
E	Largest size floating-point format available
F	32-bit floating point
L	32-bit [default number base established by **set obase**]
W	16-bit [default number base established by **set obase**]

FORMAT	
a	show the address of the item
b	Display as bits
c	ASCII character
d	decimal [32 bits]
f	floating point [32 bits]
m	one line of STORAGE_UNITS, first in hexadecimal, then as ASCII characters
n	like **m** but bytes are interpreted in reverse order
o	octal [32 bits]
p	hexadecimal pointer [32 bits]
r	reverse-map the address to a procedure name
s	null-terminated (C-style) string
x	Hexadecimal [32 bits] Note this will be the same as using **p**.

Enter a *`number`* in either decimal or hexadecimal with a leading `0`. If the leading digit of a number is a zero, the debugger assumes it is a hexadecimal number (`0123` or `0F2`). If a number begins with a decimal digit (`0-9`) but contains a hexadecimal digit (`a-f`), the debugger interprets the number as a hexadecimal number (`12A3` or `9AAF`). Note that a hexadecimal number with a leading hexadecimal digit (`F2`) must be preceded by zero (`0F2`) or it is interpreted as an identifier.

number determines how many values are displayed. The address is advanced by a number corresponding to the letter being used. For example, to display 8, 16-bit decimal values starting at address 01A8A, type the following command.

```
01a9a:Wx 8
```

The debugger responds with this output:

```
01A8A:   2074  6865  204e  2071  7565  656e  7320 7072
```

The default length is 32 bits. The default format is either decimal, octal or hexadecimal, depending on the setting of the `set obase` command. The default count is 1. Specifying an address of the form *hexadecimal_number* displays 32 bits in the current output base. The following command example displays two lines each, beginning at address 01A8A. Each line is displayed in hexadecimal format followed by an ASCII string:

```
01A8A:m 2
```

The debugger responds with this output:

```
01A8A: 50 72 6f 67 57 61 6d 20 74 6f 20 73 6f 66 76 65 ” Program
to solve ”
01A9A: 20 74 68 65 20 4e 20 71 75 65 65 6e 73 20 70 72 ” the
N queens problem”
```

The colon followed by a format specification is required to dump memory. Specifying a decimal *number* alone is the debugger command to change the current position.

References

display memory, *SPARCompiler Ada Reference Guide*

`set obase`, *SPARCompiler Ada Reference Guide*

5.13 Execution Commands

The debugger execution commands control the flow of execution of the program, including normal running, single stepping, advancing over called subprograms and executing until a specified variable changes value.

5.13.1 Run Program: r

`r` (run) starts or restarts the program from the beginning and uses the following format:

```
r [shell_arguments]
```

If *`shell_arguments`* is specified, **r** runs the program as if it executes from the shell. The debugger supports a subset of shell command arguments. It supports the following arguments for I/O redirection: >, <, >> and >& (for redirecting both standard and error output).

The debugger supports substitution for shell environment (exported) variables and *`~name`* directory shorthand. Additionally, when argument strings contain dollar signs (`$`), backquotes (`‘`), globbing meta symbols (`*`, `?`, `[]`) or in the case of 4.2 BSD UNIX, curly braces (`{}`); they are passed to the shell for evaluation. The debugger uses the shell defined in the shell environment (exported) variable. Strings enclosed in double quotation marks are passed as a single argument after removing the quotes. Other parameters are passed (`-o`, `-Pfoo`) just like the shell.

```
r [arg1 .. arg10]
```

References

`r` command, *SPARCompiler Ada Reference Guide*

`set` command, *SPARCompiler Ada Reference Guide*

5.13.2 Step One Line/Instruction: a, ai, s, si

The single-stepping commands `a` (advance), `ai` (advance instruction), `s` (step) and `si` (step instruction) execute the program for a single line or instruction. `s` (step) and `si` (step instruction) step one line or instruction, stepping *into* called subprograms. Use `a` (advance) and `ai` (advance instruction) to step one source line or instruction, stepping *over* called subprograms. `<RETURN>` repeats all single-stepping commands.

References

`a` command, *SPARCompiler Ada Reference Guide*

`ai` command, *SPARCompiler Ada Reference Guide*

`s` command, *SPARCompiler Ada Reference Guide*

`si` command, *SPARCompiler Ada Reference Guide*

5.13.3 Continue Execution: g, gw and gsg, gwg, gw

The `g` (go), `gw` (go while) and `gs` (go signal) commands execute the program from where it last breakpointed until a specific condition is reached.

If the process stops because of an OS signal, `g` continues executing the program, ignoring the signal. `gs` works in the same manner as `g` but program execution continues and the signal is passed to the program. Note that the `gs` command is equivalent to the `gx` command. The `gx` command is still valid but support for it will be discontinued in a future SCAda release.

`gw` executes program until the value of the specified variable changes and uses the following format.

```
gw name|address [, number]
```

If an `address (number)` is specified, *`number`* bytes at that address are monitored for change (default = 4). *`number`* can be 1-16. (The value being watched is checked after the execution of each machine instruction, which is quite slow on some hardware. Running `a.db` from a script can be helpful in this case.)

References

`g` command, *SPARCompiler Ada Reference Guide*

`gw` command, *SPARCompiler Ada Reference Guide*

`gs` command, *SPARCompiler Ada Reference Guide*

5.13.4 Read Debugger Commands From a File: read

The command

```
    read filename
```

causes the debugger to switch from executing keyboard instructions to executing instructions from the specified file. Additional `read` commands can occur in the file but are nested to a maximum of four levels. After executing the file, commands are read again from the keyboard (unless the command file contained an `exit` or `quit` command).

References

read command, *SPARCompiler Ada Reference Guide*

5.13.5 Terminate the debugger Session: exit, quit

Entering the command `exit` or `quit` terminates the debugger session.

References

`exit` command, *SPARCompiler Ada Reference Guide*

`quit` command, *SPARCompiler Ada Reference Guide*

5.14 Breakpoint Commands

Stop the program being debugged at a particular place in its execution by setting a breakpoint.

With `r` or `g`, the program executes until it encounters the breakpoint. If `r` or `g` are used to run a program and the possibility of an infinite loop exists, stop the program by pressing the interrupt key, `<INTR>` (usually mapped to `<CONTROL-C>`). Also, use the interrupt key to stop the debugger from completing a long display, for example if the display of a large array of large records is accidentally initiated.

The current frame is the position on the call stack. When a breakpoint is announced, the current frame is always set to the subprogram containing the breakpoint. This frame is the topmost frame of the call stack and is known as the break frame. As one moves up and down the stack, the current frame is always the topmost frame of a `cs` listing. If `li` produces a listing and disassembly, the next instruction in the current frame to execute has an asterisk (`*`) next to it (the home position). The current frame can only be changed using the call stack commands (`cu`, `cd`, `ct`, `cb`) when the program stops at a breakpoint.

References

breakpoints, *SPARCompiler Ada Reference Guide*

5.14.1 Set Breakpoint: b

The basic form of the command to set a breakpoint is:

```
b [line|subprogram] [in task]
```

This command sets a breakpoint at a specified line in the current file or at the beginning of a specified subprogram. Typically, *`line`* is a decimal number. To set a breakpoint at line 537 of the current source file, use the following command:

```
b 537
```

subprogram is the name of an Ada subprogram, task body or package. Use expanded names as well as simple identifiers (starting with STANDARD if desired). To set a breakpoint at the subprogram SORT_STAMPS, use the following command.

```
b sort_stamps
```

References

b command, *SPARCompiler Ada Reference Guide*

visibility rules, *SPARCompiler Ada Reference Guide*

expanded names, *Ada Reference Manual*

5.14.2 Command Blocks

Enter debugger commands on the same line as the breakpoint, if they are separated by semicolons. The syntax, similar to that used in Ada, is:

```
b [line|subprogram] [in task] [begin commands end]
```

For example.

```
b 38 begin p i; p dump_node(root) end
```

Alternatively, as long as the beginning block mark begin appears on the breakpoint line, the commands can appear each on a new line. The debugger prompts with a ?? prompt until the closing block mark end is entered, as illustrated here:

```
>b 38 begin
??p i
??p dump_node(root)
??end
>
```

Figure 5-8 Breakpoint Command Block

References

b command, *SPARCompiler Ada Reference Guide*

5.14.3 Set Conditional Breakpoint: b

When a conditional breakpoint is reached, execution continues transparently if the condition is false. If the condition is true, the specified commands are performed.

Possible formats for the `b` command are:

```
b [line|subprogram] [in task] when expression
b [line|subprogram] [in task] if expression then [commands]
                             [else  [commands]] end [if]
b [line|subprogram] [in task] if expression else [commands]
end [if]
```

where the following terms are used:

`line`|*subprogram*	line number or subprogram name
task	task identifier
expression	the condition at the breakpoint (a non-zero value is defined as `TRUE`)
commands	list of debugger commands to be executed

Expressions are evaluated and (as with C expressions) 0 means `FALSE`, other values mean `TRUE`. All comparison operators (<, <=, =, etc.) return 0 for false and 1 for true.

The debugger verifies that the conditional expression is valid when the breakpoint is set. The expression is verified using the visibility rules, etc., that exist when the breakpoint is hit and the expression is actually evaluated.

Specify a conditional breakpoint by using `when` *`expression`* after the breakpoint command as shown in this example:

```
b 38 when col <= row
```

This breaks the program at line 38 if col <= row.

Alternatively, `if` *`expression`* `then` can be used. Any statements in the `then` clause are executed when the `if` *`expression`* is `TRUE`. Any statements in the `else` clause are executed when the `if` *`expression`* is `FALSE`. However, the debugger announces a breakpoint only when the `if` *`expression`* is `TRUE`. For example:

```
b 38 if col < row then p col; p row; end if
```

This sets a breakpoint at line 38. The debugger executes the `then` clause and announces a breakpoint when it reaches the breakpoint *and* column is less than row.

Also, you may set debugger commands to execute when the breakpoint is hit but the condition is false. These commands take the form of an `else` clause, as shown here:

```
b 38 if i > 5 else p i; end if
```

In this example, the command `p i` between `else` and `end if` executes when the breakpoint at line 38 is reached but `i > 5` is not true. This provides a tracking facility.

The debugger supports line number breakpoints set for a specific task using the `in` clause. When the program reaches a breakpoint set with this clause, the debugger checks to see if the task executing is the one specified by the breakpoint. If not, the task is continued. If the task executing is the one specified by the breakpoint, the breakpoint is announced to the user.

References

`b` command, *SPARCompiler Ada Reference Guide*

5.14.4 Break on Return: bd, br

The `bd` and `br` commands set a breakpoint in the subprogram that called the current subprogram. The breakpoint is reached as soon as the current subprogram returns. The current subprogram is represented by the current frame. Use the following `bd` or `br` format (similar to those available for `b`).

```
bd (or br) [in task] [begin commands end]
bd (or br) [in task] when expression
bd (or br) [in task] if expression then [commands] [else
[commands]] end [if]
bd (or br) [in task] if expression else [commands]] end [if]
```

If `br` is used, a permanent breakpoint is set. If `bd` is used, the breakpoint is removed automatically when it is hit.

`bd` breakpoints are shown with a hyphen next to the breakpoint number in a `lb` listing and at the source line when the breakpoint is reached.

References

`bd` command, *SPARCompiler Ada Reference Guide*

`br` command, *SPARCompiler Ada Reference Guide*

5.14.5 Set Breakpoint at Instruction: bi

The `bi` command sets a breakpoint at an instruction address.

```
bi [instruction] [in task] [begin commands end]
bi [instruction] [in task] when expression
bi [instruction] [in task] if expression then [commands]
                           [else [commands]] end [if]
bi [instruction] [in task] if expression else [commands] end
[if]
```

The instruction address (which is obtained using `li` or `wi`) must be a hexadecimal number with leading zero.

References

`bi` command, *SPARCompiler Ada Reference Guide*

5.14.6 Set Breakpoint at Exception: bx

`bx` sets a breakpoint that is reached when the named *`exception`* occurs, before the exception is raised by the runtime system. The syntax is:

```
    bx [exception] [in task] [begin commands end]
```

Like `b`, `bd` and `bi`, `bx` breakpoints can be set for a particular task using the `in` *`task`* option and can be followed by a block of debugger commands to execute when the breakpoint is reached.

References

`bx` command, *SPARCompiler Ada Reference Guide*

5.14.7 Display Instruction: li, wi

Use `li` or `wi` to produce a display of the instructions (disassembly) to determine where to set an instruction breakpoint.

References

Section 5.12, "Display Commands," on page 5-19

`li` command, *SPARCompiler Ada Reference Guide*

`wi` command, *SPARCompiler Ada Reference Guide*

5.14.8 Delete Breakpoint: d

The `d` command deletes the listed breakpoints with the following format.

```
d all|breakpoint_number [, breakpoint_number]...
```

Specify `all` to delete all breakpoints. Multiple breakpoint numbers comprising the *`breakpoint_number`* (obtained using `lb` and displayed in brackets) must be separated by commas in this command, as illustrated:

```
d 1, 2, 3
```

References

`d` command, *SPARCompiler Ada Reference Guide*

5.14.9 Implicit Breakpoints

The debugger sets some breakpoints automatically for its own use.

- When the debugger is initially invoked and after a `set run` command, the *`current position`* and the *`home position`* are moved to point to the first executable statement of the *`main`* subprogram. If the user types either an `s` or `a` command, then the debugger automatically sets a breakpoint at the beginning of the *`main`* subprogram and runs the program. When this breakpoint is reached, it is removed and the program is stepped according to the semantics of the `s` or `a` command.
- When an `a` or `ai` command is typed and a subprogram call must be stepped over, the debugger automatically sets a breakpoint on the instruction following the call instruction and then lets the program call the procedure.

This permits the program to proceed at normal speed until the breakpoint is reached. When the breakpoint is hit, after the subprogram returns, it is removed.

- A breakpoint is automatically set at a procedure in the Ada runtime system called `SLIGHT_PAUSE`. This null procedure is called immediately prior to the runtime system abandoning execution when an unhandled exception is discovered, as described in Ada LRM 11.4.2(4). The null procedure is called to enable the debugger to recognize when a program is about to terminate because of an unhandled exception.

5.15 Positioning Commands

a.db includes commands to move the current position for debugging purposes.

5.15.1 Specify New Position

Specifying a decimal *line_number* moves the current position to that line number in the current file. In screen mode, the line number must be followed with an upper case G. Specifying a plus or minus before a number allows relative positioning. Specifying * moves the current position to the current home position.

References

line numbers, *SPARCompiler Ada Reference Guide*

5.15.2 Call Stack Bottom, Top: cb, ct

cb (call bottom) and ct (call top) are similar commands that move the current position and current frame to the bottom or top of the call stack. To display the values of local variables and parameters of a subprogram currently active on the call stack, first move the current position to that frame on the stack. This makes the local variables visible.

References

cb command, *SPARCompiler Ada Reference Guide*

ct command, *SPARCompiler Ada Reference Guide*

5.15.3 Call Stack Up, Down: cd, cu

cd (call down) and cu (call up) move up and down the call stack and take *name* or *number* as arguments:

```
cd [name|number]
cu [name|number]
```

If *name* is provided, the current frame moves down to the next frame on the call stack with that name. If *number* is specified, movement (in either direction) is to the frame having that number. The new frame becomes the current frame and the line being executed in that frame becomes the current home position, marked with a `*`.

References

`cd` command, *SPARCompiler Ada Reference Guide*

`cu` command, *SPARCompiler Ada Reference Guide*

5.15.4 Call Stack: cs

The `cs` command displays the contents of the stack, starting with the current frame and displays the frame numbers. The `cs` command has the following format:

```
cs [reg] [number] [in task]
```

If *number* is specified, only the topmost *number* frames are displayed. If `reg` is used, an extra three line hexadecimal dump is provided per frame. The first of these three lines provides the program counter, the frame pointer and the argument pointer. This is used mainly for debugging the debugger and compiler but can be useful in other circumstances. If the `in` *task* clause is used, where *task* is a task name or index obtained with the `lt` *task* command, the stack for that task is displayed. The default stack is that of the current (breakpointed) task.

References

`cs` command, *SPARCompiler Ada Reference Guide*

5.15.5 Move Current Position: e, task, v, vb

`e` (enter) or `v` (visit) provide a way to move the current position to a new file or subprogram or to change the current task using the following format.

```
e [subprogram|ada_source.a]
v [subprogram|ada_source.a]
```

The subprogram name need not be directly visible. For purposes of the `b`, `e`, `edit`, `v` and `vi` commands, all subprogram names in the program are searched. Expanded names can be given.

If a subprogram name is overloaded, a diagnostic showing the alternatives is given. The appropriate suffix (e.g., '1, '2) must then be used to disambiguate the reference.

In screen mode, invoke the `e` or `v` command by positioning the cursor on the left-most character of a subprogram name and then typing `<CONTROL-]>`. This command is compatible with the `vi` editor `<CONTROL-]>` tag command.

`task` identifies a new task to become the current task. *`task`* is the task name or address identified by using the `lt` command. `task` makes the call stack of the selected task visible. The call stack commands (`cb`, `cd`, `cs`, `ct` and `cu`) operate on the current task call stack.

`vb` moves the current source location to the source location of the indicated breakpoint

References

current position, *SPARCompiler Ada Reference Guide*

`e` command, *SPARCompiler Ada Reference Guide*

`task` command, *SPARCompiler Ada Reference Guide*

`v` command, *SPARCompiler Ada Reference Guide*

`vb` command, *SPARCompiler Ada Reference Guide*

expanded names Ada LRM 4.1.3(13)

5.15.6 Enter Editor: edit

Use `edit` to enter `vi` or the editor specified in the environment (exported) `EDITOR` variable for the specified file or subprogram. If no parameter is given, the file containing the current position is used. The default editor is `vi`**.**

Parameters are interpreted exactly the same as for the `e` command.

References

`edit` command, *SPARCompiler Ada Reference Guide*

5.16 Screen-mode Debugging

In addition to using the SCAda debugger in a line-oriented mode, a screen mode is available. Screen mode employs windows and functions similar to the `vi` editor. This section provides instructions for using the debugger in screen-mode.

References

screen mode, *SPARCompiler Ada Reference Guide*

5.16.1 Entering Screen Mode

To use the screen interface, invoke the debugger with the `-v` option, as follows:

```
a.db -v myprogram
```

or type the debugger command

```
vi
```

in response to a debugger prompt at any time during the line-mode debugging session.

To return to the line-oriented interface, type the screen command `Q`.

References

screen mode, *SPARCompiler Ada Reference Guide*

5.16.2 Screen-mode Windows

In screen mode, the debugger divides the screen into three windows. The top window, called the ‘source window’, extends from the top of the screen to a dashed line and is used for program source text. The window below the dashed line is called the ‘command window’ and is used to display debugger and program input and output. The last line on the screen is the ‘error window’ and is used for error messages from the screen interface, search patterns, etc.

The command and source windows each display a small portion of a (potentially) much larger area. For example, the source window can be moved to display a different part of the same source file or it can display part of a different source file.

The debugger keeps the last 750 lines of the text displayed in the command window in a history buffer. Typically, the command window is positioned to display the most recent 3 interactions (i.e., the bottom of the history buffer). However, this window behaves just like the source window in that it can be moved to display a different part of the history buffer.

The command window is automatically paged. If output from either a single debugger command or the user program scrolls off the top of the window, the output is stopped and `--More--` is displayed on the last line.

When in screen mode, use the screen interface commands, known as 'window commands' (they do not require the use of `<RETURN>` to complete them). In addition, use any line-oriented debugger command by typing a colon first.

References

windows, *SPARCompiler Ada Reference Guide*

5.16.3 Instruction and Source Modes

When in visual mode, the debugger supports two sub-modes, *`instruction`* or *`source`*. The `I` command toggles the submode, switching from one to the other. In source mode, the source window displays program source code and the `s` and `a` commands single step at the source statement level. Source mode is the default.

In instruction mode, the source window contains disassembled machine instructions, interspersed with source code, if available. Although the source window contains machine instructions, control it as usual. In this mode, the **s** and `a` debugger commands are interpreted as their machine instruction counterparts, the `si` and `ai` commands, respectively. The `b` command is interpreted as `bi`, setting a breakpoint at the machine instruction under the cursor. All searching and window commands are available.

References

instruction submode, *SPARCompiler Ada Reference Guide*

5.16.4 Screen-mode Window Commands

Screen-mode interface commands, known as window commands fall into two groups: debugger-related operational commands and cursor movement commands (as with `vi`).

References

window control commands, *SPARCompiler Ada Reference Guide*

5.16.5 Window Command Syntax

Most window commands are a single character and are *not* followed by `<RETURN>`. The pattern-matching commands (`/` and `?`) are followed by the pattern to be found and are terminated by either `<ESCAPE>` or `<RETURN>`.

Some commands can be preceded by an optional number. When *`number`* is given for a `<CONTROL-D>` or `<CONTROL-U>` command (scroll up or down), that number is used with all subsequent `<CONTROL-D>` and `<CONTROL-U>` commands until a new number is specified. The initial value of *`number`* is one-half the size of the window.

The window commands are applied to the window that contains the cursor. The comma command (`,`) moves the cursor from one window to the other.

In addition to this set of debugger commands specially tailored for use in the screen mode, use any debugger command by typing a colon first. In response to the colon, the screen interface scrolls up the command window one line and positions the cursor on the bottom line of the command window. Then type in the debugger command and press `<RETURN>`. The command executes and the cursor returns to where it was when the colon was entered. After each debugger command, the cursor returns to the source window.

A special help facility is provided in screen mode and is used by typing `H` which gives one of several lines of help for the pager program as well as the debugger commands at the bottom of the screen. Typing `H` again displays the next line of help.

References

debugger help facility, *SPARCompiler Ada Reference Guide*

window control commands, *SPARCompiler Ada Reference Guide*

5.17 Debugging C Programs

`a.db` can be used as a symbolic debugger for C programs. On the Solaris 2.0 operating system, C programs must be compiled with both the `-g` option and the `-xs` option to be compatible with the SCAda debugger.

The dbx-style debugger cannot debug C files that are compiled without symbolics. The `set verbose on` command must be initialized before symbolics are read for the C executable. In the `.dbrc` file, for example, it must precede a call to `a.db`. The `verbose on` parameter to the `set` command issues the warning message "`sdb symbolic information found in file ***.c`" if such a file is encountered. [Default: `off`]

5.17.1 Names, Variables and Visibility Rules

Looking up variables for C programs begins as for Ada. If the current home position is contained in a C function, the debugger finds the same definition of the name that the C compiler finds if the name is part of the text of the program corresponding to the current home position. However, if the debugger does not find a definition using conventional C visibility rules, it searches program-wide to see if external variables have the same name. If so, the value of that variable is displayed. As a result, when the current home position is contained in a C function, all external C variables are visible unless hidden by an inner declaration.

5.17.2 Type Checking

Like the C compiler, the debugger does no type checking of the actual parameters before making the function call.

For example, `p foo` displays a C variable named `foo`.

C functions can be called from the debugger. For example, if `foo` is a C function,

```
p foo(3, baz)
```

calls `foo` with parameters `3` and `baz`, where `baz` is a visible variable.

5.17.3 C Operators

The following operators are available for C programs:

`.`	`structure field selection`
`->`	`pointer dereference/structure field selection`
`[ ]`	`subscript`

Note that variable assignment, conditional breakpoints and relational expressions currently use Ada rather than C syntax even when debugging C programs.

The debugger requires the use of C/C++ operators in debugger expresssions. By default, the debugger expects Ada operators when debugging in a SCAda directory, otherwise, it assumes C/C++ operators. The `set language` command can be used to override this default.

References

`set language`, *SPARCompiler Ada Reference Guide*

5.17.4 External Variables

External variables that are declared in files that are not compiled with the **-g** option`-Z7` option (i.e., do not have symbolic information) are visible but the debugger has only their address. A message is displayed and the address of the variable is given as a 32-bit integer.

5.17.5 char * Variable Display

When displaying a variable declared as `char *`, both the address (i.e., the value of the pointer) and the string at that address is displayed. Use the * unary operator to access the individual characters or the contents of the pointer. This is useful when using `char *` as a generic pointer and not as a pointer to a string.

5.17.6 Operators

The Ada syntax and operators for assignment (`:=`), equality (=), inequality (`/=`) and so forth, rather than the C equivalents, are used in debugger commands.

5.17.7 Unary Operators

The & unary operator returns the address of its operand in the debugger. Use this operator in expressions and procedure calls.

`&` *`symbol`* returns the address of *`symbol`*.

The `*` unary operator is an indirection operator that accesses memory using the value of its operand as the address of memory. The contents of the address are displayed in the format of the type of the operand or, if the operand is a value, in the format of integer.

“We don’t know who we are until we see what we can do.”

Martha Grimes

X Window Debugging 6

a.xdb is an X/Motif-based debugger interface. It provides all the features of a.db and, in addition, provides new features not possible without the X Window System. The interface has been designed to accommodate both a.db experts and novices.

6.1 Requirements

6.1.1 SC Ada

You must have the SC Ada product correctly installed. See the *SC Ada Installation Manual* for more details.

6.1.2 X Window System

a.xdb works with X11R4, X11R5 and the XNeWS server. If you are using OpenWindows 3.2 and the XNeWS server, there is a bug where popup menus and pullright menus do not work correctly. To fix this problem, install the following patch:

```
Solaris 2.2  OpenWindows3.2, patch 101084-02 (OW, Motif 1.2)
```

Motif programs are known to cause problems with older versions of OpenWindows and the XNeWS server.

6.1.3 Window Manager

`a.xdb` has been tested with the following window managers:

```
twm, olwm (OpenWindows 3.2), mwm, vtwm
```

`a.xdb` also requires that the standard OSF keysyms be installed in the file `/usr/lib/X11/XKeysymDB`. If some or all the the keysyms are missing, you can merge them into `/usr/lib/X11/XKeysymDB` from the file *`SCAda_location`*`/lib/XKeysymDB`.

Without these keysyms, many of the key bindings used by `a.xdb` do not work and users will see error messages such as:

```
Warning: translation table syntax error:
   Unknown keysym name: osfUp
Warning: ... found while parsing
   's c <Key>osfUp:backward-paragraph(extend)'
```

6.1.4 Resource Management

A predefined set of resources for `a.xdb` and other SC Ada graphical user interfaces is contained in the directory *`SCAda_location`*`/lib/app-defaults`. Copy these files to `/usr/lib/X11/app-defaults`, or set the environment variable XAPPLRESDIR to point to *`SCAda_location`*`/lib/app-defaults`.

6.1.5 Dynamic Linking and X/Motif Programs

For each X/Motif program in SC Ada, Rational Software Corporation provides two executables, one which has the X and Motif libraries statically linked, and one which is dynamically linked.

The executables are located in *`SCAda_location`*`/sup`. The wrapper program (`a.xdb` or `a.xhs`) selects the executable and runs it with arguments appropriate for the current environment. By default, the wrapper looks for the static executable first (for `a.xdb`, it is called `xdb-static`). If that executable doesn't exist, the dynamically-linked version is used (`xdb` for `a.xdb`).

In general, the statically linked executable is more likely to run. The dynamically linked executable must have access to X/Motif libraries with the right versions and names to work properly, and these may differ for different operating system versions and system configurations. However, dynamically linking improves performance so we recommend using the dynamically linked executables if possible. To test them:

```
% cd SCAda_location/sup
% mv xhs-static xhs-static.orig
% mv xdb-static xdb-static.orig
% a.xdb
```

If `a.xdb` runs, then you may delete the `-static.orig` files to save space. If not, continue to use the statically linked programs:

```
% mv xhs-static.orig xhs-static
% mv xdb-static.orig xdb-static
```

6.2 Running a.xdb

`a.xdb` runs under the X Window System, so your X server and window manager must be running.

To start `a.xdb`, follow these steps at a shell prompt inside an xterm window:

1. If you are running `a.xdb` on a remote workstation but want it to display on your local workstation enter the following command in the xterm on the remote workstation:

   ```
   % setenv DISPLAY local_host:0
   ```

   Replace *`local_host`* with the unique network name of your local workstation

   To give the remote workstation access to the X server on your local workstation, enter the following command on your local workstation, where *`remote_host`* is the unique network name of the remote workstation:

   ```
   % xhost +remote_host
   ```

   If you do not do this, X applications cannot run, and the following error messages are displayed:

   ```
   Xlib:connection to ”<local_host>:0.0” refused by server
   Xlib:Client is not authorized to connect to server
   Error: Can’t open display: <local_host>:0.0
   ```

The command

```
% xhost
```

lists the hosts which have access to the server on your local workstation.

2. Enter the command

```
% a.xdb executable_file
```

where *executable_file* is the name of the executable you want to debug.

Alternatively, you can start `a.xdb` with no options and it will look for the file **a.**`out`. If **a.**`out` is not found, you can specify the executable by using the `Select Executable` item in the Debug pulldown menu.

3. To exit `a.xdb`, use `Quit Debugger` in the Debug pulldown menu.

6.3 Screen Layout

6.3.1 Terminology

`MB1` signifies mouse button one, the left button. `MB2` signifies mouse button two, the middle button. `MB3` signifies mouse button three, the right button.

The quickest way to learn about the screen layout is to use the context sensitive help facility of `a.xdb`. To do this use the `On Context` menu item on the Help pulldown menu. After selecting this command, the cursor is changed to a question mark (**?**). Move the cursor to the area you want help on and click `MB1`. Help about that area is displayed.

6.3.2 Popup Menus

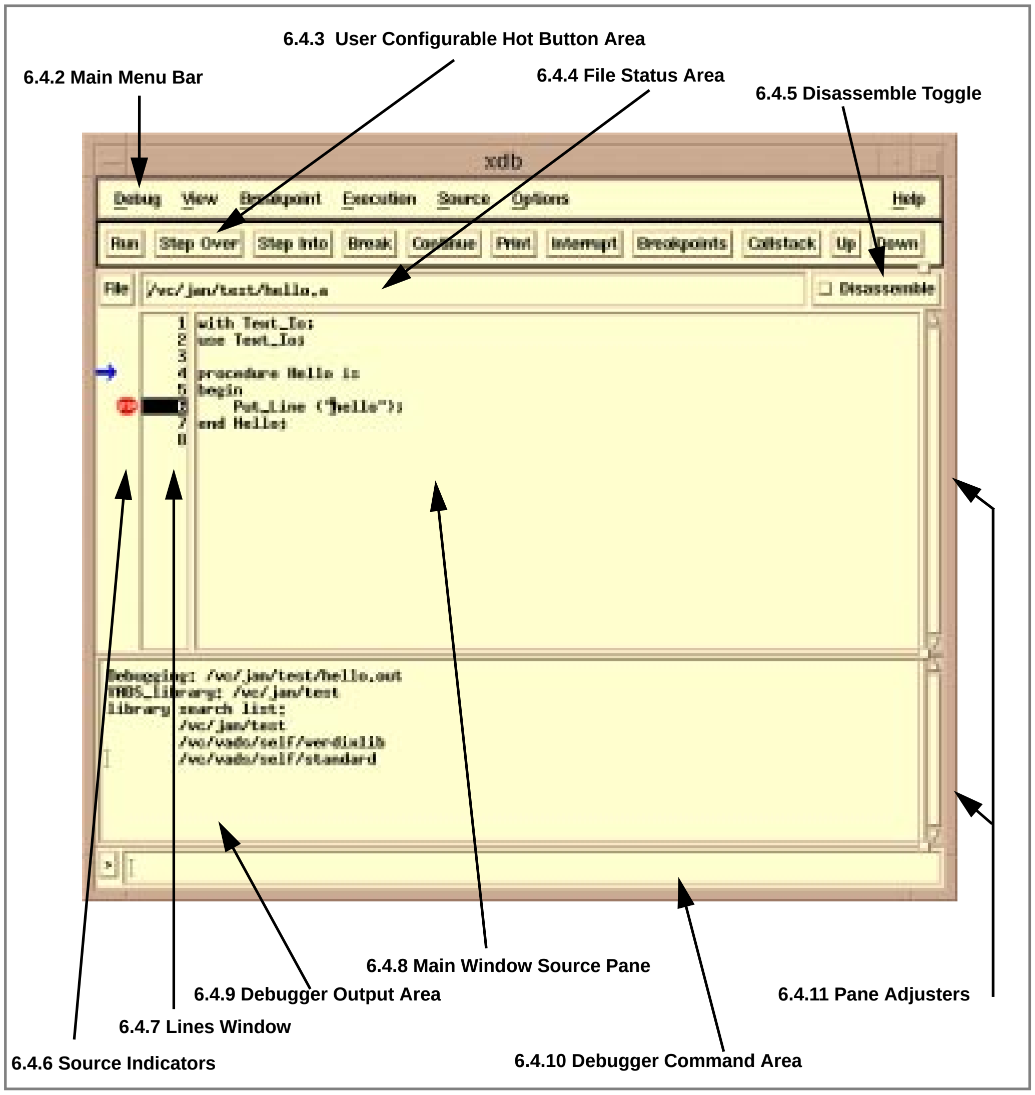
The main window has popup menus in both the source pane and the lines window. To use a popup menu follow these steps:

1. Place the pointer in either the source pane or the lines window
2. Press and hold `MB3`
3. Drag the pointer to the button you want to select and release the mouse button.

Some of the popup menus contain pullright menus indicated by a small triangle to the right of the menu item. To pick an item in a pullright menu drag the pointer on top of the item and then drag the pointer to the right.

6.3.3 Main Debugger Window - Overview

The major areas of the main window are highlighted in Figure 6-1



```
with Text_Io;
use Text_Io;

procedure Hello is
begin
    Put_Line ("hello");
end Hello;
```

```
Debugging: /vc/jan/test/hello.out
VADS_library: /vc/jan/test
library search list:
          /vc/jan/test
          /vc/vads/self/verdixlib
          /vc/vads/self/standard
```

Figure 6-1 Main WIndow of xdb screen

6.3.3.1 *Main Menu Bar*

The main menu bar provides access to the debugger operations and information. Selecting an entry in this area causes a menu to be displayed.

Menu commands whose labels include **...**, for example `Source:Run`**...**, display a dialog box to prompt for information. Some of these commands can execute without displaying thier usual dialog box. They do so by using the dialog box's default settings and operating on the currently selected object or objects.

To force the display of the dialog box for these commands, for example, to specify nondefault settings, press the `Control` key while selecting the command. The command will not execute until you click on `OK` or `Apply` in the dialog box.

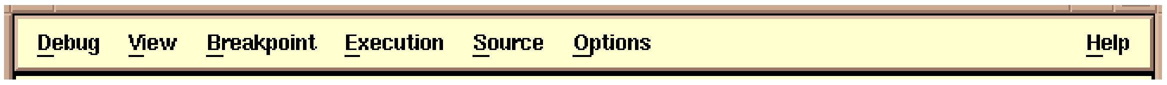


Figure 6-2 Main Menu Bar

The main menu bar contains the following pulldown menus:

Debug —provides commands for controlling the target program	
Run...	select an executable to debug, select program I/O mode
Sync	set or reset the debugger's knowledge of the target
Attach	attach to an already running executable
Detach	detach from an executable
Clear Debugger	remove all information from the debugger
Output	output window
Close Window	close the main or main task window
Quit Debugger...	exit the debugger
View – provides access to the debugger windows	
File	display a source view window on a specified file
Stack	display the call stack view window
Registers	display the registers view window
Floating Point Registers	display the floating point registesr view window

(Continued)

	Tasks	display the tasks view window
	Breakpoints	display the breakpoints view window
	Programs	display the program view window
	Any Commands...	display the view window for a user-specified command
Breakpoints — commands for setting, deleting and listing breakpoints		
	Break	set a breakpoint on the current line
	Delete Break	delete the breakpoint on the current line
	Break At Subprogram	break at the selected routine
	Break On Task	break at indicated task
	Break Down	set a breakpoint in the subprogram that called the current subprogram
	Break When...	break when the expression specified is true
	Break On Condition...	break when the condition is true
	Activate Break	activate the breakpoint on the current line
	Deactivate Break	deactivate the breakpoint on the current line
	List Breaks	list all breakpoints
Execution — commands for setting the application in motion		
	Step Line	single steps one line of source code
	Step Instruction	single steps one instruction
	Step With Signal	single steps one line and sends pending signal
	Advance Line	single steps one line of source code, skipping over any calls
	Advance Instruction	single steps one instruction, skipping over any calls
	Advance With Signal	single steps one line and sends pending signal, skipping over any calls
	Go	continues execution of the program
	Go Watch	go until the value of the variable given changes
	Go With Signal	go passing the pending signal to the program
	Run	runs or reruns the program
	Run To	runs the program to the line the cursor is on. If a breakpoint is encountered before that line, execution stops at that breakpoint.
	Interrupt	interrupts the program
	Catch Exception	catch the exception at the cursor location
	Catch All Exceptions	catch all exceptions in the program

(Continued)

	Propagate Exception	propagate the exception at the cursor location
Source — commands for printing objects and navigating source		
	Print	prints the value of the expression selected in the source window.
	Print .all	prints the dereferenced value of the expression selected in the source window.
	Assign...	pops up a dialog to request the left and right hand sides of the assignment statement.
	Visit	navigates to the source file of the routine or data selected in the source window. Each location visited is added to a stack of visited locations.
	Visit Spec	navigates to the source file of the specification of the routine selected in the source window. Each location visited is added to a stack of visited locations.
	Move Back	pops back to the previous location before the visit occurs. With Move Back you can keep popping out of the visit stack until it is exhausted.
	Search Forward	pops up the search dialog to request a pattern to search forwards through the source window.
	Search Back	pops up the search dialog to request a pattern to search backwards through the source window.
Options — commands for setting, saving, and loading options for both a.xdb and a.db		
	Set Debugger Options	set options controlling the debugger (a.db)
	Set Window Options...	set options controlling the GUI (a.xdb)
	Save Debugger Options	save the current debugger options
	Save Window Options	save the current window options
	Load Debugger Options	load debugger options from a file
	Load Window Options	load window options from a file
	Set Option File Name...	set the name of the options file

```
  (Continued)

Help — commands for getting help on a.xdb and a.db
            On Context          provide point and click help
            On Help             give help on using help
            On Window           give help on main window
            On Commands         give help on a.db commands
            On Keys             give help on key bindings
            On Version          produce the version box
```

6.3.4 User Configurable Hot Button Area

The user configurable hot button area contains user defined hot buttons. The user can add, delete, move, or modify buttons in this area using the `Set Window Options` window available on the main menu bar's Option pulldown menu.

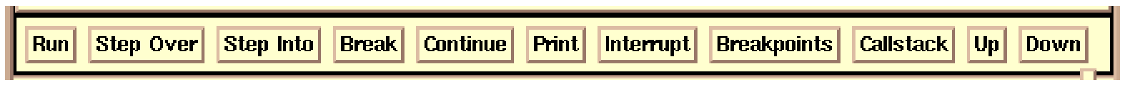


Figure 6-3 Hot Button Area

6.3.5 File Status Area

The file status area contains the name of the file being displayed in the main window's source pane.

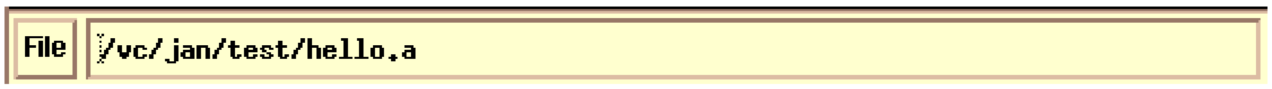
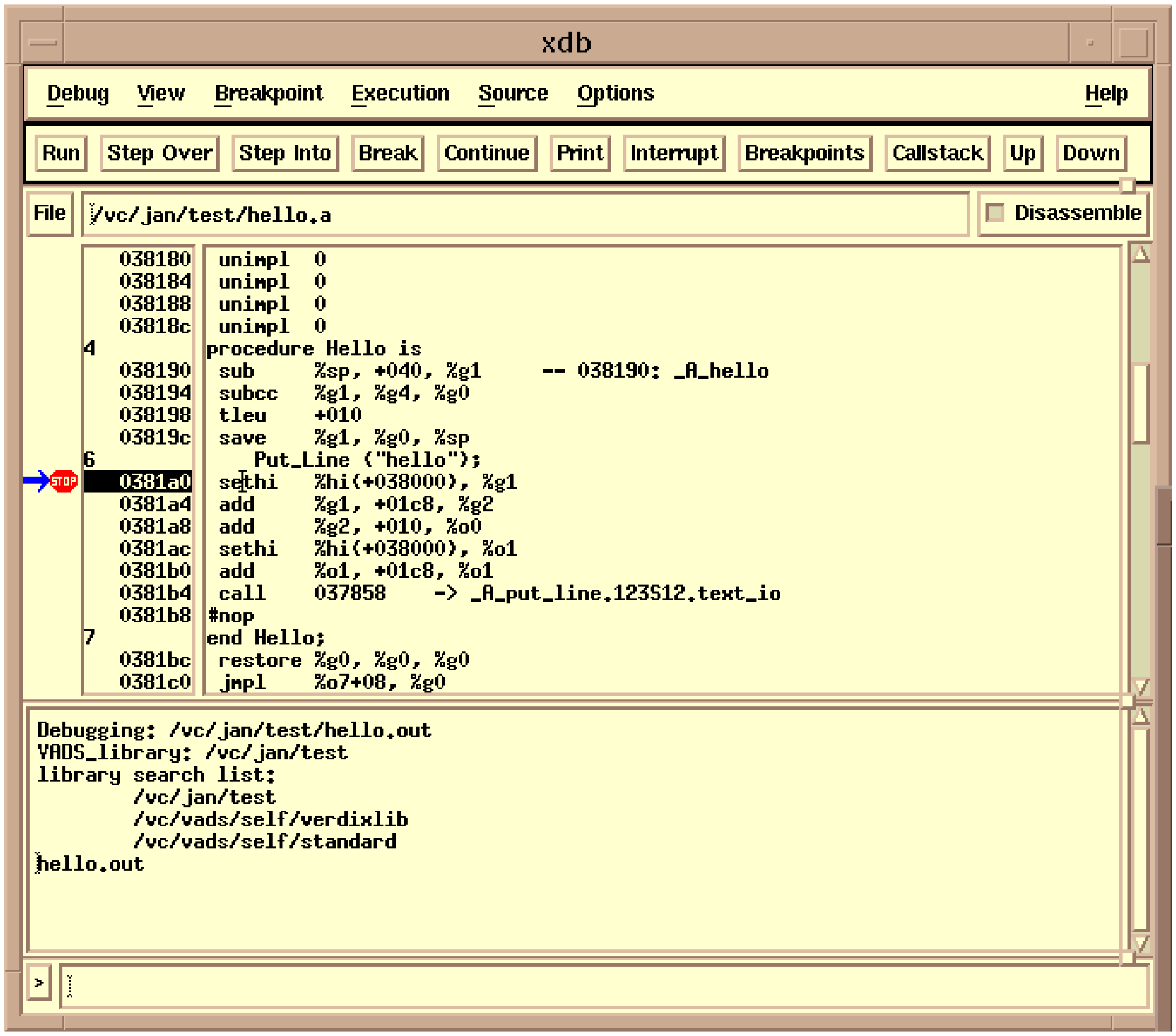


Figure 6-4 File Status Area

6.3.6 Disassemble Toggle

The `disassemble` toggle turns disassembly mode on and off. In disassembly mode, each source line is followed by its corresponding assembly code. The toggle button remains highlighted in disassembly mode.


xdb
Debug View Breakpoint Execution Source Options Help
Run Step Over Step Into Break Continue Print Interrupt Breakpoints Callstack Up Down
File
Disassemble
STOP
>

```
/vc/jan/test/hello.a

    038180  unimpl   0
    038184  unimpl   0
    038188  unimpl   0
    03818c  unimpl   0
4          procedure Hello is
    038190  sub      %sp, +040, %g1      -- 038190: _A_hello
    038194  subcc    %g1, %g4, %g0
    038198  tleu     +010
    03819c  save     %g1, %g0, %sp
6             Put_Line ("hello");
    0381a0  sethi    %hi(+038000), %g1
    0381a4  add      %g1, +01c8, %g2
    0381a8  add      %g2, +010, %o0
    0381ac  sethi    %hi(+038000), %o1
    0381b0  add      %o1, +01c8, %o1
    0381b4  call     037858     -> _A_put_line.123S12.text_io
    0381b8 #nop
7          end Hello;
    0381bc  restore %g0, %g0, %g0
    0381c0  jmpl     %o7+08, %g0
```

```
Debugging: /vc/jan/test/hello.out
VADS_library: /vc/jan/test
library search list:
        /vc/jan/test
        /vc/vads/self/verdixlib
        /vc/vads/self/standard
hello.out
```

Figure 6-5 .Dissemble Toggle

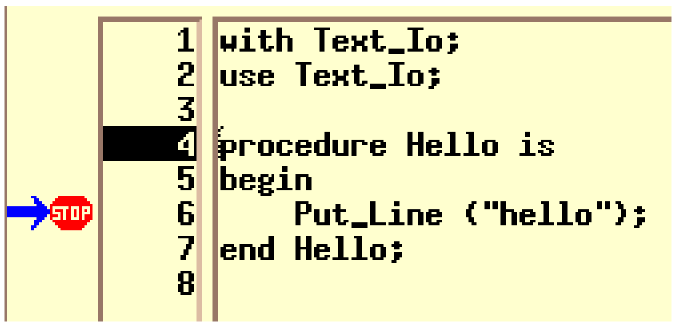
6.3.7 Source Indicators

The source indicator area contains icons to mark where the current line of execution is, and the current breakpoints. The current point of execution is displayed as an arrow. Active breakpoints are displayed as red stop signs. Inactive breakpoints are displayed as green stop signs. The source indicator area is illustrated in Figure User - 6 on this page.

6.3.8 Lines Window

The lines window contains the range of source lines visible in the main window's source pane. If the disassemble toggle is on then the lines window will contain the range of addresses.

The line indicating the current location of the cursor is highlighted. In this example, the cursor is on line 4. Note that the execution position indicated by the arrow is line 6.



```
1 with Text_Io;
2 use Text_Io;
3
4 procedure Hello is
5 begin
6      Put_Line ("hello");
7 end Hello;
8
```

Figure 6-6 Lines Window (example)

During debugging, line numbers are used primarily for setting breakpoints. Therefore there is a popup menu available in the lines window with the following options:

Break--sets a breakpoint		
	Break	set a break at current cursor location
	Break Down	set a breakpoint in the subprogram that called the current subprogram
	Break When	break when the expression specified is true
	Break On Condition	break when the condition is true
	Break At Instruction	set a breakpoint at the instruction where the cursor is located
	Break At Subprogram	break at the selected routine
Delete Break--deletes a breakpoint on the highlighted line		
Activate Break--activate a deactivated breakpoint		
Deactivate Break--deactivate an active breakpoint		
Run To--runs the program to the line the cursor is on. If a breakpoint is encountered before that line, execution stops at that breakpoint.		
List Breaks--lists all breakpoints		
Catch Exception--catch the exception at the cursor location		
Propagate Exception-propagate the exception at the cursor		

6.3.9 Main Window Source Pane

The main window's source pane contains the current source file. Normally this is the file containing the current point of execution, unless the user has changed to a different file, or moved up or down the call stack.

The source pane has a popup menu with the following options

Advance --single steps over subprogram calls		
	Advance Line	single steps to the next source line stepping over subprogram calls
	Advance Instruction	single steps one machine instruction over call instructions.
	Advance With Signal	steps and passes the signal/exception to the program
Step --single steps into subprogram calls		

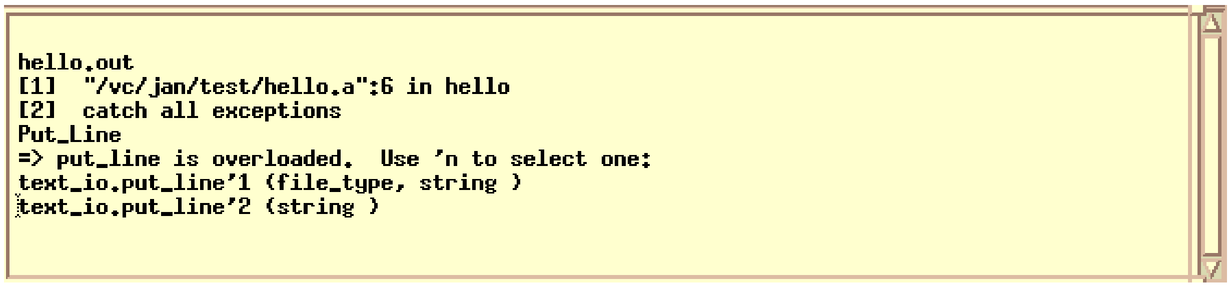
```
        Step Line              single steps one line of source code
                               stepping into subprogram calls
        Step Instruction       single steps one machine instruction
        Step With Signal       single steps and passes the
                               signal/exception to the program
Go --continues executing the program
        Go                     continues executing the program normally
        Go Watch               causes the program to execute until the
                               value of the named variable changes
        Go With Signal         continues executing the program and passes
                               the signal/exception to the program Run
                               runs or reruns the program Interrupt halts
                               the program being debugged
Run To --runs the program to the line the cursor is on.  If a breakpoint
         is encountered before that line, execution stops at that
         breakpoint.
Run --runs or reruns the program
Interrupt --interrupts the program
```

If you have selected text in the source pane, the popup menu contains the following options:

```
Print                  print the selected variable or expression
Print .all             print the dereferenced contents of the selected
                       variable or expression
Break At Subprogram    break at the selected routine
Break On Task          break at indicated task
Visit                  change the source view to the source attached to
                       the entity.
Visit Spec             change the source view to the source attached to
                       the entity.
Assign                 assign a selected variable to a named value
Call                   call the selected routine
Search Forward         search forward for the selected text
Search Back            search backward for the selected text
```

6.3.10 Debugger Output Area

The debugger output area displays the debugger response to commands, including debugger error messages.



```
hello.out
[1]  "/vc/jan/test/hello.a":6 in hello
[2]  catch all exceptions
Put_Line
=> put_line is overloaded.  Use 'n to select one:
text_io.put_line'1 (file_type, string )
text_io.put_line'2 (string )
```

Figure 6-7 Debugger Output Area

6.3.11 Debugger Command Area

The debugger command area is used for keyboard interaction with the debugger. It is located at the bottom of the screen. The prompt symbols indicate what the action the debugger will take:

>	Execute the a.db command. A new '>' prompt is displayed.
:	Execute the a.db command. Focus is returned to where the prompt was requested.
/	Search forward (from the cursor position) for the text.
?	Search backward for the text.

The debugger may display additional prompts here to request more information, such as when you enter a multi-line `a.db` command.

```
>
```

Figure 6-8 Debugger Command Area

6.3.12 Pane Adjusters

The pane adjusters allow you to adjust the size of each of the window panes in the main window. To adjust a pane, press and hold `MB1` and drag the pane up or down. Release `MB1` when the pane is the desired size.

6.3.13 View Windows

All view windows have the an `Update On` menu which allows you select the update interval of the window. The two choices are `On Breakpoint` and `On Demand`. The `On Breakpoint` option updates the view window every time the debugger stops to announce a breakpoint, single-step, or signal. The `On Demand` option updates the window only when the user hits the `UPDATE` push button.

Files... View

Selecting `Files`**...** displays a dialog box listing the contents of the current working directory and an area for entering a pathname. Selecting a file from the list or entering its path causes a source pane similar to the main window's source pane to be displayed containing the selected source file. All key bindings and popup menus available in the main window source pane are available in the File view. The File view is a static view of the source. As the current execution point changes the File view window is not scrolled to show the current execution point.

Stack View

A display of the contents of the call stack.

Registers View, Floating Point Registers View

A display of the register contents. Register values which change are highlighted with each update.

Tasks View

A display of task status. The tasks view window has a popup menu It is discussed in detail on page 13.

The radio buttons allow you to view the task as a textual list (Textual View), or as a set of icons (Graphical View).

Programs View

A display of the active programs status. (multiprogram debuggers only)

Any Commands/User Defined View

The Any Commands/User Defined View allows the user to enter any debugger command through the keyboard. This view can be used to create custom views of the output from debugger commands which are executed at a regular interval. For example, you can create a view to print out a variable `foo` after each breakpoint by using the command `p foo`.

Breakpoints View

The Breakpoints View window allows the user to interactively manipulate breakpoints. Supported operations are activate, deactivate, activate all, deactivate all, delete, delete all and visit. Double clicking in this window has the default behavior of visiting the breakpoint.

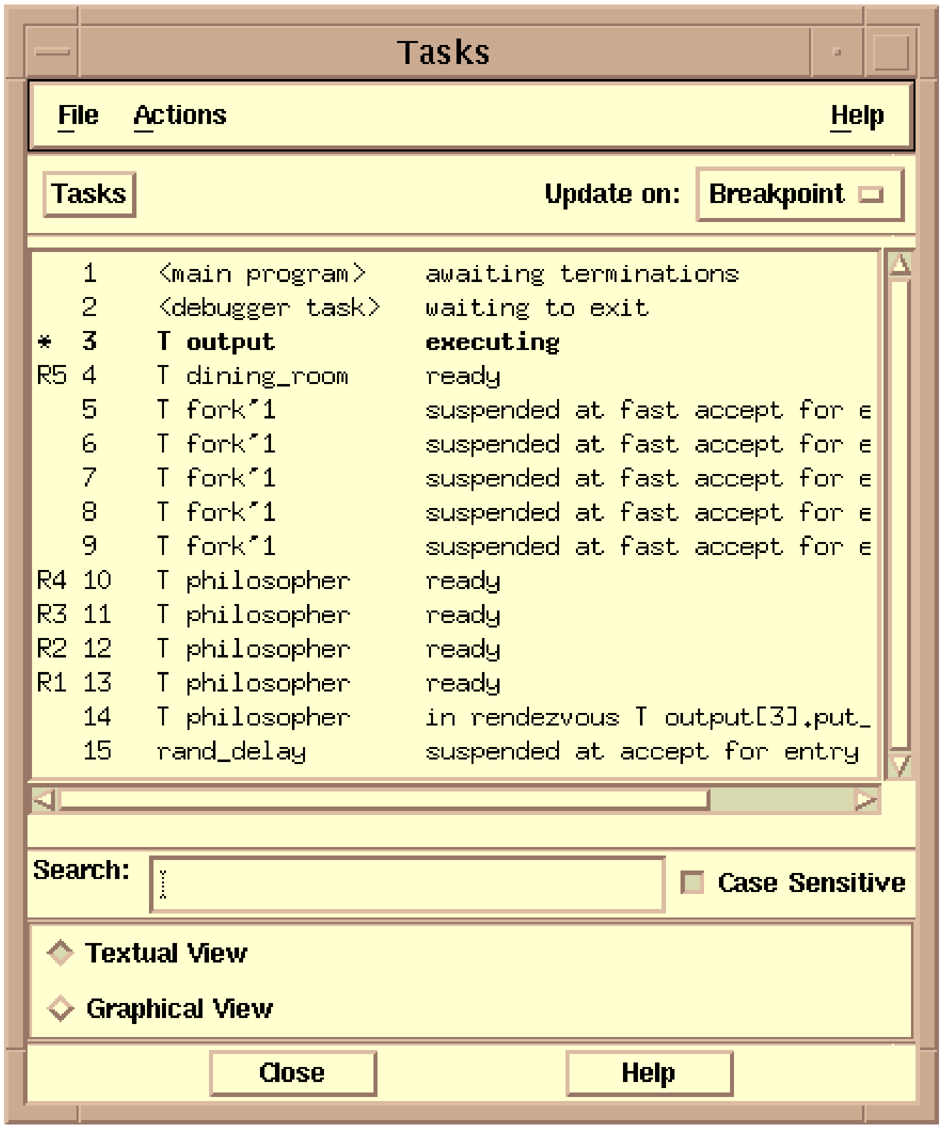
6.3.14 Tasks View

`Tasks View` displays of the status of all active tasks at an interval specified by the `Update On Menu`. In addition to displaying tasks, this window allows you to manage your tasks, and to affect their behavior through task level debugging commands. You can manage your tasks by creating new main windows which are focused on one or more tasks. See `Task Main Windows` on page 17 for more details. The `Tasks View` window allows you change the behavior of the tasks in your program through task level debugging commands. Using the task view window you can suspend, resume, abort, or change the priority of a task. In addition, you can query about the following kinds of information about the tasks.

The toolbar of the Tasks View window contains 3 entries: `File`, `Action`, and `Help`. The following menu commands are available from each of these buttons.The radio buttons allow you to view the task as a textual list, or as a

File			
File	Hide Task		hides the selected tasks from the view
File	Show Hidden Task		restores all hidden tasks
File	Filter		produces the filter box to allow filtering of the task information by state and/or regular expression pattern.
File	Sort By		
File	Sort By	Task Sequence Number	sort the task list by unique sequence number.
File	Sort By	Task Name	sort the tasks by name
File	Sort By	Task Status	sort the tasks by status
File	Close		close Tasks View window
Actions			
Actions	Task Information		prints out a more verbose output for the selected task(s) to a dialog box
Actions	Task Stack Use		prints out the stack use information for the selected task(s) to a dialog box
Actions	Thread Information		displays information about the threads layer (only applicable to systems that support threads)
Actions	Lightweight Process Information		lists information about the lightweight process layer (only applicable to systems that support lightweight processes)
Actions	Process Information		prints process information including pid, ppid, utime, stime, group and session id, flags and current status
Actions	Select Task		selects the selected task
Actions	Suspend Task		suspend the selected task(s)
Actions	Resume Task		resume the selected task(s)
Actions	Abort Task		abort the selected task(s)
Actions	Set Task Priority		brings up a dialog box to allow you to set the priority on the selected task(s)
Actions	Browse		creates a new main window focusing on the selected task

set of icons. Figure 6-9 on page 18 is an example of the Textual View.



```
    1    <main program>    awaiting terminations
    2    <debugger task>   waiting to exit
*   3    T output          executing
R5  4    T dining_room     ready
    5    T fork'1          suspended at fast accept for e
    6    T fork'1          suspended at fast accept for e
    7    T fork'1          suspended at fast accept for e
    8    T fork'1          suspended at fast accept for e
    9    T fork'1          suspended at fast accept for e
R4 10    T philosopher     ready
R3 11    T philosopher     ready
R2 12    T philosopher     ready
R1 13    T philosopher     ready
   14    T philosopher     in rendezvous T output[3].put_
   15    rand_delay        suspended at accept for entry
```

Figure 6-9 Textual View of Tasks List

The following columns of information are presented for each task:

`Queue #`	Position in the queue. This column may contain one of the following values: Rn indicates that the task is on the run queue in the nth position (R1 runs next) or Dn indicates that the task is on the delay queue in the nth position (the delay for D1 expires next). The task indicated by a + is the current task.
`Task Number`	Sequence nunber assigned to the task. This number is always 1 for the main task and is incremented every time a task is created.
`Task Name`	Task name or a T followed by the name of the task type that declared the task.
`Status`	State of the task and additional information for some states. A listing of all the task states is found under the lt command in the Debugger Reference.

A graphical interface to the Tasks View window is also available by selecting the `Graphical View` radio button. This view displays the currently active tasks as a list of icons with a label giving the task's name. There are four icons:

Task State	**Icon**
`Running`	man running
`Ready`	man in starting blocks
`Suspended`	man sitting down
`Terminated`	dead man

An example graphical display is shown in Figure 6-10.

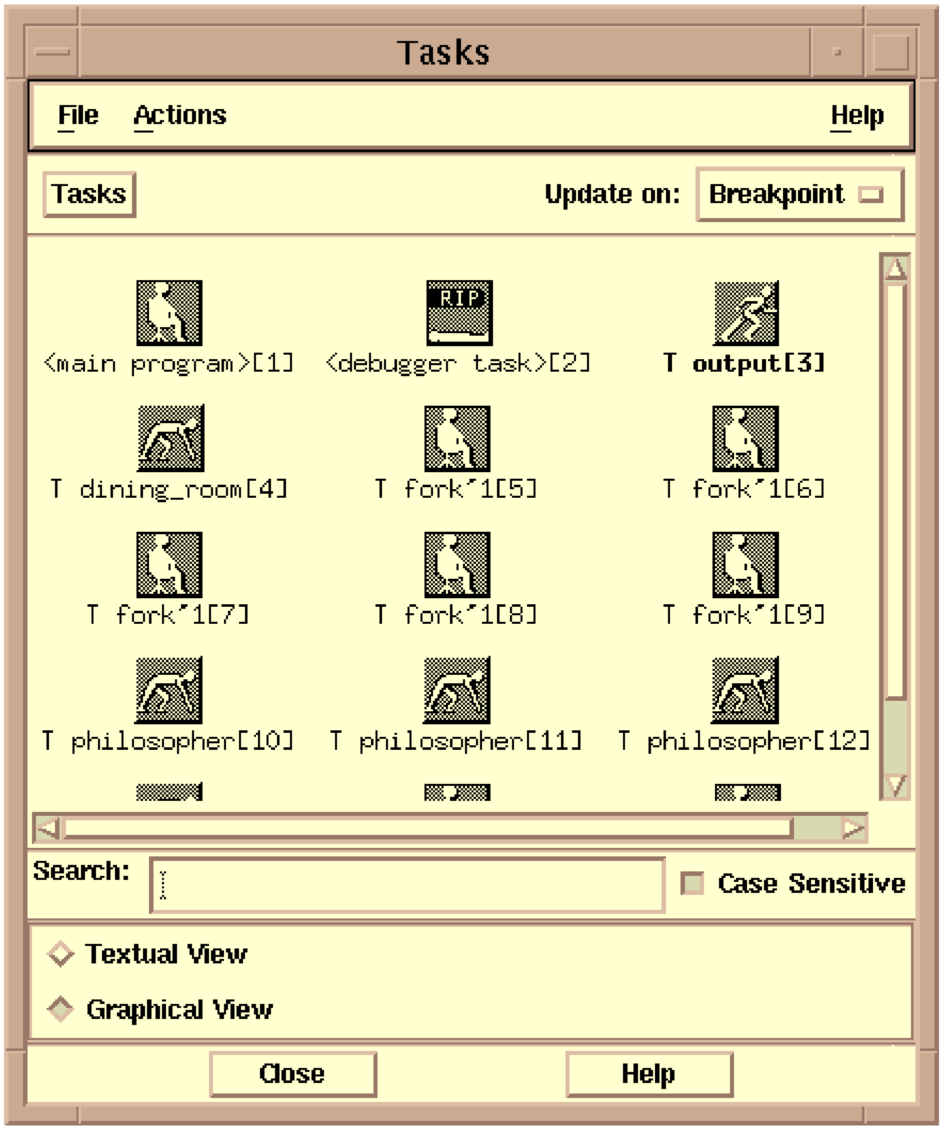


Figure 6-10 Tasks Graphical View

Caution – `Tasks View` is not supported for multiprogram debuggers.

6.3.15 Task Main Windows

You can manage your tasks by creating new main windows which are focused on one or more tasks. This creation of auxiliary main window is referred to as browsing. Each main window has a list of tasks associated with it. When you browse on one or more tasks, those tasks are removed from the main window list you selected them from, and moved to the list of tasks associated with the new main window. Any new tasks created during the program execution are added to the task list associated with the first first main window. Each main window allows you to select the task which will be the current context of that main window. The selected task is displayed in the source window and commands are executed in that task's context. The commands which are affected by a task's context are: `View:Registers`, `View:Floating Point Registers`, `View:Stack`, `View:Tasks`. If you do not select a task in the main window or windows, the following algorithm is used:

1. If there is only one main window, this main window will be in the context of the currently executing task.
2. If there is more than one main window then:
    - the currently executing task will be displayed in the main window which contains that task in it's list of tasks.
    - the remaining main windows will display the last task which was focused in this window. This task was focused either because it was previously the executing task, or was the first on the list of tasks to be browsed for an auxiliary main window.

The Tasks View Window is made up of the following areas:

`Task Menu Bar`	contains task commands
`Update On Menu`	allows the user to set the duration of update
`Tasks Button`	allows the user to update the task display on demand
`Task Display Area`	displays the active tasks as a list or as icons

Caution – `Task Main Windows` is not supported for multiprogram debuggers.

Program I/O is displayed according the the setting of an option menu to the `Run` dialog. The `Run` dialog is accessed through the `Debug` button on the `Main Menu Bar`. Three values are available:

`New xterm`	a new xterm with the title *`program name`* `Input/Output` is started up, and the program's `stdin` and `stdout` are directed to it. [Default]
`Text windows`	a new window with a scrolling text area is displayed. Program input and output will go to the new window, and program output is also copied to the output window. This mode offers no terminal emulation.
`Parent terminal`	input and output are directed to the `tty` from which `a.xdb` was started up.

To change the I/O mode, simply select from the option menu and click `Reload`.

To change the I/O mode for future runs, change the `programIOMode` resource (see Section 6.7, "Resource definitions," on page 6-33).

6.4 Keyboard Control

The default keyboard interface of `a.xdb` is similar to the interface of the `a.db` debugger. See the description of the `editStyle` resource for instructions on how to change it.

When the focus is in a source pane, most of the `a.db` screen mode commands are available. In the output window, all of the Window Control commands are supported except `I`, `[[`, `]]`, and `%`.

6.4.1 Window Control Commands

`<CONTROL-B>`	(backwards) move backward one full window
`[number]<CONTROL-D>`	(down) scroll down 1/2 window or *`number`* lines
`<CONTROL-F>`	(forward) move forward (down) one full window
`<CONTROL-G>`	print the name of file displayed in source pane
`<CONTROL-R>`	(redraw) redraw all windows (clean up display)
`[number]<CONTROL-U>`	(up) scroll up 1/2 window or *`number`* lines
`[number]G`	(go to) move to bottom or specified line of file **I**, toggle between instruction and source submodes
`Q`	leave screen mode; enter line mode
`[number]h or a`	move left one or *`number`* columns
`[number]j or b`	move down one or *`number`* lines
`[number]k or y`	move up one or *`number`* lines
`[number]l, <SPACE> or '`	move right one or *`number`* columns
`n`	(next) repeat the previous / or ? search
`yy`	yank line at cursor location (in either window) to command line (`<ESC>` required from insert mode to line-edit mode)
`[[`	move forward in the source file to the next procedure, function, package, task or declare block.
`]]`	move back in the source file to the previous procedure, function, package, task or declare block.
`/`	pattern search forward for pattern
`?`	pattern search backward for pattern
`:`	enter a debugger command line
`.`	(period) repeat the previous debugger command line
`,`	(comma) move to the other window
`0`	move to beginning of line

(Continued)	
`^`	move to first non-whitespace character on line
`$`	move to end of line
`%`	move forward or back to matching parenthesis or brace. `%` also finds the matching end when the cursor is placed on: `if`, `loop, for, while, case, record, select, function, procedure, package` or `task`. Likewise, `%` also moves the cursor back from `end if` to the corresponding `if`, back from `end procedure_name` to the corresponding procedure, etc.
`*`	move the cursor to the current home position

Figure 6-11 Window Control Commands

6.4.2 Immediate Debugger Commands

a	step to next source line over call statements
b	set a breakpoint on line containing cursor
[number]B	set breakpoint at subprogram (or qualified subprogram) name under cursor (using *number* to disambiguate an overloaded name)
cb	(call bottom) move current position and frame to bottom of stack
[number]cd	(call down) move down one frame or *number* frames on call stack
[number]cs	(call stack) display entire call stack or just *number* frames
ct	(call top) move current frame and current position to top of stack
[number]cu	(call up) move up one frame or *number* frames on call stack
d	delete breakpoint at line containing cursor
g	(go) continue program execution
p	display value of variable underneath cursor
P	delimit the expression under the cursor (for use with **p**)
P...a	print variable.all
P...*	print *variable
P...y	yank variable to command line. The variable will appear on the ommand line preceded by a **:p**.
r	(run) start or restart program execution
s	(step) step to next source line going into subprograms
yy	yank line at cursor location (in either window) to command line (<ESC> required from insert mode to line-edit mode)
[number]<CONTROL-]>	execute debugger **e** command for name of any declarable Ada entity under the cursor (using number to disambiguate an overloaded name). Placing the cursor anywhere within the first identifier of a qualified name and entering <CONTROL-]> takes you to the declaration of that qualified name.
<CONTROL-C>	interrupt the current debugger operation

Figure 6-12 Immediate Debugger Commands

When there is a prompt in the command window (either : or >) and keyboard focus is in that window, the a.db line-editing commands are available. In other text fields, such as those in the file selection boxes, all line-editing commands are supported except those that involve name completion, name listing, and command history.

6.4.3 Line-Editing

Command	**Mode**	**Meaning**
j	edit	go ahead (down) in history
A	edit	go to end of line and enter insert mode
a	edit	enter insert mode after character under cursor
k	edit	go back (up) in history
I	edit	move cursor to beginning of line and enter insert mode
0	edit	move cursor to beginning of line
^	edit	move to first non-white space on line.
c{Motion}	edit	change text [see Motion]
C	edit	change rest of line, from character under cursor
N	edit	search in the opposite direction in history for previous string
string<ESC>\	edit	complete *string* with name visible in current scope that begins with *string*. Beep sounds if 0 of >1 matches are found.
n	edit	search in the same direction in history for previous string
x	edit	delete character underneath cursor
d{Motion}	edit	delete text [see *Motion*]
D	edit	delete rest of line, from character under cursor
<RETURN>	both	transmit the current line to the debugger
<ESCAPE>	insert	end insert mode (or CHANGE mode) -- switch to edit mode
<ESCAPE>	edit	abort the current command
$	edit	move cursor to end of line
i	edit	enter insert mode before character under cursor
<CONTROL-H>	insert	erase last character, prints
<CONTROL-U>	insert	erase the entire line typed so far
string<ESC>=	edit	list all names in current scope that start with *string*. Beep sounds if no matches are found.
string<ESC>(#)	edit	complete the name using the # (as produced by the list names command above) for the name.
h	edit	move cursor one position left
l	edit	move cursor one position right
w	edit	move cursor to next word
rchar	edit	replace one character underneath cursor
?string	edit	search backward in history for a command that contains *string*
/string	edit	search forward in history for a command that contains *string*
<CONTROL-W>	insert	erase most recently typed word

Figure 6-13 Line Editing Commands

Motion

One of the following characters may be selected as a motion character for the CHANGE or DELETE editing functions. The motion character determines what is changed.

`w` - from the cursor through the rest of the word
`b` - from the beginning of the word to the cursor
`0` - from the beginning of the line to the cursor
`^` - from the first non-white space of the line to the cursor
`$` - from the cursor to the end of the line

For more information on these commands, consult the *SPARCompiler Ada Reference Guide.*

6.5 Quick Reference

The following section is a quick reference to the most commonly used features of the `a.xdb`.

How do I set breakpoints?

Breakpoints can be set in several ways. The easiest is to select the line number you want to break on in the line numbers window with `MB1`. With the cursor on top of the selected line number press and hold `MB3` to see the popup menu and select `Break`. Another way to set breakpoints is to use the key bindings. Select a line of text in the source pane with `MB1` and press `b`. In both cases, a red stop sign appears next to the breakpointed line as a visual cue. In addition to the above methods, the predefined `Break` hot button can be used in place of typing the `b` command.

How do I clear breakpoints?

Breakpoints can be cleared in a similar way. Either use the `d` (delete breakpoint) on the selected line, or use the `Delete Breakpoint` menu item in the popup menu available on the lines window. To delete all breakpoints, type `d all` in the command window.

How do I run my program?

To select the program and arguments, use the `Run...` option from the `Debug` menu. To rerun the program without changing the arguments, press the `Run` hot button. Alternatively, you can use the keyboard interface by typing `r`, or `r [`*`arguments`*`]` with focus in the command window.

How do I continue after hitting a breakpoint?

Either press the `Go` hot button or type `g` in the source pane.

How do I print variables?

Select the variable you want to print in the source pane by dragging `MB1` over the text. Either press the `Print` hot button or type the `p` command. To dereference the variable type `P *` or use the `Print *` button.

How can I see the registers?

Select the `Registers` option from the View pulldown menu. This gives you a new window containing the registers. As the registers change, they are highlighted. Specify the interval in which you want the registers to change using the option on the upper right side of the register window.

How do I modify a variable?

Select the variable you want to modify by dragging `MB1` in the source text of the main window. Press the `Assign` hot button and then enter the value you want to assign to the selected variable in the command window.

How do I make a call to a routine in the application I am debugging?

Use the keyboard interface. Move focus to the command window and type `p` *`routine_name`*`()` to print the return value of the routine.

How do I show the call stack?

Either use the `CallStack` hot button, or select the `Stack` option from the View pulldown menu. Alternatively, type `cs` in the source pane.

How do I move up and down the call stack?

Either use the `Up`, `Down`, and `Bottom` hot buttons, or use the key bindings `cu`, `cd`, and `cb` in the source pane.

How do I see disassembly?

In any source pane, press the `Disassembly` toggle button with `MB1`, or use the key binding `I` in the source pane.

How can I see another source file?

To change the source file in the main window source pane, press the `File` button in the upper left with `MB1`. This pops up a File Selection Dialog box, and allows you to select a file to view. You can also get a separate source view window by using the `File` option in the View pulldown menu. This produces a new file view window with many of the characteristics of the main source pane.

How do I control program input and output?

Program input and output are displayed according the setting of an option menu to the `Run` dialog (accessed through the `Debug` button on the `Main Menu Bar`). To change the I/O mode between `New xterm`, `text windows`, and `Parent xterm`. select the desired mode from the option menu and click `Reload`. The I/O modes are discussed in more detail in *Program I/O* on page 18.

How can I redirect the input or output of my user program?

Choose the `Run`**...** option from the Debug menu. To redirect your program's input and output to files add `< [`*`input_file`*`]` or `> [`*`output_file`*`]` to your arguments as you would from a shell. To change where your program's `stdin` and `stdout` are displayed, select one of the choices from the option menu. Click `Reload` to put your changes into effect.

How can I get help?

Use the Help menu items in the main window.

How do I exit?

Pull down the Debug pulldown menu and select `Quit Debugger`.

How do I switch to emacs?

We provide a set of emacs translations in the `app-defaults` file; set `editStyle` to `emacs` to use them. You can also use this mechanism to make your own modified tables.

6.6 Application Resources

The following custom resources alter **a.xdb**'s behavior:

Name	Type	Default
debugger	String	deb
stopColor	Pixel	Red
arrowColor	Pixel	Blue
errorBackgroundColor	Pixel	Orange
multipleArrows	Boolean	False
popupFollowPointer	Boolean	False
dialogLevel	Int	3
sourceHorizontalScroll	Boolean	False
outputHorizontalScroll	Boolean	False
viewHorizontalScroll	Boolean	True
activeButtons	String	″″
newButtons	String	″″
cannedButtons	String	default_canned_buttons
editStyle	String	vi
mainViewColumns	Dimension	80
mainViewSourceRows	Dimension	20
mainViewOutputRows	Dimension	10
mainViewCmdRows	Dimension	2
stackViewColumns	Dimension	40
stackViewRows	Dimension	10
stackViewUpdate	String	″breakpoint″
fileViewColumns	Dimension	80
fileViewRows	Dimension	10
fileViewUpdate	String	″demand″
programIOMode	String	″xterm″
programViewColumns	Dimension	80
programViewRows	Dimension	10
programViewUpdate	String	″demand″
stationViewColumns	Dimension	80
stationViewRows	Dimension	10
stationViewUpdate	String	″breakpoint″

Name	**Type**	**Default** *(Continued)*
registerViewUpdate	String	”breakpoint”
registerViewFont	XFontStruct	XtDefaultFont
registerViewFltUpdate	String	”breakpoint”
taskViewColumns	Dimension	80
taskViewRows	Dimension	10
taskViewUpdate	String	”breakpoint”
userCommandViewColumns	Dimension	40
userCommandViewRows	Dimension	10
userCommandViewUpdate	String	”breakpoint”
maxOutputWindowLines	Int	1024
xtermCommnad	String	”xterm”

Figure 6-14 Application Resources

6.7 Resource definitions

debugger	The path to the debugger used to communicate with.
stopColor	The color of the stop sign icon used to indicate breakpoints.
arrowColor	The color of the arrow icon used to indicate the point of current execution.
errorBackgroundColor	The color used as the background for windows displaying error messages.
multipleArrows	Setting this resource to TRUE causes all created file view windows to display the current position icon arrow if execution enters the source visible in one of the file view windows.
popupFollowPointer	Setting this resource to TRUE causes all popup dialogs to center on the mouse cursor. Setting this resource to FALSE centers the dialog on the main window, unless you have previously moved the dialog.
dialogLevel	How likely you are to get dialog boxes for certain events. Currently this resource only affects error messages. For values less than 3, you will get error messages displayed in the command area. Otherwise, they are displayed in dialog boxes.
sourceHorizontalScroll	Setting this resource to TRUE adds a horizontal scrollbar on the main window's source pane
outputHorizontalScroll	Setting this resource to TRUE adds a horizontal scrollbar on the main window's output window pane
viewHorizontalScroll	Setting this resource to TRUE adds a horizontal scrollbar on all view windows' text panes
activeButtons	The list of currently active hot buttons which appear at the bottom of the main window.
newButtons	The list of new buttons which you have defined.
cannedButtons	The list of buttons defined by default.

(Continued)	
editStyle	This string is used as the prefix to form names for the translation tables that a.xdb will use for text areas. For example, with the default **vi**, the following tables are used:
	viTextKeymap **--**Any multi-line text area.
	viTextCommandKeymap **--**Additional translations for the Source and Output windows.
	viTextFieldKeymap --All single-line text areas.
	We provide a set of emacs translations in the app-defaults file; set editStyle to emacs to use them. You can also use this mechanism to make your own modified tables.
mainViewColumns	The size in columns of the main window's source pane.
mainViewSourceRows	The size in rows of the main window's source pane.
mainViewOutputRows	The size in rows of the main window's debugger output pane.
mainViewCmdRows	The size in rows of the main window's debugger command pane.
stackViewColumns	The size in columns of the stack window's text pane.
stackViewRows	The size in rows of the stack window's text pane.
stackViewUpdate	The update frequency of the stack window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
fileViewColumns	The size in columns of the file window's source pane.
fileViewRows	The size in rows of the file window's source pane.
fileViewUpdate	The update frequency of the file window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
maxOutputWindowLines	The maximum number of lines buffered in the debugger output pane.
programIOMode	Inititial program I/O mode. Possible values are xterm, text and tty. [Default: xterm]
programViewColumns	The size in columns of the program window's text pane.
programViewRows	The size in rows of the program window's text pane.
programViewUpdate	The update frequency of the program window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
registerViewUpdate	The update frequency of the register window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
registerViewFont	The font used to display register views.

(Continued)	
registerViewFltUpdate	The update frequency of the floating point register window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
taskViewColumns	The size in columns of the task window’s text pane.
taskViewRows	The size in rows of the task window’s text pane.
taskViewUpdate	The update frequency of the task window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
userCommandViewColumns	The size in columns of the user command window’s text pane.
userCommandViewRows	The size in rows of the user command window’s text pane.
userCommandViewUpdate	The update frequency of the user command window. Possible values are breakpoint (update on every breakpoint) and demand (update only when update button is pressed).
xtermCommand	Command used to start up an xterm. The normal command-line options to xterm are allowed (e.g., xterm -geometry +0 +0 causes the xterm to be placed in the upper left corner. [Default: xterm -v]

6.8 Setting resources

For examples on how to set standard and custom resources for **a.xdb**, see the example application resource file in

```
SCAda_location/lib/app-defaults/Xdb        -- color version
SCAda_location/lib/app-defaults/Xdb-mono      -- monochrome version
```

“The best teacher ... is not the one who knows most, but the one who is most capable of reducing knowledge to that simple compound of the obvious and the wonderful.”
H.L. Mencken

Tutorial *7*

This chapter contains tutorials covering the use of the SC Ada compiler and the SC Ada debugger. Both line mode and screen mode use of the debugger are shown. Complete these tutorials to gain familiarity with the operation and capabilities of the SC Ada compiler and debugger. Note that much of the output in these tutorials is system specific and varies for each system. Complete the tutorials as presented using any system but your displayed output may be different than that presented in the tutorials.

7.1 SC Ada Compiler And Project Development Tutorial

This tutorial focuses on the use of the SC Ada commands in the development, implementation and organization of an Ada programming project.

To gain a better understanding of the compiler tools available, work through this tutorial on your system. The source files to begin the tutorial are found in the `*SCAda_location*/examples` directory. Complete listings of the source files are provided within the tutorial. Successful completion of the tutorial requires some knowledge of the Ada language as you must modify the provided code. All listed code is formatted using the `a.pr` (SC Ada formatter) utility.

7.1.1 Getting Help (a.help)

An interactive help utility is available for each of the SC Ada commands and concepts. If the local system administrator has installed the reference manual entries for the compiler and tools, access them by entering:

```
% man SCAda_command
```

Access on-line help with the **a.help** utility. Without a specified subject, `a.help` provides information on the use of the help utility. Otherwise, use the following format.

```
% a.help SCAda_command
```

7.1.2 Project Specification

This tutorial project is to write and test a date translation program. It must take a date of the form, "November 6, 1980" and translate it to the form, "11/6/80". If a spelling or syntax error occurs, your program returns an error message. Only dates in the 20th century are translated.

This project must be organized so that the test programs are kept in a different directory from the date conversion files.

7.1.3 Setting up the Project Library (a.mklib)

Ada development must occur inside an SC Ada library. An SC Ada library is simply a file directory that is initialized using `a.mklib`. Create a directory in which to do this tutorial and enter it. In this example, the project library is held in `/usr1/tutorial`.

```
% mkdir /usr1/tutorial
% cd /usr1/tutorial
```

Because you are unsure of the location of the Ada library on your system, use the `a.mklib` command with the `-i` (interactive) option to create an Ada library in which to develop your Ada programs.

```
% a.mklib -i
```

This command lists all of the Ada compilers available on the system. Select the version by entering its number.

```
2 versions of SC Ada are available on this machine:
  Target Name   Version     Ada Location
 1  SELF_TARGET    3.0         /usr2/ada_2.1/self
                               host_name, host_os, version_no
 2  SELF_TARGET    3.0         /usr2/ada_2.1/self_thr
                               host_name, host_os, version_no
Selection (q to quit):
```

Note – If you know the location of the SC Ada compiler, select it by entering its path name. Unless otherwise specified, the current working directory is initialized.

```
% a.mklib . /usr2/vads/ada_2.1/self/standard
```

To initialize a different directory, its complete path must be entered:

```
% a.mklib -i /usr2/work
```

To use the features provided by `verdixlib`, include it in the project library. This is done automatically when `a.mklib` is used interactively (**-i** option). If you do not use **-i**, you must manually include it using the `a.mklib` command with the path name to `verdixlib` specified.

After the library is created (the `GVAS_table`, `ada.lib` and `gnrx.lib` files and `.lines`, `.imports`, `.nets`, and `.objects` directories), you are ready to begin entering your source code.

References

`a.mklib`, *SPARCompiler Ada Reference Guide*

`verdixlib`, *SPARCompiler Ada Reference Guide*

user library, *SPARCompiler Ada Reference Guide*

7.1.4 SC Ada File Naming Conventions

Many systems have file naming conventions to make it easier for the users to keep track of their specifications, bodies and subunits. In SC Ada, the following conventions are used:

Type of Unit	**Naming Convention**
Specification	*name*.a
Body	*name*_b.a
Subunit	*name*_s.a

Figure 7-1 File Naming Conventions

It is highly recommended that these conventions be used. We recommend that the name used to identify the file be the same as the name of the main package, procedure or task defined in the file.

SC Ada does not require that Ada source files end with **.a**. The source files in these tutorials do, however, to maintain consistency with earlier releases of SC Ada.

7.1.5 Entering the Source Code

The two ways to obtain the code to complete this tutorial are to copy them from the *SCAda_location*/examples directory or enter them manually. All the source files needed in the tutorial are included in the *SCAda_location*/examples directory. Just copy the source file(s) into your project library. The five source files required to start this tutorial are as follows:

- convert.cmp
- convert_b.cmp
- convert_s.cmp
- iio.a
- test_convert.a

The last file (test_convert.a) (according to the project specification) compiles in a separate directory.

The simplest way of getting the files where you need them is to copy them all now. To do this, follow Example User - 1. Move the `test_convert.a` file into its final location later in the tutorial. Be sure you copy the files with the `.cmp` suffix after the files with the **.a** suffix. This ensures you have the correct version of each file to start the tutorial.

```
% cp /examples/test_convert.a .
% cp /examples/iio.a .
% cp /examples/*.cmp .
```

Figure 7-2 Copy the Source Files

Now change the `.cmp` suffix to `.a`. Do this for each file individually or as a group. To do it as a group, enter the following in `csh`:

```
% foreach file (*.cmp)
? mv $file $file:r.a
? end
```

or, copy each file individually and change its suffix.

```
% cp /examples/convert.cmp convert.a
% cp /examples/convert_b.cmp convert_b.a
% cp /examples/convert_s.cmp convert_s.a
```

Since all files provided in the `examples` directory have read-only permissions, you must now change the permissions on these files to allow editing. Simple editing is required as part of this tutorial.

```
% chmod +w *.a
```

Also, you can enter the files manually using a system editor. If you choose to do this, follow the SC Ada naming conventions and enter a preliminary package specification called `convert.a`. The source for this specification is listed in the next section. The `vi(1)` editor is used in this example.

7.1.6 Compiling the Program (ada and a.error)

Compile the package specification held in `convert.a` using the following command and receive an error message.

```
% ada convert.a
/usr1/tutorial/convert.a line 16, char 2:syntax error:”;” inserted
```

Figure 7-3 Compile the Package Specification

At this point, you have several options available for tracking down this bug. You can edit `convert.a`, go to line 16, and hope to insert the correct syntax. However, you can get more detailed information from the `a.error` tool.

The **-v** option (vi) of `a.error` enables you to identify and correct errors in one step with the visual editor. To list the code with embedded error messages, use `a.error` with the `-l` option (list).

Generally, `a.error` is called from the `ada` command line using the `-ev` or `-el` options. If the `-ev` option is used, an editor is automatically called to enable you to edit the file directly. It is difficult to show the operation of the editor in the documentation so we use the `-el` option. The listings produced by the `-ev` and `-el` options are virtually identical.

Type the following command to display the error listing.

```
% ada -el convert.a
**************************  convert.a  **************************
   1:package CONVERT is
   2:--
   3:-- This package is responsible for the conversion of dates from
   4:-- the English longhand notation to a numerical notation.  The
   5:-- date must be of the form ”month day, year”.  It is returned
   6:-- in the form month/day/year.  Only dates in the 20th century
   7:-- are to be translated (January 1, 1900 to December 31,1999).
   8:-- If the entered date is not syntactically correct, an error
   9:-- is returned.
  10:--
  11: type DATE_REP is
  12: record
  13:   MONTH:   INTEGER;
  14:   DAY:     INTEGER;
```

```
 (Continued)
  15:  YEAR:   INTEGER
  16: end record;
A ------^
A:syntax error: ";" inserted
  17:
  18: function GET_DATE_REP(DATE: STRING) return DATE_REP;
  19:
  20:end CONVERT;
```

The error messages indicate that we forgot a semi-colon. Edit the file and insert the semi-colon at the end of line 15.

```
% vi convert.a
.
.
```

Recompile the package specification. It should now compile correctly.

```
% ada convert.a
```

Now begin work on the body of this package. The body must implement the procedure to do the actual date translation and error checking. If you have not obtained this file, copy it from the `SCAda_location/examples` directory or enter it manually. After it is generated, compile it.

```
% ada -el convert_b.a | more
**************************  convert_b.a
****************************
       1:package body CONVERT is
       2:--
       3:-- This package body is responsible for doing the actual
      4:-- translation of the date from English to numeric format.
      5:-- MONTH_NO returns the proper month number associated with
       6:-- the entered month by comparing the entry with the
       7:-- MONTH_TAB array.  GET_DATE_REP processes the string
       8:-- entered, finding each number and the required comma.
       9:--  GET_NUMBER is called to process the numbers in the date
      10:-- string.  The values found are held in D.  If an error is
      11:-- found at any time, D.all is returned.  The value of the
      12:-- D.MONTH field will be checked by the test program.  If it
      13:-- is zero, this indicates an error has occurred.
      14:type ASTRING    is access STRING;
      15:type MONTH_LIST is array(1 .. 12) of ASTRING;
      16:MONTH_TAB : MONTH_LIST :=
      17:(1 => new STRING'("January"),
      18: 2 => new STRING'("February"),
      19: 3 => new STRING'("March"),
      20: 4 => new STRING'("April"),
      21: 5 => new STRING'("May"),
      22: 6 => new STRING'("June"),
      23: 7 => new STRING'("July"),
      24: 8 => new STRING'("August"),
      25: 9 => new STRING'("September"),
      26:10 => new STRING'("October"),
      27:11 => new STRING'("november"),
      28:12 => new STRING'("December"));
      29:
      30:function MONTH_NO(MONTH : STRING) return INTEGER is
      31:begin
      32:for I in MONTH_TAB'range loop
      33:  if MONTH = MONTH_TAB(I) then
    A ----------------------^
    A:error: RM 4.5.2: no operator visible for string "=" astring
```

```
(Continued)
     34:   return I;
     35:   end if;
     36:end loop;
     37:return 0;
     38:end MONTH_NO;
     39:
     40:function GET_DATE_REP(DATE : STRING) return DATE_REP is
     41:type ADATE is access DATE_REP;
     42:D         : ADATE   := new DATE_REP'(0, 0, 0);
     43:THE_MONTH : INTEGER;
     44:
     45:NEXT      : INTEGER := DATE'FIRST;
     46:
     47:begin
     48:while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
     49:   NEXT := NEXT + 1;
     50:end loop;
     51:
     52:if NEXT >= DATE'LAST then
     53:   return D.all;    -- error exit;
     54:end if;
     55:
    56:     -- we postpone assigning the month into d.month since 0
     57:      -- indicates an error and it is 0 now.
     58:
     59:THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
     60:if THE_MONTH = 0 then
     61:   return D.all;    -- error exit;
     62:end if;
     63:
     64:       -- get the day
     65:GET_NUMBER(D.DAY, NEXT);
   A ---------^
   A:error: RM 8.3: identifier undefined
     66:if NEXT = 0 then
     67:   return D.all;    -- error exit;
     68:end if;
     69:
     70:       -- find the comma
     71:NEXT := NEXT + 1;
     72:while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ',') loop
     73:   NEXT := NEXT + 1;
     74:end loop;
     75:
```

```
(Continued)
     76:  if NEXT >= DATE'LAST then
     77:   return D.all;    -- error exit;
     78:  end if;
     79:
     80:    -- make sure there is a numeric entry in the year field
     81:while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= '1') loop
     82:   NEXT := NEXT + 1;
     83:end loop;

     84:
     85:  if NEXT >= DATE'LAST then
     86:   return D.all;    -- error exit;
     87:  end if;
     88:
     89:            -- get the year and verify it is in the 1900s
     90:GET_NUMBER(D.YEAR, NEXT);
   A ---------^
   A:error: RM 8.3: identifier undefined
     91:if (NEXT = 0) or (D.YEAR < 1900) or (D.YEAR > 1999) then
     92:   return D.all;    -- error exit;
     93:end if;
     94:
     95:D.MONTH := THE_MONTH;
     96:return D.all;
     97:end GET_DATE_REP;
     98:end CONVERT;
```

Figure 7-4 Compile the Package Body

Several problems exits in this package body. Edit the file.

```
% vi convert_b.a
```

The first problem is indicated on line 33. It is caused by forgetting to dereference the pointer to do the string compare. Correct this by dereferencing the pointer expression by appending `.all` to it.

```
if MONTH = MONTH_TAB(I).all then
```

The other problems are the result of neglecting to define `procedure GET_NUMBER`. Insert a procedure definition following the assignment of the array `MONTH_TAB` (after line 28).

```
procedure GET_NUMBER(RET_VAL, PLACE_IN_STRING: out
INTEGER)
is separate;
```

This specifies that `GET_NUMBER` is contained in a subunit.

Compile the package body again. It should compile without errors.

```
% ada convert_b.a
```

You must define a package to display or print an integer. This requires an instantiation of `TEXT_IO.INTEGER_IO` with an `INTEGER` parameter. Place this package instantiation in a separate Ada unit named `iio.a` (if you have not done so, copy it from the *`SCAda_location`*`/examples` directory or enter it directly).

```
with TEXT_IO;
package IIO is new TEXT_IO.INTEGER_IO(integer);
-- This package enables the printing and display of
-- integers.
```

Figure 7-5 Define TEXT_IO Package

Compile this package specification. It should have no errors.

```
% ada iio.a
```

It is now time to write your test program. This program must be placed in a separate directory. Specifications for the program are described on page 2. Use of files in different locations is a common practice in project development.

Create a new Ada project library under your current working directory.

```
% mkdir tests
% cd tests
% a.mklib -i
```

Figure 7-6 Set Up the Test Program

The test program is held in `test_convert.a`. If you copied all the tutorial files into your main project library, move `test_convert.a` from that directory into tests. If you did not, copy it from the *`SCAda_location`*`/examples` directory or enter it manually.

```
% mv /usr1/tutorial/test_convert.a .
```

or

```
% cp /examples/test_convert.a .
% chmod +w test_convert.a
```

or

```
% vi test_convert.a
    .
    .
    with CONVERT;
    with TEXT_IO;
    use  TEXT_IO;
    with IIO;
    procedure TEST_CONVERT is
    --
    -- This test file contains a list of correct and incorrect dates
    -- to be tested.  Each date is analyzed by TEST_SINGLE_DATE. It
    -- calls the GET_DATE_REP procedure defined in CONVERT.
    -- The entered date is displayed.  The value of DAY.MONTH is
    -- then checked.  If it is 0, an error has occurred in the
    -- processing of the entered date string and an error message is
    -- displayed.  Otherwise, the numeric equivalent of the date is
    -- displayed.
    --
    procedure TEST_SINGLE_DATE(DATE : STRING) is
       DAY : CONVERT.DATE_REP;
       TURN_CENTURY: constant STRING(1..2) := "00";
    begin
       DAY := CONVERT.GET_DATE_REP(DATE);
       PUT('"');
       PUT(DATE);
       PUT(""" -- ");

       if DAY.MONTH = 0 then
          PUT_LINE(" ERROR");
          return;
       end if;
```

```
(Continued)
          IIO.PUT(DAY.MONTH, 2);
          PUT('/');
          IIO.PUT(DAY.DAY, 2);
          PUT('/');
          if (DAY.YEAR = 1900) then
             PUT(TURN_CENTURY);
          else
             IIO.PUT(DAY.YEAR - 1900, 2);
          end if;
          NEW_LINE;
     end TEST_SINGLE_DATE;
     begin
             -- first test some good dates

     TEST_SINGLE_DATE("January 1, 1900");
     TEST_SINGLE_DATE("December 31, 1999");
     TEST_SINGLE_DATE("November 26, 1983");
     NEW_LINE;
             -- now test some bad dates

     TEST_SINGLE_DATE("Aug. 1, 1981");   -- bad month
     TEST_SINGLE_DATE("March 16 1986"); -- no comma
     TEST_SINGLE_DATE("February ");      -- no day, no comma, no year
     TEST_SINGLE_DATE("July 3");         -- no comma, no year
     TEST_SINGLE_DATE("May 2, ");        -- comma, no year
     TEST_SINGLE_DATE("Une 19, 1885");   -- invalid year
     end TEST_CONVERT;
```

References

`ada`, *SPARCompiler Ada Reference Guide*

`a.error`, *SPARCompiler Ada Reference Guide*

7.1.7 Modify Library Search List (a.path)

You cannot successfully compile the test program until you make the compiled date conversion source files in `/usr1/tutorial` visible to the compiler in your `tests` library. Do this by connecting that directory to your library search path using the `a.path` command with the interactive option (`-I`).

```
% a.path -I
```

The following menu is displayed.

```
     1. List local library search list ?(ADAPATH)
     2. Append entire library search list to ADAPATH ?
     3. Append to library search list ?
     4. Insert into library search list ?
     5. Remove from library search list ?
     6. Remove all EXCEPT from library search list ?
     7. Cleanup the library search list ?
     8. Exit?

    Which option?  (1-8) ->
```

Enter **4** to insert your main project library (`/usr1/tutorial`) into the library search list for this directory. The following is displayed.

```
    The current SC Ada library search list path:
     1.  /self/verdixlib
     2.  /self/standard
Please enter the desired SCAda library:
```

Enter `/usr1/tutorial` and the following appears.

```
Please enter the position number of where to insert the new library:
(enter 0 to insert at the start of path)
```

Enter `0` to insert the library at the start of the path. The new library search list is displayed followed by the original menu. Enter `8` to exit.

```
    library search list:
        /usr1/tutorial
        /self/verdixlib
        /self/standard
     1. List local library search list ?(ADAPATH)
     2. Append entire library search list to ADAPATH ?
     3. Append to library search list ?
     4. Insert into library search list ?
     5. Remove from library search list ?
     6. Remove all EXCEPT from library search list ?
     7. Cleanup the library search list ?
     8. Exit?
    Which option?  (1-8) ->
```

Now anything compiled in `/usr1/tutorial` is visible to your test program. Compile the test program.

```
% ada test_convert.a
```

References

`a.path`, *SPARCompiler Ada Reference Guide*

7.1.8 Keeping Your Compilations Straight (a.make)

Things are starting to get more complicated. Four files (iio.a, convert.a, convert_b.a, test_convert.a) in two libraries are defined and implementation has just started. Soon you will have many additional files, especially if you follow good programming practices and (1) define many easily tested small units and (2) keep most of the units in individual files. Fortunately, SC Ada provides a tool, a.make, to help you keep all your files up to date. This tool determines which files need recompilation and compiles them in the correct order. It calls the linker to create an executable if the entered unit name is a procedure or integer function.

Enter the required command.

```
% a.make -v test_convert -o test_convert -f *.a
```

The -v (verbose) option lists the recompilation commands as they are executed. The test_convert argument represents the main unit being operated on; -o test_convert specifies that the executable file is named test_convert (if it is not specified, the executable is named a.out by default); the -f option indicates that all remaining non-option arguments are filenames. Thus, all files ending in .a are checked and an executable generated. The following output is displayed.

```
/usr/ada_2.1/bin/a.ld test_convert -o test_convert
a.ld error: RM 10.5: missing body for spec of convert.get_number
```

You forgot to write the convert subunit containing GET_NUMBER. This is the subunit you are using to get the day and year from the input date string.

Return to your original project library (/usr1/tutorial**)**.

```
% cd /usr1/tutorial
```

or

```
% cd ..
```

You must generate the subunit, convert_s.a. If you have not copied it from the *SCAda_location/*examples directory (it is there as convert_s.cmp), do so now or enter it manually. Because you want to test the rest of the conversion procedures, this is only a dummy version of the procedure. This is a common practice in Ada programming. This dummy version of the procedure generates final dates with a day value of 1955 and a year value of 55 (1955-1900).

```
% cp /examples/convert_s.cmp convert_s.a
% chmod +w convert_s.a
```

or

```
% vi convert_s.a
    .
    .
separate(CONVERT)
procedure GET_NUMBER(RET_VAL, PLACE_IN_STRING: out INTEGER) is
--
-- This subunit returns the integer equivalent of the day and
-- year in the date string being processed. It is currently
-- just a dummy version and should generate dates in syntactically
-- correct date strings with a day value of 1955, a year value of
-- 55 and a string position of 5.
--
begin
    RET_VAL := 1955;
    PLACE_IN_STRING := 5;
end GET_NUMBER;
```

Compile the subunit. You should get no errors.

```
% ada convert_s.a
```

References

a.make, *SPARCompiler Ada Reference Guide*

7.1.9 Generating an Executable File (a.ld)

After successfully compiling the subunit, you must again attempt to generate an executable file. Since this executable is in your `tests` directory, go to that directory. Generate the executable by calling the linker (`a.ld`). By default, SC Ada names the executable file `a.out`. Use the `-o` option and enter the new executable filename (`test_convert`).

```
% cd /usr1/tutorial/tests
% a.ld test_convert -o test_convert
```

Figure 7-7 Generate an Executable

Run the program.

```
% test_convert

"January 1, 1900" --  1/1955/55
"December 31, 1999" -- 12/1955/55
"November 26, 1983" --  ERROR

"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
"May 2,  " --  ERROR
"June 19, 1885" --  6/1955/55
```

References

`a.ld`, *SPARCompiler Ada Reference Guide*

7.1.10 Improving the Program

You now decide to improve the convert_s.a subunit so that it gets the day and year and does the exception handling required in the specification. You do this work in a temporary Ada library called development. Create a new directory under your original project library (/usr1/tutorial) and call it development. Enter the new directory and initialize it using a.mklib.

```
% mkdir /usr1/tutorial/development
% cd /usr1/tutorial/development
% a.mklib -i
```

Figure 7-8 Improve the Program

Redo convert_s.a to get the actual values from the passed string and to report errors. Copy the convert_s.a file from the /usr1/tutorial directory and enter the changes manually or copy the convert_s.cmp1 file from the *SCAda_location*/examples directory.

```
% cp /examples/convert_s.cmp1 convert_s.a
% chmod +w convert_s.a
```

or

```
% cp /usr1/tutorial/convert_s.a .(or % cp ../convert_s.a .)
% vi convert_s.a
    .
    .
with IIO;
separate(CONVERT)
procedure GET_NUMBER(DATE_STRING : STRING;
            RET_VAL, PLACE_IN_STRING : out INTEGER) is
-- This procedure calls IIO.GET to read an integer value from the
-- beginning of DATE_STRING.  It returns in RET_VAL, the integer
-- value that corresponds to the input sequence.  It returns in
-- PLACE_IN_STRING the place value in the string of the last
-- character read.  If an error is detected, the place value is
-- set to 0.
--
begin
    IIO.GET(DATE_STRING, RET_VAL, PLACE_IN_STRING);  --RM
14.3.7(13)
```

```
exception
    when DATA_ERROR =>
    PLACE_IN_STRING := 0;

end GET_NUMBER;
```

Since you changed the call to GET_NUMBER, you must now correct the procedure specification and calls in convert_b.a. Copy this file from the main project library (tutorial).

```
% cp /usr1/tutorial/convert_b.a .
```

or

```
%  cp ../convert_b.a .
```

and make the changes.

```
% vi convert_b.a
    .
    .
-- Procedure Specification: (line 30)
    procedure GET_NUMBER(DATE_STRING: string;
                 RET_VAL, PLACE_IN_STRING: out INTEGER) is separate;
-- Two calls to the procedure: (lines 68 and 93)
    GET_NUMBER(DATE(NEXT .. DATE'LAST), D.DAY, NEXT);
    GET_NUMBER(DATE(NEXT .. DATE'LAST), D.YEAR, NEXT);
```

You want to create a new executable in this directory using these two files. However, you must make the information about convert.a, iio.a and test_convert.a visible to the Ada compiler in this directory. To do this, use the copy (a.cp) command. This utility copies unit and library information for specified files to a given target library. Use the following command to copy the necessary information to the development library.

```
% a.cp convert.a -L /usr1/tutorial
/usr1/tutorial/development
% a.cp iio.a -L /usr1/tutorial
/usr1/tutorial/development
% a.cp test_convert.a -L /usr1/tutorial/tests
/usr1/tutorial/development
```

The -L *library_name* option indicates that the files are to be copied from that library. After doing this, display the ada.lib file in the development library. You see the links to these files in the other libraries.

```
% more ada.lib

!ada library
ADAPATH= /self/verdixlib /self/standard
convert:#YNLPS 2C59BDF4:convert01:/usr1/tutorial/convert.a:
iio:XBILPS 2C59C062:iio01:/usr1/tutorial/iio.a:
test_convert:#MNLSS
2C5D4454:test_convert01:/usr1/tutorial/tests/test_convert.a:
test_convert:#MNLSB
2C5D4454:test_convert01:/usr1/tutorial/tests/test_convert.a:
```

Invoke a.make to create a new executable using the two modified files (convert_b.a, convert_s.a).

```
% a.make -v test_convert -f *.a

finding dependents of: convert_b.a
finding dependents of: convert_s.a
compiling convert_b.a
     body of convert
compiling convert_s.a
     body of convert.get_number
/usr1/tutorial/development/convert_s.a, line 17, char 7:error:
    RM 8.3: identifier undefined
```

You are back in the position of finding the error and have several ways of doing this. As with the ada command, a.make provides a means to intersperse error messages using the -el or -ev options (a.make -ev test_convert). The output looks the same as with the ada command. Do not use this approach now.

The other approach is to save the output of `a.make` in a log file and then supply the log file as input to `a.error`. `a.error` sorts the errors into their respective files and produces listings or editor files as requested. Here we use this approach. The >& redirects both the output and error messages to the named file. Note that >& is used with `csh`. For `ksh`, use `>file 2>&1`.

```
% a.make test_convert >& log
% a.error -v log

with IIO;
separate(CONVERT)
procedure GET_NUMBER(DATE_STRING : STRING;
            RET_VAL, PLACE_IN_STRING : out INTEGER) is
--
-- This procedure calls IIO.GET to read an integer value from the
-- beginning of DATE_STRING. It returns in RET_VAL, the integer
-- value that corresponds to the input sequence.  It returns in
-- PLACE_IN_STRING the place value in the string of the last
-- character read.  If an error is detected, the place value is
-- set to 0.
--
begin
    IIO.GET(DATE_STRING, RET_VAL, PLACE_IN_STRING);  --RM 14.3.7(13)

exception
    when DATA_ERROR =>
---------^A                                                   ###
--### A:error: RM 8.3: identifier undefined
    PLACE_IN_STRING := 0;

end GET_NUMBER;
```

You see immediately that you forgot to include the `package IO_EXCEPTIONS` in your subprogram so that `DATA_ERROR` is not recognized. Correct the program by inserting the missing 'with

IO_EXCEPTIONS;' line at the start of the program and 'IO_EXCEPTIONS.' before DATA_ERROR (now line 18). Remove the error lines (lines containing ###). Rebuild and execute the program.

```
% a.make test_convert
% a.out

"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" --  ERROR
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
"May 2, " --  ERROR
"June 19, 1885"  --ERROR
```

You have correctly implemented procedure GET_NUMBER as the correct day and year are appearing for the first two entries. However, you still have a problem with the third entry ("November 26, 1980") as that entry is in error. For the sake of this tutorial, you now take a week off to go skiing or wind surfing (we can dream, can't we?).

7.1.11 Finding a Currently Active Unit (a.which)

You return from your trip but cannot remember which library version of `convert_b.a` you were using when the error with the third entry appeared. You want to double check and find which library version of convert was used and what file it is in. This is very important when more than one programmer is working on a project (as is often the case). To do this, use `a.which`.

You do remember you were in your temporary working directory (`development`) so you name that directory in your command line.

To locate the current body for convert, use `a.which` with the `-b` (find body) and the `-L` (name library) options.

```
% a.which -b -L /usr1/tutorial/development convert
/usr1/tutorial/development/convert_b.a
```

Figure 7-9 Find the Currently Active Unit (a.which)

If you are not in your development library, enter it now to continue work on the program.

```
% cd /usr1/tutorial/development
```

References

`a.which`, *SPARCompiler Ada Reference Guide*

7.1.12 Listing a File (a.list)

Next, display `convert_b.a` using `a.list` to refamiliarize yourself with it. The `a.list` command provides a source code listing with line numbers. Pipe it through `more` to examine each screen.

```
% a.list convert_b.a | more
```

As you are reading through the listing, you notice a mistake on line 27 in the assignment of the array, `MONTH_TAB`. The entry for November is in all lowercase instead of having the first letter capitalized. This is the entry in your test program that is generating a wrong result.

```
11 => new STRING'("november"),
```

Edit the file and change the array assignment to “November”.

```
% vi convert_b.a
```

Call `a.make` with the `-f` option to bring the library up to date again.

```
% a.make -v test_convert -f convert_b.a

a.make: reevaluating dependencies of files that have been modified:
finding dependents of: convert_b.a
compiling convert_b.a
        body of convert
compiling convert_s.a
        body of convert.get_number
/bin/a.ld test_convert
```

Note that if the problem is not so easily found, use the SC Ada debugger, `a.db`, to locate it. A tutorial on the use of the debugger follows later in this chapter.

This sequence produced an executable file with the default SC Ada name, `a.out`. Execute this file.

```
% a.out

"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
"May 2, " --  ERROR
"June 19, 1885" -- ERROR
```

It worked! It is now time to move the new and correct `convert_s.a` and `convert_b.a` files to the permanent project library (tutorial) and clean up the temporary working library (`development`).

References

`a.list`, *SPARCompiler Ada Reference Guide*

7.1.13 Cleaning Up (a.rm, a.cleanlib, a.rmlib)

The correct versions of the `convert_s.a` and `convert_b.a` files must now be moved to the permanent project library. Copy the files to the project library (`/usr1/tutorial`) and create a new executable file (`test_convert`) in your tests directory with `a.make`. Use the **-A** (add library) option to enable the compilation of the files in another directory (`tutorial`).

```
% cp convert_b.a convert_s.a ..
% cd /usr1/tutorial/tests
% a.make -v -A /usr1/tutorial -o test_convert test_convert

a.make: reevaluating dependencies of files that have been modified:
finding dependents of: /usr1/tutorial/convert_b.a
finding dependents of: /usr1/tutorial/convert_s.a
compiling /usr1/tutorial/convert_b.a in /usr1/tutorial
        body of convert
compiling /usr1/tutorial/convert_s.a in /usr1/tutorial
        body of convert.get_number
/bin/a.ld -o test_convert test_convert
```

Figure 7-10 Clean Up (a.rm, a.cleanlib, a.rmlib)

Execute `test_convert` making sure you obtain the correct result.

```
% test_convert
```

When this is done, you are ready to clean up your temporary `development` directory. Do not delete the directory because you may need it again in the development process. You want to remove all the information relating to the `convert_s.a` and `convert_b.a` files from this Ada program library. Return to that directory. First, delete the versions of the `convert_s.a`, `convert_b.a`, `log` and `a.out` files present in it.

```
% cd /usr1/tutorial/development
% rm convert_s.a convert_b.a log a.out
```

The Ada library information must now be reinitialized with one of two commands: `a.rm` or `a.cleanlib`. If `a.rm` is used, a unit name or source filename must be entered following the command. All information associated with the entered name(s) is removed from the Ada program library. `a.cleanlib` removes all separate compilation information from the library directory. Use one of these commands to reinitialize the library.

```
% a.rm convert_s.a convert_b.a
```

or

```
% a.cleanlib
```

To maintain the directory but remove all Ada library information from it, use `a.rmlib`. This command deletes the `ada.lib`, `GVAS_table`, and `gnrx.lib` files and the `.imports`, `.lines`, `.nets`, and `.objects` directories.

```
% a.rmlib
```

The library is empty. You decide you do not want to keep this directory after all. Return to your main project library, then remove the directory.

```
% cd /usr1/tutorial (or % cd ..)
% rmdir development
```

If you had wanted to remove the entire directory at the start, you can use `rm(1) -rf` command at any time after the fixed files are copied and tested.

```
% rm -rf /usr1/tutorial/development
```

References

`a.cleanlib`, *SPARCompiler Ada Reference Guide*

`a.rm`, *SPARCompiler Ada Reference Guide*

`a.rmlib`**,** *SPARCompiler Ada Reference Guide*

7.1.14 Library Disk Usage Summary (a.du)

To see what units are active in your main project library (`/usr1/tutorial`), return to it and use the `a.du` command. This utility displays the sizes (in bytes) of the various files that represent Ada units: the separate compilation information (`.nets`), the object file (`.objects`) and the line number debugging file (`.lines`).

```
% cd /usr1/tutorial
% a.du -v

  .nets  .objects  .lines
  11076     119       0 Pack Spec  of convert in convert.a
  16320    2921     668 Pack Body  of convert in convert_b.a
   8192       0       0 Subp Spec  of convert.get_number in convert_b.a
   9508     766     172 Subp Body  of convert.get_number in convert_s.a
  18788     908      84 Pack Spec (Instan.) of iio
```

Figure 7-11 Library Disk Usage Summary (a.du)

To list the imported units, use the imports (`-i`) option

```
% a.du -i

  37160  STANDARD
  50396  a_strings
  52684  file_support
  20384  integer_io$235
   9624  io_exceptions
  32044  os_files
  11952  safe_defs
  30708  system
  94992  text_io
   9537  unchecked_conversion
  39764  unsigned_types
  23988  v_i_sema
  92080  v_i_types
  ...
```

References

`a.du`, *SPARCompiler Ada Reference Guide*

7.1.15 Creating a Tags File (a.tags)

Before proceeding, create a tags file. A tags file helps in locating units even when they are distributed among a number of files and libraries. To do this, use `a.tags`.

```
% a.tags *.a
```

This utility creates a file that is used by editors such as `vi(1)` and `ex(1)` to locate a unit. Now edit a unit by name, instead of having to remember the filename.

```
% vi -t GET_NUMBER
```

The editor brings up `convert_s.a` by using the information in the tags file. The tag must be in the same case as its occurrence in the source file.

With `a.tags`, access entities declared within units, specifically to subprograms, tasks, packages and optionally to types. Further, when you are in the editor, you can position the cursor at the first character of a name and then type `<CONTROL-]>`. The editor finds the declaration of that name and switches to that declaration. Position the cursor at the start of IIO and enter `<CONTROL-]>.` The file, `iio.a`, is displayed.

Exit the file **iio.a.**

Be aware that the tags file is not kept up to date automatically as your programs change and must be remade periodically.

References

`a.tags`, *SPARCompiler Ada Reference Guide*

7.1.16 Using the SC Ada Disassembler (a.das)

It is often useful to see what machine level instructions are being executed for each Ada source line. SC Ada has an object module disassembler that is invoked using `a.das`. It interleaves your Ada source lines with machine instructions.

By default, the disassembler operates on the body of the unit named in the command line. In this case, you want to examine the body of `convert`, so enter the following command. Unless you can read the screen very quickly, redirect the output to a file or pipe it through `more`. Illustrated below is an example of the output of the disassembler. The exact output is CPU-specific and may be different for your system

```
% a.das convert | more

Unit:            convert
Library:         /usr1/tutorial
Object file:     /usr1/tutorial/.objects/convert_b02
Source file:     /usr1/tutorial/convert_b.a
Text Section:

...skipping
  35            for I in MONTH_TAB' range loop
      0037c:  addi       s0,$0,01          s0 <- 1
      00380:  lui        t5,00             t5 <- 000000
      00384:  addiu      t5,t5,08f0        t5 <- t5 + 2288
      00388:  lw         s1,-0c(fp)        s1 <- -12(fp)
      0038c:  nop
  36                if LC_MONTH = MONTH_TAB(I).ALL then
      00390:  lw         t8,-01c(fp)       t8 <- -28(fp)
      00394:  nop
      00398:  sll        t5,s0,2           t5 <- s0 << 2
      003ec:  addi       v0,$0,01          v0 <- 1
      003f0:  addi       t4,$0,07f         t4 <- 127
      003f4:  slt        t3,t4,v0          t3 <- t4 < v0
      003f8:  beq        t3,$0,8           -> 0404
      003fc: #nop
      00400:  break      08
      00404:  beq        v0,$0,52          -> 043c
      00408: #nop
  37                    return I;
      0040c:  addu       v0,s0,$0          v0 <- s0
--More--
```

Figure 7-12 Use the Disassembler (a.das)

Enter `q` to exit `-More-`.

References

`a.das`, *SPARCompiler Ada Reference Guide*

7.1.17 Useful Information (a.info, a.ls, a.vadsrc)

Several additional utilities are provided to get valuable information about programs while you develop and integrate them. These are `a.info`, `a.ls` and `a.vadsrc`.

The `a.info` command lists and/or changes the SC Ada library directives. If you invoke it at any time after your development effort starts, you see the following.

```
% a.info -i

 1.  List  the library search list (ADAPATH)?
 2.  List  visible directives along the library search list?
 3.  List  ALL directives along the library search list?
 4.  List  visible directives in the LOCAL library?
 5.  List  ALL directives in the LOCAL library?
 6.  List  INVARIANT directives in the library search list?
 7.  Add      directive to the local library?
 8.  Delete   directive from the local library?
 9.  Replace  directive value in the local library?
 10. Help  on a directive ?
 11. Quit?
Selection?  (1-11, or q for quit) -> 2
```

Figure 7-13 List and Change Library Directives (a.info)

To obtain more detailed information about your Ada libraries, enter `2` to display the directives visible on the library search path for `/usr1/tutorial`. Note that the displayed information is system dependent.

```
ARCHITECTURE:      VERSION7
ENDIAN:            BIG
HOST:              host_name
LIBRARY:           /usr/ada_2.1/self/standard/.objects/library.a
MAX_INLINE_NESTING: 5
TARGET:            SELF_TARGET
TASKING:           /usr/ada_2.1/self/standard/.objects/tasking.a
SCAda:              /usr/ada_2.1/self
VERSION:           3.0
```

Enter **11** to exit.

Another tool that provides useful information is `a.ls`. This utility provides a listing of all the compiled units in a current or specified SC Ada library. A number of options are provided to give you a variety of information. If you select the `-l` (long) and `-L` (library) options and name your tutorial library, you can find out the net file date, source file date, unit type and the unit name of the units in that library.

```
% a.ls -l -L /usr1/tutorial
Jul 30  8:44  Jul 30  8:43  Pack spec             convert
Jul 30  9:29  Jul 30  9:28  Pack Body             convert
Jul 30  9:29  Jul 30  9:28  Task Body           convert.get_number
Jul 30  9:29  Jul 30  9:28  Subp spec           convert.get_number
Jul 30  8:46  Jul 30  8:42  Pack spec (Instan)    iio
```

Figure 7-14 List Compiled Units (a.ls)

An additional tool, a.vadsrc, can display the SC Ada targets and versions available on the system. It can create or display a library configuration file. It is useful when multiple SC Ada versions are present on the same system and a default version or target processor is desired.

```
% a.vadsrc

2 versions of SC Ada are available on this machine:
    Target Name     Version      Ada Location
 1   SELF_TARGET    3.0          /usr2/ada_2.1/self
                                 host_name, host_os, version_no
 2   SELF_TARGET    3.0          /usr2/ada_2.1/self_thr
                                 host_name, host_os, version_no
```

Figure 7-15 Display Versions (a.vadsrc)

References

`a.info`, *SPARCompiler Ada Reference Guide*

`a.ls`, *SPARCompiler Ada Reference Guide*

`a.vadsrc`, *SPARCompiler Ada Reference Guide*

7.1.18 When All Else Fails (a.report)

If you are receiving product support from us, and you want to report a problem, make a suggestion or request an enhancement, use the `a.report` command. This command automatically captures identification information and prompts you for the report information. When the report is completed, the file is electronically mailed to persons determined by the system administrator when SC Ada is installed.

References

`a.report`, *SPARCompiler Ada Reference Guide*

7.1.19 Making Your Life a Little Easier (a.view)

The `a.view` command defines a number of aliases that simplify and enhance the use of the basic SC Ada commands for users of the C shell. These aliases are listed in the SC Ada Command Reference. If you use these aliases, history and timing information are entered into the `ada.history` file. With them, enter a source filename or main unit name only once and then they are reused automatically.

The SC Ada *Command Reference* discusses in detail several ways to implement and use these aliases.

References

`a.view`, *SPARCompiler Ada Reference Guide*

7.2 Debugger Tutorial

The SC Ada debugger is a tool that debugs Ada programs at the source and machine level. Like some text editors, it has two interfaces: line-mode and screen-mode. The screen-mode debugger provides a more convenient interface for most people but both modes are covered in this tutorial. Every function available in line mode is also available in screen mode. More detailed information about the operation of the debugger and the debugger commands is in the SC Ada *Reference Guide.*

This debugger tutorial presumes that you have a working knowledge of Ada programming and are able to make the minor modifications required to complete this tutorial. It presumes that you are familiar with the operation of the SC Ada compiler. If you are not, please complete the compiler tutorial described earlier in this chapter.

7.2.1 Debugger Configuration Parameters (.dbrc)

The debugger operates under the control of a set of configuration parameters. The SC Ada *Reference Guide* lists them under the `set` command. Modify them during the operation of the debugger using the `set` command. Initialize them in a `.dbrc` file located in your home directory or SC Ada library.

One of the configuration parameters is a tabs setting which defaults to 8. You may want to change it to 4 to make the output easier to read. So that you do not have to do this every time you invoke the debugger, you insert the new tabs setting (set tabs 4) in a `.dbrc` file in your home directory or SC Ada library.

Another useful capability of the debugger is the ability to save a transcript of your debugging session in a log file. This can be set up in the `.dbrc` file (our log file is named `deb.log`).

```
% cd
% vi .dbrc

set tabs 4
set log deb.log
```

Figure 7-16 Edit .dbrc File

References

`.dbrc` File, *SPARCompiler Ada Reference Guide*

`set` command, *SPARCompiler Ada Reference Guide*

7.2.2 Debugger Tutorial Project

The program in this tutorial is a utility to translate dates of the form “February 3, 1954” into the “2/3/54” format. The project includes a test program with several different test dates. If a date is entered incorrectly, an error message is printed. Only dates in the 20th century are translated. For example,

```
May 2, 1945 -- 5/2/45
September 16, 1921 -- 9/16/21
July 4 -- ERROR
November 26, 1771 -- ERROR
```

All the source code to begin this tutorial is in the *`SCAda_location/`*`examples` directory. Also, complete source listings are included within this tutorial (see page 31).

First, set up your Ada project library using `a.mklib`. For the purposes of this tutorial, the project library is in `/usr1/tutorial`. If you have used this directory for the compiler tutorial, remove all its files and subdirectories.

```
% cd /usr1/tutorial
% a.rmlib
% rm *.*
% a.mklib -i
```

Figure 7-17 Set Up the Debugger Tutorial

Otherwise, create the directory.

```
% mkdir /usr1/tutorial
% cd /usr1/tutorial
% a.mklib -i
```

For cross-development systems, if you use any cross I/O functions, you must add the cross_io library to your ADAPATH. Cross I/O is the complete implementation of Ada I/O capabilities from the target system up to the host. The library that must be added resides in `/usr2/target/cross_io`. Add this library to your search list using `a.path`:

```
% a.path -i /usr2/target/cross_io
```

For cross-development systems, include the user library in the ADAPATH. The LINK_BLOCK directive required for compilation is in this library. Add this library to your search list using `a.path`:

```
% a.path -i /usr2/target/board_conf
```

The files in ***SCAda_location*/examples** to begin this tutorial are:

```
convert.a
convert_b.deb
convert_s.a
iio.a
test_convert.a
```

Be sure you are in your project directory and copy the required files from the *SCAda_location*/`examples` directory or enter them manually. Do not forget to change the debugger suffix to `.a`. You must copy `convert_b.deb` after the other files. This ensures that you have the proper versions of the files to begin the tutorial. Note that you must also add write permission to these files as you will be editing some of them as part of the tutorial.

```
% cp /examples/convert.a .
% cp /examples/convert_s.a .
% cp /examples/test_convert.a .
% cp /examples/iio.a .
% cp /examples/convert_b.deb convert_b.a
% chmod +w *.a
```

If you chose to enter these files manually, the source code is listed on page 31.

Like any optimistic programmer, compile, link and execute the files to see if the test program works the first time. Give the executable file the name of `test_convert`.

```
% a.make test_convert -o test_convert -f *.a
% test_convert
```

Figure 7-18 Be Optimistic: Compile, Link and Execute

It does not work quite correctly as the following output shows.

```
"January 1, 1900" --  ERROR
"December 31, 1999" --  ERROR
"November 26, 1983" --  ERROR
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
"May 2, " --  ERROR
"June 19, 1885" --  ERROR
```

You decide to use the SC Ada debugger, `a.db`, to find out why good dates are showing up as errors. You have two options, using the debugger in line- or screen-oriented mode. The following sections illustrate the use of the debugger in these two modes. It is possible to toggle between modes by entering `vi` at the debugger prompt to go from line mode to screen mode or `Q` to go from screen mode to line mode.

Note – The messages displayed on your system may be slightly different than those shown in this tutorial. The debugger is constantly being improved and messages may evolve with the system.

7.2.3 Line Mode versus Screen Mode

The screen-mode interface provides a more convenient interface than the traditional line-mode for most program debugging at the source level. All commands available for the line-mode interface are also available in screen mode by prefacing them with a colon. Some of the debugger commands are directly implemented in screen mode and do not require the preceding colon. They are called immediate commands.

An additional set of commands is provided to manipulate the screen interface and provide a number of display and search capabilities. The screen interface supports a machine-level instruction submode. In this mode (access it by entering `I`), the source window contains disassembled machine instructions interspersed with source code and the step (`s`), advance (`a`) and break (`b`) commands are interpreted at the instruction level (`si`, `ai` and `bi`).

Use of these additional commands, instruction sub-mode, the immediate and the regular debugger commands, is shown in the screen-mode debugger tutorial found later in this chapter.

At any time in a debugging session, switch between line and screen mode. Use of line or screen mode is at your discretion. Generally, screen-mode operation provides an easier interface to the debugger. It cannot be used, however, if the input (`-i`) option is used when invoking the debugger. Other times, line mode is more appropriate to your needs. For this reason, examples demonstrating the use of the debugger in both modes are provided in the following sections. A detailed discussion of both line and screen mode is in the SC Ada *Reference Guide*.

References

Line Mode and Screen Mode Command Syntax, *SPARCompiler Ada Reference Guide*

Screen Mode, *SPARCompiler Ada Reference Guide*

7.2.4 Date Translation Source Code

The following is a listing of the code used to implement and test the date conversion utility. The test program is held in `test_convert.a`. The conversion utility is held in `convert.a`, `convert_b.a`, `convert_s.a` and `iio.a`. All code is formatted using `a.pr`

```
*********************  convert.a ***********************
package CONVERT is
--
-- This package is responsible for the conversion of dates from
-- the English longhand notation to a numerical notation.  The
-- date must be of the form "month day, year". It is returned in
-- the form month/day/year.  Only dates in the 20th century are
-- to be translated (January 1, 1900 to December 31,1999).  If
-- the entered date is not syntactically correct, an error is
-- returned.
--
type DATE_REP is
record
MONTH:INTEGER;
DAY:INTEGER;
YEAR:INTEGER;
end record;

function GET_DATE_REP(DATE: STRING) return DATE_REP;

end CONVERT;
********************  convert_b.a  ************************
package body CONVERT is
--
-- This package body is responsible for doing the actual
-- translation of the date from English to numeric format.
-- MONTH_NO returns the proper month number associated with the
-- entered month by comparing the entry with the MONTH_TAB array.
-- GET_DATE_REP processes the string entered, finding each number
```

(Continued)

```
-- and the required comma.  GET_NUMBER is called to process the
-- numbers in the date string. The values found are held in D. If
-- an error is found at any time, D.all is returned. The value of
-- the D.MONTH field will be checked by the test program.If it is
-- zero, this indicates an error has occurred.
--
type ASTRING    is access STRING;
type MONTH_LIST is array(1 .. 12) of ASTRING;
MONTH_TAB : MONTH_LIST :=
(1 => new STRING'("january"),
 2 => new STRING'("february"),
 3 => new STRING'("march"),

 4 => new STRING'("april"),
 5 => new STRING'("may"),
 6 => new STRING'("june"),
 7 => new STRING'("july"),
 8 => new STRING'("august"),
 9 => new STRING'("september"),
10 => new STRING'("october"),
11 => new STRING'("november"),
12 => new STRING'("december"));

procedure GET_NUMBER(DATE_STRING: string; RET_VAL,
           PLACE_IN_STRING:  out INTEGER) is separate;

function MONTH_NO(MONTH : STRING) return INTEGER is

begin
for I in MONTH_TAB'range loop
   if MONTH = MONTH_TAB(I).all then
       return I;
   end if;
end loop;
return 0;
```

(Continued)

```
end MONTH_NO;

function GET_DATE_REP(DATE : STRING) return DATE_REP is
type ADATE is access DATE_REP;
D         : ADATE    := new DATE_REP’(0, 0, 0);
THE_MONTH : INTEGER;

NEXT      : INTEGER := DATE’FIRST;


begin
while (NEXT <= DATE’LAST) and then (DATE(NEXT) /= ’ ’) loop
   NEXT := NEXT + 1;
end loop;

if NEXT >= DATE’LAST then
   return D.all;    -- error exit;
end if;

        -- we postpone assigning the month into d.month since 0
        -- indicates an error and it is 0 now.

THE_MONTH := MONTH_NO(DATE(DATE’FIRST .. NEXT - 1));
if THE_MONTH = 0 then
   return D.all;    -- error exit;
end if;

        -- get the day
GET_NUMBER(DATE(NEXT .. DATE’LAST), D.DAY, NEXT);
if NEXT = 0 then
   return D.all;    -- error exit;
end if;


        -- find the comma
```

(Continued)

```
NEXT := NEXT + 1;
while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
   NEXT := NEXT + 1;
end loop;

        -- make sure there is more string left to hold the year
NEXT := NEXT + 1;
if NEXT >= DATE'LAST then
   return D.all;    -- error exit;
end if;

        -- get the year and verify it is in the 1900s
GET_NUMBER(DATE(NEXT .. DATE'LAST), D.YEAR, NEXT);
if (NEXT = 0) or (D.YEAR < 1900) or (D.YEAR > 1999) then
   return D.all;    -- error exit;
end if;

D.MONTH := THE_MONTH;
return D.all;
end GET_DATE_REP;
end CONVERT;
```

```
**************************  iio.a  ********************************
with TEXT_IO;
package IIO is new TEXT_IO.INTEGER_IO(integer);

--This package enables the printing and display of integers.
--
```

```
 (Continued)
********************  convert_s.a  ************************
with IO_EXCEPTIONS;
with IIO;
separate(CONVERT)
procedure GET_NUMBER(DATE_STRING : STRING; RET_VAL,
          PLACE_IN_STRING : out INTEGER) is
--
-- This procedure calls IIO.GET to read an integer value from the
-- beginning of DATE_STRING.  It returns in RET_VAL, the integer
-- value that corresponds to the input sequence.  It returns in
-- PLACE_IN_STRING the place value in the string of the last
-- character read.  If an error is detected, the place value is
-- set to 0.
--
begin
IIO.GET(DATE_STRING, RET_VAL, PLACE_IN_STRING); --RM 14.3.7(13)

exception
when IO_EXCEPTIONS.DATA_ERROR =>
PLACE_IN_STRING := 0;

end GET_NUMBER;
end GET_NUMBER;



***********************  test_convert.a  ****************************
with CONVERT;
with TEXT_IO;
use  TEXT_IO;
with IIO;
procedure TEST_CONVERT is
--
-- This test file contains a list of correct and incorrect dates
-- to be tested.  Each date is analyzed by TEST_SINGLE_DATE. It
```

(Continued)

```
-- calls the GET_DATE_REP procedure defined in CONVERT.
-- The entered date is displayed.  The value of DAY.MONTH is
-- then checked.  If it is 0, an error has occurred in the
-- processing of the entered date string and an error message is
-- displayed.  Otherwise, the numeric equivalent of the date is
-- displayed.
--
procedure TEST_SINGLE_DATE(DATE : STRING) is
DAY : CONVERT.DATE_REP;
TURN_CENTURY: constant STRING(1..2) := ”00”;
begin
DAY := CONVERT.GET_DATE_REP(DATE);
PUT(’”’);
PUT(DATE);
PUT(””” -- ”);

if DAY.MONTH = 0 then
  PUT_LINE(” ERROR”);
  return;
end if;

IIO.PUT(DAY.MONTH, 2);
PUT(’/’);
IIO.PUT(DAY.DAY, 2);
PUT(’/’);
if (DAY.YEAR = 1900) then
  PUT(TURN_CENTURY);
else
  IIO.PUT(DAY.YEAR - 1900, 2);
end if;
NEW_LINE;
end TEST_SINGLE_DATE;
begin
        -- first test some good dates

```

(Continued)

```
TEST_SINGLE_DATE(”January 1, 1900”);
TEST_SINGLE_DATE(”December 31, 1999”);
TEST_SINGLE_DATE(”November 26, 1983”);
NEW_LINE;
        -- now test some bad dates

TEST_SINGLE_DATE(”Aug. 1, 1981”);   -- bad month
TEST_SINGLE_DATE(”March 16 1986”);  -- no comma
TEST_SINGLE_DATE(”February ”);     -- no day, no comma, no year
TEST_SINGLE_DATE(”July 3”);         -- no comma, no year
TEST_SINGLE_DATE(”May 2,  ”);       -- comma, no year
TEST_SINGLE_DATE(”June 19, 1885”);  -- invalid year
end TEST_CONVERT;
```

Figure 7-19 Debugger Tutorial Source Code

7.2.5 Getting On-Line Help from the Debugger

Several forms of on-line help are available to users of the SC Ada debugger. The following command,

```
 help [subject]   -- line mode
:help [subject]   -- screen mode
```

displays information about the entered subject. If the subject is omitted, a list of debugger commands for which help is available is displayed.

An additional help tool is available for use in screen mode by entering

```
H
```

This prints a single line of information at the bottom of the screen that provides a brief summary of the screen mode commands. Several lines of help are available. Press `H` again for the next line of text help. Press any other key to exit the help utility.

References

`help`, *SPARCompiler Ada Reference Guide*

7.3 Line-oriented Debugging Session

To invoke the debugger in line mode, enter the following:

```
% a.db test_convert
```

The debugger displays the following information.

```
Debugging: /usr1/tutorial/test_convert
SC Ada library: /usr1/tutorial
library search list:
                /usr1/tutorial
                /self/verdixlib
                /self/standard
```

Enter `set all` to examine the debugger settings.

```
>set all
alert_freq: 100     async: off            case: off
c_types: local      event: 0              except_stack: off
input: pty          lines:  10            log: deb.log
number: on          obase: 10             output: pty
page: on            persist: off          prompt: ">"
safe: off           signal  b: hup..pipe, term..tstp, chld..32
signal  g: cont     signal gs: alrm       source: /usr1/tutorial
                    /verdixlib /standard
step_alert: 1000    tabs:  4              trace: ,v off,s off
SC Ada directory: /usr1/tutorial          verbose: off
xrate:    5.000     with on: all
run: test_convert
```

The tabs setting is 4 and a log file is produced (`deb.log`). These changes from the default settings are a result of your entries in the the `.dbrc` file (see *Debugger Configuration Parameters* (`.dbrc` on page 27). You decide you do not want to generate a log file. To change this entry, use the command history and line editing features available with the line-mode debugger to change it. The SC Ada *Reference Guide* discusses these commands in detail.

Press <ESC> to enter edit mode. In this mode, every time you enter k, the previous command is displayed. Entering j advances one command. When a command is displayed, it can be edited and executed. Pressing <RETURN> executes the command and exits edit mode and back to insert mode.

Enter <ESC> to enable edit mode. Enter k to display the last command (set all). The cursor is at the end of the line. Enter cb to change the last word. The cursor moves to the beginning of all and a $ appears at its end. Enter log off and press <RETURN>. The command should read set log off. Of course, you can enter the new command (set log off) directly, but you do not learn as much doing that.

Use the list command (l**)** to display the start of your program. Enter l.

```
> l
 5* procedure TEST_CONVERT is
 6  --
 7  -- This test file contains a list of correct and incorrect dates
 8  -- to be tested.  Each date is analyzed by TEST_SINGLE_DATE. It
 9  -- calls the GET_DATE_REP procedure defined in CONVERT.
10  -- The entered date is  displayed.  The value of DAY.MONTH is
11  -- then checked.  If it is 0, an error has occurred in the
12  -- processing of the entered date string and an error message is
13  -- displayed.  Otherwise, the numeric equivalent of the date is
14< -- displayed.
```

The ***** on line 5 indicates this is the current home position for this file. The **<** on line 14 indicates it is the current line. Another **l** starts at the current line. A w displays a window around the current line.

Start the program execution using one of four commands.

r (run)	Runs/reruns a program from its beginning. This command processes new invocation parameters and handles I/O redirection, if necessary.
g (go)	Executes the program from where it last stopped. This command cannot process new invocation parameters but uses the current parameters.
a (advance)	Advances one source line over subprogram calls. If the program has not yet started, this command advances one source line.

s (step) Steps one source line into a called subprogram. If the program has not yet started, this command advances one source line.

Since you want to step through the program and no invocation parameters are required, enter a to begin program execution. Execution jumps to the call to TEST_SINGLE_DATE on line 44. This is the next line for which the compiler has generated code.

```
> a
test_convert
stopped at ”/usr1/tutorial/test_convert.a”:44 in test_convert
 44      TEST_SINGLE_DATE(”January 1, 1900”);
```

To step into TEST_SINGLE_DATE, enter s.

```
> s
stopped at ”/usr1/tutorial/test_convert.a”:16 in test_single_date
 16      procedure TEST_SINGLE_DATE(DATE: STRING) is
```

”, ”repeat command”>Continue to enter s or simply press <RETURN> to step through this procedure and into the GET_DATE_REP function. Use <RETURN> to repeat the most recent a, ai, s, si, l, li, / or **?** command. Entering r, g, gw or gs causes the <RETURN> key to lose its memory.

```
> <RETURN> -- or s
stopped at ”/usr1/tutorial/test_convert.a”:20 in test_single_date
 20     DAY := GET_DATE_REP;
> <RETURN> -- or s
stopped at ”/usr1/tutorial/convert_b.a”:43 in get_date_rep
 43      function GET_DATE_REP(DATE_STRING) return DATE_REP is
```

Enter s or <RETURN> three more times to step through this function until you reach a loop statement (line 51).

```
> <RETURN> -- or s
stopped at "/usr1/tutorial/convert_b.a":45 in get_date_rep
 45       D : ADATE := new DATE_REP'(0,0,0);

> <RETURN> -- or s
stopped at "/usr1/tutorial/convert_b.a":48 in get_date_rep
 48        NEXT : INTEGER := DATE'FIRST;

> <RETURN> or s
stopped at "/usr1/tutorial/convert_b.a":51 in get_date_rep
 51      while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
```

You now want to get past this loop without manually executing it by stepping through all its iterations. Do this with the following sequence of commands:

1. Enter **l** to list the next 10 lines of the program.

```
> l
51*     while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
52          NEXT := NEXT + 1;
53       end loop;
54
55        if NEXT >= DATE'LAST then
56           return D.ALL;  -- error exit;
57        end if;
58
59        -- we postpone assigning the month into d.month since 0
60<      -- indicates an error and it is 0 now.
```

In the debugger listings, the asterisk (*****) is located at the next line to execute.

2. Enter b 55 to set a breakpoint at the first line following the while loop.

```
>b 55
```

3. Enter g to execute the program. The program executes the loop statement and stops at the breakpoint.

```
> g
[1]  stopped at "/usr1/tutorial/convert_b.a":55 in get_date_rep
      55   if NEXT >= DATE'LAST then
```

Enter l to examine the if statement. Note the * and = on line 55. The * indicates this is the next line to execute while the equal sign (=) indicates a breakpoint exists at that line. When you list this section of code, you see the if statement is used to detect an error. Enter s to continue stepping through the program. If an error is detected, you step to statement 56; if not, you step to statement 62, the next executable statement after the if.

```
> l
55*=          if NEXT >= DATE'LAST then
56                      return D.all;    -- error exit;
57            end if;
58
59       -- we postpone assigning the month into d.month since 0
60         -- indicates an error and it is 0 now.
61
62          THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
63            if THE_MONTH = 0 then
64<                     return D.all;    -- error exit;
>s
 stopped at "/usr1/tutorial/convert_b.a":62 in get_date_rep
  62   THE_MONTH := MONTH_NO(DATE(DATE'FIRST..NEXT-1));
```

It worked - no errors up to this point.

Enter s to step into the subprogram MONTH_NO.

```
> s
 stopped at "/usr1/tutorial/convert_b.a":33 in month_no
  33function MONTH_NO(MONTH : STRING) return INTEGER is
```

You decide you do not want to step into this function, but rather advance past it and check the value of THE_MONTH to see if any errors occurred in the function. If the value of THE_MONTH is 0, an error occurred.

Luckily, the SC Ada debugger has an easy way to exit a subprogram. The command (bd) inserts a breakpoint at the first machine instruction following the return of the current subprogram. Enter bd to set the breakpoint. Enter g to execute the program to the breakpoint. You return to the end of the function call (MONTH_NO) on line 62. The hyphen on the listing indicates that the breakpoint is not on the first instruction executed by this statement. This breakpoint is automatically deleted after it is hit.

```
> bd
> g
[2]-   stopped at "/usr1/tutorial/convert_b.a":62 in get_date_rep
  62    THE_MONTH := MONTH_NO(DATE(DATE'FIRST..NEXT-1));
```

Enter a to advance to the next line. Enter w to display the source code surrounding your current location.

```
> a
stopped at "/usr1/tutorial/convert_b.a":63 in get_date_rep
63       if THE_MONTH = 0 then
>w
58
59        -- we postpone assigning the month into d.month since
60          -- 0 indicates an error and it is 0 now.
61
62         THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
63*<         if THE_MONTH = 0 then
64              return D.all;    -- error exit;
65          end if;
66
67               -- get the day
```

Enter p the_month to check the value of THE_MONTH.

```
> p the_month
0
```

THE_MONTH is zero, so something is wrong in MONTH_NO. Check the current call stack using the call stack (cs) command. This displays the procedure parameters so you can be sure that the input to the procedure is as expected. Enter cs.

```
> cs
#   line           procedure             params
1   63             get_date_rep  (date = ”January 1, 1900”)
2   20             test_single_date (date = ”January 1, 1900”)
3   44             test_convert()
```

The parameters look correct. The easiest thing to do is to set a breakpoint at MONTH_NO and rerun the program. Enter b month_no to set the breakpoint at that procedure. Enter r to rerun the program.

```
> b month_no
> r
 [1] stopped at ”/usr1/tutorial/convert_b.a”:55 in get_date_rep
   55        if NEXT >= DATE’LAST then
```

Execution stops at the earlier breakpoint you set (line 55). You do not need it anymore so delete it. List the currently active breakpoints by entering lb and delete the breakpoint using the delete command (d) followed by the breakpoint number (enter d 1).

```
> lb
[1]  ”/usr1/tutorial/convert_b.a”:55 in get_date_rep
[3]  ”/usr1/tutorial/convert_b.a”:33 in month_no
>d 1
```

If you want to delete the breakpoint you just hit without listing all the breakpoints, use the number given in brackets ([1], [2], etc.) at the start of the breakpoint announcement with the delete command (d 1, d 2, d 3, etc.).

After doing this, enter g to continue execution and you arrive at the MONTH_NO function. Enter cs 1 to take a quick look at its parameter. This command displays only the top entry on the call stack.

```
> g
 [3] stopped at ″/usr1/tutorial/convert_b.a″:33 in month_no
  33                function MONTH_NO(MONTH: STRING) return INTEGER is
> cs 1
#   line         procedure       params
1   33             month_no         (month = ″January″)
```

Enter s twice to step into the loop that compares the parameter to the table of months.

```
> s
stopped at ″/usr1/tutorial/convert_b.a″:35 in month_no
35                   for I in MONTH_TAB′ range loop

> <RETURN> -- or s
stopped at ″/usr1/tutorial/convert_b.a″:36 in month_no
36                   if MONTH = MONTH_TAB(I).ALL then
```

Use the print (p) command to look at the first table entry. To get the value of the object and not the access type, you must use .all. Enter p month_tab(i).all.

```
   > p month_tab(i).all
   ″january″
```

This should match your passed parameter (January). You see that a ”J” is used on input, but the table has a ”j”. You must convert the input string to all lower case. Enter exit or quit to exit the debugger.

```
>quit
```

Modify convert_b.a. Enter the following changes yourself or copy the new convert_b file from the *SCAda_location*/examples directory. It is there as convert_b.deb1.

```
% cp /examples/convert_b.deb1 convert_b.a
% chmod +w convert_b.a
```

or

```
% vi convert_b.a
.
.
package body CONVERT is
.
.
.
function MONTH_NO(MONTH: STRING) return INTEGER is
LC_MONTH: STRING(MONTH'RANGE);
function TO_LOWER (CH: CHARACTER) return CHARACTER is
    STANDARD_DIFFERENTIAL: constant INTEGER :=
        CHARACTER'POS ('A') - CHARACTER'POS ('a');

begin --to_lower
    if CH in 'A' .. 'Z' then
        return CHARACTER'VAL (
        (CHARACTER'POS (CH) - STANDARD_DIFFERENTIAL)
                        );
    else
        return CH;
    end if;
end TO_LOWER;

function TO_LOWER (STR: STRING) return STRING is
    RETURN_STRING: STRING (STR'range) := STR;

begin --to_lower
    for INDEX in STR'range loop
        RETURN_STRING (INDEX) := TO_LOWER (STR (INDEX));
    end loop;
    return RETURN_STRING;
end TO_LOWER;
begin --month_no
LC_MONTH := TO_LOWER(MONTH);
for I in MONTH_TAB' range loop
    if LC_MONTH = MONTH_TAB(I).ALL then
        return I;
    end if;
end loop;
return 0;
```

```
 (Continued)
   end MONTH_NO;
   .
   .
   .
   end;
```

Recompile and link `convert_b.a`. Attempt to execute `test_convert` again.

```
% a.make -o test_convert test_convert
% test_convert
```

This time you get the following output.

```
    "January 1, 1900" --  1/ 1/00
    "December 31, 1999" -- 12/31/99
    "November 26, 1983" -- 11/26/83
    "Aug. 1, 1981" --  ERROR
    ** MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error"
RAISED
```

Unhandled exceptions are easy to figure out with the debugger since the debugger executes the program until the exception is raised. Then it stops and shows where the problem is. A diagnostic display describes what is wrong.

Reinvoke the debugger and enter `r` to run it. You find a problem. Note that your exact message is system dependent.

```
% a.db test_convert

> r
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
process received signal Trace/BPT trap [5]
 stopped 16 instructions after "/usr1/tutorial/convert_b.a":84 in
            get_date_rep
 84         while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
>
```

The debugger generates the "`process received signal Trace/BPT trap [5]`" message in response to a UNIX signal.

To continue execution of the debugger and get the actual debugger error message, enter `gs`.

```
> gs
MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
Exception state information was lost as runtime searched for a
handler
Use the a.db command "set except_stack" and then re-run your
program
```

The SC Ada debugger, by default, uses high-speed algorithms to perform stack unwinding (`EXCEPT_STACK off`). Unfortunately, if an exception occurs, this stack unwinding can remove information needed by the debugger to further describe the problem. To display this information, disable the high-speed exception stack unwinding process. Enter `set except_stack` to do this and enter `r` to rerun the program.

```
> set except_stack
> r
test_convert
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
process received signal Trace/BPT trap [5]
 stopped 16 instructions after "/usr1/tutorial/convert_b.a":84 in
            get_date_rep
 84         while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
```

Now enter gs to display the additional exception information.

```
>gs
MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
 (program is now stopped on line where exception was raised)
 stopped 16 instructions after "/usr1/tutorial/convert_b.a":84 in
get_date_rep
 84      while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
>
```

It is possible to display the actual instruction on which the exception is raised using either the wi or li command. Enter wi to display the instructions surrounding the instruction where execution stopped. The instruction location is expressed as a hexadecimal number with a leading zero. The instruction indicating the position where execution stopped is marked with an asterisk (*****). Note the exact instructions displayed are dependent on your system.

```
> wi
  02718:  #or      %o1, 0, %g1
  0271c:   add     %g2, %g1, %o0
  02720:   subcc   %i2, %g0, %g0
  02724:   te      +05
  02728:   add     %fp, -028, %o1
  0272c:*  call    018318    -> _A_get_number.30B12.convert
  02730:  #nop
  02734:   or      %o3, 0, %i1
  02738:   st      %o2, [%i2+04]
 102          if NEXT = 0 then
```

If you used li, the instruction on which the exception is raised is the first line listed. What is important is that you can examine the execution of your source program at the assembly level. You can step (si) or advance (ai) over single instructions and set a breakpoint at an instruction (bi).

Another capability of the SC Ada debugger enables you to examine the current machine register contents at any time. Enter `reg` to display the register contents when the program stopped. Display individual registers (see SC Ada *Reference Guide* (`reg`)). This display is machine dependent.

```
> reg
g0:        0  o0:    1e9b4  l0: 11001084  i0:    1e9b4   pc:    19e28
g1:    17ea0  o1:    30658  l1:    1956c  i1:    1e9b0  npc:    19e2c
g2:    18178  o2:    17f31  l2:    19570  i2:        1    y:  1000000
g3: ffffffff  o3:    181a0  l3: ff000009  i3:        0  psr: 11401083
g4: f7e00cf8  o4: fc92d3ec  l4:        b  i4:        1  impl ver nzvc ec
g5:        0  o5: f82ae6ac  l5: ff00000b  i5:    17f31     1   1 0100  0
g6:        0  o6: f7fffac8  l6: f81b1c00  i6: f7fffb20 ef pil s ps et cwp
g7:        0  o7:    19f50  l7: f847af58  i7:    19608  1   0 1  0  0   3
>
```

Use `p` to display the values of `NEXT` and `DATE’LAST` to see if you can shed some additional light on the problem. Enter `p next`. Enter `p date’last`.

```
> p next
14
> p date’last
13
```

NEXT must never be greater than `DATE'LAST`. The problem is that the loop to find the comma in the date is wrong. This loop starts on line 108 in `convert_b.a`. Enter `w 108` to display the loop.

```
> w 108
103                 return D.all;    -- error exit;
104             end if;
105
106             -- find the comma
107             NEXT := NEXT + 1;
108*<           while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
109                 NEXT := NEXT + 1;
110             end loop;
111
112            -- make sure there is more string left to hold the year
```

Check the out-of-bounds condition first and do not use the subscripting if `NEXT` is too big. The two clauses of the `while` loop must be swapped and the `and` changed to an `and then` to prevent the program from evaluating both sides of the expression even if the left side is false. Enter `exit` or `quit` to exit the debugger. Edit `convert_b.a` and modify line 108 to read as follows

```
> quit
% vi convert_b.a
.
.
while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ',') loop
.
.
```

Recompile and link `convert_b.a` and attempt to run it again. This time a new problem appears.

```
% a.make -o test_convert test_convert
% test_convert

"January 1, 1900" --  1/ 1/00
"June 19, 1985" --  12/31/99
"November 26, 1900" -- 11/26/83
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
MAIN PROGRAM ABANDONED -- EXCEPTION "END_ERROR" RAISED
```

Invoke the debugger. Since you know an exception is raised, enter `set except_stack` to disable the stack unwinding. Enter `r` to run the program. You get the following output.

```
% a.db test_convert
> set except_stack

> r
"January 1, 1900" --  1/ 1/00
"June 19, 1985" --  12/31/99
"November 26, 1900" -- 11/26/83
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
MAIN PROGRAM ABANDONED -- EXCEPTION "END_ERROR" RAISED
 (program is now stopped on line where exception was raised)
 stopped 2 instructions after "/standard/integer_io_s.a":157 in
    get'7 (INSTANTIATION)
 157                  raise END_ERROR;
      ----------------^
=> RM 11.3(3): RAISE statement raised named exception
```

An error occurred in one of the routines in the standard library directory. The ability to display information about errors occurring in other libraries and debug into those libraries, especially the standard Ada libraries, is a powerful and very useful capability of the SC Ada debugger.

Enter `cs` to display the call stack.

```
> cs

 # line  procedure         params
 4  157  get'7 (from="  ", item=2, last=5)
 5   15 get_number (date_string="  ", ret_val=99008, place_in_string=5)
 6  119  get_date_rep (date="May 2,  ")
 7   20  test_single_date (date="May 2,  ")
 8   54  test_convert ()
>
```

By looking at the program, call stack and output, you see the problem is occurring when the input includes the month, date and comma but no year following the comma (line 54 in `test_convert.a`).

```
TEST_SINGLE_DATE("May 2,  ");      -- comma, no year
```

To more easily debug this problem, execute the program up to this call and stop. Do this by setting a breakpoint at this specific call using the `when` option of the breakpoint (`b`) command. Enter:

```
> b TEST_SINGLE_DATE when DATE = "May 2,  "
```

Make sure there are *two* spaces after the comma. Enter **r** to rerun the program. Execution stops at this call.

```
> r
test_convert
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83

"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
[1]  stopped at "/usr1/tutorial/test_convert.a":16 in
test_single_date
  16      procedure TEST_SINGLE_DATE(DATE : STRING) is
```

Enter `p date` to display the value of `DATE` to double check that you are in the right place.

```
> p date
"May 2,  "
```

Now you are at the call that is causing the exception. Knowing that the problem is occurring when there is an error in the year field of the date string, you decide a good place to start is with the procedure call that returns the year from the date string (line 119 in `convert_b.a`).

```
GET_NUMBER(DATE(NEXT..DATE'LAST), D.YEAR, NEXT);
```

Since this line is in `convert_b.a`, use the enter, `e convert_b.a`, command to go to the start of that file. Then enter `/d.year` to use the search capability of the debugger to locate the procedure call. You can search for `GET_NUMBER` but remember that the procedure is called twice and you want the second call.

```
> e convert_b.a
"/usr1/tutorial/convert_b.a"::1
1< package body convert is
> /d.year
119<             GET_NUMBER(DATE(NEXT..DATE'LAST), D.YEAR, NEXT);
```

Enter `b` to set a breakpoint at the call. Enter `g` to continue program execution.

```
> b
> g
[2]  stopped at "/usr1/tutorial/convert_b.a":119 in get_date_rep
 119             GET_NUMBER(DATE(NEXT..DATE'LAST), D.YEAR, NEXT);
```

You suspect that the problem is with the string defined as the first parameter to `GET_NUMBER`. Enter `s` to step into the procedure. Enter `p date_string` to display the value of `DATE_STRING`, its first parameter. It contains no valid numerical characters.

```
> s
 stopped at "/usr1/tutorial/convert_s.a":4 in get_number
   4  procedure GET_NUMBER(DATE_STRING: STRING; RET_VAL,
> p date_string
"  "
```

Enter **s** two more times (remember you can use RETURN also) until you reach procedure GET in `integer_io_s.a`.

```
> s
 stopped at "/usr1/tutorial/convert_s.a":15 in get_number
  15    IIO.GET(DATE_STRING, RET_VAL, PLACE_IN_STRING); --RM
14.3.7(13)
> <RETURN> or s
 stopped at "/usr/ada_2.1/self/standard/integer_io_s.a":122 in
get'1
122     procedure get   (from:  in string;
```

Enter `cs` to display the call stack.

```
> cs
 # line  procedure         params
 1 142  get'7 (from="  ", item=2, last=0)
 2   15  get_number (date_string="  ", ret_val=99008, place_in_string=5)
 3  119  get_date_rep (date="May 2,  ")
 4   20  test_single_date (date="May 2,  ")
 5   54  test_convert ()
```

The string (`from`) to be analyzed has no valid numerical characters in it. This causes `procedure GET` to attempt to read past the end of the string and an END_ERROR exception to be raised. The error is in `convert_b.a`. You must be sure a valid numeric entry is in the year portion of the date string before calling GET_NUMBER to retrieve it. Enter `exit` or `quit` to exit the debugger.

```
>quit
```

You must now fix `convert_b.a`. Do this manually, following the instructions below or copy it from the *SCAda_location*/`examples` directory. It is there as `convert_b.a`.

```
% cp /examples/convert_b.a .
% chmod +w convert_b.a
```

or

```
% vi convert_b.a
```

1. Remove the following two lines from the file (lines 112 and 113).

```
    -- make sure there is more string left to hold the year
NEXT := NEXT + 1;
```

2. Add the following lines immediately before the call to `GET_NUMBER` that returns the year value (now line 116).

```
    -- make sure there is a numeric entry in the year field
while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= '1') loop
   NEXT := NEXT + 1;
end loop;

if NEXT >= DATE'LAST then
return D.all;    -- error exit;
end if;
```

Call `a.make` to recompile and link `convert_b.a` and execute the program.

```
% a.make -o test_convert test_convert
% test_convert

"January 1, 1900" --  1/ 1/00
"December 31, 1999" --  12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
"December 17 1986" --  ERROR
"January " --  ERROR
"July 3" --  ERROR
"May 2, " --  ERROR
"June 19, 1885" -- ERROR
```

Voila - it works!

Note – If you have completed the line-oriented debugging session, you must start this tutorial with the original `convert_b.a` and executable files. Recopy the first file from the *`SCAda_location/`*`examples` directory where it is held as `convert_b.deb`.

```
% cp /examples/convert_b.deb convert_b.a
% chmod +w *.a
```

Reinitialize your project library.

```
% a.cleanlib
```

Relink the files to produce a new executable and you are ready to begin.

```
% a.make -v test_convert -o test_convert -f *.a
```

you have probably discovered, working with the debugger in line mode is somewhat tedious. It is possible to switch from line mode to screen mode at any time by entering `vi` at the debugger prompt. Return to line mode from screen mode by entering `Q`. The next section describes the process for debugging this program in screen mode.

References

`a`, *SPARCompiler Ada Reference Guide*

`b`, *SPARCompiler Ada Reference Guide*

`bd`, *SPARCompiler Ada Reference Guide*

`cs`, *SPARCompiler Ada Reference Guide*

`e`, *SPARCompiler Ada Reference Guide*

`g`, *SPARCompiler Ada Reference Guide*

`gs`, *SPARCompiler Ada Reference Guide*

`l`, *SPARCompiler Ada Reference Guide*

`lb`, *SPARCompiler Ada Reference Guide*

`p`, *SPARCompiler Ada Reference Guide*

`r`, *SPARCompiler Ada Reference Guide*

`reg`, *SPARCompiler Ada Reference Guide*

`<RETURN>`,*SPARCompiler Ada Reference Guide*

`s`, *SPARCompiler Ada Reference Guide*

`set`, *SPARCompiler Ada Reference Guide*

`w`, *SPARCompiler Ada Reference Guide*

`wi`, *SPARCompiler Ada Reference Guide*

7.4 Screen-oriented Debugging Session

To invoke the debugger in screen mode, enter the following:

```
% a.db -v test_convert
```

The debugger displays the following information.

```
   4  with IIO;
   5* procedure TEST_CONVERT is
   6  --
   7  -- This test file contains a list of correct and incorrect dates
   8  -- to be tested.  Each date is analyzed by TEST_SINGLE_DATE. It
   9  -- calls the GET_DATE_REP procedure defined in CONVERT.
  10  -- The entered date is displayed.  The value of DAY.MONTH is
  11  -- then checked.  If it is 0, an error has occurred in the
  12  -- processing of the entered date string and an error message is
  13  -- displayed.  Otherwise, the numeric equivalent of the date is
  14  -- displayed.
  15  --
  16          procedure TEST_SINGLE_DATE(DATE : STRING) is
  17              DAY : CONVERT.DATE_REP;
  18              TURN_CENTURY: constant STRING(1..2) := "00";
*-----------------------------------------------------test_convert.a--
    Debugging: /usr1/tutorial/test_convert
    SCAda_library: /usr1/tutorial
    library search list:
    /usr1/tutorial

        /self/verdixlib
        /self/standard
```

This display has several important features. The screen is divided into three windows. The top window is the source window and extends from the top of the screen to the dashed line (default size is the top 2/3 of the screen). It displays the program source or machine-level instruction text. The window below the dashed line is the command window and displays debugger and program input and output. The last line on the screen is the error window and displays error or other messages (currently blank).

The ***** (next to line 5) indicates it is the next line to execute. At the right end of the dashed line is the name of the source file being displayed (`test_convert.a`). At the left side of the dashed line is an asterisk (*****). This asterisk bounces back and forth between columns 1 and 2 with every debugger command. This movement is useful because it indicates something is happening even if no other sign of it exists. For example, if a small loop is contained entirely in the source window and a breakpoint is set in the loop, entering `g` (go) executes the loop one time and stops at the same breakpoint. The screen contents are identical except that the ***** on the dashed line changes columns.

Use all line-mode debugger commands in screen mode by preceding the command with a colon (:), e.g., `:reg`, `:exit`, `:task`, etc.. In addition to the line-mode commands that must be preceded by a colon, you have two additional classes of screen-mode commands available to you, immediate and window control commands.

The immediate commands include many of the more common debugger commands. They do not require a preceding colon and are not followed by `<RETURN>` but rather execute immediately. These commands execute a program (`g`, `r`, `a`, `s`), perform call stack operations (`cb`, `cd`, `cs`, `ct`, `cu`), display variables (`p`, `P..p`, `P..a`, `P..*`), manipulate breakpoints (`b`, `B`, `d`) and control debugger operation (`<CONTROL-]>`, `<CONTROL-C>`). Use of many of these is demonstrated in this debugger tutorial.

The window control commands control the screen interface. They are case sensitive. They do not need to be preceded by a colon or followed by `<RETURN>`. For those of you familiar with `vi(1)`, many of the commands are very similar. Note that these commands apply to the window that contains the cursor.

The best way to become familiar with these commands is to use them. Enter each command listed below and note the change on your screen.

1. Screen Movement and Display

```
<CONTROL-F>    (move forward one window)
<CONTROL-B>    (move backward one window)
<CONTROL-D>    (move down 1/2 window (8 lines) - can be preceded by a
               number)
<CONTROL-U>    (move up 1/2 window (8 lines)- can be preceded by a
               number)
<CONTROL-R>    (redraw screen)
5C             (change size of window where cursor is to 5 lines
               (source))
C              (return window sizes back to default (source = 15))
G              (move to end of file - can be preceded by a line number)
*              (display screen that contains the current cursor
               position)
```

2. Line Movement (*h*, *l*, *j*, *k* commands can be preceded by a number)

```
h              (move 1 column to the left)
l              (move 1 column to the right)
j              (move down 1 line)
k              (move up 1 line)
0 or ^         (move to beginning of line)
w              (move to next word)
$              (move to end of line)
```

3. Searching (/ and ? lines must be followed by <RETURN>)

```
]]             (move forward to next procedure, function, package task
               or declare block (TEST_SINGLE_DATE))
[[             (move back to next procedure, function, package, task
               or declare block (test_convert))
/ iio.put      (move down to first occurrence of IIO.PUT - line 30)
n              (repeat search - line 32)
? end if       (move up to next if - line 28 (end if;))
%              (move to start of if block matching the end if - line
               25 (can go both ways))
```

4. Information

```
<CONTROL-G>    (print name of file displayed in source window in error
               window)
H              (display help line)
H              (display next help line)
```

5. Cursor Control

```
,              (move cursor to other window (command, in this case))
,              (move cursor to other window (back to source window))
```

6. Instruction Submode

```
I              (enter instruction submode)
I              (exit instruction submode - enter source submode)
```

7. Debugger Commands (any command available in line mode is also available in screen mode but must be preceded by a colon .)

```
:task          (list current task - none active so debugger indicates
               this)
.              (repeat last debugger command (commands preceded by a
               colon))
:1             (go to line 1)
```

8. Line Mode Operation

```
Q              (enter line mode)
vi             (enter screen mode (from line mode))
```

This is a quick overview of the screen-mode commands. The SC Ada *Reference Guide* (screen mode) discusses them in detail.

Enter `:set all` to examine the current debugger parameter settings. The following is displayed in the command window.

```
-----------------------------------------------------test_convert.a--
:set all
case: off             except_stack: off     input: pty
lines:  10            log: deb.log          number: on
obase: 10             output: pty           page: on
persist: off          prompt: ">"           safe: off
source: . /usr1/tutorial                    tabs:  4
SCAda directory: /usr1/tutorial              verbose: off
-  More  -
```

When the `- More -` is displayed, this indicates that additional information is to be displayed in the command window but it does not fit. You have several options available.

`<SPACE>`	display next window of text, portion of current screen scrolls off
`<RETURN>`	display next line of text, one line will scroll off
`g`	enlarge command window to display entire message (normal window settings are restored at the next command or by entering **C**)
`x`	abort command
`p`	turn off paging for both debugger commands and program output
`q`	do not display remaining information but enter it into history buffer and write it to the specified log file (if one is specified)

Enter `g` to display the entire message and the following is displayed.

```
 37            IIO.PUT(DAY.YEAR - 1900, 2);
 38        end if;
 39        NEW_LINE;
 40    end TEST_SINGLE_DATE;
 41  begin
 42        -- first test some good dates
 43
 44*   TEST_SINGLE_DATE("January 1, 1900");
 45    TEST_SINGLE_DATE("December 31, 1999");
 (Continued)
*---------------------------------------------------test_convert.a--
:set all
alert_freq: 100       async: off           case: off
c_types: local        event: 0             except_stack: off
input: pty            lines:  10           log: deb.log
number: on            obase: 10            output: pty
page: on              persist: off         prompt: ">"
safe: off             signal  b: hup..pipe, term..tstp, chld..32
signal  g: cont       signal gs: alrm      source: /usr1/tutorial

        /verdixlib /standard
step_alert: 1000      tabs:  4             trace: ,v off,s off
SCAda directory: /usr1/tutorial              verbose: off
xrate:    5.000       with on: all
run: test_convert
```

Note that the tabs setting is 4 and a log file is being produced (`deb.log`). These changes from the default settings are a result of your entries in the `.dbrc` file(see *Debugger Configuration Parameters (*`.dbrc`*)* on page 27). You decide you do not want to generate a log file for the current debugger session. To change this entry, use the command history and line editing features available.

In edit mode, every time you enter `k`, the previous line oriented debugger command (that is, a command preceded by a :), is displayed. Entering `j` advances one command. When a command is displayed, it can be edited. The SC Ada *Reference Guide* (line editing) lists all line editing commands. Pressing `<RETURN>` executes the command and exits edit mode.

Enter `:` then enter `<ESC>` to get into edit mode. Enter `k` to display the previous command (`set all`). The cursor is at the end of the line. Enter `cb` to change the last word. The cursor moves to the beginning of the word and a `$` appears at its end (`al$`). Enter `log off` (the command should read `:set log off`) and enter `<RETURN>`. The command executes and you return to insert mode. Of course, you can just directly enter the command, `:set log off`, but you do not learn anything new that way.

We are now ready to run the program. Start the program using one of four commands.

`r` (run)	starts programs from their beginning. Processes invocation parameters and handles I/O redirection.
`g` (go)	starts execution from where the program last stopped. Cannot process invocation parameters but uses the current parameters.
`a` (advance)	advance one source line over subprogram calls. If the program has not yet started, advances one source line.
`s` (step)	advance one source line into a called subprogram. If the program has not yet started, advances one source line.

Because you want to step through the program and it has no invocation parameters, enter `a` (advance) to begin execution. It is an immediate command so it does not need to be preceded by a colon or followed by `<RETURN>`.

This command executes the program up to the next source line in the current unit. Execution stops at the first call to `TEST_SINGLE_DATE` on line 44. This is the first line for which code is generated.

```
   37             IIO.PUT(DAY.YEAR - 1900, 2);
   38         end if;
   39         NEW_LINE;
   40     end TEST_SINGLE_DATE;
   41 begin
   42         -- first test some good dates
   43
   44*    TEST_SINGLE_DATE(”January 1, 1900”);
   45     TEST_SINGLE_DATE(”December 31, 1999”);
   46     TEST_SINGLE_DATE(”November 26, 1983”);
   47     NEW_LINE;
   48         -- now test some bad dates
   49
   50     TEST_SINGLE_DATE(”Aug. 1, 1981”);       -- bad month
   51     TEST_SINGLE_DATE(”March 16 1986”);      -- no comma
-*-------------------------------------------------------test_convert.a--
        /verdixlib /standard
step_alert: 1000      tabs:  4              trace: ,v off,s off
SCAda directory: /usr1/tutorial               verbose: off
```

```
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

Enter `s` to step into `TEST_SINGLE_DATE`. Note that the `*` is now at line 16 (marked here with a `1` to the left of the line number). Enter `s` again to move to line 20 (marked with a `2`). Notice that the asterisk on the dashed line changes columns. Also note that the use of `<RETURN>` to repeat the most recent debugger command is not available in screen mode (as it is in line mode).

```
    9  -- calls the GET_DATE_REP procedure defined in CONVERT.
   10  -- The entered date is displayed.  The value of DAY.MONTH is
   11  -- then checked.  If it is 0, an error has occurred in the
   12  -- processing of the entered date string and an error message is
   13  -- displayed.  Otherwise, the numeric equivalent of the date is
   14  -- displayed.
   15  --
(1)16      procedure TEST_SINGLE_DATE(DATE : STRING) is
   17          DAY : CONVERT.DATE_REP;
   18          TURN_CENTURY: constant STRING(1..2) := "00";
   19      begin
(2)20*         DAY := CONVERT.GET_DATE_REP(DATE);
   21          PUT('"');
   22          PUT(DATE);
   23          PUT(""" -- ");
*-----------------------------------------------------test_convert.a--
        /verdixlib /standard
step_alert: 1000      tabs:  4             trace: ,v off,s off
SCAda directory: /usr1/tutorial              verbose: off
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

Enter `s` four times to step into and through the `GET_DATE_REP` function. You stop at the declarations that cause code to be generated by the Ada compiler (numbers 1,2,3, and 4 on the left, in the figure below). You step through this function until you reach the loop statement on line 51.

```
    42
(1) 43  function GET_DATE_REP(DATE : STRING) return DATE_REP is
    44      type ADATE is access DATE_REP;
(2) 45      D          : ADATE    := new DATE_REP’(0, 0, 0);
    46      THE_MONTH : INTEGER;
    47
(3) 48      NEXT      : INTEGER := DATE’FIRST;
    49
    50  begin
(4) 51*     while (NEXT <= DATE’LAST) and then (DATE(NEXT) /= ’ ’) loop
    52          NEXT := NEXT + 1;
    53      end loop;
    54
    55      if NEXT >= DATE’LAST then
    56          return D.all;    -- error exit;
 *------------------------------------------------------convert_b.a--

        /verdixlib /standard
step_alert: 1000      tabs:  4             trace: ,v off,s off
SCAda directory: /usr1/tutorial              verbose: off
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

You want to get past this loop without manually executing it by stepping through all its iterations. You see the next statement after the loop is line 55. Enter `j` (or down-arrow) four times to move down to that line (55) and enter `b` to set a breakpoint there. In screen mode, the debugger shows lines with breakpoints by putting an = next to the line number.

To skip over the loop and stop execution at the breakpoint, enter `g` (go).

```
   42
   43  function GET_DATE_REP(DATE : STRING) return DATE_REP is
   44      type ADATE is access DATE_REP;
   45      D          : ADATE   := new DATE_REP'(0, 0, 0);
   46      THE_MONTH : INTEGER;
   47
   48      NEXT      : INTEGER := DATE'FIRST;
   49
   50  begin
   51      while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
   52          NEXT := NEXT + 1;
   53      end loop;
   54
   55*=    if NEXT >= DATE'LAST then
   56          return D.all;    -- error exit;
*------------------------------------------------------convert_b.a--

        /verdixlib /standard
step_alert: 1000      tabs:  4             trace: ,v off,s off
SC Ada directory: /usr1/tutorial              verbose: off
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

Enter `6` and then `<CONTROL-D>` to list more of the source code (down to line 62). When you do, you see the `if` statement is used to detect an error. Enter `s` to continue stepping through the program. If an error is detected, you step to statement 56; if not, you step to statement 62, the next executable statement after the `if`.

You make it to line 62 so no errors occurred yet.

```
   48      NEXT      : INTEGER := DATE'FIRST;
   49
   50   begin
   51       while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
   52           NEXT := NEXT + 1;
   53       end loop;
   54
   55=      if NEXT >= DATE'LAST then
   56           return D.all;    -- error exit;
   57       end if;
   58
   59          -- we postpone assigning the month into d.month since 0
   60          -- indicates an error and it is 0 now.
   61
   62*      THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
-*---------------------------------------------------convert_b.a--
        /verdixlib /standard
step_alert: 1000      tabs:  4              trace: ,v off,s off
SC Ada directory: /usr1/tutorial               verbose: off
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

Enter `s` to step into `MONTH_NO`.

```
   26          10 => new STRING'("october"),
   27          11 => new STRING'("november"),
   28          12 => new STRING'("december"));
   29
   30      procedure GET_NUMBER(DATE_STRING: string; RET_VAL,
   31                        PLACE_IN_STRING: out INTEGER) is separate;
   32
   33*     function MONTH_NO(MONTH : STRING) return INTEGER is
   34      begin
   35          for I in MONTH_TAB'range loop
   36              if MONTH = MONTH_TAB(I).all then
   37                  return I;
   38              end if;
   39          end loop;
   40          return 0;
-*------------------------------------------------------convert_b.a--
        /verdixlib /standard
step_alert: 1000      tabs:  4              trace: ,v off,s off
SC Ada directory: /usr1/tutorial               verbose: off
xrate:    5.000       with on: all
run: test_convert
:set log off
test_convert
```

You decide at this point, that you do not want to step into this function. You want to advance over it and check the value of THE_MONTH after it executes. If THE_MONTH is 0, then an error occurred in the subprogram.

Luckily, the SC Ada debugger has an easy way to exit a subprogram you have entered. The command (bd) inserts a breakpoint at the first machine instruction following the return of the current subprogram. Enter :bd to set the breakpoint. Enter g to execute the program to the breakpoint. The cursor returns to the end of the function call (MONTH_NO) on line 62. The hyphen on the breakpoint line listing in the command area indicates that the breakpoint is not on the first instruction executed by this statement. This breakpoint is automatically deleted after it is reached.

```
    55=         if NEXT >= DATE'LAST then
    56              return D.all;    -- error exit;
    57          end if;
    58
    59          -- we postpone assigning the month into d.month since 0
    60          -- indicates an error and it is 0 now.
    61
    62*         THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
    63          if THE_MONTH = 0 then
    64              return D.all;    -- error exit;
    65          end if;
    66
    67               -- get the day
    68          GET_NUMBER(DATE(NEXT .. DATE'LAST), D.DAY, NEXT);
    69          if NEXT = 0 then
-*-----------------------------------------------------convert_b.a--
xrate:    5.000      with on: all
run: test_convert
:set log off
test_convert
:bd
[2]- stopped at "/usr1/tutorial/convert_b.a":62 in get_date_rep
  62          THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
```

Enter `a` to advance to the next line. Check the value of `THE_MONTH` by moving the cursor (enter `h` as many times as necessary) on top of any character in `THE_MONTH` and enter `p`. This displays the name of the variable and its value in the command window.

```
   56            return D.all;    -- error exit;
   57        end if;
   58
   59        -- we postpone assigning the month into d.month since
   60        -- 0 indicates an error and it is 0 now.
   61
   62        THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
   63*       if THE_MONTH = 0 then
   64            return D.all;    -- error exit;
   65        end if;
   66
   67             -- get the day
   68        GET_NUMBER(DATE(NEXT .. DATE'LAST), D.DAY, NEXT);
   69        if NEXT = 0 then
   70            return D.all;    -- error exit;
-*-------------------------------------------------------convert_b.a--
:set log off
test_convert
:bd
[2]- stopped at "/usr1/tutorial/convert_b.a":62 in
     get_date_rep
  62         THE_MONTH := MONTH_NO(DATE(DATE'FIRST .. NEXT - 1));
THE_MONTH
0
```

THE_MONTH is 0, so something is wrong in MONTH_NO. Check the current call stack using the cs (call stack) command. Also, this displays the procedure parameters so you can be sure that the input to the procedure is as expected. Enter cs and the following is displayed in the command window.

```
*---------------------------------------------------------
convert_b.a--
 62          THE_MONTH := MONTH_NO(DATE(DATE’FIRST .. NEXT - 1));
THE_MONTH
0
 # line  procedure         params
 1   63  get_date_rep (date=”January 1, 1900”)
 2   20  test_single_date (date=”January 1, 1900”)
 3   44  test_convert ()
```

The parameters look correct. You decide the easiest thing to do is to set a breakpoint at the call to MONTH_NO and rerun the program. Enter k or <up-arrow> to return to the procedure call (line 62). Enter b to insert a breakpoint and enter r to rerun the program.

Execution stops at the earlier breakpoint you set (line 55). Enter d to delete this breakpoint. Enter g to continue program execution to the breakpoint you want.

Enter `s` to step into `MONTH_NO`. Check the value of `MONTH`. Move the cursor (enter `l`) to it and enter `p`. Since it is a parameter, check its value by entering `1cs` to display only the top entry on the call stack. Enter `1cs`.

```
   26          10 => new STRING'("october"),
   27          11 => new STRING'("november"),
   28          12 => new STRING'("december"));
   29
   30      procedure GET_NUMBER(DATE_STRING: string; RET_VAL,
   31                        PLACE_IN_STRING: out INTEGER) is separate;
   32
  33*    function MONTH_NO(MONTH : STRING) return INTEGER is
   34     begin
   35          for I in MONTH_TAB'range loop
   36              if MONTH = MONTH_TAB(I).all then
   37                  return I;
   38              end if;
   39          end loop;
   40          return 0;
-*-------------------------------------------------------convert_b.a--
2   20  test_single_date (date="January 1, 1900")
3   44  test_convert ()
test_convert
MONTH
"January"
 # line  procedure         params
 1   33  month_no (month="January")
```

The parameter looks correct so you decide to step into the first iteration of the loop that compares the parameter to the table of months. You expect a match with the table entry. Enter **s** twice and the current position (indicated by the `*`) is on line 36.

To look at the first table entry, you have several options. Use the `:p` command since it is a complex expression. Notice that if you just type `:p month_tab(i)`, you get the value of the access type, not the value of the object. To get the value of the object, use `.all` and the command reads `:p month_tab(i).all`.

A second alternative involves the use of the P capabilities of screen mode. Position the cursor anywhere over MONTH_TAB and enter P. An @ appears over the last letter (MONTH_TA@). Enter P again to move the @ to the end of the expression (MONTH_TAB(I@). Type P as many times as needed. When the @ is overwriting the), enter a. The value of MONTH_TAB(I).ALL is displayed.

```
   26          10 => new STRING'("october"),
   27          11 => new STRING'("november"),
   28          12 => new STRING'("december"));
   29
   30      procedure GET_NUMBER(DATE_STRING: string; RET_VAL,
   31                       PLACE_IN_STRING: out INTEGER) is separate;
   32
   33      function MONTH_NO(MONTH : STRING) return INTEGER is
   34      begin
   35          for I in MONTH_TAB'range loop
   36*             if MONTH = MONTH_TAB(I).all then
   37                  return I;
   38              end if;
   39          end loop;
   40          return 0;
-*-----------------------------------------------------convert_b.a--
MONTH
"January"
 # line  procedure         params
 1   33  month_no (month="January")
:p month_tab(i).all
"january"
MONTH_TAB(I).all
"january"
```

This should match your passed parameter (January), but it does not. You see that a J is used on input, but the table has a j. Change the input string to all lower case. Enter :exit or :quit to exit the debugger.

Modify convert_b.a. Enter the following changes yourself or copy the new convert_b file from the *SCAda_location*/examples directory. It is called convert_b.deb1.

```
% cp /examples/convert_b.deb1 convert_b.a
% chmod +w convert_b.a
```

or

```
% vi convert_b.a
```

```
% vi convert_b.a

package body CONVERT is
    .
    .
    .
function MONTH_NO(MONTH: STRING) return INTEGER is
    LC_MONTH: STRING(MONTH’RANGE);
    function TO_LOWER (CH: CHARACTER) return CHARACTER is
    STANDARD_DIFFERENTIAL: constant INTEGER :=
        CHARACTER’POS (’A’) - CHARACTER’POS (’a’);

    begin --to_lower
    if CH in ’A’ .. ’Z’ then
        return CHARACTER’VAL (
            (CHARACTER’POS (CH) - STANDARD_DIFFERENTIAL)
                     );
    else
        return CH;
    end if;
    end TO_LOWER;

    function TO_LOWER (STR: STRING) return STRING is
    RETURN_STRING: STRING (STR’range) := STR;

    begin --to_lower
    for INDEX in STR’range loop
        RETURN_STRING (INDEX) := TO_LOWER (STR (INDEX));
    end loop;
    return RETURN_STRING;
    end TO_LOWER;
begin --month_no
    LC_MONTH := TO_LOWER(MONTH);
    for I in MONTH_TAB’ range loop
    if LC_MONTH = MONTH_TAB(I).ALL then
        return I;
    end if;
    end loop;
    return 0;
```

```
 (Continued)
end MONTH_NO;
    .
    .
    .
end;
```

Figure 7-20 Source Listing for convert_b.deb1

Recompile and link `convert_b.a`. Attempt to execute `test_convert` again.

```
% a.make -o test_convert test_convert
% test_convert
```

This time you get the following output.

```
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
** MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
```

Unhandled exceptions are easy to figure out with the debugger since the debugger executes the program until an exception is raised. Then it stops and shows where the problem is. It displays a diagnostic describing what is wrong. Reinvoke the debugger.

```
% a.db -v test_convert
```

Enter `r` to execute the program. Enter `<SPACE>` in response to the `-More-` in the command window and the following is displayed:

```
  101      GET_NUMBER(DATE(NEXT .. DATE'LAST), D.DAY, NEXT);
  102      if NEXT = 0 then
  103          return D.all;    -- error exit;
  104      end if;
  105
  106               -- find the comma
  107      NEXT := NEXT + 1;
  108*     while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
  109          NEXT := NEXT + 1;
  110      end loop;
  111
```

```
 (Continued)
  112          -- make sure there is more string left to hold the year
  113      NEXT := NEXT + 1;
  114      if NEXT > DATE'LAST then
  115          return D.all;    -- error exit;
*-----------------------------------------------------convert_b.a--
Aug. 1, 1981" --  ERROR
process received signal "Trace/BPT trap" [5]
--> range check
 stopped 7 instructions after "/usr1/tutorial/convert_b.a":108 in
    get_date_rep
 108               while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
```

To continue execution of the debugger and get the actual debugger error message, enter `:gs`. The following is displayed in the bottom window of the screen:

```
MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED

Exception state information was lost as runtime searched for a handler
Use the a.db command "set except_stack" and then re-run your program
```

The SC Ada debugger, by default, uses high-speed algorithms to perform stack unwinding (except_stack off). Unfortunately, if an exception occurs, this stack unwinding can remove information needed by the debugger to further describe the problem. To display this information, disable the high-speed exception stack unwinding algorithms. Enter `:set except_stack` to do this. Enter `r` to rerun the program. Enter `g` in response to the `-More-` to display the entire message and the following is displayed in the command window. Note that the exact message displayed depends on your system.

```
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
process received signal "Trace/BPT trap" [5]
--> range check
 stopped 7 instructions after "/usr1/tutorial/convert_b.a":108 in get_date_rep
 108            while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
```

Now enter `:gs` to display the additional exception information. Enter `g` to display the entire message.

```
:gs

MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
 (program is now stopped on line where exception was raised)
 stopped 7 instructions after "/usr1/tutorial/convert_b.a":108
   in get_date_rep
 108            while (DATE(NEXT) /= ',') and (NEXT <= DATE'LAST) loop
  -----------------------^
=> RM 4.1.1(4): index value outside range of index subtype
```

It is possible to display the actual instruction on which the exception is raised using the list instructions (`:li`) or window instructions (`:wi`) command. However, screen mode offers an instruction submode. In the instruction submode, the source window contains disassembled machine instructions interspersed with source code. In this mode the `s` (step), `a` (advance) and `b` (set breakpoint) commands are interpreted at the instruction level (`si`, `ai` and `bi`). All searching and window commands are available in the instruction submode. The `I` (uppercase) command toggles between source and instruction submodes.

Enter I to display the instruction being executed when the exception occurs. Note that the exact instructions displayed are dependent on your system.

```
  0400c2c:   slt        t4,s0,t0      t4 <- s0 < t0
  0400c30:   beq        t4,$0,8       -> 0400c3c
  0400c34:  #nop
  0400c38:   break      08
  0400c3c:   slt        t3,t1,s0      t3 <- t1 < s0
  0400c40:   beq        t3,$0,8       -> 0400c4c
  0400c44:  #nop
  0400c48:*  break      08
  0400c4c:   subu       t8,s0,t0      t8 <- s0 - t0
  0400c50:   addu       t4,s1,t8      t4 <- s1 + t8
  0400c54:   lb         t6,00(t4)     t6 <- (0(t4) & 0ff) (sign extended)
  0400c58:   nop
  0400c5c:   addi       t5,$0,02c     t5 <- 44
  0400c60:   bne        t6,t5,8       -> 0400c6c
-*-------------------------------------------------------convert_b.a
MAIN PROGRAM ABANDONED -- EXCEPTION ”constraint_error” RAISED
 (program is now stopped on line where exception was raised)
 stopped 7 instructions after ”/usr1/tutorial/convert_b.a”:108 in get_date_rep
 108              while (DATE(NEXT) /= ’,’) and (NEXT <= DATE’LAST) loop
  -----------------------^
=> RM 4.1.1(4): index value outside range of index subtype
```

Each instruction is expressed as a hexadecimal number with a leading zero. The instruction indicating the current execution position is marked with an asterisk (*****). Note that the instruction set can be different for your system and your output may not exactly match the above. What is important is that you can examine and debug your program at the assembly level.

Enter I to return to source submode.

Another capability of the SC Ada debugger enables you to examine the current machine register contents at any time. Enter `:reg` to display the register contents when the program stops. To display all the registers at once, enter `g` when the `-More-` appears. This display is system dependent. Individual registers can be displayed.

```
-*---------------------------------------------------------convert_b.a--
:reg
g0:        0  o0:     1e9b4  l0: 11001080  i0:     1e9b4   pc:     19e28
g1:    17ea0  o1:     30658  l1:     1956c  i1:     1e9b0  npc:     19e2c
g2:    18178  o2:     17f31  l2:     19570  i2:         1    y:   1000000
g3: ffffffff  o3:    181a0  l3:         0  i3:         0  psr: 11401087
g4: f7e00cf8  o4: f8506fb4  l4:         9  i4:         1 impl ver nzvc ec
g5:        0  o5:     24c00  l5: ff000009  i5:     17f31    1   1 0100  0
g6:        0  o6: f7fffac8  l6: f81b1c00  i6: f7fffb20 ef pil s ps et cwp
g7:        0  o7:     19f50  l7: f8506f58  i7:     19608  1   0 1  0  0   7
```

Display the values of `NEXT` and `DATE'LAST` to see if you can shed some light on the problem. To display the value of `NEXT`, move the cursor (enter `h`) to it and enter `p`. To display the value of `DATE'LAST`, move the cursor to `DATE` (use `l`) and press `P`. A `@` overwrites the last letter of `DATE (DAT@)`. Enter `P` again to move the cursor to the last character of the next part of the name expression (`T` in `LAST - DATE'LAS@`). Enter `p` now and the value of `DATE'LAST` is displayed. Use the `:p` command to display its value (if you use only `p`, you get the value of `DATE` only).

```
NEXT
14
DATE'LAST
13
```

NEXT must never be greater than DATE’LAST. The problem is that the loop to find the comma in the date is wrong. This loop is found starting on line 108 in convert_b.a. This is currently displayed on the screen

```
106   -- find the comma
107   NEXT := NEXT + 1;
108*  while (DATE(NEXT) /= ’,’) and (NEXT <= DATE’LAST) loop
109       NEXT := NEXT + 1;
110   end loop;
```

Check the out-of-bounds condition first and do not use the subscripting if NEXT is too big. The two clauses of the while loop must be swapped and the and changed to an and then to prevent the program from evaluating both sides of the expression even if the left side is false. Enter :exit or :quit to leave the debugger. Edit convert_b.a and change line 108 to read as follows:

```
% vi convert_b.a
.
.
while (NEXT <= DATE’LAST) and then (DATE(NEXT /= ’,’) loop
.
.
```

Recompile and link convert_b.a and run it again. This time a new problem appears.

```
    % a.make -o test_convert test_convert
    % test_convert

”January 1, 1900” --  1/ 1/00
”December 31, 1999” -- 12/31/99
”November 26, 1983” -- 11/26/83
”Aug. 1, 1981” --  ERROR
”March 16 1986” --  ERROR
”February ” --  ERROR
”July 3” --  ERROR
** MAIN PROGRAM ABANDONED -- EXCEPTION ”END_ERROR” RAISED
```

Invoke the debugger. Enter `set except_stack` to disable the stack unwinding and get the most information about the exception. Enter `r` to execute the program. The message from the debugger is too long to display in the command window. Enter `g` to display the entire message when the `-More-` is displayed. This adjusts the size of the source and command windows to display the entire message.

```
% a.db -v test_convert
  152          -- test for end_error
  153          --
  154          i := from'first;
  155          loop
  156              if i > from'last then
*--------------------------------------------------------integer_io_s.a--
test_convert
"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
MAIN PROGRAM ABANDONED -- EXCEPTION "END_ERROR" RAISED
 (program is now stopped on line where exception was raised)
 stopped 2 instructions after "/standard/integer_io_s.a":157
 in get'7 (INSTANTIATION)
 157                  raise END_ERROR;
      ----------------^
=> RM 11.3(3): RAISE statement raised named exception
```

An error occurred in one of the routines in the standard library directory. The ability to display information about errors occurring in other libraries and debug in those libraries, especially the standard Ada libraries, is a powerful (and very useful) capability of the SC Ada debugger.

Enter `cs` to display the call stack.

```
  152         -- test for end_error
  153         --
  154         i := from’first;
  155         loop
  156             if i > from’last then
  157*                raise END_ERROR;
  158             end if;
  159             exit when from(i) /= ’ ’ and then from(i) /= ascii.ht;
  160             i := i + 1;
  161         end loop;
  162
  163         getnum(from, result, int_last, error);
  164         if error or int_last = 0 then
  165             raise DATA_ERROR;
  166         else
-*-------------------------------------------------------integer_io_s.a--
=> RM 11.3(3): RAISE statement raised named exception
 # line  procedure         params
 4  157  get’7 (from=”  ”, item=2, last=5)
 5   15  get_number (date_string=”  ”, ret_val=99008, place_in_string=5)
 6  119  get_date_rep (date=”May 2,  ”)
 7   20  test_single_date (date=”May 2,  ”)
 8   54  test_convert ()
```

By looking at the program, call stack and output, you see the problem is occurring when the input includes the month, date and comma but no year following the comma (line 54 of `test_convert.a`).

```
TEST_SINGLE_DATE(”May 2, ”);      -- comma, no year
```

To more easily debug this problem, you want to execute the program up to this call and stop. Do this by setting a breakpoint at this specific call using the when option of the breakpoint (`b`) command. Enter

```
:b TEST_SINGLE_DATE when DATE = “May 2,  ”
```

Make sure there are two spaces after the comma. Enter `r` to rerun the program. Execution stops at this call.

```
   15  --
   16*=    procedure TEST_SINGLE_DATE(DATE : STRING) is
   17          DAY : CONVERT.DATE_REP;
   18          TURN_CENTURY: constant STRING(1..2) := ”00”;
   19      begin
   20          DAY := CONVERT.GET_DATE_REP(DATE);
   21          PUT(’”’);
   22          PUT(DATE);
   23          PUT(””” -- ”);
   24
   25          if DAY.MONTH = 0 then
   26              PUT_LINE(” ERROR”);
   27              return;
   28          end if;
   29
*------------------------------------------test_convert.a--
”December 31, 1999” -- 12/31/99
”November 26, 1983” -- 11/26/83

”Aug. 1, 1981” --  ERROR
”March 16 1986” --  ERROR
”February ” --  ERROR
”July 3” --  ERROR
```

Display the value of `DATE` to double check you are in the right place (use `p`, `:p date` or `lcs`).

```
:p DATE
DATE
”May 2,  ”
```

You are now at the call in `test_convert.a` with the value “May 2, ” that is causing the exception. Knowing that the problem is occurring when an error is in the year field of the date string, you decide a good place to start is with the procedure call that returns the year from the date string (line 119 in `convert_b.a`).

```
GET_NUMBER(DATE(NEXT..DATE’LAST), D.YEAR, NEXT);
```

Because this call is in the GET_DATE_REP procedure in convert_b.a, we must get to this procedure. To do this, move the cursor to GET_DATE_REP and enter <CONTROL-]>.

```
   75
   76   function GET_DATE_REP(DATE : STRING) return DATE_REP is
   77       type ADATE is access DATE_REP;
   78       D          : ADATE   := new DATE_REP'(0, 0, 0);
   79       THE_MONTH : INTEGER;
   80
   81       NEXT      : INTEGER := DATE'FIRST;
   82
   83   begin
   84       while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= ' ') loop
   85           NEXT := NEXT + 1;
   86       end loop;
   87
   88       if NEXT >= DATE'LAST then
   89           return D.all;    -- error exit;
-*-------------------------------------------------------convert_b.a--
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
DATE
"May 2,  "
"/usr1/tutorial/convert_b.a":get_date_rep:76
```

Enter /d.year to use the search capability of the debugger to locate the exact call to GET_NUMBER in GET_DATE_REP. You can search for GET_NUMBER, but remember that the procedure is called twice and you want the second call.

Enter b to set a breakpoint at that call and enter g to continue program execution.

```
  112          -- make sure there is more string left to hold the year
  113          NEXT := NEXT + 1;
  114          if NEXT > DATE'LAST then
  115              return D.all;    -- error exit;
  116          end if;
  117
  118              -- get the year
  119*=         GET_NUMBER(DATE(NEXT .. DATE'LAST), D.YEAR, NEXT);
  120          if NEXT = 0 then
  121              return D.all;    -- error exit;
  122          end if;
  123
  124          D.MONTH := THE_MONTH;
  125          return D.all;
  126      end GET_DATE_REP;
*--------------------------------------------------convert_b.a--
"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
DATE
"May 2,  "
"/usr1/tutorial/convert_b.a":get_date_rep:76
```

You suspect that the problem is with the string defined as the first parameter to GET_NUMBER. Enter s to step into GET_NUMBER. Use p or :p date_string to display the value of DATE_STRING("   ").

```
   3  separate(CONVERT)
   4* procedure GET_NUMBER(DATE_STRING : STRING; RET_VAL,
   5                        PLACE_IN_STRING : out INTEGER) is
   6  --
   7  -- This procedure calls IIO.GET to read an integer value from the      8
-- beginning of DATE_STRING.  It returns in RET_VAL, the integer
   9  -- value that corresponds to the input sequence.  It returns in
  10  -- PLACE_IN_STRING the place value in the string of the last
  11  -- character read.  If an error is detected, the place value is
  12  -- set to 0.
  13  --
```

```
 (Continued)
  14  begin
  15   IIO.GET(DATE_STRING, RET_VAL, PLACE_IN_STRING);  --RM 14.3.7(13)
  16
  17  exception
-*------------------------------------------------------convert_s.a--
"May 2,  "
"/usr1/tutorial/convert_b.a":get_date_rep:76
:p date_string
DATE_STRING
"  "
```

Enter s two times to step through the program until you reach procedure GET in integer_io_s.a. Enter cs to display the call stack. You see the value of “from” is “ ”. It has no valid numerical characters in it. This causes the GET procedure to attempt to read past the end of the string and an END_ERROR exception is raised.

```
  141
  142*     procedure get   (from:  in string;
  143                       item:  out num;
  144                       last:  out positive)
  145      is
  146          result  : integer;
  147          error   : boolean := false;
  148          i       : integer;
  149          int_last: integer;  -- In case getnum returns a last of 0
  150      begin
  151          --
  152          -- test for end_error
  153          --
  154          i := from'first;
  155          loop
-*------------------------------------------------------integer_io_s.a--
"  "
 # line  procedure         params
 1  142  get'7 (from="  ", item=2, last=0)
 2   15  get_number (date_string="  ", ret_val=99008, place_in_string=5)
 3  119  get_date_rep (date="May 2,  ")
 4   20  test_single_date (date="May 2,  ")
 5   54  test_convert ()
```

The error is in `convert_b.a`. You must make sure a valid numeric entry is in the year portion of the date string before calling `GET_NUMBER` to retrieve it. Enter `:exit` or `:quit` to exit the debugger and correct `convert_b.a`. Copy this corrected version of `convert_b.a` from the *`SCAda_location/`*`examples` directory (it is there as `convert_b.a`) or make the changes manually.

```
% cp /examples/convert_b.a .
% chmod +w convert_b.a
```

or

```
% vi convert_b.a
```

1. Remove the following two lines from the file (lines 112 and 113):

```
-- make sure there is more string left to hold the year
NEXT := NEXT + 1;
```

2. Add the following lines immediately before the call to `GET_NUMBER` that returns the year value (now line 116).

```
-- make sure there is a numeric entry in the year field
while (NEXT <= DATE'LAST) and then (DATE(NEXT) /= '1') loop
    NEXT := NEXT +1;
end loop;

if NEXT >= DATE'LAST then
    return D.all;  -- error exit
end if;
```

Call `a.make` to recompile and link the file.

```
% a.make -o test_convert test_convert
```

Execute the program. Voila - it works!

```
% test_convert

"January 1, 1900" --  1/ 1/00
"December 31, 1999" -- 12/31/99
"November 26, 1983" -- 11/26/83

"Aug. 1, 1981" --  ERROR
"March 16 1986" --  ERROR
"February " --  ERROR
"July 3" --  ERROR
"May 2, " --  ERROR
"June 19, 1885" -- ERROR
```

As you can see, the SC Ada debugger is a valuable tool not only for tracing down errors, but also for viewing and understanding the relationship of the source code to the flow of program execution.

The line-mode and screen-mode tutorials have briefly illustrated the use of many of the SC Ada debugger commands. Additional commands enable you to display and assign values to memory locations, move up and down the call stack, edit files, and display task information. The *SPARCompiler Ada Reference Guide* discusses these commands in detail.

References

`a`, *SPARCompiler Ada Reference Guide*

`b`, *SPARCompiler Ada Reference Guide*

`bd`, *SPARCompiler Ada Reference Guide*

`cs`, *SPARCompiler Ada Reference Guide*

`e`, *SPARCompiler Ada Reference Guide*

`g`, *SPARCompiler Ada Reference Guide*

`gs`, *SPARCompiler Ada Reference Guide*

`l`, *SPARCompiler Ada Reference Guide*

`lb`, *SPARCompiler Ada Reference Guide* `p`, *SPARCompiler Ada Reference Guide*

`r`, *SPARCompiler Ada Reference Guide*

`reg`, *SPARCompiler Ada Reference Guide*

`<RETURN>`, *SPARCompiler Ada Reference Guide*

`s`, *SPARCompiler Ada Reference Guide*

`set`, *SPARCompiler Ada Reference Guide*

`w`, *SPARCompiler Ada Reference Guide*

`wi`, *SPARCompiler Ada Reference Guide*

Index