SPARCompiler Ada Reference Guide

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

Part No.: 801-4863-11
Revision B, August 1994

Contents

Figures

Tables

"Thou art the book,
The library whereon I look."

Henry King

SPARCompiler Ada Files and Libraries 1

1.1 Topics

This chapter discusses the following SPARCompiler Ada file, directory, unit, and library topics:

Ada compilation units	Ada units and dependencies in Ada
File Formats	Types of files used by Ada
SC Ada Release Directories	Description of the directories provided with Ada
SC Ada User Libraries	Definition and contents of Ada libraries created by users for source file compilations

Hereafter, SPARCompiler Ada is referred to as SC Ada.

1.1.1 Ada Compilation Units — units and dependencies

Each Ada source file contains the text for a single Ada compilation, which consists of one or more compilation units. A compilation unit is the smallest piece of Ada code that compiles successfully.

A compilation unit is any of the following:

Subprogram declaration
Generic declaration
Package declaration
Generic instantiation
Subprogram body
Library unit body (subprogram body or package body)
Subunit (subprogram body, package body or task body)

A source file contains one or more units, even though all units in the file are not used for the application. The compiler, not knowing which of the units are required for the application, produces a separate object file for each unit in the file. The object file is stored in the directory `.objects`. The prelinker, `a.ld`, determines which units are required for the application and invokes the linker, passing the appropriate object filenames as parameters.

References

Section A.2, “Source File Limits,” on page A-2, Appendix A.

Unit Dependencies

When compiling a unit that references other units, the compiler checks that each reference is valid. It checks actual data types in referencing units against specified data types in the referenced unit.

This checking is automatic when compiling with `a.make`. It is enabled when using `ada`, but Ada compilation units must be compiled in dependency order. Dependency order means if one unit depends on specifications or definitions provided in another unit, the unit containing those specifications or definitions must be compiled first.

The files `fact.a` and `prob.a`, shown here, illustrate a simple case of such a dependency. When the file containing `FACTORIAL` (`fact.a`) is compiled, a record is made in the Ada library that a function named `FACTORIAL` can be called and that `FACTORIAL` requires an `INTEGER` argument and returns an `INTEGER` result, as shown in this code segment:

```
-- file: fact.a --
function FACTORIAL ( N: INTEGER ) return INTEGER is
 begin
if N <= 0 then return 1;
else return N * FACTORIAL ( N - 1 );
end if;
end FACTORIAL;
```

Another compilation, the file `prob.a` that uses `FACTORIAL`, is shown here. The separate compilation information from the compiled unit `FACTORIAL` is used to check that it is called with parameters of the correct number and type in the file `prob.a`, as shown in this code segment:

```
-- file: prob.a --
with FACTORIAL;
function PROBABILITY ( NUM_ITEMS : INTEGER ) return FLOAT
is
begin
return 1.0 / FLOAT ( FACTORIAL ( NUM_ITEMS ));
end PROBABILITY;
```

References

order of compilation, section 10.3 in *Ada Reference Manual*

1.1.2 File Formats — types of files used by SC Ada

This section introduces the various types of files used by SC Ada and briefly describes the file structure.

File formats of several types represent the interface between various portions of the SC Ada tools. For example, the `IL` (Intermediate Language) file generated by the front end of the compiler is used as input to the code generator.

The details of these files are generally of interest only for specialized applications involving the creation or modification of object or other files by tools other than those supplied with SC Ada.

Ada Source Files

Ada source files are ordinary text files. Edit them with any text editor.

The SC Ada compiler no longer requires that Ada source filenames end with the extension `.a`. Specify any suffix. For example, `text_io.spec`, `foo.bar`, and `example.ada` are all valid source filenames.

The root name of a source filename is that part which remains when the suffix is removed. For example, the root name of `text_io.spec` is `text_io`.

SC Ada imposes no restriction on the location of subprogram specifications and bodies. It is often useful to place them in separate files or to distribute them among several program libraries.

Source files must be compiled within an SC Ada library directory. Create an SC Ada library directory anywhere in the file system with the `a.mklib` tool.

When a unit is compiled, an entry is made in the `ada.lib` file. The third field of this entry specifies the name of a file, which is always in the `.nets` subdirectory. If the compiled unit has an associated object file, then that file exists in both the `.objects` and `.lines` subdirectories.

Source File Structure and Restrictions

Character Set - SC Ada provides the full graphic_character textual representation for programs. The character set for source files and internal character representations is ASCII.

Lexical Elements, **Separators**, and **Delimiters** - Ada uses normal text files as input.

References

Section A.2, “Source File Limits,” on page A-2

Object Files

When an Ada source file compiles, the compiler produces a set of object files (one for each unit) that are in the `.objects` directory. The compiler names each unit by replacing the source file extension with a unique two-digit number. The object file format evolved from the Berkeley UNIX 4.2 BSD `a.out` format. The symbol table and string table are still the same but we have altered the `a.out` header to add information that better supports embedded systems. Often object files are referred to as relocatable files, because the linker relocates the files in memory when it produces the executable file.

Executable Files

SC Ada executable files are the end result of the linking process, which links the object files for the units required by an application. SC Ada produces object executable files with the default name `a.out`. Specify an executable filename other than the default when you link your application.

Lines Files

The Ada compiler produces lines files for use by the debugger. They are in the `.lines` directory. Lines files contain the information required by the debugger to map an instruction address to a source file and line number and to support debugging optimized code. Only compilation units that generate code have lines files.

Nets Files

The Ada compiler produces nets files, which contain separate compilation information. They are in the `.nets` directory. When an Ada unit is compiled, its name is entered in the program library with a pointer to the associated net file. The compiler produces one net file for each compilation unit.

The compiler uses this information when a library is `with`ed by another compilation unit. `a.make` uses the information to compute the correct compilation order and to check for units that require recompiling because they are out of date. `a.ld` uses the compilation information to determine which object files must be linked to build a program, the exception tables, and elaboration tables. Finally, the debugger uses the information to provide symbol name, type, and address information for symbolic debugging.

Archive Files

Archiving files provides a compact and efficient method for creating archive libraries of object files and for indexing these files to enable high-speed access by the SC Ada prelinker. Archive files are created by using the standard archiving method provided with your host operating system.

1.1.3 SC Ada Release Libraries — contents of the SC Ada libraries

SC Ada provides a number of directories, including configuration directories, support directories, object file and executable file directories, and runtime libraries. The *SPARCompiler Ada User's Guide* gives a listing and brief description of each SC Ada directory. Appendix A of the *SPARCompiler Ada Programmer's Guide* gives detailed descriptions of the contents and functionality of the user configuration directory, `usr_conf`.

This section details most of the directories that are classified as part of the SC Ada runtime system. These directories are accessed when you compile applications or are used to compile certain programs. The following sections describe the packages contained in the `standard`, `publiclib`, and `verdixlib` libraries. Filenames are in parentheses.

A major SC Ada runtime library, `vads_exec` is not included in this discussion. This library contains the routines that provide the user interface to many of the runtime services including interrupt handling, mailboxes, memory management, semaphores, tasking operations, and stack operations.

This section lists the contents of the `examples` directory.

References

configuring the `usr_conf` library and directories provided with SC Ada, *SPARCompiler Ada User's Guide*
`vads_exec` library, *SPARCompiler Ada Runtime System Guide*

1.1.4 standard

The `standard` library contains specifications and bodies for predefined Ada packages.

Note – Do not recompile the files in this directory in this library.

Packages in `standard` other than those defined in the *Ada Reference Manual* support the predefined packages or provide interfaces to SC Ada RTS services. The following packages are supported.

`A_STRINGS` implements a set of variable-length string operations based on the type definition:

```
type STRING_REC(len: natural) is record
         s: string(1..len);
end record;
type A_STRING is access STRING_REC;
```

This representation provides the best opportunity for optimizations in the future while having convenient reference. We recommend its use for variable-length strings (`a_strings.a, a_strings_b.a`).

`ADA_DEFS` contains the Ada Kernel implementation definitions that are Solaris threads specific. (*ada_location*`/self_thr/standard`). (`a_defs.a`)

`ADA_KRN_DEFS` contains the Ada Kernel type definitions (`ada_krn_defs.a`)

References

Ada Kernel, *SPARCompiler Ada Runtime System Guide*

`ADA_KRN_I` contains the interface to the Ada Kernel services `ada_krn_i.a`)

References

Ada Kernel, *SPARCompiler Ada Runtime System Guide*

C_STRINGS implements a set of variable-length string operations based on the type definition:

```
type C_STRING is access string (1.integer'last);
--WARNING: this package mimics the behavior of
--ASCII.nul-terminated strings commonly used in
--the C programming language. It should be used
--to represent strings that are to be passed to
--and from host OS utilities, for example.
```

(c_strings.a, c_strings_b.a):

CALENDAR implements the predefined package CALENDAR (calendar.a, calendar_b.a, calendar_s.a).

CLOSE_ALL closes all open files. Generally, the RTS calls it on program exit (close_all.a).

CURRENT_EXCEPTION provides the name, in string form, of a raised exception. function EXCEPTION_NAME returns the current exception name. It must be called from an exception handler. If EXCEPTION_NAME is called from outside an exception handler, a zero length string is returned (curr_except.a).

```
package CURRENT_EXCEPTION is
         function EXCEPTION_NAME return string;
         pragma BUILT_IN(EXCEPTION_NAME);
end;
```

Note – Use V_I_EXCEPT.EXCEPTION_CURRENT and V_I_EXCEPT.EXCEPTION_NAME to get the current exception name when outside the exception handler.

DATES provides functions for commonly-used manipulations of dates (dates.a, dates_b.a).

DIRECT_IO implements the predefined package DIRECT_IO (dir_io.a, dir_io_b.a).

ENUMERATION_IO implements the body for the generic package ENUMERATION_IO defined in the predefined package TEXT_IO (enum_io_s.a).

ERRNO contains enumeration type definitions for error codes returned by OS functions and the interface to the errno variable (errno.a).

ERRNO_SUPPORT contains the routines to get and put the UNIX errno value for the current task. This addresses the multiprocessor case where errno needs to be accessed through special OS provided services (errno_sup.a).

FCNTL and FNCTL(2) are the OS file control packages (fcntl.a).

FILENAMES defines a set of utilities for dealing with filenames on operating systems that support SC Ada. It is portable and its use is recommended for creating portable system utilities. For example, the following code calls a subprogram DO_IT for all files in a directory with names matching the pattern '*str*.a' (filenames.a, filenames_b.a):

```
THIS_DIR: FILE_NAMES.FIND_FILE_INFO :=
           FILE_NAMES.INIT_FIND_FILE (to_A( "*str*.a"));
         FILE: A_STRINGS.A_STRING;
begin
         loop
                 file := FILE_NAMES.FIND_FILE(THIS_DIR);
                 doit(file);
         end loop;
exception
when FILE_NAMES.NO_MORE_FILES => ...
```

FILE_SUPPORT supports file activities for Ada standard I/O. Use this package for lower-level file activities other than those defined by Ada TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO (file_spprt.a, file_spprt_b.a).

FIXED_IO implements the body for the generic package FIXED_IO defined in the predefined package TEXT_IO (fixed_io_s.a).

FLOAT_IO implements the body for the generic package FLOAT_IO defined in the predefined package TEXT_IO (float_io_s.a).

HEX supports INTEGER'IMAGE and INTEGER'VALUE (as functions rather than attributes) but using hexadecimal strings rather than decimal (hex.a, hex_b.a).

IFACE_INTR provides interface to the OS signal handling services (iface_intr.a).

INTEGER_IO implements the body for the generic package INTEGER_IO defined in the predefined package INTEGER_IO (integer_io_s.a).

IO_EXCEPTIONS implements the predefined package IO_EXCEPTIONS specification (io_excpt.a).

IOCTL and IOCTL_FMT supply an interface to OS I/O control functions (ioctl.a, ioctl_fmt.a).

KRN_CALL_I contains the interface to the routines resident in the user program that calls the kernel services. Only present for VADS MICRO (*ada_location*/self/standard) (krn_call.a)

KRN_CPU_DEFS contains the kernel program's type definitions that are CPU-specific. Only present for VADS MICRO (*ada_location*/self/standard) (krn_cpu_defs.a)

KRN_DEFS contains the kernel program's type definitions. Only present for VADS MICRO (*ada_location*/self/standard) (krn_defs.a)

KRN_ENTRIES contains the kernel program's service entry IDs and arguments. Only present for VADS MICRO (*ada_location*/self/standard) (krn_entries.a)

LANGUAGE defines the prefixes and suffixes in the link names generated by the system linker. Use it in interface programming. It provides portability across operating systems (language.a).

LIBC supplies an interface to Solaris 2.1 library functions (libc.a).

LINK_BLOCK defines the communication structure for the debugger, the runtime kernel, and the user program (link_block.a, link_block_b.a).

LOW_LEVEL_IO implements LOW_LEVEL_IO as described in the *Ada Reference Manual* to access physical devices. Replace the null subprograms contained in the body of the package with your own routines (lowlevel_io.a).

MACHINE_TYPES supplies definitions of byte (8-bit) and word (16-bit) unsigned types (mach_types.a).

MACHINE_CODE defines machine code statements described in *Ada Reference Manual* 13.8 (machine_code.a).

MEMORY supplies PEEK, POKE, and COPY on untyped memory. Because these routines circumvent the normal memory protection provided by Ada, use this package with extreme caution (memory.a, memory_b.a).

NUMBER_IO contains support routines used by the TEXT_IO in FIXED_IO, FLOAT_IO, and INTEGER_IO package bodies (number_io.a, number_io_b.a).

OS_FILES supplies a machine-independent interface to low-level I/O operations. Write programs using OS_FILES if the files contain untyped binary information (os_files.a, os_files_b.a).

OS_SIGNAL provides the interface to the OS's signal services used by the Ada RTS. It also contains the typed definitions for the signal structures and the signal number constants (os_signal.a).

OS_SYNC provides the interface to the mutex, condition variable and counting semaphore data structures, and services provided by Solaris threads (ada_location/self_thr/standard) (os_thread.a).

OS_THREAD provides the interface to the Solaris thread services. It is only present for Solaris MT Ada (ada_location/self_thr/standard) (os_thread.a)

OS_TIME provides the interface to the UNIX time services. (os_time.a)

OS_VARIANT contains host OS specific routines used by the OS_FILES package body (os_variant.a, os_variant_b.a).

PERROR contains routines for getting (GET_MSG) or putting (PERROR) the message text associated with error codes returned by Solaris 2.1 functions (perror.a, perror_b.a).

RAW_DUMP prints regions of memory in hexadecimal representation (raw_dump.a).

SAFE_SUPPORT provides Ada tasking safe support for I/O (safe_sup.a, safe_sup_b.a)

SEQUENTIAL_IO implements the predefined package SEQUENTIAL_IO (seq_io.a, seq_io_b.a).

SHARED_IO triggers generation of shared object code for generic packages in TEXT_IO for the common Ada types. This package improves disk usage by ensuring that only one instantiation of these I/O packages exists for each SC Ada installation. (shared_io.a).

SIMPLE_IO provides unprotected subprograms that can be called from an ISR (UNIX signal handler) (simple_io.a).

STATUS supplies a definition of the Solaris 2.1 STATUS_BUFFER data type. This type is returned by calls to stat(2) and fstat(2) (status.a, status_b.a).

STRINGS defines types and routines for manipulating normal Ada strings (strings.a, strings_b.a

STRLEN, STRNCPY duplicates the corresponding C functions (strlen.a, strlen_b.a, strncpy.a, strncpy_b.a).

SYSTEM is the predefined Ada package SYSTEM (system.a).

TEXT_IO implements the predefined package TEXT_IO (text_io.a, text_io_b.a).

TEXT_SUPPRT contains support routines used by TEXT_IO and NUMBER_IO package bodies (text_sup.a, text_sup_b.a).

TTY and TTY_SIZES supply a definition of the Solaris 2.1 tty(4) data structures (tty.a, tty_b.a, tty_sizes.a).

U_ENV supplies a definition of the command line argument interface (u_env.a).

UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION specify the predefined generic library subprograms (unchecked.a).

UNIX supplies a direct interface to the most common Solaris 2.1 system calls (unix.a, unix_b.a).

UNIX_DIRS supplies a specialized interface for the Solaris 2.1 directory manipulation utilities (unix_dirs.a, unix_dirs_b.a).

UNIX_LIMITS is an interface to the Solaris 2.1 limit commands (unix_limits.a).

STATUS supplies a definition of the Solaris 2.1 STATUS_BUFFER data type. This type is returned by calls to stat(2) and fstat(2) (status.a, status_b.a).

STRINGS defines types and routines for manipulating normal Ada strings (strings_.a).

STRLEN, STRNCPY duplicates the corresponding C functions (strlen.a, strlen_b.a, strncpy.a, strncpy_b.a).

SYSTEM is the predefined Ada package SYSTEM (system.a).

TEXT_IO implements the predefined package TEXT_IO (text_io.a, text_io_b.a).

TEXT_SUPPRT contains support routines used by TEXT_IO and NUMBER_IO package bodies (text_sup.a, text_sup_b.a).

TTY and TTY_SIZES supply a definition of the Solaris 2.1 tty(4) data structures (tty.a, tty_b.a, tty_sizes.a).

U_ENV supplies a definition of the command line argument interface (u_env.a).

UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION specify the predefined generic library subprograms (unchecked.a).

UNIX supplies a direct interface to the most common Solaris 2.1 system calls (unix.a,unix_b.a).

UNIX_DIRS supplies a specialized interface for the Solaris 2.1 directory manipulation utilities (unix_dirs.a, unix_dirs_b.a).

UNIX_LIMITS is an interface to the Solaris 2.1 limit commands (unix_limits.a).

UNIX_PRCS supplies a specialized interface to the Solaris 2.1 process-control utilities (unix_prcs.a).

UNIX_TIME is an interface to UNIX time functions (unix_time.a).

UNSIGNED_TYPES is supplied to illustrate the definition of and services for the unsigned types supplied in this version of SC Ada (unsigned.a).

Caution – Use `package UNSIGNED_TYPES` with caution. We do not give any warranty, expressed or implied, for the effectiveness or legality of this package The package is supplied in comment form because the actual package cannot be expressed in normal Ada - the types are not symmetric about 0 as required by the *Ada Reference Manual*.

`USER_DEFS` contains the type definitions for services resident in the user program. Only present for VADS MICRO (*ada_location*`/self/standard`) (`user_defs`).

`UTIMES` provides an interface to the `utimes(2)` function of BSD UNIX and the `utimes(2)` function for UNIX System V (`utimes.a, utimes_b.a`).

`V_ADA_INFO` provides the interface to the `ada.lib INFO` directive parameters placed in the executable by `a.ld` (for example, `PROCESSOR_TYPE`) (`v_ada_info.a`).

`V_BITS` is the inline equivalent of `V_I_BITS` and contains an identical interface (`v_bits.a`, `v_bits_b.a`).

This routine results in faster code because call overhead is eliminated. The `V_BITS` functions assume that the parameters passed to them are accessible in a specific way. While this method works in the majority of cases, in some cases it does not. It does not work on a RISC machine, for example, where any of the parameters passed to one of these functions is stored at some location in memory rather than in a register. If your program does not compile using this package, chances are the parameters are not stored in the way that the function expects.

Caution – The SC Ada compiler does not correctly print error messages in this case - the compiler detects an error but it does not flag the correct line. One simple work-around is to use `package V_I_BITS` instead.

`V_I_ALLOC` interfaces to the SC Ada memory allocation services (`v_i_alloc.a`).

This package provides the interfaces to the heap memory allocators for the user space. Normally, it is not needed by the user since using “new” and instantiating `UNCHECKED_DEALLOCATION` work. However, you can bypass the

normal Ada mechanisms and use these functions directly. If you write a new memory allocator to supplant the default, this package provides the spec to write it against.

V_I_BITS supplies BIT_AND, BIT_OR, BIT_XOR, BIT_NEG, BIT_SRA, BIT_SRL, and BIT_SRL on integers (v_i_bits.a).

V_I_CALLOUT interfaces to the program, task and idle callout services (v_i_callout.a).

V_I_CIFO interfaces to the CIFO type definitions used by Ada tasking (v_i_cifo.a).

V_I_CSEMA is the low-level interface to counting (nonbinary) semaphores (v_i_csema.a).

V_I_EXCEPT is the low level interface to the Ada exception services such as getting the ID, PC, and string name of the current Ada exception or installing a callout for raised exceptions (v_I_except.a).

References

list of services, *SPARCompiler Ada Runtime System Guide*

V_I_INTR interfaces to the SC Ada interrupt services (v_i_intr.a).

V_I_LIBOP interfaces to library routines called by the compiler. It has support for image and value attributes, val/pos, bit copy and test, catenation, memory copy, zero and compare, fixed point mantissa, and string copy (v_i_libop.a).

V_I_MBOX is the low-level interface to the runtime mailbox services (v_i_mbox.a).

V_I_MEM is the low-level interface to the runtime memory services - fixed, flex, and heap pools (v_i_mem.a).

V_I_MUTEX is the low level interface to the ABORT_SAFE mutes services (v_i_mutex.a).

References

list of services, *SPARCompiler Ada Runtime System Guide*

V_I_PASS interfaces to the SC Ada passive task data structures and support services (v_i_pass.a).

V_I_RAISE interfaces to the SC Ada exception support services (v_i_raise.a).

V_I_SEMA is the low level interface to binary semaphores (v_i_sema.a).

V_I_SIG interfaces to the SC Ada interrupt entry signal services (v_i_sig.a).

V_I_TASKOP interfaces to the Ada tasking subprograms called by the compiler to implement the Ada tasking semantics (v_i_taskop.a).

V_I_TASKS provides the low-level interface to binary semaphores (v_i_tasks.a).

V_I_TIME interfaces to the SC Ada time subprograms (v_i_time.a).

V_I_TIMEOP interfaces to the SC Ada time operator subprograms (v_i_timeop.a).

V_I_TYPES supplies types used in the SC Ada Runtime System. It includes the type definitions for TIME_T, ALLOC_T, TEST_AND_SET_T, and FLOATING_POINT_CONTROL_T. (v_i_types.a).

V_SEMA is the inline equivalent of V_I_SEMA and contains an identical interface (v_sema.a).

V_TAS provides inline test-and-set capability (v_tas.a, v_tas_b.a).

V_USR_CONF_I contains the interface for configuring the user library (v_usr_conf_i.a).

Note – This file has been moved from the usr_conf directory

X_CALENDAR provides extensions to CALENDAR, such as SET_CLOCK and DELAY_UNTIL (xcalendar.a).

References

Machine Code Insertions, package MACHINE_CODE, and package UNSIGNED_TYPES, *SPARCompiler Ada Programmer's Guide*

1.1.5 verdixlib

The SC Ada library `verdixlib` contains Ada packages that provide mathematical functions and other capabilities to the user. These packages are proprietary to SunSoft. Both `verdixlib` and `standard` are automatically on the path of any SC Ada library created by `a.mklib` (unless a parent library is given), making their packages available to user programs. Additional information is supplied in the header of each package specification file.

Note – Do not recompile the files in this directory.

`COMMAND_LINE` provides easy access to the command line arguments used to invoke a program. These closely follow the C language conventions and enable the program to access the environment variables as well (`cmd_line.a`).

`COMPLEX_ARITH` provides functions for complex value arithmetic using generic floating types and functions for composition and decomposition of the complex data type (`complex_body.a`, `complex_spec.a`).

`DATES` provides functions for commonly-used manipulations of dates (`dates.a`, `dates_b.a`).

`MATH` provides mathematical constants, exponential, logarithmic, circular trigonometric, inverse circular trigonometric, hyperbolic trigonometric, polar conversion, and Bessel functions (`math_spec.a`, `math_body.a`).

`ORDERING` provides various generic sorting and permutation routines and is instantiated with a variety of data types (`ordering_b.a`, `ordering_s.a`).

`REPORT` provides functions for reporting the pass / fail / not-applicable results of tests (`report_spec.a`, `report_body.a`).

`UNIX_CALLS` provides Ada language routines for performing many of the most common and useful UNIX system calls (`unixcallspec.a`, `callbody.a`).

1.1.6 publiclib

`publiclib` contains public domain and other Ada packages not supported by SunSoft but used in programming. Complete source code is provided. The headers of the package specification files supply additional information.

`BIT_FLG_FIX` provides integer to bit-field conversions. It is used with `CURSES` (`bit_flg_fix.a`).

`C_PRINTF` is an implementation of `printf` for C programs converted to Ada (`c_printf.a`, `c_printf_b.a`).

`C_TO_A_TYPE` provides Ada equivalents for C types (`c_to_a_type.a`).

`CHARACTER_TYPE` provides character class comparisons like the C `ctype` macros (`char_type.a`).

`CURSES` is an interface to the `curses` library (`curses_body.a`, `curses_spec.a`).

`U_RAND` is a package implementing a random number generator (`u_rand.a`).

`VSTRINGS` is a package implementing variable length strings (`vstring_body.a`, `vstring_spec.a`).

1.1.7 examples

`examples` is a directory containing SC Ada programming examples using the libraries listed above. Each file contains directions on compiling and linking its program. Copy the files to a user Ada library and compile.

`alloc_exer.a` `alloc_exer_b.a`	Memory allocation exercise package
`arguments.a`	Tests the `COMMAND_LINE` interface in `verdixlib`

`convert.a`
`convert.cmp`
`convert_b.a`
`convert_b.cmp`
`convert_b.deb`
`convert_b.deb1`
`convert_s.a`
`convert_s.cmp`
`convert_s.cmp1`
`iio.a`
`test_convert.a`
	Files needed to complete the compiler and debugger tutorials

`build_iface.a`	routine to build interface code for any subroutine called from C or FORTRAN
`date.a`	Show date/time using `package CALENDAR` from `standard` library
`example_exer.a`	Example main program that uses memory allocation
`hanoi.a`	Tower of hanoi with screen-oriented display
`hello.a`	Prints Hello, world.
`mortgage.a`	Uses the `package MATH` from `verdixlib`
`permute_list.a`	Uses the `package ORDERING` from `verdixlib`
`phl.a`	A screen-oriented implementation of dining philosophers
`queens.a`	Solve the eight queens problems

`random.a`	Use `CALENDAR` from `standard` to generate random numbers
`README`	Guide to example programs

`.menu`
`slidedoc01`
`slidedoc02`
`slidedoc03`
`slidedoc04`
`slideshow.a`
	Show interface to `CURSES` in `publiclib`

`sort_file.a`	A tree-based sort program for text files
`sort_ints.a`	Sort files of integers using `TEXT_IO` and `ORDERING`
`termbody.a` `termspec.a`	Terminal interface, used by `HANOI`

`.cal`
`uc.1.man`
`uc.p`
`uctran.a`
	A calendar reminder program

`xview_examples`	Example programs that exercise a variety of XView packages.

1

SC Ada User Libraries — libraries created for source file compilations

Definition of an SC Ada Library

Often, a library is described as a file or directory that contains a set of specialized routines for a particular application or development system. While this is true for Ada, libraries have a more specialized meaning; they are directories that are initialized to contain subdirectories and files that are required by the development system before an Ada source file is compiled in that directory. Because they are normal directories, libraries contain any number of items besides Ada source code, and SC Ada files. The specialized files and directories that reside in an SC Ada library are shown in Figure 1-1.

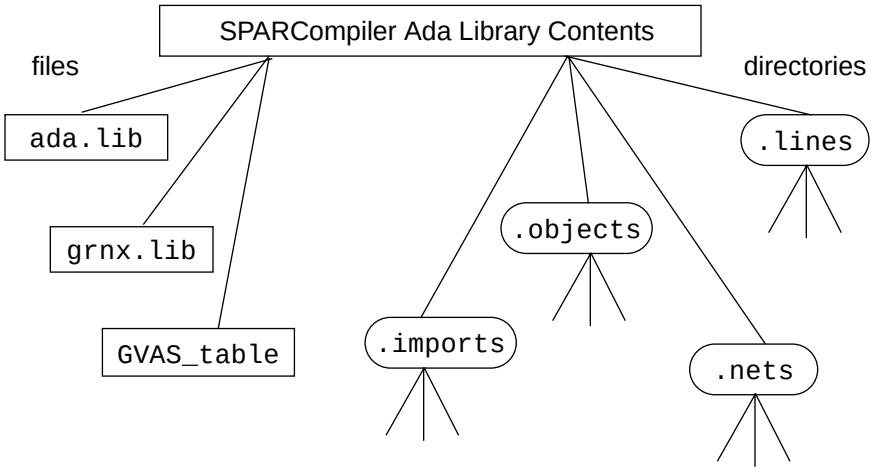

Figure 1-1 SPARCompiler Ada Library Contents

All Ada source file compilations occur in an SC Ada library. SC Ada includes a complete set of tools for creating, managing, and deleting libraries.

Every directory converted to an SC Ada library contains the `ada.lib`, `grnx.lib`, and `GVAS_table` files and the directories `.imports`, `.objects`, `.nets`, and `.lines`. Before you compile a program in a directory, run `a.mklib` on that directory, converting it to an SC Ada library and be sure these files and directories are present. If you attempt to use the compiler from a directory that is not an SC Ada library, the compiler fails.

The following sections describe each of the SC Ada library files and subdirectories.

References

SC Ada library management tools, *SPARCompiler Ada User's Guide*

1.1.8 `ada.lib` *File*

The `ada.lib` file is a user-modifiable file that contains file mapping information, library search path information, and linker directives. Use `a.cleanlib`, `a.info`, `a.mklib`, `a.mv`, `a.path`, `a.rm` or `a.rmlib` to modify the `ada.lib` file.

A description of the `ada.lib` file is useful in understanding compiler operation.

Figure 1-2 is a sample `ada.lib` file:

```
!ada library
ADAPATH= /ada_location/self/verdixlib
/ada_location/self/standard
HOST:INFO:host_name:
LIBRARY:LINK:/ada_location/self/standard/.objects/library_name:
 UNIX_HOST:DEFINE:BOOLEAN:TRUE:
hex 1F32ECD2:YNLPS#:hex01:
hex 1F32ECE4:XNLPB#:hex_b01:
low_level_io_2A0CF1E3:YNLPS#:lowlevel_io01:
low_level_io_2A0CF1DA:XNLPB#:lowlevel_io02:
```

Figure 1-2 Example of `ada.lib` File

The first line is an internal identification of the file and must not be changed.

The second line contains the library search list, an ordered list of predecessor libraries. Line length is restricted only by the line length limitation of the system. Multiple `ADAPATH` lines are allowed.

Subsequent lines in the `ada.lib` are of four types: `INFO` directives, `LINK` directives, `DEFINE` directives, `INCLUDE` directives, and compilation results. Each of these lines has at least three fields separated by colons and the line ends with a colon.

`INFO` directives have the word `INFO` in the second field, a name of some significance to the compiler and other SC Ada tools in the first field, and a value in the third field.

`LINK` directives have the word `LINK` in the second field. The first field is a name (having some specific meaning to the linker) or the word `WITH`*n* directing the linker to link the value in the third field (a filename or library) each time a program is linked using this library.

DEFINE directives have the word DEFINE as the second field.

INCLUDE directives have the word INCLUDE as the second field. The first field is the source filename. The third field is the include filename. This directive is the result of a successful compilation in a library when preprocessing is enabled.

Although we have just discussed the format of directives in the ada.lib file, you seldom need to change these directives. This information is presented to help you understand the role of directives. Typically, you create a master ada.lib file and reference that master file for your compilations.

Other lines in an ada.lib file are the result of a compilation (or attempted compilation) in the library and have this form:

```
    unit:type
time_created:ada_library_file:suffix_value:[pathname:]
```

For example, in this line from the ada.lib:

```
    low_level_io_2A0CF1DA:XNLPB#:lowlevel_io02:
```

low_level_io is the name of the compiled unit. XNLPB# is the type field explained below. 2A0CF1DA is an 8-digit hexadecimal number indicating the time the unit is created and lowlevel_io02 is the name of files created at compile time in the .nets, .lines, and .objects directories. The name of these files and the name of the unit correspond to the name of the source file, in this case, lowlevel_io.a. The example above has no *suffix_value* because the suffix of the source file matches the machine's default suffix (.a). If the extension does not match the default, it is listed here. The default suffix can be changed using the DEFAULT_SRC_EXT_INFO directive. The example above has no pathname because lowlevel_io.a is in the current directory. If the source is in a different directory, /rc/test for example, it is represented in the ada.lib as:

```
low_level_io_2A0CF1DA:XNLPB#:lowlevel_io02:/rc/test/lowlevel_io.a:
```

The ada.lib file contains one entry for each unit specification or body; unit is always recorded in lower case. The value field is a concatenation of the filename containing the unit and a 2-character sequence incremented for each Ada library entry from a particular file. The type field contains a combination of letters that categorize the unit. The manner in which the letters are used divides them into three groups. The groups and meanings are listed here. (a.ls presents this information in a more user-readable form.)

The `type` field can begin with zero or more of the following special characters:

`L`	Arises from `pragma LINK_WITH`
`V`	Indicates `INCLUDE` file dependence
`W`	Shadow body, dummy body for a spec which does not need a body
`X`	The package has elaboration code
`Y`	The unit cannot raise exceptions (no exception table is needed)
`Z`	Designates both `X` and `Y`
`D`	Defines an inline
`U`	Uncompiled (See NOTE below.)
`M`	Can be a ‘main' program
`T`	Has a task
`E`	`pragma ELABORATE`
`B`	No body needed
`R`	Unit contains the body of a generic

Note – ‘Normal' is a place holder that means “not generic, not an instantiation, and not shared.” An uncompiled unit (special character `U`) is entered into the `ada.lib` file when the compiler is invoked with the `-d` option. An uncompiled unit is entered into the `ada.lib` file when the `-f` option to `a.make` is used.

The last four characters of the type field contains one letter from each of the following four columns:

1	2	3	4
N normal	L library	S subprogram	S spec
G generic	S separate	P package	B body
I instantiation		T task	
S shared			

Note – All type entries are padded to a minimum length of 6 characters. The pad character is #.

Examples:

`MNLSS#`	Possible main program, normal library-level subprogram spec
`NLPB##`	Normal library-level package body
`UNSTB#`	Uncompiled normal separate task body
`GLPS##`	Generic library package spec

Entries for generic instantiations have a slightly different form giving the generic names and a sequential number concatenated into `INSTXX` in the *value* field. The following example illustrates a typical entry for an instantiation:

```
float_io$text_io$27:SLPB#:INST28XX:
```

This indicates a shared instantiation of `package FLOAT_IO` from `package TEXT_IO` rather than the generic body.

References

[ChapNumber,ChapNumberA]>-44
[ChapNumber,ChapNumberA]>-51
“a.make — recompile source files in dependency order” on page 2-60
[ChapNumber,ChapNumberA]>-45

1.1.9 GVAS_table *File*

The GVAS_table file keeps track of which virtual addresses are free. The attachment of a virtual address to each compiled unit enables the SC Ada compiler to reuse information about compiled units rapidly.

The GVAS (Global Virtual Address Space) holds the DIANA nets for your Ada code. DIANA is the data structure that SC Ada uses to represent the separate compilation information for each compiled unit. The files containing this information are known as DIANA nets or nets files. Each SC Ada library has its own GVAS. The GVAS is large but it is possible that all the GVAS for a library become allocated. If this happens, the compiler reports that the GVAS is exhausted and begins reusing previously allocated space. Compilation times can increase due to relocation of DIANA nets which are expensive. If this occurs and the message "GVAS exhausted" is displayed, use a.cleanlib to clear your GVAS. Note that you must recompile units in this library after running a.cleanlib.

gnrx.lib *File*

The gnrx.lib file contains a descriptor for each generic body, for each request for a generic instantiation, and for each actual generic instantiation. This file, like GVAS_table and ada.lib, is maintained and used by the compiler.

The file gnrx.lib is a binary file and is not readable.

.imports *Directory*

The files in the .imports of an Ada library can consume large amounts of disk space. The compiler and the tools can operate without the .imports directory being present.

The .imports directory is a DIANA net cache, which holds nets from other Ada libraries that you have withed in this library. For example, if your program says "with text_io;" then the compiler imports the net for TEXT_IO into the .imports directory.

Each Ada library has a Global Virtual Address Space (GVAS) assigned to it. As units are compiled, the net that is generated is put at a specific area in the appropriate GVAS. When a net is brought in from another library because of an Ada with statement, it is relocated into the local GVAS. This relocated net is put into the .imports directory so it does not need to be relocated each time units that with the net are recompiled, making recompilations faster.

If nets are assigned to the same GVAS address in the importing library, no new net copy is made. However, a small file is created reflecting the position of the net.

Each net in the `.imports` directory is slightly different from the original net in the parent library because they occupy different GVAS areas. When you remove nets from the `.imports` directory the compiler must re-import and relocate the nets when recompiling the units that `with` them. If you remove the `.imports` directory, the compiler must relocate the net every time it recompiles, which extends compilation time but is not burdensome normally.

If you are not compiling in your Ada libraries often, conserve disk space by cleaning the `.imports` directory.

`.objects` *Directory*

The `.objects` directory contains the object files generated for each Ada compilation unit in the library. The `.o` file is a special object module created by `a.ld` that contains information required for linking. The 01, 02, etc. files are object module files created for each compilation unit.

`.nets` *Directory*

The `.nets` directory contains files holding the separate compilation information for each compiled unit. SC Ada uses the DIANA intermediate representation for each unit and stores this representation in the directory `.nets`. These files are referred to as ‘net' files in this document.

References

[ChapNumber,ChapNumberA]>-4
“Overview of System Components,” *SPARCompiler Ada User’s Guide*

`.lines` *Directory*

The `.lines` directory contains line number reference files for use by the debugger and disassembler. This information appears only in the `.lines` files and is not present in the executable. The debugger and disassembler use these files to map address and lines in the source code. Files in the `.lines` directory are binary.

1.1.10 `.LINK_INFO` *and* `.MAKE_INFO` *Files*

Each invocation of `a.ld` (and later `a.make`) writes a file named `.LINK_INFO` (or `.MAKE_INFO`) in the SC Ada user library. These files contain summaries of the dependency analysis of the last link or make, as well as the elaboration order. This information is used in later links or makes to reduce prelink and make times.

“Men have become the tool of their tools.”

Thoreau

Command Reference 2

SC Ada includes the components listed. The reference manual pages that follow are arranged alphabetically,

`ada`	Invoke the Ada compiler
`a.app`	Invoke the Ada preprocessor
`a.ar`	Create an archive library of Ada objects
`a.cleanlib`	Reinitialize library directory
`a.cp`	Copy unit and library information
`a.das`	Disassemble object files
`a.db`	Debug Ada and C source code programs
`a.du`	Summarize disk usage for Ada libraries
`a.error`	Analyze and disperse error messages
`a.header`	Print the information stored in a unit 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 previously compiled units
`a.list`	Produce a source code listing
`a.ls`	List compiled units
`a.make`	Recompile source files in dependency order

(Continued)

`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 Ada unit from a library
`a.rmlib`	Remove a compilation 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 Ada
`a.view`	Provide aliases and history for a C shell user
`a.which`	Find a compiled unit
`a.xref`	Print cross-reference information for a given Ada unit or library

ada — *invoke the Ada compiler*

Syntax

ada [*options*] [*source_file*]... [*object_file*.o]...

Arguments

object_file.o

Non-Ada object filenames. These files are passed to the linker and are linked with the specified Ada object files.

options

Options to the compiler are:

-A

(disassemble) Disassemble the units in the source file after compiling them. Follow -A with arguments that further define the disassembly display (for example, -Aa, -Ab, -Ad, -Af, -Al, -As).

a

Add hexadecimal display of instruction bytes to disassembly listing

b

Disassemble the unit body [default]

d

Also print the data section (if present)

f

Use the alternative format for output

l

Put the disassembly output in file "*filename*.das"

s

Disassemble the unit spec

-a *filename*

(archive) Treat *filename* as an object archive file created by ar. Since some archive files end with .a, -a distinguishes archive files from Ada source files.

`-B`*static/dynamic*
(static) If *static* is indicated, the Ada program is compiled and linked statically. The default is *dynamic*.

Follow `-A` by arguments that further define the disassembly display (for example, `-Aa`, `-Ab`, `-Ad`, `-Af`, `-Al`, `-As`).) If *static* is indicated, the Ada program is compiled and linked statically. The default is *dynamic*.

`-c`
(no control) Suppress the control messages generated when `pragma PAGE` and/or `pragma LIST` are encountered.

`-D` *identifier type value*
(define) Define an identifier of a specified type and value.

`-d`
(dependencies) Analyze for dependencies only. Do not do semantic analysis or code generation. Update the library, marking any defined units as uncompiled. `a.make` uses the `-d` option to establish dependencies among new files. This option attempts to do imports for any units referenced from outer libraries. This reduces relocation and disk space usage.

`-E`
`-E` *directory*
(error output) Without a file or directory argument, `ada` processes error messages using `a.error` and directs a brief message to standard output; the raw error messages are left in *source_file.err*. If a file path name is given, the raw error messages are placed in that file. If a directory argument is given, the raw error output is placed in *dir/source*`.err`. Use the file of raw error messages as input to `a.error`. Use either `-e` or `-E` option.

`-e`
(error) Process compilation error messages using `a.error` and send it to the standard output. Only the source lines containing errors are listed. Only use one `-e` or `-E` option.

`-Ef`*error_file source_file*
(error) Process *source_file* and place any error messages in the file indicated by *error_file*. Note that no space is between the `-Ef` and *error_file*.

`-El`

`-El` *directory*

(error listing) Same as the `-E` option, except that a source listing with errors is produced.

`-el`

(error listing) Intersperse error messages among source lines and direct to standard output.

`-Elf`*error_file source_file*

(error listing) Same as the `-Ef` option, except that a source listing with errors is produced.

`-ev`

(error `vi(1)`) Process syntax error messages using `a.error`, embed them in the source file and call the environment editor `ERROR_EDITOR`. (If `ERROR_EDITOR` is defined, define the environment variable `ERROR_PATTERN`. `ERROR_PATTERN` is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specified, `vi(1)` is invoked.

The value of the environment variable `ERROR_TABS`, if set, is used instead of the default tab settings (8).

`-F`

(full `DIANA`) Do not trim the `DIANA` tree before output to net files. To save disk space, the `DIANA` tree is trimmed so that all pointers to nodes that did not involve a subtree that define a symbol table are nulled (unless those nodes are part of the body of an inline or generic or certain other values to be retained for the debugging or compilation information). Generally, the trimming removes initial values of variables and all statements.

`-G` (GVAS)

Display suggested values for the `MIN_GVAS_ADDR` and `MAX_GVAS_ADDR` `INFO` directives.

`-K`

(keep) Keep the intermediate language (`IL`) file produced by the compiler front end. The `IL` file is placed in the `.objects` directory, with the filename *unit_name.i*.

`-L` *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

`-l`*file_abbreviation*
(library search) This is an option passed to the Solaris 2.1 linker, `ld(1)` telling it to search the specified library file. Do not use a space between the `-l` and the file abbreviation.

`-M` *unit_name*
(main) Produce an executable program by linking the named unit as the main program. *unit_name* must be compiled already. It must be either a parameterless procedure or a parameterless function returning an integer. The executable program is named `a.out` unless overridden with the `-o` option.

`-M` *source_file*
(main) Produce an executable program by compiling and linking *source_file*. The main unit of the program is assumed to be the root name of the `.a` file (for `foo.a` the unit is `foo`). Only precede one `.a` file by `-M`. The executable program is named `a.out` unless overridden with the `-o` option.

`-N`
(no code sharing) Compile all generic instantiations without sharing code for their bodies. This option overrides the `SHARE_BODY INFO` directive and the `SHARE_CODE` or `SHARE_BODY` pragmas.

`-O[0-9]`
(optimize) Invoke the code optimizer. An optional digit (no space is before the digit) provides the level of optimization. The default is `-O4`.

Note – Each level of optimization is cumulative. For example, optimization level 4 (`O4`) incorporates all the features of optimization levels 1 through 3, plus hoisting invariants from loops and address optimizations.

`-O`
Full optimization.

`-O0`
No optimization.

`-O1`
Copy propagation, constant folding, removing dead variables, subsuming moves between scalar variables.

`-O2`
Add common subexpression elimination in basic blocks.

`-O3`
Add global common subexpression elimination.

`-O4`
Add hoisting invariants from loops and address optimizations.

`-O5`
Add range optimizations, instruction scheduling, and one pass of reducing induction expressions.

`-O6`
No change.

`-O7`
Add one more pass of induction expression reduction.

`-O8`
Add one more pass of induction expression reduction.

`-O9`
Add one more pass of induction expression reduction and add hoisting expressions common to the `then` and the `else` parts of `if` statements.

Hoisting from branches (and cases alternatives) can be slow and does not always provide significant performance gains so it can be suppressed.

`-o` *executable_file*
(output) This option is used in conjunction with the `-M` option. *executable_file* is the name of the executable rather than the default, `a.out`.

`-P`
Invoke the Ada preprocessor.

`-R` *ada_library*
(recompile instantiation) Force analysis of all generic instantiations, causing reinstantiation of any that are out- of-date.

`-S`
(suppress) Apply `pragma SUPPRESS` to the entire compilation for all suppressible checks.

`-sh`
(show) Display the name of the tool executable but do not execute it.

`-T`
(timing) Print timing information for the compilation.

`-v`
(verbose) Print compiler version number, date, and time of compilation, name of file compiled, command input line, total compilation time, and error summary line. Disk usage information about the object file is provided. With `OPTIM` the output format of compression (the size of optimized instructions) is as a percentage of input (unoptimized instructions).

`-w`
(warnings) Suppress warning diagnostics.

source_file
Name of the source file to compile.

Description

The `ada` command executes the Ada compiler and compiles the named Ada source file. The file must reside in an SC Ada library directory. The `ada.lib` file in this directory is modified after each Ada unit is compiled.

By default, `ada` produces only object and net files. If the `-M` option is used, the compiler invokes `a.ld` automatically and builds a complete program with the named library unit as the main program.

The compiler generates object files compatible with the host linker.

Give non-Ada object files (`.o` files produced by a compiler for another language) as arguments to `ada`. These files are passed to the linker and linked with the specified Ada object files.

Specify command line options in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

Several SC Ada compilers can be simultaneously available on a single system. Because the `ada` command in any *ada_location*`/bin` on a system executes the correct compiler components based upon visible library directives, the option `-sh` option prints the name of the components actually executed.

`a.db` or `a.das` generate program listings with a disassembly of machine code instructions.

Note – If two files of the same name from different directories are compiled in the same Ada library using the `-L` option (even if the contents and unit names are different), the second compilation overwrites the first. For example: The compilation of

```
/usr2/directory2/foo.a -L /usr2/ada_2.1/test
```

overwrites the compilation of

```
/usr2/directory1/foo.a -L /usr2/ada_2.1/test
```

in SC Ada library `/usr2/ada_2.1/test`.

Diagnostics

The diagnostics produced by the SC Ada compiler are intended to be self-explanatory. Most refer to the *Ada Reference Manual*. Each Reference Manual reference includes a section number and, optionally, a paragraph number enclosed in parentheses.

References

“a.app — preprocess Ada source” on page 2-11
“a.das — disassemble object files” on page 2-20
“a.db — debug Ada and C source code programs” on page 2-23
“a.error — analyze and disperse error messages” on page 2-31
“a.ld — build an executable program from previously compiled units” on page 2-53
“a.info — list or change SC Ada library directives” on page 2-42
“a.make — recompile source files in dependency order” on page 2-60
“Syntax” on page 2-66
optimizations, *SPARCompiler Ada User’s Guide*
Ada preprocessor, `pragma OPTIMIZE_CODE`, and
suppress checks (`pragma` suppress), *SPARCompiler Ada Programmer’s Guide*
`ld(1)`, *Solaris Developer Documentation*

a.app — *preprocess Ada source*

Syntax

```
a.app [options] [in_file [out_file]]
```

Arguments

in_file
Name of the Ada source file to preprocess.

options
Options to the a.app command are:

-D *identifier type value*
(define) Define an identifier of a specified type and value.

-L *library_name*
(Library) Operate in SC Ada library *library_name* (the current working directory is the default).

-s
(strip) Strip control and inactive lines from the output source.

-w
(warnings) Suppress warning diagnostics.

out_file
Name of the output file.

Except for the -s (strip) option, which is available only when invoking a.app directly, these options are recognized by ada.

Description

Invoke the Ada preprocessor by either including the `-P` option on the command line of the invocation of `ada`, `a.make`, and `a.tags`, or by including the `APP INFO` directive in the `ada.lib`.

The syntax of the `INFO` directive is:

`APP:INFO:`*boolean_value*`:`

If *boolean_value* is set to `TRUE`, the compiler invokes `a.app` automatically before compiling the source; any other value for *boolean_value* has no effect.

The `-P` option to `ada` takes precedence over the `APP INFO` directive.

When the compiler invokes `a.app`, it creates a temporary output file and discards it at the end of the compile. All diagnostics are in reference to the original input file. If an error is encountered, then *out_file* is not created.

If no files are specified on the command line, they default to standard input and standard output.

References

“a.info — list or change SC Ada library directives” on page 2-42
preprocessing Ada programs, *SPARCompiler Ada Programmer’s Guide*.

a.ar — *create an archive library of Ada object files*

Syntax

```
a.ar [options] [-L library_name] archive_name [unit_name]
    [-O object_list]
```

Arguments

archive_name
Name of the archive library to be created.

options
Options to a.ar are:

-A
(all) Include in the archive objects from all libraries on the ADAPATH.

-f
(force) Create the archive even if some units are out-of-date or need elaboration.

-L *library_name*
(library) Find the specified unit in library *library_name* (or use all units in library *library_name* if no *unit_name* is specified).

-O *object_list*
(objects) Add the objects in *object_list* to the archive.

-s
(suppress) Suppress error and warning messages regarding units that are out-of-date or need elaboration. It must be used in conjunction with -f.

-sh
(show) Display the name of the tool executable to do not execute it.

-V
(verify) Print the archiving commands without executing them (the archive is not created).

-v
(verbose) Print the archiving commands prior to executing them.

unit_name
Name of an Ada unit.

Description

When used below, “closure” is defined as the set of units that contains:

- Named unit specification
- All units named in the context clause of the specified unit
- Its parent, if the specified unit is a subunit
- Bodies for all units in the closure
- Subunits for all units in the closure, if any exist
- Instantiations created by all units in the closure, if any exist
- All units named in the context clause of all units in the closure

If a unit name is specified, `a.ar` creates an archive library of all objects corresponding to the units in the closure of unit *unit_name*. By default, only those objects in the current SC Ada library (i.e., the current working directory or the library specified by `-L` *library_name*) are added to the archive. If an object is in the closure but is located in a library that is on the current library `ADAPATH`, it is added to the archive only if the `-A` option is specified.

If no *unit_name* is specified, all objects in the current SC Ada library are included in the archive. Again, objects that are in the closure but do not reside in the current library are included in the archive only if the `-A` option is specified.

`a.ar` invokes the UNIX tool `ar` to create the archive library.

Note that elaboration code is not executed for Ada units that are extracted from an archive library. Thus if a unit requiring elaboration is linked in this manner, the elaboration is not performed and the program may be erroneous. By default, `a.ar` generates an error message if any object to be added to the archive requires elaboration, and the archive is not created. The `-f` option forces `a.ar` to create the archive, including in it the object requiring elaboration However an error message generates unless the `-s` option, which suppresses error messages, is used.

If any units whose objects are to be included in the archive are determined to be out-of-date, an error message generates and the archive is not built. Again, `-f` forces `a.ar` to create the archive, including these out-of-date units (and `-s` suppresses any error messages).

Be careful when using the `-f` option to force into the archive units that are out-of-date or needing elaboration. Any such erroneous unit linked in your program can make your program erroneous.

The `-O` option adds objects to the archive that not normally included by `a.ar` (e.g. non-Ada object files). This option must come after the archive and unit names on the command line.

2

a.cleanlib — *reinitialize library directory*

Syntax

```
a.cleanlib [options] [ada_library]
```

Arguments

ada_library
Name of the library in which a.cleanlib is to operate. If no library is specified, the current working directory is assumed.

options
Options to the a.cleanlib command are:

-c
(check) Remove all erroneous libraries from ADAPATH [default].

-F
(force name) Allow the cleaning of an SC Ada library having a reserved name.

-f
(force) Clean the SC Ada library structure even if components are missing or if lock files exist.

-v
(verbose) Report libraries removed. The -v option provides output only when used with -c.

Description

The command empties the files ada.lib, gnrx.lib, and GVAS_table of all separate compilation information and removes the contents of the directories .imports, .nets, .lines, and .objects from the named library or, if no library is specified, from the current library directory. a.cleanlib preserves all library directives in the ada.lib.

a.cleanlib preserves all non-compilation information contained in ada.lib, including the library search list and any directives.

If a.cleanlib cannot find every library component, it aborts without removing any information unless the -f (force) option is given.

The `-F` option allows `a.cleanlib` to clean a library having a reserved name (`standard`, `verdixlib`, `publiclib`).

Files

`ada.lib`	Library reference file
`gnrx.lib`	Generic instantiation reference file
`GVAS.lock, gnrx.lock`	Lock the library while reading or writing special library files
`GVAS_table`	Address assignment file
`.imports`	Imported Ada units directory
`.lines`	Line number reference files directory
`.nets`	SC Ada `DIANA` net files directory
`.objects`	SC Ada (global) object files directory

References

“Syntax” on page 2-66
“a.rm — remove an Ada unit from a library” on page 2-83

2

a.cp — *copy unit and library information*

Syntax

```
a.cp unit_name [, ...] [options] target_directory
a.cp source_file [, ...] [options] target_directory
```

Arguments

options
Options to the a.cp command are:

-b
(body) Copy the bodies of the named units.

-F
(force name) Enable copy of units to protected libraries, that is, standard, verdixlib, publiclib.

-f
(force) Do not report matching errors if unit name is not found.

-L *library_name*
(library) Copy from SC Ada library *library_name* (the current working directory is the default).

-s
(spec) Copy the compilation information for the specifications of the named units.

-u
(unit) Force the next name to be treated as a unit even if it contains a period (.).

-V
(verify) List the units to copy but do not copy them.

-v
(verbose) List the units as they are copied.

source_file
Name of an Ada source file.

target_directory
Directory to which the unit and library information is copied. This directory must be an SC Ada library.

unit_name
Name of an Ada unit or subunit. Unit names with dotted notation such as `aaa.bbb` or `aaa.bbb.ccc` are interpreted as the names of Ada source files unless the `-u` option is specified.

Description

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

When *source_file* is specified, the corresponding files in `.nets`, `.lines`, and `.objects` are copied for each unit defined in *source_file*, and the appropriate entries are created in the `ada.lib` file in the target directory.

A variety of options copy specifications and bodies separately. The `-u` (unit) option enables references to units whose names contain a period (.). Without the `-u` option, a name containing a period is treated as a source filename.

You can specify *unit_name* and *source_file* with regular expressions. For example, `a.cp "f*"` copies all units beginning with the letter "`f`." The command, `a.cp "f*.a"` copies units in source files that begin with the letter "f."

References

"a.ls — list compiled units" on page 2-58
"a.mv — move unit and library information" on page 2-69

2

a.das — *disassemble object files*

Syntax

```
a.das [options]unit_name
a.das -E value -n object_file
```

Arguments

object_file
Name of object file to disassemble. Use this argument with the `-n` option.

options
Options to the `a.das` command are:

`-A`
(assembly) Output the entire source file with the assembly listing.

`-a`
(all) Add hexadecimal display of instruction bytes to disassembly listing.

`-b`
(body) Disassemble the indicated subprogram body. [default]

`-d` [*format*]
(data) Print the data section (if present). Follow `-d` by arguments that indicate the output format, one from each of the following two groups:

B or b in bytes
W or w in words
L or l in longwords

x hexadecimal output
d decimal output

With no arguments, the `-d` option defaults to `-dWx`. No space is required between the `-d` and the qualifying letters.

`-E` *value*
Specify the endian value for targets that support both big and little endian. *value* is `b` for big endian and `l` for little endian. Default: [b].

`-f`
(format) Specify an alternative format for output. Use of the `-f` option takes the tabs out of the disassembly output. The default has the disassembly tabbed in, so the source lines are closer to the left margin.

`-i`
(instructions) Print the number of machine code instructions generated for each line of source code. `-i` can be followed by arguments that indicate the output format:

- `d` Do not print the disassembly, just print the instruction count.
- `n` Order the output by source line number. The default orders the output by instruction count.

`-L` *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

`-n` *object_file*
(no source) Disassemble object files. No SC Ada library files are required.

`-p`
(profiling) Read the `mon.list` file (generated by `a.prof -d`) and insert source line execution percentages in listings.

`-S` *source_file*
(source) Disassemble all units in the named source file.

`-s`
(spec) Disassemble the indicated subprogram specification.

`-sh`
(show) Display the name of the tool executable but do not execute it.

unit_name
Ada unit name. If `a.das` is not run from within the SC Ada library containing the unit, the `-L` command must be used to name the library where the unit is held.

Description

`a.das` is an object module disassembler that interleaves Ada source lines and assembler instructions. Unlike disassembling from within `a.db`, it needs no target hardware to operate. `a.das` disassembles any Ada unit except those containing generics. Without the `-s` or `-b` option, it disassembles the given unit body.

`a.das` requires only the Ada *unit_name* when run from the SC Ada library containing the unit.

`a.das` disassembles object files produced by the compiler when it is invoked using the `-n` option and naming the file directly.

List the data section (if present) in various forms with the `-d` option.

ada_location`/bin/a.das` is a wrapper program that executes the correct executable based upon directives visible in the `ada.lib` file. This enables multiple SC Ada compilers to exist on the same host. The `-sh` option prints the name of the actual executable file.

References

“Instruction and Source Submodes” on page 3-134

a.db — *debug Ada and C source code programs*

Syntax

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

Arguments

a.db_options
Options to the a.db command are:

-a *PID*
Invoke the debugger on the currently executing process (*PID*). The debugger *does not* join the process group of that process. Use ps or jobs to get the *PID*.

-ag *PID*
Invoke the debugger on the currently executing process (*PID*). The debugger *does* join the process group of that process. Use ps or jobs to get the *PID*. This enables Control-c.

-c
Debug C programs. This option avoids error messages relating to missing Ada libraries.

-e *entry_point*
(entry) Specify the program entry point. This option starts a program at an address different than the default starting address.

-I *argument_list*
(interface) Pass arguments defined in *argument_list* down when the debugger interface process is invoked.

-i *filename*
(input) Read input from the specified file.

-L *library_name*
(library) Read program compilation information from the specified library, rather than the current directory, as though you were operating in the specified library. This option is for debugging Ada programs only.

-N *target_node*
(node) Specify target node name to supercede the VADS_TARGET_NAME environment variable.

`-r` *"executable_file* [*executable_file_options*]*"*
(run) Initialize `set run` with *executable_file* and *executable_file_options.*

`-sh`
(show) Display the name of the debugger executable but do not execute. This option is useful if multiple version of SC Ada are on a system.

`-t` *filename*
(terminal) Read terminal state from a file. Used only when the debugger is being run in the background.

When the debugger runs in the background, it cannot reliably get the state of the controlling terminal, as that state changes as you run other programs in the foreground. However, the output of the program being debugged depends on the set up of your terminal. To ensure that the output is displayed in a consistent way, we provide the program `tty_state` in *ada_location*`/sup/diag`. `tty_state` must be run in the foreground and it dumps the terminal state to a file. Invoke this program as follows:

```
tty_state -f filename -w
```

Supply that same filename to the debugger with the `-t` option. Note that you can print the `tty` state that is written to filename by typing:

```
tty_state -f filename -r
```

`-v`
(visual) Invoke the screen-mode debugger directly.

executable_file
Name of file to execute and debug. If *executable_file* is not specified, the debugger searches for `a.out`. If only a root filename is given (`foo`, as opposed to `/vc/sbq/foo`), the debugger searches the directories on the `PATH` environment (exported) variable for an executable file `foo` just as the shell does. Note that if "." is not on your `PATH`, you must enter `a.db ./foo`.

executable_file_options
Command line options that pertain to the *executable_file* being debugged. All command line options that follow the name of the executable are assumed to belong to the program being debugged.

References

command file input, display debugger executable, executable file, invoking the debugger, and screen mode, *SPARCompiler Ada User's Guide*
"screen mode — screen-oriented debugger interface" on page 3-132

Description

`a.db` is a symbolic debugger for SC Ada and for C programs. On the Solaris 2.1 operating system, C programs must be compiled with both the `-g` option and the `-xs` option to be compatible with the SC Ada debugger.

Specify invocation options to the debugged program on debugger invocation.

Note – All command line options that follow the name of the executable are assumed to belong to the program being debugged.

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. The `-r` option initializes `set run` to a string made up of the *executable_file* and the *executable_file_options* enclosed in quotes. Enclosing shell commands that pertain to the executable file within the quotes, output redirection for example, ensures that they are not interpreted by the shell to apply to the debugger itself.

Any single unit or token on the command line can be up to 511 characters long.

Detailed descriptions of interactive `a.db` commands are provided in this reference, which is available also online using `a.help` or the debugger internal `help` command.

Use the `quit` command to leave the debugger and return to the shell.

References

debugging C programs, *SPARCompiler Ada User's Guide*

Invocation File

In addition to the invocation line, supply parameters to `a.db` using a `.dbrc` file. During debugger initialization, `a.db` checks for `./.dbrc`. If that does not exist, it checks for `$HOME/.dbrc`. The `.dbrc` file contains only `set` commands. These commands execute before any other commands, including those in the input file specified on the command line but not before command line options.

A `set source` command in the `.dbrc` file can specify the location of an `ada.lib` for the debugging session other than the default `ada.lib`. If a `set source` command is present, the debugger searches the directories specified in the `set source` command for the first directory that contains an `ada.lib`. The debugger uses that directory to obtain the `DIANA` net files and the line number files produced by the compiler.

References

“set — set debugger parameters” on page 3-142

Start-up Environment

The debugger establishes the debugging environment when it starts up. Certain key parameters are displayed on the screen to verify what and where it is debugging. See Figure 2-1.

```
% a.db /vc/wheels/atst/phl
Debugging: /vc/atst/phl
ada_library: /vc/atst
library search list:/vc/wheels/atst
   /usr2/ada_2.1/self/verdixlib
   /usr2/ada_2.1/self/standard
   /vc/install/build/tasking
>
```

Figure 2-1 Example of Debugger Start-up Environment

The first line of the example is the invocation of the debugger on the file `phl`, the dining philosophers program copied from the `examples` directory and compiled.

The first line of output shows the full path and name of the program being debugged. `set source` is initialized to this path automatically. `set run` is initialized to this path with the executable name automatically. This facilitates subsequent invocations of the debugger on this executable file. Any options that follow the executable filename are assumed to be for the executable and are sent to `set run` unless the `-r` option is used.

The *ada_library* is the name of the SC Ada library directory used for this debugging session.

The library search list is derived from the `ada.lib` file in the *ada_library*. It shows the Ada library directories that are searched when the debugger looks for an Ada unit. The search list prints in the same order that the debugger searches it.

The last line, "`>`," is the debugger prompt.

References

Ada library directory, *SPARCompiler Ada User's Guide*

Redirecting Program and Debugger Input/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 exist to using redirection:

- 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 is:

```
set input debug.in
load
r
quit
```

Run the debugger in 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 &
```

References

redirecting input/output *SPARCompiler Ada User's Guide*

Files

`ada.lib`	Library reference file
`gnrx.lib`	Generic instantiation reference file
`GVAS.lock, gnrx.lock`	Lock the library while reading or writing special library files
`GVAS_table`	Address assignment file
`.imports`	Imported Ada units directory
`.lines`	Line number reference files directory
`.nets`	Ada network control files directory
`.objects`	Ada object files directory

a.du — *summarize disk usage for Ada libraries*

Syntax

```
a.du [options] [ada_library, ...]
```

Arguments

ada_library
Name(s) of the SC Ada library in which to operate. [Default: current working directory]

options
Options to the a.du command are:

`-i`
(imports) Display only information for imported units.

`-f`
(force) Display information even if the library is incomplete.

`-v`
(verbose) Display information in verbose mode.

Description

a.du lists the size in bytes for all compiler-generated files in the specified SC Ada libraries. If no library is specified, the current directory is assumed.

Default output is in six columns without headers:

1. Size of files in .nets directory
2. Size of files in .objects directory
3. Size of files in .lines directory
4. Spec or body
5. Unit name
6. Source file (if any)

The -v option prints headers and additional information.

ada_location/bin/a.du is a wrapper program that executes the correct executable based upon directives visible in the ada.lib file. This enables multiple SC Ada compilers to exist on the same host.

Files

`GVAS_table`	Address assignment file
`.imports`	Imported Ada units directory
`.lines`	Line number reference files directory
`.nets`	Ada network control files directory
`.objects`	Ada object files directory

a.error — *analyze and disperse error messages*

Syntax

```
a.error [options] [error_file]
```

Arguments

error_file
Name of error file to analyze. This error file is generated by invoking the ada or a.make command with the -E *error_file* option.

options
Options to the a.error command are:

-e *editor*
(editor) Insert the error messages in the source file and invoke the specified editor.

-f
(force) Force a listing even if pragma LIST(OFF) is encountered.

-l
(listing) Produce a listing on the standard output.

-n
(no) Do not display line numbers.

-s
(short) Display only the error messages and the lines associated with them.

-t*number*
(tabs) Change the tab settings, overriding the value of the environment variable ERROR_TABS, if it is set. A default tab setting of 8 is applied if neither ERROR_TABS nor the -t option is used.

-V
(validation) Do not change formfeeds to a two character representation of '^''L'. If *error_file* is not produced by the SC Ada compiler, the output is similar to that of a.list.

`-v`
(`vi`) Embed error messages in the source file and call the environment editor `ERROR_EDITOR`. (If `ERROR_EDITOR` is defined, define the environment variable `ERROR_PATTERN`. `ERROR_PATTERN` is an editor search command that locates the first occurrence of '`###`' in the error file.) If no editor is specified, `vi(1)` is called.

`-w`
(warnings) Ignore warnings.

Description

Generally, `a.error` is called from the `ada` command, but use it separately. `a.error` analyzes and optionally disperses diagnostic error messages produced by the SC Ada compiler. It looks at the specified error file or standard input, determines the source file and line number to which the error refers, determines whether the error is ignored or not, and outputs the associated source line followed by the error line(s).

`a.error` inserts the error lines into the source file and invokes the `vi(1)` editor if the `-v` option is given. Error lines placed into files this way are of two types. The first gives the position of the error and the second identifies it. Multiple errors on a single line are referenced by sequential alphabetic characters. See Figure 2-2.

```
     subtype T is range 1..1f;
-----------------^A                              ###
---------------------------^B                    ###
--### A: syntax error: “identifier” inserted
--### B: lexical error: deleted
```

Figure 2-2 Example of `a.error` Output

Because all error lines are flagged with ###, use the `vi(1)` editor command `:g/###/d` to delete them. However, any source lines containing ### are deleted also; consequently, do not use ### in any source with which `a.error -v` may be used.

In the case of source files with multiple links, `a.error` creates a new copy of the file with only one link to it.

Diagnostics

`a.error` produces diagnostics indicating `no errors` if `-v` is used and no errors are detected, and `no such file or directory` if invoked with an invalid filename.

References

“ada — invoke the Ada compiler” on page 2-3
“a.make — recompile source files in dependency order” on page 2-60

2

a.header — *print the information stored in a unit net header*

Syntax

a.header [*options*] *unit_name*|*net_file_name* [-L *library_name*]

Arguments

net_file_name
Name of the net file to be used.

options
Options to the a.header command are:

-addr
Print the base GVAS address.

-all
Print all information in the net header. [default]

-b
Print the information in the unit body net. [default]

-body
Same as -b.

-cmdline
Print the command line options the unit was compiled with.

-copyright
Print the copyright.

-define
Print the DEFINE directives visible when the unit was compiled.

-deps
Print the list of dependencies for the unit.

-gvas_timestamp
Print the date and time of the GVAS table at the time the unit was compiled.

-info
Print the INFO directives visible when the unit was compiled.

`-l`
Print additional information, where applicable.

`-long`
Same as `-l`.

`-L` *library_name*
Find the specified unit in library *library_name* (or all units in *library_name* if no net or unit name is given).

`-net`
Same as `-n`.

`-options`
Print the command-line options and the visible `INFO` and `DEFINE` directives the unit was compiled with. (`-options = -cmdline -define -info`)

`-priority`
Print the priority of the unit, if applicable.

`-single`
Print the information for each dependency on a single line.

`-size`
Print the size of the `DIANA` net for the unit.

`-s`
Print the information in the unit spec net.

`-spec`
Same as `-s`.

`-sh`
Print the name of the tool executable but do not execute it.

`-timestamp`
Print the date and time the unit was compiled.

`-type`
Print the string representation for the unit type.

`-v`
Print labels for the various pieces of information.

`-verbose`
Same as `-v`.

`-version`
Print the net version number for the unit net file.

unit_name
Name of Ada unit.

Description

The information listed below is stored in the net file corresponding to a unit. `a.header` prints the following information in the order listed:

`type`
The unit type. The string that categorizes a unit is printed. This string corresponds to the type field in the `ada.lib` entry for that unit.

The `-l/-long` option prints, in parenthesis, an encrypted integer value that corresponds to the type string, following the string.

`timestamp`
The date and time the unit was compiled. The 26-character string corresponding to the date and time, as produced by the UNIX `ctime(3v)` routine, is printed.

The `-l/-long` option prints, in parenthesis, the long integer representation for the date/time, following the string.

copyright
The copyright message.

net version number
The internal net version number.

`GVAS` timestamp
The date and time of the `GVAS_table` file at the time the unit was compiled. The 26-character string corresponding to the date and time, as produced by the UNIX `ctime(3v)` routine, is printed.

The `-l/-long` option prints, in parenthesis, the integer representation for the date/time, following the string.

priority
: If a unit is a main program (as indicated by the unit type), its priority is stored in the net. If this information is in the net, its integer value is printed. If the information is not in the net, nothing is printed.

compilation option size
: The number of characters in the string(s) that contain the command line options, visible `INFO` directives and visible `DEFINE` directives the unit was compiled with.

command line options
: The command line options the unit was compiled with. All command line arguments are printed on a single line, separated by a single space.

`INFO` directives
: The list of `INFO` directives visible at the time the unit was compiled. Each `INFO` directive, of the form `name:INFO:value:` is printed on its own line.

`DEFINE` directives
: The list of `DEFINE` directives visible at the time the unit was compiled. Each `DEFINE` directive, of the form `name:DEFINE:type:value` is printed on its own line.

number of dependencies
: The number of units the specified unit is dependent upon.

dependency list
: Information regarding each of the units this unit depends on. The default is to print only the names of these units, with each unit name on its own line. The `-l/-long` option causes the following information regarding the units in the dependency list to be printed as well. Each field, or piece of information, is printed on its own line. After the direct dependency field, a blank line is printed to easily group the information regarding each dependency.

`type`
: The unit type. The string that categorizes a unit, and an encrypted integer value (in parenthesis), are printed. The string corresponds to the type field in the `ada.lib` entry for that unit.

`timestamp`
The date and time the unit was compiled. The 26-character string corresponding to the date and time, as produced by the UNIX `ctime(3v)` routine, and the integer representation for that date/time (in parenthesis) is printed.

base `GVAS` address
The base address the unit net file has been assigned in `GVAS`. A hexadecimal value, preceded by a `0x`, is printed.

direct dependency
A boolean value indicating whether or not the specified unit is directly dependent upon this unit. If the unit is on the specified unit `WITH` list, it is a direct dependency, and the value `TRUE` is printed. Otherwise, it is an indirect dependency and `FALSE` is printed.

base `GVAS` address
The base address the unit net file has been assigned in `GVAS`. A hexadecimal value, preceded by a `0x`, is printed.

size of the `DIANA` net
The size, in bytes, of the unit `DIANA` net. A hexadecimal value, preceded by a `0x`, is printed.

By default, a "short" version of all information in the net header is printed, with no label to indicate the meaning of the information.

The `-l/-long` option prints some additional information, where applicable, as described above.

The `-v/-verbose` option prints labels preceding each piece of information.

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 the order as listed above and cannot be changed. For example, the command

```
% a.header -timestamp -type
```

always gives the type followed by the timestamp. The order of the options does not affect the order of the output.

Note that when reading/interpreting the output from `a.header`, it is necessary to know whether or not the output contains the priority information (unless, of course, you are not using the `-all` option (the default), and did not specify `-priority`). You can do this in two ways:

- The unit type string indicates that it is a main unit by the presence of an `M` in the string. If the type string contains an `M`, it is a main unit and the priority information is contained in the output.
- Invoke `a.header -priority` *unit_name*. If no output exists from this command, the output being read does not contain the priority information.

It is possible to give a specific net file name instead of a unit name. The `-n` option indicates that the name given is a net file name and not a unit name.

If neither a unit name nor a net file name have been specified, and `-L` *library_name* is given, the net header information for all units in the library *library_name* is given. For each unit, the information as described above is preceded by the unit and source file name and followed by the string `##########`, e.g.,

```
unit_name1 (source_file1.a)
net header info
##########
unit_name2 (source_file2.a)
net_header_info
##########
```

ada_location`/bin/a.header` is a wrapper program that executes the correct executable based on directives visible in the `ada.lib` file. Therefore multiple SC Ada compilers can exist on the same host. The `-sh` option prints the name of the executable file.

References

“ada.lib File” on page 1-24

2

a.help — *invoke the interactive help utility*

Syntax

a.help [*options*] [*subject*]

Arguments

options

Options to the a.help command are:

-p *pager*

(pager) Use *pager* as the paging program. The complete path name must be given with surrounding quotes if additional options to the paging program are desired.

-sh

(show) Display the name of the tool executable but do not execute it.

subject

Name of subject for which help information is displayed. To display a list of the subjects for which help is available, enter:

```
% a.help vads_intro
```

Description

a.help provides on-line help for each of the SC Ada utilities and for debugger commands and concepts.

Reference manual entries *for the compiler and tools* are available on-line by using the man command if the local system administrator has installed them. Obtain a list of topics by typing:

man ada

Obtain an entry for a specific command with:

man *ada_command*

Without a specified subject, a.help provides information on use of the help utility and prompts for additional subject names. Use q to exit from a.help.

Without the `-p` option, `a.help` uses the paging program defined by the environment variable PAGER, requiring the full path name, including surrounding quotes, for additional options. If PAGER is not defined, the default is used.

ada_location`/bin/a.help` is a wrapper program that executes the correct executable based upon directives visible in the `ada.lib` file. This enables multiple SC Ada compilers to exist on the same host. The `-sh` option prints the name of the actual executable file.

On-Line Help from the Debugger

Access on-line help for the debugger as well as the compiler and tools during a debugging session by typing:

`help` [*subject*]

or, while in screen mode:

:**`help`** [*subject*]

If the subject is omitted, a list of debugger commands is displayed. Obtain this overview by typing `intro` after a help prompt. Get help with the `help` command by typing `help` at a help prompt.

Files

ada_location`/sup/help_files/*`

References

on-line help in the debugger, *SPARCompiler Ada User's Guide*

2

a.info — *list or change SC Ada library directives*

Syntax

```
a.info [options]
```

Arguments

options

Options to the a.info command are:

-a *name value*

(add) Add the INFO or LINK directive *name* with the specified *value*. When used with the -D option, the -a option requires an additional field:

```
a.info -D -a name var_type value.
```

-A[ll]

(all) Search all directories on the path.

-D

Operate on a DEFINE directive. The -D option must appear before a -a, -d, -r or -q option on the command line.

-d *name [value]*

(delete) Delete the INFO or LINK directive *name* with the specified *value*. If no *value* is specified, all directives of that *name* are deleted.

-F

(force) Override protection of named libraries to allow changes to directives.

-h [*directive_name*]

(help) Display help information about valid directives(s). Entering -h with no parameters displays information about all directives. Entering the name of a single directive displays information about that directive. Note that you can also use wildcards in *directive_name* (e.g., foo*).

-f

Operate in silent mode.

-I

(invariant) List the invariant directives in the library path specified. a.info disallows the replacing, adding, setting or deleting of invariant directives. Invariant directives are those directives whose values must not be changed.

`-i`
(interactive) Operate in interactive mode.

`-L` *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

`-l`
(list) Display visible directives and their corresponding values. [default]

`-q` *name* [*value*]
(query) This boolean function returns `TRUE` if the directive *name* exists. If *value* is specified, `TRUE` is returned if the directive *name* with *value* exists.

`-r` *name value*
(replace) Replace the `INFO` or `LINK` directive *name* with the specified *value.*

`-s` *name value*
(set) Combine the `-q` and `-a` or the `-q` and `-r` options; that is, query directive *name,* and if it is found, replace it with *name* and *value* or query directive *name,* and if it is not found, add it with *value.*

`-v`
(verbose) Display maximum information for visible and hidden directives.

Description

`a.info` is used to examine, add, delete, replace, and query `INFO`, `LINK`, and `DEFINE` directives and their values.

An `INFO` directive is an entry in the `ada.lib` file that provides information to the compiler and SC Ada tools regarding the characteristics of the release and the type of code generated.

A `LINK` directive is an entry in the `ada.lib` file that provides information to the linker.

A `DEFINE` directive is an entry in the `ada.lib` file that provides information to the Ada preprocessor.

INFO and LINK directives have the format: *name*:*type*:*value*: DEFINE directives have an additional field: *name:type:var_type:value: name* is the name of the directive. *type* can be the word LINK, INFO or DEFINE. *value* is one of the possible values for the directive of that type. *var_type* is one of STRING, TEXT, BOOLEAN, INTEGER or REAL.

Without options, a.info displays all visible directives in the current library.

The -i option executes a.info in interactive mode. In this mode, all directives that can be added are listed, and all command line actions can be performed interactively. a.info prompts for the desired directive names and values. The notation !.* or "" indicates that any value is acceptable. You cannot change the ADAPATH using a.info in the interactive mode (use a.path).

Help information describing each of the directives valid on your system is available through the menu displayed when the -i option is selected. Selecting the entry, Help on a directive?, displays a list of valid directives. Selecting a directive name from the list displays syntax and descriptive information about the directive.

Follow the operating system documentation, ed(1) to form regular expressions, shown in the options.

INFO *Directive Names*

SC Ada supports the following INFO directives, which use the indicated syntax:

APP:INFO:*boolean*:	Automatically invokes a.app
ARCHITECTURE:INFO:*value:*	Generate code optimized for either Version 7 (VERSION7) or for Viking (VIKING) architecture. [Default: VERSION7]
AUTO_INLINE:INFO:*boolean*:	Does automatic inlining
COMMENT:INFO:*any_value*:	Includes user comments in ada.lib
CPU_LIMIT:INFO:*cpu_seconds:*	Limits time used by the front end of the compiler

(Continued)

DEBUG_XREF:INFO:*boolean:*	Perform extra checks to ensure no references are being missed or erroneously added.
DEFAULT_SRC_EXT:INFO:*suffix_value:*	Specify default source file suffix
DEFER_INSTANTIATIONS:INFO*:* *boolean*:	Prevent compiler from performing body instantiations until explicitly requested
ENDIAN:INFO:*endian_value*	Specify endian type of target processor
FLOAT_REGISTER_VARIABLES:INFO: *boolean*:	Use the floating point registers
HOST:INFO:*host_name*:	Specifies the host system name
MAX_GVAS_ADDR:INFO:*integer*:	Specify the maximum address boundary of GVAS
MAX_INLINE_NESTING:INFO:*integer*:	Specify the maximum depth of nested inline subroutine expansion
MAX_VIRTUAL_ADDR:INFO:*integer*:	Specify the maximum address boundary of virtual memory
MIN_GVAS_ADDR:INFO:*integer*:	Specify the minimum address boundary of GVAS
MULTISOURCE_FE:INFO:*boolean*:	Accept multiple source files in batches
PARALLEL_CODE_GEN:INFO:*boolean*:	Invoke the fe/optim/cg in parallel, using pipes for the intermediate language input and output instead of temporary files
READ_ONLY_LIBRARY:INFO:boolean	Specifies that SC Ada library is not modifiable.
SHARE_BODY:INFO:*boolean*:	Set default for pragma SHARE_BODY
STATIC_LINKING:INFO:*value*:	Links programs statically (static) rather than dynamically (dynamic). [Default: dynamic].
TARGET:INFO:*boolean*	Name of target processor
TARGET_C_LIBRARY:INFO: *library_name*:	Name of alternate library to use when linking

(Continued)

TARGET_C_P_LIBRARY:INFO: *library_name*:	Name of alternate profiling library to use when linking
TARGET:INFO:*target_processor*:	Name of target processor
UNSAFE_LIBRARY_SEARCHES:INFO: *boolean*:	Suppresses checks on ada.lib file version.
USE_LAST_LINK_INFO:INFO:*boolean*	Speed links retaining information from last link
VADS:INFO:*ada_location*:	Pointer to release area
VERSION:INFO:*version_number*:	Current version of SC Ada
XREF:INFO:*boolean*:	Print cross-reference information for a given Ada unit or library.

APP — This directive, when set to TRUE, causes the compiler to invoke the Ada preprocessor, a.app, automatically before compiling the source. The -P option to ada that invokes a.app takes precedence over the directive. If set to FALSE, this directive has no effect.

ARCHITECTURE — This directive controls whether code is generated for SPARC Version 7 architecture or SPARC Version 8 (Viking) architecture. Code generated for the Version 7 architecture runs on the Viking architecture. However, code that is optimized for the Viking can be generated by setting this directive to VIKING. Code generated for the Viking does not run on Version 7. To have code run correctly on any SPARC, leave this directive set to VERSION7. If the code is only going to run on Viking SPARCs, set it to VIKING. [Default: VERSION7]

CPU_LIMIT — This directive limits the CPU time used by the SC Ada compiler front-end. The limit applies to compilation, but not to execution of the user program. The directive can occur in any ada.lib file on the search path.

Caution – This directive is intended for use only as a backstop. The SC Ada library can be left in an inconsistent state when terminated in this way.

DEBUG_XREF — If DEBUG_XREF is set to TRUE, the compiler does additional internal checks to ensure that no references are being missed or erroneously added. If these checks are done and a missing or erroneous reference exists, the compiler issues a warning to that effect. These checks require additional time and space resources. [Default: FALSE]

DEFAULT_SRC_EXT — This directive enables you to specify a default source file suffix. The compiler runs slightly faster when most of the source files have a suffix that matches the default. Valid suffixes must begin with a period (.) and not contain another period (.) in the name

DEFER_INSTANTIATIONS — This directive prevents the compiler from performing body instantiations until they are explicitly requested by invoking the ada command with the -R option or by invoking a.make. Use of this directive can save time if generic bodies are being repeatedly modified and recompiled.

ENDIAN — This directive enables you to specify the endian type of the target system. Possible endian values are BIG and LITTLE. The default is BIG.

FLOAT_REGISTER_VARIABLES — This directive enables the use of register variables for floating point numbers. Its default value is TRUE.

HOST — This directive specifies the name of the host. The Ada preprocessor a.app uses this value.

MAX_GVAS_ADDR — This INFO directive specifies the maximum boundary of GVAS. The -G option to the ada command displays the suggested value for this directive.

MAX_INLINE_NESTING — This INFO directive specifies the maximum depth of nested inline subroutine expansions. Valid values are integers between 0 and 50. When 0, no inline expansions are performed. Be careful when specifying a value larger than 5 as the size of the compiler code becomes quite large. The default depth of nesting is 5.

The compiler limits the inline nesting depth of directly recursive routines to 4.

MAX_VIRTUAL_ADDR — This INFO directive specifies the maximum boundary of virtual memory.

MIN_GVAS_ADDR — This INFO directive specifies the minimum boundary of GVAS. The -G option to the ada command displays the suggested value for this directive.

MULTISOURCE_FE — When this directive is set to TRUE, the a.make and ada commands invoke the front end with as many as 20 files at a time. If the front end finds an error in one file, it does not compile subsequent files. However, a.make attempts to recompile the subsequent files if it determines that they do not depend on the erroneous file. Currently, only one file is passed to the front end if the APP_INFO directive is TRUE or any of the following options are on the command line: -d, -P, -e or -E.

PARALLEL_CODE_GEN — When this directive is set to TRUE, it invokes the front end, optimizer, and code generator in parallel, using pipes for the intermediate language (IL) input and output rather than temporary files. The file descriptors for the pipes are passed to the fe, optim, and cg through the -pi *number* (input pipe) and -po *number* (output pipe) options. For example, given the following:

```
fe -po 1 | optim -pi 0 -po 1 | cg -pi 0
```

the IL is read from stdin and written to stdout. This fails if anything besides the IL stream is written to the standard output file. The ada tool creates pipes and passes the file descriptions of these pipes to the tools. The pipes are usually file descriptions 3 through 6.

This directive makes the compiler slightly faster on uniprocessor machines. It significantly increases the speed of the compiler on multiprocessor systems. Multisource compiles show the greatest speed improvements. Default: [FALSE]

READ_ONLY_LIBRARY — If this directive is set to TRUE, it specifies that the SC Ada library is not to be modified by SC Ada. The SC Ada compiler, make utility and library management tools will not modify read-only libraries.

The presence of this directive in an SC Ada library allows SC Ada to perform optimizations during library operations. No lock files are created in read-only libraries. Units compiled within read-only libraries are assumed to be up to date, and timestamps on the units are not compared with the last-modified times of the source files. The reduction in file system operations allowed by the presence of this directive increases the performance of the SC Ada compiler and tools, especially when SC Ada libraries are accessed over a network.

The effects of this directive are a super-set of the effects of the UNSAFE_LIBRARY_SEARCHES directive. That directive does not suppress the checks for file timestamps performed by a.make and a.ld, which are suppressed by the READ_ONLY_LIBRARY directive.

SHARE_BODY — This INFO directive specifies the default method the compiler uses to perform generic instantiation. If TRUE, the compiler compiles instances so the code for the body of the generic is shared among the instances. This is the default. If FALSE, each instance of a generic causes the compiler to regenerate the code for the generic body. Override the default with pragma SHARE_BODY.

STATIC_LINKING — This directive defines whether programs are linked dynamically (dynamic) or statically (static). [Default: dynamic]

TARGET — This directive informs the compiler and other tools of the target processor for which code is being generated.

For self host applications, *target_processor* is SELF_TARGET.

TARGET_C_LIBRARY — This directive provides the name of the library to use when linking if libc (the default) is not used.

TARGET_C_P_LIBRARY — This directive provides the name of the profiling library to use when linking if libc (the default) is not used.

UNSAFE_LIBRARY_SEARCHES — SC Ada tools read the ada.lib files on the ADAPATH during initialization. When an operation on a library is performed, the tools check that the version of the ada.lib file in memory matches the version on the disk. Setting this directive to TRUE suppresses these checks and can increase the performance of the SC Ada tools. However, as its name implies, use this directive with great caution.

The suppression of these checks can lead to serious errors if one SC Ada tool modifies the ada.lib file while another tool is running. It is the user's responsibility to ensure that this never happens. [Default: FALSE]

USE_LAST_LINK_INFO — When this directive is set to TRUE, a.ld creates a file, .LAST_LINK, in the local Ada library that retains the list of units, their types, seals and their dependencies in the following format:

```
unit_nameXseal {non_trivial_dependent_numbers}*
```

where

X is '|' if *unit_name*'s body has elaboration code
X is '>' if *unit_name*'s body has no elaboration code
X is ':' if *unit_name*'s spec has elaboration code
X is '<' if *unit_name*'s spec has no elaboration code

seal is the timestamp value for the compilation date of unit_name

NON_TRIVIAL_DEPENDENT_NUMBERS is a list of numbers for units that *unit_name* depends upon, which, if regarded in transitive closure produce the direct dependencies of *unit_name*. The numbers identify the unit lines of the .LAST_LINK file. The numbers are biased by the lines of header information, currently two lines.

Note – The..LAST_LINK file has two header lines, giving the number of units involved, and the timestamp of the link. The order of the .LAST_LINK file is significant and represents the elaboration order decided for the last link. The a.ld processor attempts to use this order again, unless dependency changes prevent it. The .LAST_LINK file acts to stabilize links, in that elaboration order tends to remain steady. Use the .LAST_LINK file to encourage a specific elaboration order. Editing the .LAST_LINK file is considered dangerous, as the dependent numbers are changed. The .LAST_LINK information is normally useful even when linking different programs in succession, as long as a number of units are shared by the programs. If set to FALSE, this directive has no effect. [Default: FALSE]

VADS — This directive provides a pointer to the directory containing a particular version of SC Ada. Its provides SC Ada tools with a base location for the release area.

VERSION — This directive specifies which version SC Ada library this is and what version of other SC Ada tools have worked in this Ada library.

XREF — This directive generates cross-reference information for the unit(s) compiled with it. This information is stored in the net file and is used by the a.xref tool to generate cross-reference listings for designated units and/or libraries. [Default: FALSE]

`LINK` *Directive Names*

SC Ada supports the following LINK directives, which use the indicated syntax:

`LIBRARY:LINK:`*filename*:	RTS system library for programs without tasking
`MIN_TASKING:LINK:`*filename*:	RTS system library for programs with tasking but which do not have `abort` or `select with terminate`.
`STARTUP:LINK:`*object_filename*:	Name of a startup file
`TASKING:LINK:`*filename*:	RTS system library for programs with tasking
`WITH`*n*`:LINK:`*string*:	Pass files and/or commands to the linker

`LIBRARY` — This directive provides the name of the file that contains the SC Ada Runtime System library for programs that do not use tasking.

`RMIN_TASKING` — This directive provides the name of the file that contains the SC Ada Runtime System Library for programs that use tasking but contain no `abort` or `select with terminate` statements. Since the program does not need this capability, it has been eliminated in the file to reduce the size of the program and to get rid of unnecessary checks.

`STARTUP` — This directive provides the name of the object file that contains the program start-up routine. The `STARTUP` directive causes `a.ld` to include the object file in the link before any other object file.

`TASKING` — This directive provides the name of the file that contains the SC Ada Runtime System Library for programs that use tasking. The CPU linker selects the correct link entry to use.

`WITH`*n* — This directive enables you to add files and/or commands to the list of files/commands that `a.ld` sends to the linker. *n* is any integer. Any break in the sequence `WITH1`, `WITH2`,... causes the prelinker to stop processing `WITH`*n* directives. If a set of `WITH`*n* directives have the names `WITH1`, `WITH2`, `WITH4`, and `WITH5`, only `WITH1` and `WITH2` are processed. Because `WITH3` is missing, `WITH4` and `WITH5` are not processed.

Files

ada_location`/sup/legal.all`	List of legal directives for the current implementation.

References

"a.ld — build an executable program from previously compiled units" on page 2-53
"a.make — recompile source files in dependency order" on page 2-60
"register variables" on page 3-125
on-line help in the debugger, *SPARCompiler Ada User's Guide*

a.ld — *build an executable program from previously compiled units*

Syntax

a.ld [*options*] *unit_name* [*linker_options*]

Arguments

linker_options
All arguments after *unit_name* pass to the linker. These are options for the linker, archive libraries, library abbreviations or object files.

options
Options to the a.ld command are:

-DO
(objects) Use partially linked objects instead of archives as an intermediate file if the entire list of objects cannot be passed to the linker in one invocation. This option is useful because of limitations in the archiver on some hosts (but not Solaris 2.1).

-DT
(time) Displays how long each phase of the prelinking process takes.

-Du *unit_list*
(units) Traces the addition of indirect dependencies to the named units.

-Dx
(dependencies) Displays the elaboration dependencies used each time a unit is arbitrarily chosen for elaboration.

-DX
(debug) Debug memory overflow (use in cases where linking a large number of units causes the error message "local symbol overflow" to occur).

-E *unit_name*
(elaborate) Elaborate *unit_name* as early in the elaboration order as possible.

-F
(files) Print a list of dependent files in order and suppress linking.

`-L` *library_name*
(library) Collect information required for linking in *library_name* instead of the current directory. However, place the executable in the current directory.

`-o` *executable_file*
(output) Use the specified filename as the name of the output rather than the default, `a.out`.

`-sh`
(show) Display the name of the tool executable but do not execute it.

`-T`
(table) List the symbols in the elaboration table to standard output.

`-U`
(units) Print a list of dependent units in order and suppress linking.

`-V`
(verify) Print the linker command but suppress execution.

`-v`
(verbose) Print the linker command before executing it.

`-w`
(warnings) Suppress warning messages.

unit_name
Name of an Ada unit. It must name a non-generic subprogram. If *unit_name* is a function, it must return a value of the type `STANDARD.INTEGER`. This integer result passes to the shell as the status code of the execution.

Description

`a.ld` collects the object files to make *unit_name* a main program and calls the `ld(1)`linker to link together all Ada and other language objects required to produce an executable image in `a.out`. The utility uses the net files produced by the Ada compiler to check dependency information. `a.ld` produces an exception mapping table and a unit elaboration table and passes this information to the linker. The elaboration list generated by `a.ld` does not include library level packages that do not need elaboration. Similarly, packages that contain no code that can raise an exception no longer have exception tables.

`a.ld` reads instructions for generating executables from the `ada.lib` file in the SC Ada libraries on the search list. Besides information generated by the compiler, these directives include `WITH`*n* directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Place any number of `WITH`*n* directives in a library but number them contiguously beginning at `WITH1`. The directives are recorded in the library `ada.lib` file and have the following form.

`WITH1:LINK:`*object_file*:
`WITH2:LINK:`*archive_file*:

Place `WITH`*n* directives in the local SC Ada libraries or in any Ada library on the search list.

A `WITH`*n* directive in a local SC Ada library or earlier on the library search list hides the same numbered `WITH`*n* directive in a library later in the library search list.

Use `a.info` to change or report library directives in the current library.

ada_location`/bin/a.ld` is a wrapper program that executes the correct executable based upon directives visible in the `ada.lib` file. This enables multiple SC Ada compilers to exist on the same host. The `-sh` option prints the name of the actual executable file.

Files

`a.out`	Default output file
`.nets`	Ada `DIANA` net files directory
`.objects/*`	Ada object files
ada_location`/standard/*`	Startup and `standard` library routines

Diagnostics

Self-explanatory diagnostics are produced for missing files, and so forth. Additional messages are produced by the `ld` linker.

References

"a.info — list or change SC Ada library directives" on page 2-42
"LINK Directive Names" on page 2-51
`ld(1)`, *Solaris Developer Documentation*

a.list — *produce source code listing*

Syntax

```
a.list [-n -V] source_file
a.list [-n -V] -p prof_file
```

Arguments

options
Options to the a.list command are:

-n
(no) Suppress line numbers.

-p
(profiling) Read the mon.list file (generated by a.prof -d) and insert source line execution percentages in listings.

prof_file
Name of file containing profiling information. If no file is listed, mon.list is the default.

source_file
Name of Ada source file.

-V
(validation) Do not change the formfeeds to a two-character representation pf "6″L.".

Description

a.list produces a listing for programs containing no errors that closely resembles the output of a.error. The listing is written to the standard output and can be piped or redirected to a file.

References

"a.error — analyze and disperse error messages" on page 2-31
"a.prof — analyze and display profile data" on page 2-79

2

a.ls — *list compiled units*

Syntax

```
a.ls [options] [unit_name]
```

Arguments

options
Options to the a.ls command are:

-a or -All
(all) List all units visible in libraries on the library search list.

-b
(body) Limit output to unit bodies.

-f *source_file*
(file) List only units in *source_file*.

-F
(suffix) List unit bodies with a trailing #.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

-l
(long) List net file date, source file date, unit, and unit type.

-M
(main) List main units.

-s
(specification) Limit output to unit specifications.

-v
(verbose) List source filename, source file date, net file date, and unit.

-1
(one/single) Print output in a single column.

unit_name
Name of an Ada unit. *unit_name* is expressed as a regular expression. See the following Description for specifications for regular expressions.

Description

`a.ls` provides a list of the units compiled in the current or specified SC Ada library. Options give more or less extensive information, change the format of the list or provide a list of compiled units occurring in specified source files.

You can specify *unit_name* as a regular expression (similar to regular expressions in `csh(1)`) to match groups of units. If the regular expression contains any of the shell meta characters, quote the expression, for example, `a.ls` "f*" will list all units beginning with the letter "f".

If no match is found for the input criteria, `a.ls` returns a non-zero exit status.

Specifications for Regular Expressions

- Special characters are `?`, `*`, `[]`, and `{}`.
- Any character except a special character matches itself.
- `?` matches a single character.
- `*` matches any number of characters.
- A non-empty string `s` or bracketed `[s]` matches any character in `s`. In `s`, `\` has no special meaning. Use `[`only as the first character. A substring, *a*`-`*b*, with *a* and *b* in ascending ASCII order, stands for the inclusive range of ASCII characters.
- Specify alternative matches with {*first*, *second, third*}.

Without the `-l` or `-v` options, `a.ls` prints output in multiple columns. `-l` and `-v` cannot be used with the `-s` option.

The options `-F`, `-l`, and `-v` (in increasing order of listing detail) are mutually exclusive. If more than one of these three is given, the listing is that with the most detail.

References

regular expressions in `csh(1)`, *Solaris Developer Documentation*

2

a.make — *recompile source files in dependency order*

Syntax

```
a.make [options] [unit_name]... [ld_options] [-f source_file ...]
```

Arguments

ld_options

Options are passed directly to the linker and a.make does not process.

options

`-A` *ada_library* `[-A` *ada_library*`] ...`

(add) Bring the listed libraries up to date if necessary.

`-All`

(all) Bring all libraries on the library search path up-to-date.

`-C "`*compiler*`"`

(compiler) Use the string *compiler* in recompiling the required units. Only use this option to pass options to the compiler that a.make does not recognize.

`-D` *identifier type value*

(define) Define an identifier of a specified type and value.

`-Dv`

(debug) Print debugging information.

`-d`

(dependencies) List the file-to-file dependencies. Note that dependency information for instantiation bodies is not listed.

`-E`

`-E` *directory*

(error output) Without a file or directory argument, a.make processes error messages using a.error and directs a brief output to standard output; the raw error messages are left in *ada_source*.err. If a file path name is given, the raw error messages are placed in that file. If a directory argument is supplied, the raw error output is placed in *dir/source*.err. Use the file of raw error messages as input to a.error. Only use one -e or -E option.

`-e`
(error) Process compilation error messages using `a.error` and send it to standard output. Only the source lines containing errors are listed. Use either the `-e` or `-E` option, not both.

`-Ef`*error_file source_file*
(error) Process source_file and place any error messages in the file indicated by *error_file*. Note that no space is between the `-Ef` and *error_file*.

`-El`
`-El` *directory*
(error listing) Same as the `-E` option, except that a source listing with errors is produced.

`-el`
(error listing) Intersperse error messages among source lines and direct to standard output.

`-Elf`*error_file source_file*
(error listing) Same as the `-Ef` option, except that a source listing with errors is produced.

`-ev`
(error `vi(1)`) Process syntax error messages using `a.error`, embed them in the source file, and call the environment editor `ERROR_EDITOR`. (If `ERROR_EDITOR` is defined, define the environment variable `ERROR_PATTERN`. `ERROR_PATTERN` is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specified, `vi(1)` is called.

`-f` *source_file_list*
(files) Treat remaining non-option arguments as filenames in the current SC Ada library to consider for compilation. All units in these files are brought up-to-date; use `-f` with one of the other options to print actions or dependencies without executing them, but must be the last option given.

`-F` *filename*
(named file) Consider for compilation the source files listed in *filename*. This option is similar to `-f` option, except that listing the source files in *filename* allows a list that may otherwise exceed the command-line limit. In *filename*, list multiple filenames on a single line with any number of blanks or tabs separating them.

`-I` *source_file*
(if) List actions that are taken if *source_file* changes.

`-i`
(ignore errors) Do not suppress compilation if file is dependent upon another file that did not compile successfully.

`-L` *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

`-l` *"linker"*
(linker) Use the string *linker* in linking the required units. This option provides unusual options to `a.ld` when using `a.make`.

`-O[0-9]`
(optimize) Invoke the code optimizer. An optional digit (no space is before the digit) provides the level of optimization. The default is `-O4`.

Note – Each level of optimization is cumulative. For example, optimization level 4 (`O4`) incorporates all the features of optimization levels 1 through 3, plus hoisting invariants from loops and address optimizations.

`-O`
Full optimization

`-O0`
No optimization

`-O1`
Copy propagation, constant folding, removing dead variables, subsuming moves between scalar variables

`-O2`
Add common subexpression elimination within basic blocks

`-O3`
Add global common subexpression elimination

`-O4`
Add hoisting invariants from loops and address optimizations

`-O5`
Add range optimizations, instruction scheduling and one pass of reducing induction expressions

`-O6`
No change and instruction scheduling

`-O7`
Add one more pass of induction expression reduction

`-O8`
Add one more pass of induction expression reduction

`-O9`
Add one more pass of induction expression reduction and add hoisting expressions common to the `then` and the `else` parts of `if` statements

Hoisting from branches (and cases alternatives) can be slow and does not always provide significant performance gains so it can be suppressed.

`-o` *executable_file*
(output) Use the specified filename as the name of the output rather than the default, `a.out`.

`-P`
(preprocessor) Invoke the Ada Preprocessor.

`-S`
(suppress) Apply `pragma SUPPRESS` to the entire compilation.

`-sh`
(show) Display the name of the tool executable but do not execute it.

`-T`
(timing) Print timing information for the compilation. This option is not used directly by `a.make` but is passed to `ada` which prints the timing information for the files compiled.

`-U`
(units) List the list of dependent units in order but do not link.

`-v`
(verbose) List the recompilation commands as they are executed.

`-V`
(verify) List the recompilation commands to execute but do not execute them.

`-w`
(warnings) Suppress warning diagnostics.

Description

`a.make` *unit_name* determines which files must be recompiled in order to produce a current executable file with *unit_name* as the main unit.

`a.make -f` *source_file_list* determines which units in *source_file_list* must be recompiled. Use the `-f` option anytime you are starting in a new library. The command

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

makes all Ada files in a library in the correct order. You do not need to specify a main unit in this case, but if you do, that main unit is linked. This option provides the mechanism to bootstrap `a.make` and start things off.

`a.make` *unit_name* `-f` *source_file_list* considers for compilation all files that are needed by *unit_name*, regardless of whether they are included in *source_file_list*. If a file is included in *source_file_list* but is not needed by *unit_name*, it is not considered for compilation. If a file is needed by *unit_name* but is not included in *source_file_list*, it nevertheless is considered for compilation. However, the `-f` *source_file_list* option makes the dependency information about all units in *source_file_list* visible to the library.

`a.make` has no knowledge of any source file (`foo.a`) until that file is compiled in a way that changes the program library. Unless the `-f` option is used, this requires that `foo.a` be compiled “by hand” at least once. Unless the `-U` or `-d` option is given, the file must compile successfully or else the program library remains unchanged. In any case, syntax errors must be corrected before the file is “seen” by `a.make`.

`a.make` uses `DIANA` net files to determine the correct order of compilation and elaboration.

ada_location`/bin/a.make` is a wrapper program that executes the correct executable based upon directives visible in the `ada.lib` file. This enables multiple SC Ada compilers to exist on the same host. The `-sh` option prints the name of the actual executable file.

Supplied names and unknown options are passed to `a.ld`.

Files

`ada.lib`	Library reference file
`gnrx.lib`	Generic instantiation reference file
`GVAS.lock`, `gnrx.lock`	Lock the library while reading or writing special library files
`GVAS_table`	Address assignment file
`.imports`	Imported Ada units directory
`.lines`	Line number reference files directory
`.nets`	Ada network control files directory
`.objects`	Ada object files directory

References

“a.app — preprocess Ada source” on page 2-11
“a.ld — build an executable program from previously compiled units” on page 2-53
compiling Ada programs (optimization), *SPARCompiler Ada User’s Guide*
`pragma OPTIMIZE_CODE`, *SPARCompiler Ada Programmer’s Guide*

2

a.mklib — *create an SC Ada library directory*

Syntax

a.mklib [-F -f -v -i] [-t *target*] [*new_ada_library*] [*parent_ada_library*]

Arguments

new_ada_library
Name of directory to initialize as an SC Ada library. If no directory is specified, the current working directory is initialized.

options
Options to the a.mklib command are:

-f
(force) Create SC Ada library structure even if some components are already present.

-F
(force name) Allow creation of an SC Ada library with a restricted name.

-i
(interactive) Display all versions of SC Ada installed on the system and prompt for selection of SC Ada version unless modified with the -t option.

-t *target*
(target) Create a library for a specific target machine.

-v
(verbose) Display the library search list and target directives.

parent_ada_library
Name of an existing SC Ada library. If a parent library is named, the library search list in the new ada.lib consists of the parent library and the parent library path. As a result, Ada units in the new library reference all Ada units defined by the parent library and all units that are accessible from the parent library.

Description

`a.mklib` creates and initializes a new SC Ada library directory, creating three files (`GVAS_table`, `ada.lib`, and `gnrx.lib`) and four directories (`.lines`, `.imports`, `.nets`, and `.objects`). The library search list in the new `ada.lib` consists of the parent library and the parent library path. As a result, Ada units in the new library can reference all Ada units defined by the parent library and all units that are accessible from the parent library.

Note – We recommend using `a.mklib` with the `-i` (interactive) option. It provides a choice of the available releases. The `-i` option provides the contents of the `VADS_END` file for each release on the system. The `VADS_END` file contains information about the host, host operating system, version number, and dates for each compiler.

The `-v` option cannot be used with the `-i` option.

The tool `a.vadsrc` creates a local configuration file called `.vadsrc`. Place it either in the current directory or in your home directory. Future libraries are created using the *parent_ada_library* specified by the *target* entry in this file.

When no *parent_ada_library* is specified on the command line, `a.mklib` searches the `/etc/VADS` file for a unique entry for *target*. If multiple target entries exist for *target* in `/etc/VADS` (such as when more than one version of SC Ada is installed for *target*), then `a.mklib` searches for a unique *target* entry in a `.vadsrc` file in the local directory, and then in a `.vadsrc` file in your home directory. If the *target* entry in the `/etc/VADS` file is not unique and no `.vadsrc` file exists, an error message results, and `a.mklib` requires the `-t` target option, the `-i` option or a *parent_ada_library* specified on the command line.

The `-t` *target* option specifies a particular target machine. Obtain a list of available targets with the `-i` option or with the tool `a.vadsrc`. These values are case-insensitive, for example, `SELF_TARGET` or `self_target`.

The `-f` option forces initialization of an SC Ada library structure, overwriting any existing components and deleting any existing lock files.

`a.mklib` prohibits the creation of libraries named `standard`, `verdixlib` or `publiclib`. The `-F` options overrides this restriction.

Example

If you are positioned at the directory `/usr2/babbage/code` and the SC Ada library *parent_library* exists, the command

```
a.mklib new_library parent_library
```

creates the library directory `/usr2/babbage/code/new_library` and provides access to the Ada compilation units compiled previously in the *parent_library* library directory. Any units available to *parent_library* from other libraries are now available from *new_library* as well.

However, if *parent_library* is not an SC Ada library, `a.mklib` issues an error message.

Files

`/etc/VADS`	SC Ada version reference file
`.vadsrc`	Local configuration file
`~/.vadsrc`	Your home directory configuration file

Diagnostics

An error is reported and no action is taken (without the `-f` option) if *new_ada_library* contains any SC Ada components or lock files, or if the name specified exists but is not a directory.

References

“a.cleanlib — reinitialize library directory” on page 2-16
“a.rmlib — remove compilation library” on page 2-85
“a.vadsrc — display versions and create library configuration file” on page 2-98
“SPARCompiler Ada Library Contents” on page 1-22
creating an SC Ada library, *SPARCompiler Ada User’s Guide*

a.mv — *move unit and library information*

Syntax

```
a.mv unit_name [, ...] [options] target_directory
a.mv source_file [, ...] [options] target_directory
```

Arguments

options
Options to the a.mv command. These are:

-b
(body) Move the bodies of the named units.

-F
(force name) Move units to protected libraries, that is, standard, verdixlib, publiclib.

-f
(force) Do not report matching errors if unit name is not found.

-L *library_name*
(library) Move from SC Ada library *library_name* (the current working directory is the default).

-s
(spec) Move the compilation information for the specifications of the named units.

-u
(unit) Force the next name to be treated as a unit even though it contains a period (.).

-V
(verify) List the units to move but do not move them.

-v
(verbose) List the units as they are moved.

source_file
Name of an Ada source file.

target_directory
Directory to which the unit and library information is to move. This dire ctory must be an SC Ada library.

unit_name
Name of an Ada unit or subunit. Unit names with dotted notation such as `aaa.bbb` or `aaa.bbb.ccc` are taken to be the names of Ada source files unless the `-u` option is specified.

Description

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

If *source_file* is specified, the corresponding files in `.nets`, `.lines`, and `.objects` are moved for each unit defined in *source_file*, the appropriate entries are deleted from the `ada.lib` in the source library created in the `ada.lib` in the target directory.

A variety of options move specifications and bodies separately. The `-u` (unit) option enables references to units whose names contain a period (.). Without the `-u` option, any name that contains a period is treated as a source filename.

Specify *unit_name* and *source_file* with regular expressions. For example, `a.mv "f*"` moves all units beginning with the letter "`f`." The command, `a.mv "f*.a,"` moves all units in source files that begin with the letter "`f`."

References

"a.cp — copy unit and library information" on page 2-18
"Specifications for Regular Expressions" on page 2-59

a.path — *report or change SC Ada library search list*

Syntax

```
a.path [options] [ada_library1 [ada_library2]]
```

Arguments

options

Options to the a.path command are:

-a *ada_library1* [*ada_library2*]
(append) Append *ada_library1* after *ada_library2*. With a single argument, append *ada_library1* to the end of the library search list. Both *ada_library1* and *ada_library2* must be SC Ada libraries.

-all *ada_library*
(all) Append the ADAPATH of *ada_library* to the ADAPATH of the current library.

-d
(dependency check) Check for circularities in the path. This option provides additional path cycle information when used with -v.

-f
(force) Override checks on libraries to allow the use of erroneous or nonexisting library names with a.path options.

-c
(cleanup) Remove from ADAPATH all erroneous libraries.

-i *ada_library1* [*ada_library2*]
(insert) Insert *ada_library1* before *ada_library2*. With a single argument, insert *ada_library1* at the beginning of the list. Both *ada_library1* and *ada_library2* must be SC Ada libraries.

-I
(interactive) Operate in interactive mode.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

`-r` *ada_library1*
(remove) Remove *ada_library1* from the library search list.

`-v`
(verbose) Display path as it is changed.

`-x` *ada_library1*
(except) Remove all libraries except *ada_library1* from the list.

Description

`a.path` changes or reports the list of library names to search during compilation. This list is maintained in the `ada.lib` file in the current SC Ada directory. During compilation, any program units not in the current library are searched for in the SC Ada libraries listed on the search list. If the unit is not in the first SC Ada library, it is searched for in the second and so on in listed order. When `a.path` is used with no options, it reports the contents of the current library search list, one library to a line. `a.path` flags any incomplete library on its command line unless the `-f` option is specified

When the `-I` (interactive) option is specified, the following menu appears. Implement all other options through this menu. To operate in a different *ada_library*, include the `-L` *library_name* option on the `a.path` command line. See Figure 2-3.

```
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) ->
```

Figure 2-3 Example of `a.path -i` (interactive) Menu

Removing a library name from the library search list does not remove compilation information from the referenced libraries.

The maximum length of each element in the library search list is the maximum line length of the system. However, the library search list itself is unlimited.

References

“ada.lib File” on page 1-24

2

a.pr — *format source code*

Syntax

a.pr [*options*] [*source_file*]

Arguments

options

The two types of a.pr options are configuration file options and command line options.

.prrc *Configuration File Options [default shown in brackets]*

align_cmts *where*

Align comments to the right of the longest line (line) or the longest line containing a comment (comment). [comment is the default.]

chars *number*

Specify maximum number of characters of code per line, including comment and indentation; any line extending over this limit is continued on the next line; valid range is from 20 to 500 [132].

comment *case*

Print all comments in the specified case: upper, lower, same. [same]

ident *case*

Print all identifiers in the specified case: upper, lower, same. [upper]

indent *number*

Specify amount of indentation between levels; valid range is from 1 to 8. [8]

lines *number*

Specify maximum number of lines allowed on a page; valid range is from 1 to 1000. [55]

margin *number*

Specify starting margin for top-most level; valid range is from 0 to 15. [0]

no_page

Paginate only when pragma PAGE is encountered. [page]

`no_warning`
Suppress warning messages regarding line length greater than desired. [provide warnings]

`page` *number*
Set page size; perform pagination with blank lines; valid range is from 1 for 1000. [paginate using form feeds]

`page_lu`
Start each library unit (indicated by a `with` clause) on a new page. [do not start on new page]

`record` *where*
Print `record` on either the same line (`same`) or on the next one (`next`). [`same`]

`reserved` *case*
Print all reserved words in the specified case: `upper`, `lower`, `same`. [`lower`]

`tabs` *number*
Print tabs for indentation whenever the number of spaces needed for indentation is greater than or equal to the specified number; valid range is from 0 to 8; if `tabs 0` is specified, indentation is performed with blanks. [8]

`a.pr` *Command Line Options [default shown in brackets]*

`-ac`
(align comment) Align comments to the right of the longest line that contains a comment. [default]

`-al`
(align line) Align comments to the right of the longest line, regardless of whether it contains a comment. [`-ac`]

`-c` *number*
(characters) Specify maximum number of characters of source code allowed on a line. Valid range is from 20 to 500. [132]

`-cl`
(comments lower) Print comments in lowercase. [`-cs`]

`-cs`
(comments same) Print comments as in source code. [default]

`-cu`
(comments upper) Print comments in uppercase. [`-cs`]

`-i` *number*
(indent) Specify indentation between levels. Valid range is from 1 to 8. [8]

`-il`
(identifiers lower) Print identifiers in lowercase. [`-iu`]

`-is`
(identifiers same) Print identifiers as in source code. [`-iu`]

`-iu`
(identifiers upper) Print identifiers in uppercase. [default]

`-l` *number*
(lines) Specify maximum number of lines allowed on a page. Valid range is from 1 to 1000. [55]

`-m` *number*
(margin) Specify starting margin for top-most level. Valid range is from 0 to 15. [0]

`-nl`
(no page library unit) Do not start a new page for each library unit. [default]

`-np`
(no pagination) Specify no pagination. Pagination occurs only when `pragma PAGE` is encountered. [`-pg`]

`-nw`
(no warnings) Suppress warning messages regarding line length. [`-w`]

`-p` *number*
(page) Specify page size. Valid range is from 1 to 1000. [`-pg`]

`-pg`
(pagination) Paginate using form feeds. [default]

`-pl`
(page library) Start a new page when a library unit is encountered. [`-nl`]

`-rl`
(reserved lower) Print reserved words in lowercase. [default]

`-RN`
(record next) Print record on the line following `type` or `for`. [`-RS`]

`-rs`
(reserved same) Print reserved words as in source code. [`-rl`]

`-RS`
(record same) Print record on the same line as `type` or `for`. [default]

`-ru`
(reserved upper) Print reserved words in uppercase. [`-rl`]

`-t` *number*
(tabs) Specify tabs for indentation whenever the number of spaces needed is greater than or equal to the specified number. If `-t 0` is specified, indentation is performed with spaces. Valid range is from 0 to 8. [8]

`-w`
(warning) Provide warning messages regarding line lengths greater than desired. [default]

source_file
Name of the Ada source file to format.

Description

`a.pr` reformats Ada source code according to options specified in a runtime configuration file with the name `.prrc`. Tailor `a.pr` for individual Ada coding standards. Place the configuration file either in your current working directory or your home directory.

Additionally, specify options on the command line that override those in the configuration file.

Invoked without a filename, `a.pr` reads its input from standard input.

Error and warning messages are written to `stderr`.

Note that command line options only override corresponding `.prrc` options. For example, a `-iu` command line option overrides a `set indent lower .prrc` option but has no effect on a `set commands upper .prrc` option.

Only one `.prrc` file is used. If a `.prrc` file is found in the current working directory, the `.prrc` file in the home history is completed ignored. If a `.prrc` file is not found in the current working directory, the `$HOME/.prrc` file is used (if it exists).

References

formatting source code, *Solaris Developer Documentation*

a.prof — *analyze and display profile data*

Syntax

```
a.prof [options] [filename [monitor...] ]
```

Arguments

filename
Name of the Ada source file to analyze.

monitor
The monitor file, mon.out, contains frequencies for address ranges. The UNIX monitor() subroutine produces it .

options
Options to the a.prof command are:

-a
(addresses) Display symbol addresses. This disambiguates overloaded and mysterious symbols.

-d
(disassembly) Generate source line profiling information in mon.list. a.list and a.das use this information.

-l
(list) Sort output by symbol value.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

-s
(summary) Produce a summary profile file in mon.sum. Useful when more than one profile file is specified.

-sh
(show) Display the executable tool path name but do not execute it.

-z
(zero) Display routines which are not used.

Description

Statistical profiling is a simple way for you to determine the relative CPU usage of all the parts of your Ada programs. SC Ada offers profiling Ada libraries, which cause Ada programs to be interrupted at regular intervals and the program instruction counter examined and saved.

`a.prof` interprets the file produced by a program linked with the `profile_conf` directory.

Solaris 2.1 kernel support for profiling makes this method effective and unobtrusive for self-host development (see *ada_location*`/profile_conf`).

You control the accuracy of profiling by the duration for which you run your programs and the amount of memory allocated to profiling accounting.

When used properly, profiling provides an accurate description of the CPU utilization of your entire Ada Program, including the runtime system.

`a.prof` interprets the monitor file produced by the execution of an Ada program. The symbol table of the named executable (or `a.out` by default) is read and correlated with the monitor file (`mon.out` by default). For every external symbol, the percentage of time spent executing between that symbol and the next, as well as the total time, are printed.

For multiple monitor files, the output represents the sum of the profiles.

Files

`a.out`	Executable
`mon.list`	For source line profile
`mon.out`	For profile
`mon.sum`	For summary profile

References

Statistical Profiler, *SPARCompiler Ada Programmer's Guide*

a.report — *report deficiency or suggestion*

Syntax

```
a.report
```

Description

a.report enables customers with SC Ada Support contracts to submit problems to Sun Answer Centers.

Sun Answer Center assigns a unique Service Order (SO) and acknowledges the SO.

a.report captures the date, time, site name, user ID, and Ada configuration information automatically. (See the sample report at the end of this entry.) It prompts for the name of source files illustrating the problem to be appended to the report and invokes vi(1) or the editor defined by the environment editor variable to enable editing of general text that describes the problem. The formatting lines supplied by a.report separate parts of the text for the Sun Answer Center's automated processing. Abort report generation at any point by means of the Intr key (usually mapped to Control–c).

A warning is displayed if the report length is greater than that acceptable by most network mailers. In that case, be certain that no intervening systems are between the Sun Answer Center and the report site. If so, save the report in a file and send it to the Sun Answer Center via magnetic tape.

When a report is complete, the entire file is electronically mailed to persons determined by the system administrator when SC Ada was installed. Usually, this list includes the site Ada administrator. Add additional mail destinations and/or save a copy of the report by supplying a filename when prompted.

Submit customer reports to the Sun Answer Center by telephone or by electronic mail.

Please ask your Sun sales representative for more information: contract price, nearest Sun Answer Center, and so forth.

Figure 2-4 is a sample customer report generated by a.report:

```
==========================================

New Report    Date/time: Wed Aug  7 16:37:01 EDT 1991
User: John Moore    Login: moore
Return-Path: Will use mail header for acknowledgement address
Category: 1-Compilation of Ada    Urgency: 3-Serious
Reference: BR007
Description: Identifier undefined at lines 7 and 8
SPARC SunOS Release 4.1, SPARCompiler Ada 1.0
VADS_END: Tue Jul 17 13:11:42 PDT 1990, 1.0(d)
OS Release: SunOS Release 4.1.1 (NSE_kernel) #2: Sun Jul 8 12:27:47
EDT 1990
Host: havoc    Hostid: 41000a73    Host#: 41000a73    Report#: 2
==========================================

a.info -A :
FLOATING_POINT_SUPPORT: MC68881
HOST: SPARC
LIBRARY: ada_location/.objects/library.a
TARGET: SELF_TARGET
TASKING: ada_location/standard/.objects/tasking.a
VADS: ada_location
VERSION: 1.0
==========================================

Please enter problem description below:
During the compilation of the bug.a file the following error occurs:

      line 7, char 7: error: RM 8.3: identifier undefined
      line 8, char 7: error: RM 8.3: identifier undefined
==========================================

- --==========++++++++++########## file bug.a
with TEXT_IO;

procedure hello is

begin

      put (“Hello, world.”);
      new_line;

end hello;
- --==========++++++++++########## end
==========================================
```

Figure 2-4 Example of Report Generated by `a.report`

a.rm — *remove an Ada unit from a library*

Syntax

```
a.rm [options] unit_name
a.rm [options] source_file
```

Arguments

options
Options to the a.rm command are:

-b
(body) Delete the bodies of the named units.

-F
(force name) Allow the removal of an SC Ada library having a reserved name.

-f
(force) Ignore warnings and protections.

-i
(interactive) Prompt for confirmation before deleting any unit information.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

-s
(spec) Remove the unit(s) specification information.

-u
(unit) Force the next name to be treated as a unit even though it ends in .a. Specify this option to a.rm if *unit_name* contains a period (.).

-v
(verbose) List the units as they are removed.

-V
(verify) List the units to remove but do not remove them.

source_file
Name of an Ada source file.

unit_name
Name of an Ada unit or subunit. Unit names with dotted notation such as `aaa.bbb` or `aaa.bbb.ccc` are taken to be the names of Ada source files unless the `-u` option is specified.

Description

When *source_file* is specified, the corresponding files in `.nets`, `.objects`, and `.lines` are removed for each unit defined in *source_file* and the appropriate entries are deleted from `ada.lib`. A name containing a period is taken to be an Ada source filename unless the `-u` option is given.

Specify *unit_name* and *source_file* with regular expressions. For example, `a.rm "f*"` deletes all units beginning with the letter "`f`." However, it does not delete units in source files beginning with "`f`." The command, `a.rm "f*.a,"` deletes units in source files that begin with the letter "`f`."

References

"a.cp — copy unit and library information" on page 2-18
"Specifications for Regular Expressions" on page 2-59
"a.mv — move unit and library information" on page 2-69

a.rmlib — *remove compilation library*

Syntax

```
a.rmlib [-f -F] [ada_library]
```

Arguments

options
Options to the a.rmlib command are:

-f
(force) Clean the SC Ada library structure even if some components are missing or lock files exist.

-F
(force name) Clean the SC Ada library structure of a library having a restricted name.

ada_library
Name of the SC Ada library in which to operate. If no library is specified, the current working directory is assumed.

Description

a.rmlib removes all SC Ada library components from *ada_library* or from the current library if no argument is given. It removes three files (GVAS_table, ada.lib, and gnrx.lib), four directories (.lines, .imports, .nets, and .objects), and lock files (if present) and the -f option is used. The directory itself, all Ada source files, and other files and directories are left untouched.

If a.rmlib cannot find every library component or lock files exist, it aborts without removing any files unless the -f (force) option is given.

Without the -F option, a.rmlib cannot operate in a library bearing the name standard, verdixlib or publiclib.

The directory name for the removed library is left in dependent library paths. This blocks compilation in any dependent libraries until a.path is used to remove the path entry that specifies this directory. Compilation proceeds also if an SC Ada library is recreated in the named directory from which the library information is removed.

Diagnostics

An error is reported and no action is taken (without the `-f` option) if *ada_library* contains an incomplete set of components or a lock file.

An error message is issued if any files or directories are not accessible for deletion.

References

"a.cleanlib — reinitialize library directory" on page 2-16
"Syntax" on page 2-66

a.symtab — *display symbol information for all static package variables and constants*

Syntax

```
a.symtab symbol_table_file [options]
    [ unit_name [unit_name] ... ] [ -L library_name ]
```

Arguments

options
The following options are available for a.symtab:

`-c`
(closure) Process all units in the closure of the specified unit. See below for a definition of closure.

`-b`
(body) Process the specified unit(s)' body only.

`-L` *library_name*
(library) If no units are specified, all units in library library_name are processed. If a unit is specified, a.symtab looks in the specified library for the given unit(s).

`-r`
(renamed) Include renamed objects in the symbol listing.

`-s`
(specification) Process the specified unit(s)' specification only.

symbol_table_file
Name of file containing a list of symbols in the executable image for the units being processed.

unit_name
Name of Ada unit for which symbols are to be displayed.

Description

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. Variables and constants nested inside subprograms are not listed.

If the -c option is given, all units in the closure of the specified unit(s) are processed, where closure is defined as the smallest set of units that contains:

1. the specified unit
2. its specification, if the unit is a body
3. its body, if the unit is a specification
4. its subunits, if any
5. its parent, if the unit is a subunit
6. all units named in the context clause of the specified unit(s)
7. all units named in the context clauses of the units in the context clause of the specified unit(s), etc.

Thus, if -c is used and the unit specified is a main program, all units that are needed to build that program are processed.

If neither -s or -b are specified, both the specification and body of the specified units (if they exist) are processed.

a.symtab uses information stored in the DIANA net files to determine the type and size of the package objects. The DIANA net also contains the name of the symbol that corresponds to the base address for this object, and the offset from that base address. For example, given the following package:

```
package examples is
   var1 : integer;
   var2 : float;
end;
```

The symbol containing the base address for the package variables would be _A_examples..STATIC. The offset for var1 might be 010, the offset for var2 might be 018. In order to determine the absolute address of var1 and var2, a.symtab must have access to the absolute address for the symbol _A_examples..STATIC. a.symtab requires as input a *symbol_table_file* that contains a list of symbols in the executable image for the units being processed.

The Solaris nm tool can be used to create this *symbol_table_file*. When using nm, the output must be in the easily parseable, Berkeley (4.3BSD) format, with the symbol name preceded by its value and a single character indicating its type, for example, "0000406b88 T _A_examples..STATIC". On systems where a

System V output format is the default, there is most likely an option that will produce the Berkeley format (on Silicon Graphic's IRIX 5.0, the option is `-B`; on SunOS 5.0, the option is `-p`).

For example, given a main program, `main`, the commands

```
a.ld main -o main.out
nm main.out > main.names
    a.symtab main.names -c main > symtab.out
```

use the symbol table information in main.names to generate the symbol information listing.

Symbol Listing Content and Format

Information regarding all static symbols in the specified packages is listed. Each package listed is followed by the symbols in that package. The packages are not ordered in any way; the listing is essentially random. The listing of the symbols within the packages is in the order that they were declared.

The following information is given in the symbol listing:

- level
- symbol name
- format
- size
- address

Each of these is discussed in the next sections.

Level
: The level # indicates the level of the symbol name. Each package is at level 1. All objects within that package follow and begin at level 2. If the object is a record, it components follow the object name and begin at level 3, with nested records having correspondingly higher level numbers.

Symbol Name
: For packages, this is the package name with a "..SPEC" (for specifications) or "..BODY" (for bodies) suffix appended. For objects and components, the simple name of the object is given.

Format
This is a single character that describes the symbol's base type. The following format characters are possible:

a -- array
b -- boolean
e -- enumeration
f -- all float and fixed point types
h -- access objects and objects of type system.address
i -- all integer types
r -- record
- -- package

Size
The size, in bits, of the symbol is given. If the object is a record or array, the total object size is given.

Note that this field is 0 for package (level 1) entries.

Address
The absolute (runtime) address of the symbol is listed.

Note that this field is 0 for package (level 1) entries.

Note, however, that arrays are a special case. The line following an array entry (indicated by the 'a' format character), is an entry for the array element. The level is the level of the array + 1, the name is the element type, the format is the same as for objects of that type, the size is the bit size of the element and the address is 0. If the element type is a record, the components are then listed as for record objects, with the address field containing the address of that component in the first array element.

Error Messages

If, for any reason, some of the desired information cannot be obtained about a symbol, the symbol listing entry is put to `stderr` (instead of `stdout`), with an appropriate message. Possible errors and their messages are:

Dynamic Variable

If a variable is not static (for example. is a record or array that is dynamically constrained), the size and address are not obtainable. In this case, the entry sent to `stderr` would look like:

```
1       dynamic_var             a           NOT STATIC
```

Symbol Not Found In Symbol Table File

It is possible for a unit not to have symbol information in the *symbol_file_table* given as input. This can arise due to units not linked in because of selective-linking, because there is an extra unit in a library that is not needed and is thus not linked into the executable image or because the *symbol_file_table* is out of date). In this case, the entry sent to `stderr` might look like:

```
1  foo     i    4     base address _A_unitname..STATIC not found
```

[fields compacted in example to not extend over 80 chars]

Constants That Have Been Folded

The compiler "folds" constants whenever possible, using the static value of the constant wherever the constant is referenced. Thus, it is possible for a constant to not have an address. In this case, the entry sent to `stderr` might look like:

```
1       const                   i           4          FOLDED
```

Objects That Have Been Given Address Clauses

Currently, `a.symtab` does not attempt to figure out the address of objects that have been given address clauses (e.g. "for foo use at ..."). In this case, the entry set to `stderr` might look like:

```
1       var_with_addr_cause     i           4          ADDR CLAUSE
```

Note that this restriction is temporary, `a.symtab` will eventually handle address clauses whenever possible.

Example

Figure 2-5 is a set of example units, and the output to both `stdout` and `stderr` when `a.symtab` `nm.out -c main` is invoked.

```
with system;
package example is

    type rec1 is record
        f1 : character;
        f2 : short_integer;
        f3 : string(1..10);
    end record;

    type rec2 is record
        f : float;
        r : rec1;
    end record;
    type arr_type is array(1..10) of rec2;

    type int_array is array(integer range <>) of integer;

    int      : integer := 5;
    r1       : rec1;
    r2       : rec2;
    rec2_arr : arr_type;
    int_arry : int_array(5..10);

    const         : constant integer := 1;    -- FOLDED
    dynamic_arr   : int_array(1..int);        -- NOT STATIC

    var_with_addr : integer;                  -- ADDR CLAUSE
    for var_with_addr use at int'address;

end example;

package body example is

    addr     : system.address;
    flt      : float;
end example;
with example;
procedure main is
begin
    null;
end;
                                                    (Continued)
```

```
(Continued)

 stdout:

 1  example..BODY                                  -    0      0
 2  addr                                  h    32     100092D0
 2  flt                                   f    64     100092D8
 1  example..SPEC                                  -    0      0
 2  int                                   i    32     10000040
 2  r1                                    r    112    10000048
 3  f1                                    e    8      10000048
 3  f2                                    i    16     1000004A
 3  f3                                    a    80     1000004C
 4  character                                     e    8      0
 2  r2                                    r    176    10000058
 3  f                                     f    64     10000058
 3  r                                     r    112    10000060
 4  f1                                    e    8      10000060
 4  f2                                    i    16     10000062
 4  f3                                    a    80     10000064
 5  character                                     e    8      0
 2  rec2_arr                              a    1920   10000070
 3  rec2                                          r    192    0
 3  f                                     f    64     10000070
 3  r                                     r    112    10000078
 4  f1                                    e    8      10000078
 4  f2                                    i    16     1000007A
 4  f3                                    a    80     1000007C
 5  character                                     e    8      0
 2  int_arry                              a    192    10000160
 3  integer                                       i    32     0
 3  integer                                       i    32     0
 1  system..SPEC                                  -    0      0
 2  max_rec_size                          i    32     10000020

                                                        (Continued)
```

```
(Continued)

stderr:

 2  const                                    i    32     FOLDED
 2  dynamic_arr                                 a   NOT STATIC
 2  var_with_addr                         i    32    ADDR CLAUSE
 2  system_name                              e    8      FOLDED
 2  storage_unit                             i    32     FOLDED
 2  memory_size                              i    32     FOLDED
 2  min_int                                  i    32     FOLDED
 2  max_int                                  i    32     FOLDED
 2  max_digits                               i    32     FOLDED
 2  max_mantissa                             i    32     FOLDED
 2  fine_delta                               f    64     FOLDED
 2  tick                                     f    64     FOLDED
 2  sig_status_size                          i    32     FOLDED
 2  byte_order                               e    8      FOLDED
 2  supports_invocation_by_address            b    8     FOLDED
 2  supports_preelaboration                  b    8      FOLDED
 2  make_access_supported                    b    8      FOLDED
 2  no_addr                                  h    32     FOLDED
 2  no_task_id                               i    32     FOLDED
 2  no_program_id                            i    32     FOLDED
 2  no_long_addr                             i    32     FOLDED
```

Figure 2-5 `a.symtab`

a.tags — create a source file cross reference of units

Syntax

```
a.tags [options] source_file ...
```

Arguments

options
Options to the a.tags command are:

-a
(append) Append to the tags file.

-B
(backward) Record backward searching patterns (?).

-D *identifier type value*
(define) Define an identifier of a specified type and value (used with the -P (preprocessor option)).

-F
(forward) Record forward searching patterns (*). [default]

-f *tags_file*
(file) Override the default output file, tags, with a file named *tags_file*.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

-P
(preprocessor) Invoke the Ada Preprocessor (a.app) for each file being processed.

-t
(types) Create tags for types.

-v
(vgrind) Generate an index with line numbers for vgrind(1) on the standard output.

-w
(warnings) Suppress warning messages.

`-x`
(cross) Generate an indexed list of all tags on the standard output.

source_file
Name(s) of the Ada source file(s) to create the `tags` file.

Description

`a.tags` makes a `tags` file from the specified Ada source(s). The operation is similar to the `ctags(1)` command with modifications for Ada-specific features.

Each line of the `tags` file lists the object name, the file in which it is defined, and search patterns for locating each object definition. Editors, such as `vi(1)` or `ex(1),` use the `tags` file to locate units and, if the `-t` option is used, to locate types as well.

Create the `tags` file with the command:

```
a.tags *.a
```

For example, to edit unit `END_PROG` without specifying the file that contains it, type the following command:

```
vi -t END_PROG
```

When using `ex(1)` or `vi(1)` with the `-t` option, the command line must contain the desired unit or type in the same case (upper or lower) as its occurrence in the source file.

Ada allows unit name overloading, and `a.tags` requires special conventions to access different units having the same name. Ada specifications are named by prefacing the Ada simple name with `s#`. Bodies are named with the unmodified Ada name. Stubs for separates are named by prefacing the Ada simple name with `stub#`.

Nested packages, subprograms, types, generics, and task definitions are always listed, both with their full Ada expanded name and with any tag prefaces added to the simple name. Simple names for nested units are listed only if the simple name is unique across all other tags. Thus, you can use the simple name if it is unique and can always use the full name.

Fully qualified overloaded names in a file are not differentiated. However, the tag identifies the correct file and repeated application of the search pattern finds the desired subprogram. The search pattern is generalized to match all versions of the overloaded subprogram; this generalization can cause the pattern to recognize things other than the desired unit. Identical fully qualified names across files are not handled.

The `-x` and `-v` options provide listings on the standard output; all other options refer to the file tags generated for use by `ex(1)` or `vi(1)`.

References

`ctags(1)`, *Solaris Developer Documentation*

2

a.vadsrc — *display versions and create library configuration file*

Syntax

```
a.vadsrc [options] [directory]
```

Arguments

directory
Name of the directory in which to create the library configuration file.

options
Options to the a.vadsrc command are:

`-i`
(interactive) Show all versions of SC Ada installed on the system and prompt for a selection.

`-t` *target*
(target) Create a .vadsrc file for a specific target machine.

`-v`
(verbose) Print the contents of any .vadsrc file in the current directory or in your HOME directory.

Description

If multiple SC Ada targets or versions are on the same system, a.vadsrc is useful to control the default version or target processor for which libraries are created.

With no option, a.vadsrc reports the installed SC Ada versions.

If -i (interactive) is used, the tool prompts for selection of an SC Ada version and creates a .vadsrc file in the current or specified directory. With this option, the contents of the VADS_END file are listed for each release present on the system. The VADS_END file contains information about the host system, host operating system, version number, and dates for each compiler on the system.

Files

`/etc/VADS`	SC Ada version reference file

References

"Syntax" on page 2-66

2

a.view — *provide aliases and history for C shell user*

Syntax

```
source a.view
```

Description

a.view defines a number of aliases that simplify and enhance the use of the basic SC Ada commands for C-shell users. The aliases allow a source filename to be set once, and thereafter, alias commands use it until it is changed. Similarly, a main unit name need be entered only once. (It need not be entered at all if it is the same as the last specified filename prefix.) Compilation and linking aliases enter history and timing information into the ada.history file.

To use the aliases without alteration, put the following line in the .login file. This line must appear at the beginning of scripts using these aliases.

```
source ada_location/bin/a.view
```

Aliases defined in a.view are summarized here. The term “tracking” indicates whether or not the main unit name is set to the same as the filename prefix.

Aliases

a
: Compile established filename, put errors in ./ada.errors/*filename* and history entry in ada.history.

ad
: Compile and run the debugger.

ah
: List last entry in ada.history.

al
: List established filename using PAGER (the environment pager set by the user). If PAGER is undefined, more is used.

ald
: Link the established main unit.

`am`
Execute `a.make` using filename specified in `sm` and put errors in `ada.errors/`*unit_name*`.m`.

`ao`
Compile and optimize code.

`av`
Edit the established filename with `vi(1)`.

`ax`
Execute the established main unit.

`axtime`
Execute a main unit and put timing entry into `ada.history`.

`e`
List erroneous lines and diagnostics from last compilation of established filename.

`el`
List established filename with diagnostics from last compilation interspersed. If `PAGER` is defined, it is used. Otherwise, `more` is used.

`ev`
Edit the established filename with `vi(1)` with diagnostics from last compilation interspersed.

`s` *name*
Set source filename prefix. If new working directory, then set tracking on. If tracking is on, then set main unit. If an extension is given, `s` sets the extension.

`sb` *name*
Set source filename prefix and main unit; set tracking on. If an extension is given, `sb` sets the extension.

`se` *name*
Set file extension.

`sm` *name*
Set main unit and set tracking off so that the main unit name does not change with `s` command.

`so` *level*
Set optimization level. Set the optimization level once.

`sp`
Print settings of filename prefix and main unit. The extension and optimization settings print also.

`vs`
List status for the last executed SC Ada command.

In the commands that take *name*, additional arguments are ignored and any trailing `.a` is stripped. (The prefix is desired for the filename.) In addition, only the tail component of name (the part following the last `/`) is used to set the main unit. (Main unit is an Ada unit name, which does not allow "`/`.")

The intention of this convention is to allow the use of filename substitution for easy specification of a full filename and main unit. For example, if the current directory contains the files `tasking_limit_test.a` (Ada source) and `tasking_limit_test.out` (executable object), and if no other files begin with `tas`, the command `s tas*` sets the filename prefix to `tasking_limit_test` and the main unit to the same string.

When the main unit name differs from the filename, use the `sm` command.

In all other commands, additional arguments are passed to the underlying SC Ada command. The following command causes the linker to search the `termlib` library in addition to standard libraries:

```
ald -ltermlib
```

Files

`ada.errors`	Directory containing error files from compilations
`ada.history`	History of compilations and results

Diagnostics

Warnings are produced if any `set` command is used in a non-SC Ada library directory, or if the specified source file does not exist in the library.

a.version — *Display if licensed for Multithreaded Ada*

Syntax

```
a.version
```

Description

If you are licensed for MT (Multithreaded Ada), `a.version` displays the following message:

```
"you are enabled to run sunpro.mpmt.ada"
```

If you are not licensed for MT Ada, `a.version` displays an error message.

2

a.which — *Find a compiled unit*

Syntax

```
a.which [options] unit_name
```

Arguments

options
Options to the a.which command are:

-b
(body) Give the location of the body.

-L *library_name*
(library) Operate in SC Ada library *library_name* (the current working directory is the default).

-s
(special) Convert *unit_name* to all uppercase.

-v
(verbose) Give the library search list.

unit_name
Ada unit name.

Description

Use a.which to list the name of the source file that defines the version of *unit_name* visible in the current SC Ada library. The program library search sequence can be printed also. The -b (body) option lists the source file location of the unit body. Without this option, the unit specification is located.

a.xref — *print cross-reference information for a given Ada unit or library*

Syntax

a.xref [*options*] [*unit_name* [*unit_name...*]] [-L *library_name*]

Arguments

options
Options to the a.xref command are:

-b
(body) Cross-reference the specified unit(s) body. [default]

-c *class_name*
(class) Cross-reference only those symbols that are of class *class_name*. See a description of the various class names under *Class*.

-i *class_name*
(ignore)Ignore all symbols that are of class *class_name*. See a description of the various class names under *Class*.

-L *library_name*
(library) If no units are specified, all units in library *library_name* are cross-referenced. If a unit or units are specified, a.xref looks in the specified library for the given units.

-n *symbol_name*
(name) Cross-reference only those symbols that have the name *symbol_name* (regular expressions are allowed in the symbol name).

-p
(predefined) Include predefined symbols (e.g., integer, boolean) in the cross-reference listing.

-s
(spec) Cross-reference the specified unit(s) specification.

unit_name
Name(s) of the Ada units to be cross-referenced.

Description

`a.xref` generates a cross-reference listing for units compiled with the `XREF INFO` directive set to `TRUE`. If this directive is not visible or is `FALSE` at the time the unit is compiled, `a.xref` generates a warning and is not able to list cross-reference information for that unit.

`a.xref` generates cross-reference listings for one or more Ada units or an entire library. If multiple units are cross referenced, the cross-reference information for all units is coalesced and given in one listing (i.e. the information is not grouped by unit).

unit_name and *symbol_name* can be specified with regular expressions. For example, `a.xref` "f*" cross-references all units beginning with the letter f. `a.xref -n "x*" -n "y?"` cross-references all symbols beginning with x and all symbols beginning with y that are followed by a single character.

Because Ada names are typically long and can make the cross-reference listing difficult to read, three separate tables are maintained: the Source File Table, the Unit Table and the Expanded Prefix Table. These tables are each sorted lexicographically and each entry is assigned a number. These table entry numbers are referenced in the cross-reference listing.

The Source File Table contains a list of all Ada source files that have definitions or references listed in the cross-reference listing, the date and time the file was last modified and the name of the directory that contains the file. The Unit Table contains a list of all compilation units that have definitions or references in the cross-reference listing, the unit type (either spec or body), the date and time the unit was last compiled, and the name of the Ada library that contains the unit. The Expanded Prefix Table contains a list of all prefixes of expanded names for symbols. For example, if the expanded name for symbol *variable_name* is *package_name.subprogram_name.variable_name*, the prefix in the Expanded Prefix Table is *package_name.subprogram_name*.

Cross Reference Information

The following information is given in the cross-reference listing:

Name

The simple name for the symbol.

Expanded Prefix

The number that corresponds to the appropriate entry in the Expanded Prefix Table for the symbol's expanded name, preceded by the letter E, e.g. E4 stands for the fourth entry in the Expanded Prefix Table. If the symbol has no expanded prefix (e.g. the symbol is for a compilation unit or a predefined type), this field contains a -.

Class

The class type for the symbol. Class types are as follows:

`dtype` — derived type
`gtype` — generic formal type
`stype` — subtype
`ltype` — limited private type
`ptype` — private type
`type` — type

`comp` — record component
`const` — constant
`discr` — discriminant
`enum` — enumeration literal
`entry` — task entry
`iter` — iteration variable
`nstmt` — named statement
`label` — label
`task` — task
`var` — variable

`ipar` — in parameter
`iopar` — in out parameter
`opar` — out parameter

`func` — function
`gfunc` — generic function
`ifunc` — function instantiation

`proc` — procedure
`gproc` — generic procedure
`iproc` — procedure instantiation

`pack` — package
`gpack` — generic package
`ipack` — package instantiation

`op` — operator
`gop` — generic operator

Type

If the class of the symbol is such that the symbol has a type, e.g. variable, constant, parameter, etc, the name of the symbol type is given. For functions, the return type is given. This name is in the form *En.type_name* where *En* is the corresponding entry into the Expanded Prefix Table.

Size

For constants, variables, record components and discriminants, parameters and types, the size (in bits) of that symbol is given.

Address/Offset

For global package variables, the suffix for the symbolic address for the variable is given. For example, given the following package:

```
package examples is
      var1 : integer;
      var2 : float;
end;
```

The symbolic address for the package variables could be:

```
var1 :   _A_examples..STATIC+010
var2 :   _A_examples..STATIC+018
```

In this case, the section name is `STATIC` and is abbreviated by `S`. Other possible section names and their abbreviations are:

"`STATIC..BODY`", abbreviated by "`S..B`"
"`CONST`", abbreviated by "`C`"
"`CONST..BODY`", abbreviated by "`C..B`"

Thus, for our example symbolic `address _A_examples..STATIC+10`, the cross-reference listing contains the string `S+010`. The Expanded Prefix field in the cross-reference listing contains the entry into the Expanded Prefix Table

corresponding to the package name (examples in this example). Catenating `_A_`, the expanded prefix name and the unabbreviated symbolic address suffix gives you the symbolic address for this field.

If this package has been linked into a program executable, the symbol `_A_examples..STATIC` is found in the executable image. The address of that symbol gives the base address of the package. The hex offset (+010 and +018 in our example) is added to the base address to determine the runtime address of the package variable.

Note that it is not always possible to determine the address of a constant. If possible, the compiler "folds" constants, using the static value of the constant wherever the constant is used. If the constant is folded, it is not part of the executable image and has no address. In these cases, the Address/Offset field of the cross-reference listing contains the word "`FOLDED`".

Definition

The locations of symbol definitions are of the form *line_number.Fn.Un*, where *Fn* is the entry into the Source File Table corresponding to the source file that contains the definition, and *Un* is the entry into the Unit Table corresponding to the unit that contains the definition.

References

Reference information is of the form *ref_type.line_number.Fn.Un*.

Where *ref_type* indicates the type of the reference, and can be one of:

`call` — subprogram/entry call
`goto` — goto
`inst` — instantiation
`raise` — raise
`set` — modification
`ref` — reference (non-modifying)

line_number is the source code line number of the reference, *Fn* is the corresponding entry into the Source File Table for the file containing the reference and *Un* is the entry into the Unit Table for the compilation unit that contains the reference.

The references are sorted first by file number; all references to this symbol from file #1 (if any) are first, followed by all references to this symbol in file #2 (if any), etc. They are then sorted by line number, in ascending order.

Note – If a field in the cross-reference listing does not apply to a particular symbol, that field contains a `-`.

Default Listing

By default, the cross-reference information is sorted by the symbols simple name. For two symbols with the same simple name, they are further sorted by expanded name prefix (e.g., for two symbols, `unit1.x` and `unit2.x`, unit1 x is listed first, since unit1 is lexicographically less than unit2.

For example, given the following two Ada source files in Figure 2-6:

```
example1.a:                                 example2.a
1 package example1 is                   1  with example1;
2     type int is new integer;          2  procedure example2 is
3     one : int := 1;                   3    xxx : example1.int;
4                                       4  begin
5     function add_one(x : integer)     5    xxx :=
6                 return int;           6     example1.add_one(0);
7  end;                                 7  end;
8
9  package body example1 is
10
11     function add_one(x : integer) return int is
12     begin
13         return int(x) + one;
14     end;
15  end;
```

Figure 2-6 Example of `a.xref` Source Code

The cross-reference listing for the command `a.xref -L` in a library containing `example1.a` and `example2.a` looks like:

```
***** SC Ada Cross-Reference Listing *****

***** date: Wed Apr 28 15:51:44 1993
***** args: /rc/ada_2/1/sup/a.xref -L .

Name                    EPT #   Class  Type                  Size  Addr/Offset
----                    -----   -----  ----                  ----  -----------
      Definition    References
      ----------    ----------
"+"                     E1      op     E1.int                 -     -
      2.F1          call.13.F1.U2
add_one                 E1      func   E1.int                 -     -
      5.F1          call.6.F2.U4
add_one                 E1      func   E1.int                 -     -
      11.F1
example1                -       pack   -                      -     -
      1.F1          ref.1.F2.U4
int                     E1      dtype  integer                32    -
      2.F1          ref.3.F1.U1  ref.6.F1.U1  ref.11.F1.U2  ref.13.F1.U2
                    ref.3.F2.U4
one                     E1      var    E1.int                 32    S+010
      3.F1          set.3.F1.U1  ref.13.F1.U2
x                       E2      ipar   integer               32    -
      5.F1          ref.13.F1.U2
x                       E2      ipar   integer               32    -
      11.F1
xxx                     E3      var    E1.int                 32    -
      3.F2          set.5.F2.U4

::::: Source File Table :::::

File No   File Name       Date of Last Mod               Directory Name
-------   ---------       ----------------              --------------

F1        example1.       Tue Apr 27 13:00:52 1993      /vc/carol/xref
F2        example2.a      Tue Apr 27 13:00:53 1993      /vc/carol/xref
#
```

```
::::: Unit Table :::::

Unit No   Unit Name                                  Type  Date Compiled
-------   ---------                                  ----  -------------
               Ada Library Name
               ----------------

U1        example1                                spec  Tue Apr 27 13:01:35 1993
               /vc/carol/xref
U2        example1                                body  Tue Apr 27 13:01:35 1993
               /vc/carol/xref
U3        example2                                spec  Tue Apr 27 13:01:38 1993
               /vc/carol/xref
U4        example2                                body  Tue Apr 27 13:01:38 1993
               /vc/carol/xref
::::: Expanded Prefix Table :::::

Prefix No    Expanded Prefix
---------    ---------------
E1           example1
E2           example1.add_one
E3           example2
```

Figure 2-7 Example of `a.xref` Output

"Huge and Mighty Forms that do not live like living men moved slowly through the mind by day and were a trouble to my dreams."

Wordsworth

Debugger Reference 3

3.1 Summary

Reference entries for each debugger command and concept (in *italic*) listed are on the following pages in alphabetical order.

`a`	Step one source line over calls
`address`	Address memory directly
`ai`	Step one machine instruction over calls
assignment (:=)	Assign a value to a variable, register or memory location(s)
`async`	Operate debugger in asynchronous mode
`ax`	Advance, pass the signal to the program
`b`	Set breakpoint at a line or beginning of a subprogram
`bd`	Set breakpoint after current subprogram
`bi`	Set breakpoint at machine instruction
`br`	Set permanent breakpoint at return
breakpoints	Control program execution
`bx`	Set a breakpoint when an Ada exception occurs
call stack	Display current state of program
`cb`	Move to the bottom frame of the call stack

(Continued)

`cd`	Move down on the call stack
command history	See line editing.
command syntax	Syntax of debugger commands
`core file`	Debugging a program that produced a core file
`cs`	Display the call stack
`ct`	Move to the top frame of the call stack
`cu`	Move up on the call stack
current frame	Current position on the call stack
current position	Current position in a source file
`d`	Delete breakpoints
disassembly	Display disassembled source code
display memory	Display raw memory
`e`	Move to a new source file
`edit`	Invoke the editor on a subprogram or file
examine	Display variables, files, debugger parameters, and so forth
`exit`	Terminate the debugger session
expressions	Arithmetic expressions in the debugger
files	Specify files to be used by the debugger
`g`	Continue executing the program
`gw`	Continue executing until a variable changes
`gx`	Continue execution and pass the signal to the program
help	Print help text
home position	Execution point in the current frame
inline expansions	Debugging inline expansions
invocation	Invoking the debugger
`l`	Display part of a source program
`lb`	List all currently set breakpoints
`li`	List disassembled instructions

(Continued)

line editing	Command history and line editing functions
line numbers	Move to a specified line
`lt`	List all active tasks
`lu`	List UNIX processes
overloading	Disambiguate overloaded names
`p`	Display the value of a variable or expression
procedure calls	Call subprograms from the program
`quit`	Terminate the debugger session
`r`	Run the program
`read`	Read debugger commands from a file
`reg`	List the current machine register contents
Return	Re-execute debugger command
`return`	Return from all called subprograms
`s`	Step one source line, into subprograms
screen mode	Screen-oriented debugger interface
search (? /)	Search forward/backward in the current file for a pattern
`set`	Set debugger parameters
`si`	Single step one machine instruction into program
`signals`	set/ignore signals
`stop`	Stop the debugger or program
strings	String printing, assignment, procedure calling, and use
`sx`	Single step, pass the signal to the program
`task`	Identify a new current task
terminal control	Catching program input/output
`vi`	Switch the debugger into screen mode
visibility rules	Determine which identifiers are visible at a breakpoint

(Continued)

`w`	List a group of source lines surrounding a line
`wi`	List disassembled code with original code
`x`	Monitor memory location(s)

a — *(advance) step one source line over calls*

Syntax

a — Pressing Return repeats

Description

a single steps to the next source line for which the compiler generated code, stepping *over* subprogram calls.

Other stepping commands are ai, s, si, and gw. The s command steps one source line *into* called subprograms.

If the program is not started or if it terminates, a starts the program, stepping one source line. For Ada programs, this steps over all the library unit elaborations.

Use two debugger parameters, alert_freq and step_alert, to track the number of instructions that are stepped. Using the default settings, a message is displayed after the first 1000 instructions are stepped (step_alert). After that, every 100 additional instructions stepped (alert_freq), generates a new message. In line mode, these messages are periods - one after the initial number of instructions are stepped with a new period displayed for each 100 additional instructions stepped.

In Screen Mode

The a command is directly supported in screen mode: type a to single step one source line over subprogram calls. It is not necessary to press Return in screen mode.

With safe mode set on, the screen mode command becomes aa.

The number of instructions stepped appears as a number on the dashed line separating the command and source window. This number is first displayed after the first 1000 instructions are stepped (step_alert). This number is incremented for every 100 instructions stepped after the initial display (alert_freq).

In instruction submode, a is interpreted as ai.

Note – The a command does not advance over entry calls, only procedure calls.

References

“ai — (advance instruction) single step machine code over calls” on page 3-8
“screen mode — screen-oriented debugger interface” on page 3-132
“Instruction and Source Submodes” on page 3-134
“set — set debugger parameters” on page 3-142
step one command, *SPARCompiler Ada User’s Guide*

address — address memory directly

Description

Some commands (`li`, `wi`, `bi`) take an address as a parameter, which is expressed as a hexadecimal number (with a leading 0).

In addition, you can type in an address using the form *number*. The number is either decimal or hexadecimal.

References

“display memory — display raw memory” on page 3-51

3

ai — *(advance instruction) single step machine code over calls*

Syntax

`ai` Pressing Return repeats

Description

`ai` single steps one machine instruction over call instructions.

Other stepping commands are `a`, `s`, `si`, and `gw`. The `si` command single steps one instruction *into* called subprograms.

If the program has not executed or if it terminates, `ai` starts the program, stepping one instruction. This relocates the current position from the main subprogram to the actual starting subprogram preceding your program.

In Screen Mode

Precede this command with `:` and follow with Return.

In instruction submode, `a` is interpreted as `ai`.

References

“Instruction and Source Submodes” on page 3-134
step one command, *SPARCompiler Ada User’s Guide*

:= — *assign a value*

Syntax

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

Arguments

address
A number that represents the address of a storage unit in memory. *address* is represented by a decimal number, hexadecimal number (with a leading zero) or by an expression (e.g., `$epb - 32`).

expression
A scalar expression or a string. If it is an arithmetic expression, its final value must be of `type INTEGER` or `type FLOAT`.

name
Any Ada or C expression that identifies a scalar object or a register name (preceded by a dollar sign). If name is a debugger keyword, precede it with a backslash or it results in a syntax error.

number
The number of bytes that are modified. If the value of *expression* is an integer, *number* must be 1, 2, 3 or 4. If the value of *expression* is a floating point number, *number* must be either 4 or 8. [Default: 4]

Description

This command modifies memory. After the memory or register is modified, a line of the form:

address: *new* / *old*

is printed. *address* is where the value is written; *new* is the value that is written; *old* is the previous value. For variable names, *address* is not printed.

In Screen Mode

Precede this command with : and follow with Return.

Examples

```
>0200034 := "this is it"
>date := "January 5, 1991"
>0f7ffeac1,8 := FLOAT_NUM * 3.5
>a.b(3) := 3
```

Note – No type checking is performed between the name and the expression.

References

"Summary" on page 3-1
"expressions — arithmetic expressions in the debugger" on page 3-65
"strings — string operations and support" on page 3-151
value assignment, *SPARCompiler Ada User's Guide*
expressions, section 4.1 in *Ada Reference Manual*

asynchronous debugging — *run the debugger in asynchronous mode*

Description

Asynchronous debugging is now supported in SC Ada. With asynchronous debugging, debugger can accept and execute commands while a program is running.

Note that the debugger is asynchronous only in the sense that it continues to accept commands after the program ha 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.

The original debugger still works in exactly the same way until it enters asynchronous mode. Do this in one of two ways: through the `-A` command line option or the `async` option to the debugger's `set` command.

Once you put the debugger into asynchronous mode, commands that set the program running no longer have the additional effect of making the debugger wait for a breakpoint or signal in the program before prompting for additional commands. These commands that set the program running are `g`, `gx`, `gw`, and `r`. The single stepping commands still wait for the single step to complete before accepting new commands. After setting the program running asynchronously, the debugger announces that the “Starting program running”, and prompts for a new command.

Once the program is running in asynchronous mode, most commands that would normally put the program in motion again have no effect, other than to produce the “Starting program running” message again. This applies to the stepping commands, also. Thus, if you type `g`, and get the “Starting program running” message, and then type `s`, the debugger does nothing to the program, and simply spits out the “Starting program running” message.

There are two exceptions: Whenever you type the `r` command, the debugger runs the program from the start, just like it used to. The other exception is the `gw` command. This command lets the program continue running, but it sets the specified data breakpoint.

If you enter screen mode while the program is running asynchronously, the debugger puts a `+` character in the prompt position on the screen dividing line, rather than an asterisk.

Most commands work the same way while the program is running as while the program is stopped. That is, the debugger tries to execute them, and produces error messages if it has problems. For example, you can set breakpoints while the program is running, and it will stop when and if it hits them. If you hit `Control-c` while the program is running, it will stop.

For several commands, the debugger has to do extra work in asynchronous mode. The `reg` command always reads up a new set of registers from the program. The `p`, `:=` (assignment), `cs`, `task`, and `gw` commands cause the debugger to first establish an "instantaneous" call frame environment in which to execute the command. Thus, successive executions of the `cs` command, will produce different results.

Of course, the debugger cannot really establish this environment instantaneously. Time passes between the moment when the debugger first reads up the `PC`, for example, and when it reads up the stack frame associated with the `PC`. Thus, information obtained from these commands can be unreliable. It's also possible that the debugger may be able to successfully complete the command the first time it's invoked and then have problems with the next invocation.

If you're debugging a tasking program, particularly an multiprocessor program, it's likely that the instantaneous environment that the debugger establishes is in an idle task in the non-Ada threads layer. In this case, you can use the task command to set up a program context in which the debugger can look up the names of variables. In asynchronous mode, this "task" is then automatically selected every time the debugger establishes an execution environment, rather than using the currently executing task.

Input/Output

If you're using set input `pty` (the default), the debugger's handling of program input is different in asynchronous mode. Normally, when you set the program running, the debugger stops looking for command input, but continues to read up characters from it's own standard input. It assumes that these keystrokes are directed towards the program's standard input and passes them along to the program.

In asynchronous mode, however, all keystrokes are assumed to be debugger commands whether or not the program is running. Two commands have been added to the debugger, `put` and `put_line`, to allow characters to be sent to

the program's input. They 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.

Although your `put` command writes characters to the program's standard input, there are no guarantees that the program will read them up. If there are unread characters when the program announces a breakpoint or signal, or when you switch the debugger to synchronous mode, the debugger flushes them and emits a warning message.

Another difference in I/O handling in asynchronous mode is that the debugger never switches to the `tty` settings of the program being debugged. Currently, the debugger resets its `tty` to look like the program's `tty` whenever the program is set in motion.

Program output in asynchronous mode is handled the same way as in synchronous mode. When the program produces output in asynchronous mode, however, it may be mixed in with the characters being typed as debugger command input or be mixed with the output of the debugger.

Any difficulties caused by this different behavior can be circumvented by creating an X window or screen, and then typing the `tty` shell command to find the new window's device name. This name can then be used in set input and set output commands to separate the program I/O from the debugger's I/O. For example, if you create a new screen on a machine named picard and type:

```
% picard : tty
```

and get

```
/dev/ttyq22
```

then type the debugger commands:

```
>set input /dev/ttyq22
>set output /dev/ttyq22
```

then all program I/O happens in the new screen and was not mixed with the debugger's I/O.

References

-A Option, “invocation — invoking the debugger” on page 3-76

“put — (put) send characters to program input” on page 3-114
“put_line — (put line) send characters to program input, append new line” on page 3-116
“set — set debugger parameters” on page 3-142
“task — print current task or choose a new current task” on page 3-156

ax — *(advance signal) advance, pass the signal to the program*

Syntax

```
ax
```

Description

When a signal exception occurs in a program being debugged, first it is passed to the debugger. The debugger announces the signal exception, the location at which it occurs, and stops, waiting for commands. To continue advancing as though the signal does not occur, use `a` or `ai`. To continue advancing and pass the signal to the program, use the `ax` command. This is useful in debugging programs that do explicit signal handling.

Use two debugger parameters, `alert_freq` and `step_alert`, to track the number of instructions that are stepped. Using the default settings, a message is displayed after the first 1000 instructions are stepped (`step_alert`). After that, every 100 additional instructions stepped (`alert_freq`), generates a new message. In line mode, these messages are periods - one after the initial number of instructions are stepped with a new period displayed for each 100 additional instructions stepped.

This command cancels the Return key memory.

In Screen Mode

Precede this command with `:` and follow with Return.

The number of instructions stepped appears as a number on the dashed line separating the command and source window. This number is first displayed after the first 1000 instructions are stepped (`step_alert`). This number is incremented for every 100 instructions stepped after the initial display (`alert_freq`).

References

“a — (advance) step one source line over calls” on page 3-5
“set — set debugger parameters” on page 3-142
“signals — set / ignore signals” on page 3-148

3

b — *(break) break at a line or beginning of a subprogram*

Syntax

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

Arguments

commands

A sequence of one or more debugger commands that execute automatically when the breakpoint is reached. Use the following format:

```
begin commands end
```

You can enter the commands on the same line, separated by semicolons, or enter each command on its own line as long as the first `begin`, `then` or `else` is on the same line as the `b` keyword. An execution command (`a`, `ai`, `s`, `si`, `g`, `gw`, `r`) in the commands sequence must be last. The second method (separate lines) is recommended. As each command of commands is entered on its own line, the debugger prompts with `??` for each new command until the sequence terminates with `end` (or an `else` in the case of an `if...then...else`).

expression

An Ada expression that is evaluated each time the breakpoint is reached. This evaluation takes place in the environment of the location of the breakpoint. If *expression* is `FALSE` (0), the breakpoint is not announced and the program continues. If *expression* is `TRUE` (nonzero), the breakpoint is announced.

Use the `if` statement to set a conditional breakpoint that conditionally executes debugger commands when the breakpoint is reached. *expression* is evaluated each time the breakpoint is reached. If it is `TRUE`, the breakpoint is announced and any commands in a `then` clause execute. If the *expression* is `FALSE`, the commands in an else clause execute.

line

Line number at which the breakpoint is set. Typically, *line* is a decimal number; however, all the options specified in the line number section of this reference are supported.

subprogram

Name of an Ada subprogram, task or package. If the subprogram name given is a simple identifier, all subprograms, tasks, and packages (with an elaboration subprogram) in the program are visible.

The subprogram can be given as an expanded name (starting with `standard` if desired). The leftmost simple name of an expanded name must be directly visible from the current context or must be the name of a library unit.

If the subprogram name has multiple definitions, it is overloaded. The debugger prints a diagnostic showing the alternatives. Retype the `b` command attaching a '1, '2, ... suffix (matching an alternative shown in the diagnostic) to the subprogram name to disambiguate it. Alternatively, sometimes it is sufficient to use an expanded name to disambiguate it.

task

Task number of the task at which the breakpoint is announced. Obtain the task number with the `lt` command. The breakpoint is announced only for the specified task.

Description

This command sets a breakpoint. You can set breakpoints at a line in the current file, at the beginning of a subprogram or in a task. To set a breakpoint at a line in another file or subprogram, use the `e` (enter) command to locate the correct source file first.

If *subprogram* or *line* is not specified, a breakpoint is set at the current position by using the line part of the current position three-part identifier (file, line number, and instruction address). In line mode, the current position is marked with `<`.

You can set a breakpoint only in an instance of a generic. You cannot enter the source file of a generic, set a breakpoint, and have that breakpoint exist in all instances of the generic.

Each breakpoint set with b has a number. The number is displayed when the breakpoint is reached or when the `lb` command lists all breakpoints. Use the number to delete individual breakpoints with the `d` command.

In Screen Mode

The current position is the line in the source window that contains the cursor. When the cursor is located in the source window, typing `b` is the same as a line mode `b` command without any parameters. That is, a breakpoint is set on the line containing the cursor. The screen-mode `b` command is acknowledged immediately by the appearance of `=` to the left of the line on which the breakpoint is set.

To set a breakpoint at the beginning of a subprogram while in screen mode, position the cursor on top of any letter of any occurrence of the name of a subprogram and press `B`. This has the same effect as typing `b` subprogram in line mode. An acknowledgment message is displayed at the bottom of the screen to indicate that the breakpoint has is set.

If the subprogram name is overloaded, the debugger prints a diagnostic showing the alternatives. Retype the `B` command preceding it with a number (matching an alternative shown in the diagnostic) to disambiguate it. That is, typing a number *n* before the `B` command has the same effect as typing `b subprogram'n` in line mode.

To set a conditional breakpoint in screen mode, type `":"` and enter the line-mode command.

In instruction submode, `b` is interpreted as `bi`.

Examples

```
>b 537                          set breakpoint at line 537 of
                                current source file)
>b SORT_STAMPS                  (set breakpoint at subprogram
                                SORT_STAMPS)
>b SORT when FIRST = "January"(conditional breakpoint)
>b MONTH_NO begin               (command block)
??p month
??p date'string
??g
??end
```

References

“breakpoints — control program execution” on page 3-26
“current position — current position in a source file” on page 3-46
“d — (delete) delete breakpoints” on page 3-47
“e — (enter) move to a new source file” on page 3-60
“Instruction and Source Submodes” on page 3-134
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
“lt — (list tasks) list all active tasks” on page 3-93
“visibility rules — determine visible identifiers at a breakpoint” on page 3-160
command blocks, set conditional breakpoints, and set breakpoints, *SPARCompiler Ada User’s Guide*
expanded names, section 4.1.3(13) in *Ada Reference Manual*

3

bd — *(break down) break after current subprogram*

Syntax

```
bd [of task] [begin commands end]
bd [of task] when expression
bd [of task] if expression then commands [else commands] end [if]
bd [of task] if expression else commands end [if]
```

Arguments

commands

A sequence of one or more debugger commands that execute automatically when the breakpoint is reached. Use the following format:

```
begin commands end
```

You can enter the commands on the same line, separated by semicolons, or enter each command on its own line as long as the first `begin`, `then` or `else` is on the same line as the `b` keyword. An execution command (`a`, `ai`, `s`, `si`, `g`, `gw`, `r`) in the commands sequence must be last. The second method (separate lines) is recommended. As each command of commands is entered on its own line, the debugger prompts with `??` for each new command until the sequence terminates with `end` (or an `else` in the case of an `if...then...else`).

expression

An Ada expression that is evaluated each time the breakpoint is reached. This evaluation takes place in the environment of the location of the breakpoint. If *expression* is `FALSE` (0), the breakpoint is not announced and the program continues. If *expression* is `TRUE` (nonzero), the breakpoint is announced.

Use the `if` statement to set a conditional breakpoint that conditionally executes debugger commands when the breakpoint is reached. *expression* is evaluated each time the breakpoint is reached. If it is `TRUE`, the breakpoint is announced and any commands in a `then` clause execute. If the *expression* is `FALSE`, the commands in an else clause execute.

task

Task number of the task in which the breakpoint is announced. The `lt` command obtains the task number. The breakpoint is announced only for the specified task.

Description

`bd` sets a breakpoint in the subprogram that called the current subprogram, that is, one frame down from the current frame. The breakpoint is reached *immediately* when the current entity returns. The current subprogram is the one represented by the current frame. *Immediately* means that the `bd` breakpoint is set in the first machine instruction following the current subprogram return. This may not be on a source statement boundary. The breakpoint is removed automatically when it is reached. To get to the beginning of the next statement, use the `a` command.

Usually, `bd` is used for stopping at the end of the current subprogram after entering it with the `s` or `si` command.

The simplest form of this command, `bd`, is used most often. But, like the `b` and `bi` commands, you can specify the `bd` command for a particular *task* with the `of` *task* clause. A set of commands at the breakpoint, can be automatically executed at a breakpoint with a `begin` *commands* `end` block. Set a conditional `bd` breakpoint by using a `when` or `if` statement.

In Screen Mode

Precede this command with `:` and follow with Return.

References

“b — (break) break at a line or beginning of a subprogram” on page 3-16
“breakpoints — control program execution” on page 3-26
“cd — (call down) move down on the call stack” on page 3-34
“cs —. (call stack) display the call stack” on page 3-40
“d — (delete) delete breakpoints” on page 3-47
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
“lt — (list tasks) list all active tasks” on page 3-93
break down, *SPARCompiler Ada User’s Guide*

3

bi — *(break instruction) break at machine instruction*

Syntax

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

Arguments

commands

A sequence of one or more debugger commands that execute automatically when the breakpoint is reached. Use the following format:

```
begin commands end
```

You can enter the commands on the same line, separated by semicolons, or enter each command on its own line as long as the first `begin`, `then` or `else` is on the same line as the `b` keyword. An execution command (`a`, `ai`, `s`, `si`, `g`, `gw`, `r`) in the commands sequence must be last. The second method (separate lines) is recommended. As each command of commands is entered on its own line, the debugger prompts with `??` for each new command until the sequence terminates with `end` (or an `else` in the case of an `if...then...else`).

expression

An Ada expression that is evaluated each time the breakpoint is reached. This evaluation takes place in the environment of the location of the breakpoint. If *expression* is `FALSE` (0), the breakpoint is not announced and the program continues. If *expression* is `TRUE` (nonzero), the breakpoint is announced.

instruction

Address of a machine instruction. The address is a hexadecimal number (with a leading 0).

task

Task number of the task at which the breakpoint is announced. The `lt` command obtains the task number. The breakpoint is announced only for the specified task.

Description

`bi` sets a breakpoint at a specific machine instruction. Use `li` or `wi` to display the instructions (disassembly), and pinpoint exactly where to set the instruction breakpoint.

Set `bi` breakpoints for a particular task using the `of` *task* option.

Execute a block of debugger commands automatically when the breakpoint is reached by appending a `begin-end` block. Set a conditional `bi` breakpoint with a `when` or `if` statement.

Each breakpoint set with `bi` has a number. The number is displayed when the breakpoint is reached or when the `lb` command lists all breakpoints. Use the number to delete individual breakpoints with the `d` command.

In Screen Mode

Precede this command with `:` and follow with Return.

In instruction submode, `b` is interpreted as `bi`.

An `=` indicates that a breakpoint is set on that line.

References

“b — (break) break at a line or beginning of a subprogram” on page 3-16
“breakpoints — control program execution” on page 3-26
“d — (delete) delete breakpoints” on page 3-47
“Instruction and Source Submodes” on page 3-134
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
“li — (list instructions) list disassembled instructions” on page 3-84
“lt — (list tasks) list all active tasks” on page 3-93
“wi — (window instruction) list disassembled and original code” on page 3-162
set breakpoint at instruction, *SPARCompiler Ada User’s Guide*

3

br — *(break return) set permanent breakpoint at return*

Syntax

```
br [of task] [begin commands end]
br [of task] when expression
br [of task] if expression then commands [else commands] end [if]
br [of task] if expression else commands end [if]
```

Arguments

commands

A sequence of one or more debugger commands that execute automatically when the breakpoint is reached. Use the following format:

```
begin commands end
```

Enter the commands on the same line, separated by semicolons, or enter each command on its own line as long as the first `begin`, `then` or `else` is on the same line as the `b` keyword. An execution command (`a`, `ai`, `s`, `si`, `g`, `gw`, `r`) in the commands sequence must be last. The second method (separate lines) is recommended. As each command of commands is entered on its own line, the debugger prompts with `??` for each new command until the sequence terminates with `end` (or an `else` in the case of an `if...then...else`).

expression

An Ada expression that is evaluated each time the breakpoint is reached. This evaluation takes place in the environment of the location of the breakpoint. If *expression* is `FALSE` (0), the breakpoint is not announced and the program continues. If *expression* is `TRUE` (nonzero), the breakpoint is announced.

task

Task number of the task at which the breakpoint is announced. The `lt` command obtains the task number. The breakpoint is announced only for the specified task.

Description

The `br` command sets a permanent breakpoint (as opposed to the `bd` command) at the last-executed (return) instruction of the current subprogram. The breakpoint is not deleted after it is hit.

On RISC machines which have delay slots, if the return instruction is followed by an instruction in the delay slot, the break is set in the delay slot if possible. If the CPU does not support setting breaks in delay slots, it is set the break in the previous instruction.

`br` does not work for inlines. If you set a `br` in an inline, the break is set at the return from the procedure which contains the inline.

`br` also does not work for C.

When the code generator puts out more than one return instruction in a subprogram, setting one `br` puts a break at every return statement. If you already have another kind of break at one or more of the return instructions, `br` fails.

References

“bd — (break down) break after current subprogram” on page 3-20
“breakpoints — control program execution” on page 3-26

3

breakpoints — control program execution

Description

A breakpoint is a location (a point) in a program where the debugger is instructed to suspend (to break) the program execution. The debugger has five commands that set breakpoints: `b`, `bi`, `bd`, `br`, and `bx`. When execution commands are given (`a`, `ai`, `g`, `gx`, `r`, `s`, or `si` command), the debugger ensures that when execution reaches set breakpoints, the program "breaks" — that is, the program stops executing.

While the program is running, the debugger does not accept commands, but input to the program or the debugger can be typed ahead. When the program reaches a breakpoint, its execution is suspended. In line mode, the debugger announces the breakpoint, as shown in the following example.

```
[2] stopped at "/vc/sbq/tst3/hs.a":95 in check
95  i : integer := 256;
```

The first line of the announcement begins with a breakpoint number in brackets ([2] in the example above). The remainder of the announcement message pinpoints the location of the breakpoint. The name in quotes is the name of the source file and the number following the colon is the line number in that file where the program stopped. The name following the word *in* is the name of the subprogram (package or task) that contains the source line.

If you attempt to set a breakpoint at a passive task, passive interrupt entry or nonpassive interrupt entry, a label corresponding to the entity is displayed (`PASSIVE ACCEPT`, `PASSIVE ISR` or `NON_PASSIVE ISR`).

You can define a maximum of 64 breakpoints at a time. In addition to your 64 breakpoints, a breakpoint is set automatically by Ada at a procedure in the Ada runtime system called `SLIGHT_PAUSE`. This enables the debugger to recognize when a program is about to terminate because of an unhandled exception.

You can set breakpoints in a generic instantiation. To do so, enter a subprogram in the instance (e.g. `e foo`) and set the breakpoint(s). You can also set breakpoints in a generic instantiation if you are currently executing code inside the instance (e.g. you've stepped into the subprogram of an instance).

If you position yourself in the source of a generic unit through some other method than described above, you cannot set breakpoints in the source of the generic unit. Attempting to do this causes an error message to be displayed: `no instructions at this line`. This happens, for example, if you position yourself in the source of the generic unit by using its source file name as the argument to the e command.

In Screen Mode

Breakpoints are not announced explicitly in screen mode. The debugger scrolls the source window, if necessary, to ensure that the line containing the breakpoint is on the screen. Since the breakpoint is the current home position, it is marked with an *, as well as =, and the cursor is on the line.

```
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);
```

As usual, the debugger signals that it is waiting for input by putting an * in column 1 or 2 of the dashed line. No * is in either of these columns while the program is running, and the debugger does not act immediately on new commands.

Typing ahead in screen mode is not recommended. Input to the program and input to the debugger are easily confused on the screen, and usually, typing ahead places unwanted command characters in the source window. (Use Control-r to refresh the screen.)

References

“b — (break) break at a line or beginning of a subprogram” on page 3-16
“bd — (break down) break after current subprogram” on page 3-20
“bi — (break instruction) break at machine instruction” on page 3-22
“br — (break return) set permanent breakpoint at return” on page 3-24
“bx — (break exception) break when an Ada exception occurs” on page 3-28
“d — (delete) delete breakpoints” on page 3-47
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
breakpoint commands and implicit breakpoints, *SPARCompiler Ada User’s Guide*

3

bx — *(break exception) break when an Ada exception occurs*

Syntax

```
bx [NOT] [exception] [of task] [begin commands end]
bx [NOT] [exception] [of task] when expression
bx [NOT] [exception] [of task] if expression then commands
                               [else commands] end [if]
bx [NOT] [exception] [of task] if expression else commands end [if]
```

Arguments

commands

A sequence of one or more debugger commands that execute automatically when the breakpoint is reached. Use the following format:

```
begin commands end
```

You can enter the commands on the same line, separated by semicolons, or enter each command on its own line as long as the first `begin`, `then` or `else` is on the same line as the `b` keyword. An execution command (`a`, `ai`, `s`, `si`, `g`, `gw`, `r`) in the commands sequence must be last. The second method (separate lines) is recommended. As each command of commands is entered on its own line, the debugger prompts with `??` for each new command until the sequence terminates with `end` (or an `else` in the case of an `if...then...else`).

exception

An Ada exception.

expression

An Ada expression that is evaluated each time the breakpoint is reached. This evaluation takes place in the environment of the location of the breakpoint. If *expression* is `FALSE` (0), the breakpoint is not announced and the program continues. If *expression* is `TRUE` (nonzero), the breakpoint is announced.

Use the `if` statement to set a conditional breakpoint that conditionally executes debugger commands when the breakpoint is reached. *expression* is evaluated each time the breakpoint is reached. If it is `TRUE`, the breakpoint is announced and any commands in a `then` clause execute. If the *expression* is `FALSE`, the commands in an `else` clause execute.

NOT
All exceptions except the one listed are announced.

task
Task number of the task in which the breakpoint is announced. The lt command obtains the task number. The breakpoint is announced only for the specified task.

Description

bx sets a breakpoint that is reached when the named *exception* occurs (for example, bx constraint_error). If the *exception* field is omitted, a breakpoint is announced when any exception occurs.

For example:

```
>bx constraint_error
```

Like b, bd, and bi breakpoints, set bx breakpoints for a particular task, using the of *task* option and follow it with a block of debugger commands to execute when the breakpoint is reached.

Each breakpoint set with bx is given a number. The number displays when the breakpoint is reached or when all breakpoints are listed by the lb command. Use the number to delete individual breakpoints with the d command.

The use of the bx NOT command has some restrictions. bx and bx NOT *exception* cannot be used in the same task. However, bx *exception* and bx NOT *other_exception* can be used in the same task. Thus, the following are legal:

```
bx constraint_error
bx not numeric_error
```

The following can be used to ignore several exceptions in the same task.

```
bx not constraint_error of task b
bx not numeric_error of task b
```

This option can be used to set breaks which would otherwise conflict in separate tasks.

```
bx of task a
bx constraint_error of task b
bx not constraint_error of task c
```

In Screen Mode

Precede this command with : and follow with Return.

References

“breakpoints — control program execution” on page 3-26
“d — (delete) delete breakpoints” on page 3-47
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
“lt — (list tasks) list all active tasks” on page 3-93
“read — read debugger commands from a file” on page 3-121
set breakpoint at exception, *SPARCompiler Ada User’s Guide*

call stack — display current state of program

To represent the current state of the program being debugged, the debugger 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 have been called but have not returned to their caller. When the program executes, the subprogram that is executing currently 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 `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`). Changing the current level on the call stack changes the variables that are directly visible.

```
> 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()
```

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 changes only when moving up and down on the call stack. When the execution position changes, the viewing position changes to match it.

References

"cb — (call bottom) move to the call stack bottom frame" on page 3-33
"cd — (call down) move down on the call stack" on page 3-34
"cs —. (call stack) display the call stack" on page 3-40
"ct — (call top) move to the call stack top frame" on page 3-42
"cu — (call up) move up on the call stack" on page 3-43
"current position — current position in a source file" on page 3-46
"home position — execution point in current frame" on page 3-73

cb — *(call bottom) move to the call stack bottom frame*

Syntax

```
cb
```

Description

cb moves both the current position and the current frame to the bottom or lowest frame on the call stack. For Ada programs, this is the frame corresponding to the main program.

The call stack is represented with the breakpointed subprogram at the top of the stack. Use cs to display the call stack.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the cs command displays for the frame: to the left is the frame number, followed by the name of the subprogram, package or task that the frame represents, followed by the names and values of actual parameters, if any. Inline frames are marked with a + character immediately after the frame number.

Use cb to see local variables and parameters in the bottom frame of the call stack.

In Screen Mode

Type cb to move to the bottom of the stack in screen mode. Pressing Return afterward is not necessary.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

References

"cs —. (call stack) display the call stack" on page 3-40
call stack bottom, *SPARCompiler Ada User's Guide*
main program, section.10.1(8) in *Ada Reference Manual*

3

cd — *(call down) move down on the call stack*

Syntax

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

Arguments

name
Name of a frame on the call stack.

number
Number of a frame on the call stack. If *number* is 0, `cd` moves the current position to the home position in the current frame. This is useful after moving away from the current frame, for example, in a new file with the `e` command. The `*` command is a synonym for `cd 0`.

Description

`cd` moves the current position down one frame on the call stack. If *name* or *number* is specified, the current position moves down to the next frame on the call stack with that name or number. (This moves in either direction — `up` or down.) The `cs` command displays the contents of the stack, starting with the current frame and shows the frame numbers.

The call stack is represented with the breakpointed subprogram being at the top of the stack. The call stack is displayed with the `cs` command, starting with the current frame. Each frame is shown with its number. `cd` moves down to the frame of the procedure that called the top frame.

After the `cd` command executes, the new frame becomes the current frame and the line executing in that frame becomes the home position. The line containing the current home position is marked with `*` when it is displayed on the screen in screen mode or by a display command (`l`, `li`, `w`, `wi`) in line mode.

After executing the `cd` command, the debugger prints a one-line display corresponding to the new current frame. This line is the same line that the `cs` command displays for the frame, to the left is the frame number, followed by the name of the subprogram, package or task that the frame represents, followed by the names and values of actual parameters, if any exist. Inline frames are marked with a `+` character immediately following the frame number.

To display the values of local variables and parameters of a procedure currently active on the call stack, move the *current frame* to that frame on the stack. This makes the local variables visible.

In Screen Mode

Type `cd` in screen mode. Pressing Return afterward is not necessary.

The *name* option cannot be used directly in screen mode, but the *number* option is supported. However, the *number* parameter must precede `cd` (unlike line mode where the *number* follows `cd`). For example, to move to stack frame 5 on the call stack in screen mode, type `5cd`.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

References

“cs —. (call stack) display the call stack” on page 3-40
“current frame — current position on the call stack” on page 3-45
“inline expansions — debugging inline expansions” on page 3-74
“line numbers — move to a specified line” on page 3-91
“visibility rules — determine visible identifiers at a breakpoint” on page 3-160
call stack down, *SPARCompiler Ada User’s Guide*

3

command syntax — syntax of debugger commands

Most debugger commands are of the form: keyword *parameters.*

In line mode, debugger keywords are not case sensitive. For example `b 357`, set breakpoint at line 357 can be entered as `B 357`. However, certain identifiers, pathnames, and C variables for example, are case sensitive.

In screen mode, due to several special cases, debugger keywords are case sensitive. Examples:

`B` (break at procedure) vs. `b` (break at line)

`C` (change window size) vs. `cs`, `ct`, `cu`, `cd`, `cb` (call stack commands)

`G` (move to the bottom of the view) vs. Control-g (print the file and line)

`H` (help lines) vs. `h` (move cursor left)

`P..p` (to print dotted names) vs `p` (print simple name)

In line mode, enter a list of commands on a single line, separated by semicolons (except commands with parameters that the shell interprets). In screen mode, precede a list of commands with a colon.

Line-mode commands execute when Return is pressed. The single exception is a breakpoint command followed by a block of commands as illustrated here. The breakpoint is set after the final Return after the end keyword.

The debugger uses Ada syntax for comments: characters between the double dash (--) and Return are ignored.

While in line mode, Return repeats 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 repeats in this way is marked in the documentation with the phrase ‘Return repeats.' In screen mode, repeat the previous command line with a period. To require certain single-letter commands (`a`, `g`, `r`, and `s`) to be typed twice for safety, set the debugger safe parameter to on (`set safe on`).

You can enter a number as either decimal or hexadecimal. If hexadecimal, begin the number 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: precede a hexadecimal number that has a leading hexadecimal digit (F2) with zero (0F2) or it is interpreted as an identifier.

When a name can be either a debugger keyword or a variable name, the debugger interprets it as a keyword. Precede the name with a backslash (\) to force the debugger to interpret it as a variable.

Frequently, the documentation for the debugger refers to the Interrupt key (Intr). The system command `stty` enables this function to be assigned to any convenient key (often Control-c). Intr halts the program being debugged, if it is running or the debugger current operation. The debugger responds immediately to Intr with a prompt for the next command.

Control characters (e.g., Control-z) have their usual meaning.

References

“stop — stop the debugger or program” on page 3-150
command syntax, *SPARCompiler Ada User’s Guide*
`stty(1)`, `tty(4)`, *Solaris Developer Documentation*

3

core file — debugging a program that produced a “core” file

Description

UNIX programs produce “core” files when certain signals occur that are neither caught nor ignored. The core file contains a complete snapshot of the program's state at the time the signal occurred. See your UNIX signal documentation for the list of signals that cause a core file to be produced.

The debugger automatically reads a core file if it exists in the directory the debugger was invoked from. Alternatively, the path name of a core file can be given explicitly with the `-C core_file_name` debugger option.

After reading the core file the debugger informs you that it is using the core file image:

```
[using memory image in “core” file from program “foo”]
```

It then announces the signal that caused the core file to be produced. This announcement looks identical to the announcement produced if the signal had occurred while running the program from the debugger. It includes the signal name, the source file and line, and the subprogram name where the signal occurred.

You can then use any of the debugger's commands to interrogate the state of the program at the point it produced the core file. For example, you can display the call stack, display variable values, examine registers, list task information, etc.

You cannot continue the program from its core file state. All execution commands (a, ai, s, si, g, gw, gx, r) cause the program to be restarted. You can also type 'set run' to reset the program to its normal startup state (e.g. you might do this if you were not really interested in the core file state).

In some situations, the debugger produces warning and/or error messages about a core file. If error messages are produced the core file is ignored. Here are some of the more common messages:

```
Warning: program name from “core” file does not match
executable name
```

The program executable's path name stored in the core file does not match the program executable's path name given on the debugger's invocation line. This does no necessarily indicate that the core file was not produced by the given program. Therefore, this is just a warning.

```
=> “core” file ignored .. it is older than the executable
```

The file modification time of the program executable is more recent than the file modification time of the core file. Therefore, the core file was not produced by the executable.

```
=== memory address is out-of-bounds: 02a520
=> cannot read opcode at PC 02a520
- (was the “core” file produced by this program?)
=> “core” file ignored
```

The value of the PC register recorded in the core file does not correspond to a valid text address in the program executable (nor does it correspond to a data address in the core file memory image). Therefore, the core file was not produced by the executable

3

cs —. *(call stack) display the call stack*

Syntax

```
cs [reg] [number] [of task]
```

Arguments

number
Number of frames to display.

`reg`
Provide hexadecimal dump information.

task
Task identifier. This identifier is in the ADDR column of output from the `lt` command.

Description

`cs` displays the call stack, starting with the current frame. *number* specifies the topmost number of frames to be displayed (`0` means all). The `of` *task* clause displays the call stack for a specific task with the task identifier.

```
# 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 leftmost column of the display contains a number for each frame. Use this number with the `cd` and `cu` commands. Frames that correspond to inline expansions are marked with a `+` character immediately after the frame number.

The debugger recognizes passive tasks, passive interrupt entries, and nonpassive interrupt entries, and indicates these entities on the call stack. The following labels are used:

`PASSIVE ACCEPT`	Current frame is an accept of a passive task entry
`PASSIVE ISR`	"Wrapper" procedure the compiler puts into the exception vector table to call a passive interrupt entry
`NON_PASSIVE ISR`	"Wrapper" procedure the compiler puts into the exception vector table to call an interrupt entry for a nonpassive task

Other stack commands are `cb`, `cd`, `cu`, and `ct`.

If the keyword `reg` is used, an extra three-line hexadecimal dump is provided per frame. The first line shows the values of the PC, FP, and AP registers. (On most machines, the FP and AP registers are the same register). The second line displays a row of 32-bit words, starting with the word pointed to by FP and then moving down to lower addressed words. The third line displays a row of 32-bit words, starting with the word pointed to by the AP and then moving up, to higher addressed words.

In Screen Mode

Type `cs` in screen mode; pressing Return afterward is not necessary. However, in screen mode the *number* parameter precedes the `cs` (unlike line mode where the number follows `cs`). For example, to see the top three frames of the call stack in screen mode, `type 3cs`.

References

"inline expansions — debugging inline expansions" on page 3-74
"lt — (list tasks) list all active tasks" on page 3-93
call stack and display call stack, *SPARCompiler Ada User's Guide*

ct — *(call top) move to the call stack top frame*

Syntax

```
ct
```

Description

`ct` moves the current frame and current position to the top of the stack that is also the breakpointed frame. When a process breakpoints, the current position is initialized to be the top of the stack. Use `cs` to display the call stack.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the `cs` command displays for the frame, to the left is the frame number, followed by the name of the subprogram, package or task that the frame represents, followed by the names and values of actual parameters, if any exist. Inline frames are marked with a + character immediately following the frame number.

Local variables and parameters of the subprogram at the top of the call stack are made visible with `ct`.

In Screen Mode

Type `ct` to move to the top of the stack in screen mode; pressing Return afterward is not necessary.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

References

“cs —. (call stack) display the call stack” on page 3-40
“inline expansions — debugging inline expansions” on page 3-74
call stack top, *SPARCompiler Ada User’s Guide*

cu — *(call up) move up on the call stack*

Syntax

cu [*name*|*number*]

Arguments

name
Name of a frame on the call stack.

number
Number of frames to display.

Description

cu moves up one frame on the call stack. If *name* or *number* is provided, cu moves up to the next frame on the call stack with that name or number.

The new frame becomes the current frame and the line executing in that frame becomes the current home position, marked with *.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the cs command displays for the frame, to the left is the frame number, followed by the name of the subprogram (package or task) that the frame represents, followed by the names and values of actual parameters, if any exist. Inline frames are marked with a + character immediately following the frame number.

Use cu 0 or * to move to the home position in the current frame. This is helpful after moving into a new file using the e command.

In Screen Mode

Type cu in screen mode; pressing Return afterward is not necessary. The *name* option cannot be used in screen mode, but the *number* option is supported; however, the *number* parameter precedes cu instead of following it as in line mode. For example, to move to stack frame 5 on the call stack in screen mode, type 5cu.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

References

“inline expansions — debugging inline expansions” on page 3-74
“line numbers — move to a specified line” on page 3-91
call stack down, *SPARCompiler Ada User’s Guide*

current frame — current position on the call stack

Description

The current frame is the current position on the call stack. When a breakpoint is announced, the current frame is always set to the breakpointed subprogram, package or task. This frame is the topmost frame of the call stack. In moving up and down the call stack (`cd` and `cu` commands), the current frame is the frame moved to most recently. The current frame is always the topmost frame of a `cs` command listing.

Each frame has a current instruction associated with it. For the topmost non-inline frame, this is the instruction that executes next. For all other non-inline frames, it is the call instruction that called the next higher non-inline frame. For inline frames, the current instruction is the same as the current instruction of the next lower non-inline frame. This instruction address, plus the source line and filename that generated it, constitute the home position for that frame. In showing disassembled instructions, the home position (marked with `*`) is displayed next to the current instruction for the current frame.

The current frame can be changed only with the call stack commands (`cu`, `cd`, `ct`, and `cb`) or by running the program to a new breakpoint.

The current frame plays a central role in establishing what program names are currently visible at any point during a debugging session.

When a breakpoint is reached or a call stack command executes, the current position is always set equal to the home position of the current frame. Examining the program source or instructions changes the current position. Return to home position in the current frame by typing

```
*
```

This command is described in the section on line numbers. It has the effect of moving the current position back to the home position of the current frame.

References

“current position — current position in a source file” on page 3-46
“inline expansions — debugging inline expansions” on page 3-74
“line numbers — move to a specified line” on page 3-91
“visibility rules — determine visible identifiers at a breakpoint” on page 3-160

current position — current position in a source file

Description

The current position is represented by a three-part identifier consisting of file, line number, and instruction address. This identifier represents the present location in the source program.

The `e` command is the only command that permits changing the current position to an arbitrary file. The `e` command, without any parameters, displays the current position (except the instruction address).

The debugger keeps track of the current position to make certain frequently used commands more convenient. In particular, the commands `b`, `l`, `li`, `w`, and `wi` use the current position as a default parameter. This section explains how the current position is initialized, how it is changed, and how to find out what it is.

When a breakpoint occurs or after a call stack command executes, the current position is set to the home position of the current frame. For the top frame of the call stack, this corresponds to the breakpoint. For all other frames, the home position is the location of the call to the next higher frame. You can change the current position with the call stack commands (`cu`, `cd`, `ct`, and `cb`), the `e` command, the line number command, the listing commands (`l`, `li`, `w`, and `wi`), and the searching commands (`/` and `?`).

In line mode, if the source line containing the current position is displayed, a `<` appears to the left of the source line.

In Screen Mode

The current position is always the line in the source window that contains the cursor.

d — *(delete) delete breakpoints*

Syntax

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

Arguments

`all`
Delete all breakpoints.

breakpoint_number
Number assigned to each set breakpoint.

Description

`d` deletes the listed breakpoints. All breakpoints are deleted if `all` is used.

Multiple breakpoint numbers comprising the *breakpoint_number* (obtained using `lb` and displayed in brackets) must be separated by commas in this command as illustrated here:

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

Delete a breakpoint at any time.

In scripts and command blocks, it is possible to delete a break at the current position without using the breakpoint number by simply using `d` alone. For example,

```
b if condition then
    d
    b if other_condition then return read all else g; end if
else
    g end;
```

In Screen Mode

`d` deletes a breakpoint set on the line in the source window that contains the cursor. If a source line contains a breakpoint, `=` appears to the left of the line.

Delete a breakpoint by moving the cursor to a line with = and typing d. Pressing Return is not necessary. The = disappears, showing that the breakpoint is gone.

References

"b — (break) break at a line or beginning of a subprogram" on page 3-16
"bd — (break down) break after current subprogram" on page 3-20
"bi — (break instruction) break at machine instruction" on page 3-22
"bx — (break exception) break when an Ada exception occurs" on page 3-28
"lb — (list breakpoints) list all currently set breakpoints" on page 3-83
break point commands and delete breakpoints, *SPARCompiler Ada User's Guide*

disassembly — display disassembled source code

Description

Display disassembled machine instructions using the list instructions (`li`) and display window instructions (`wi`) commands. These commands use the following format:

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

`li` (list instructions) lists a specified number of disassembled instructions (source code with corresponding assembly language code) and repeats by pressing Return. Similarly, `wi` (window instructions) prints a window of disassembled code surrounding a specified line or instruction address (hexadecimal number).

Figure 3-1 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;
```

Figure 3-1 Example of Output from the `li` Command

In addition to the `li` and `wi` commands for displaying instructions, the debugger operates in instruction submode of screen 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` and the `I` commands. Use the `p` command with registers and, toggle in and out of instruction submode with the `I` command.

References

“bi — (break instruction) break at machine instruction” on page 3-22
“Instruction and Source Submodes” on page 3-134
“li — (list instructions) list disassembled instructions” on page 3-84
“wi — (window instruction) list disassembled and original code” on page 3-162

display memory — display raw memory

Syntax

[p] *hexadecimal_address*[:*display*][*number*]
[p] *decimal_address*:*display*[*number*]
[p] *name*:*display* [*number*]

Arguments

decimal_address
Memory address in decimal notation.

display
One- or two-character code indicating how the contents of the memory address are displayed. See the Description section for a listing of these codes.

hexadecimal_address
Memory address in hexadecimal notation (begins with a leading 0).

name
Ada or C object to display. This can be a complex expression. *name* can be a register name of the form *$register_name* in which case the contents of the register are displayed.

If *name* is a debugger keyword, it must preceded with a backslash or a syntax error results. For example, `b:=3` results in a syntax error because `b` is a debugger keyword but `\b :=3` is legal.

number
Number of values to display.

Description

Use the `p` command to display memory. In screen mode, if you want to display memory at a variable location, use the `P..y` facility. After the variable is yanked to the command line, enter `:m` Return following the variable name to display memory at that variable location.

In the last syntax form (`p` *name*:*display* [*number*]), *name* is evaluated and the address of the named object is used. *name* can be an Ada object or a C variable.

The address of the item can be specified with an expression, e.g., `$pc+4`

The precedence of operators is that the C unary operators * and & have the highest precedence, followed by the : in display memory, and then the binary operators. For example, if the source is C, the debugger evaluates `p*`*address*`:`*display* as `p(*`*address*`):`*display*. Use parentheses to establish a different precedence, `p*(`*address*`:`*display*`)`.

For binary operators, `p$pc+4:m` is evaluated, by default, as `p$pc+(4:m)`. Use parentheses within binary expressions preceding the : to avoid ambiguity, `p($pc+4):m`. Note that the trailing m in this expression is a format character (discussed below).

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]

(Continued)

Format	
r	Reverse-map the address to a procedure name
s	Null-terminated (C-style) string
x	Hexadecimal [32 bits]
z	Show the address of the dope vector for records and unconstrained arrays

You can enter *number* in either decimal or hexadecimal with a leading zero. 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 (1–9) but contains a hexadecimal digit (A–F), the debugger interprets the number as a hexadecimal number (`12A3` or `9AAF`). Precede a hexadecimal number with a leading hexadecimal digit (`F2`) 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 `01A9A`, type the following command:

```
01a9a:Wx 8
```

The debugger responds with this output:

```
01a9a:    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 `01A9A`. Each line displays in hexadecimal format followed by an ASCII string:

```
01a9a:m 2
```

The debugger responds with this output:

```
01a9a:  20 74 68 65 20 4e 20 71 75 65 65 6e 73 20 70 72 " the N queens pr"
01aaa:  6f 62 6c 65 6d  a  0  0  0  1  0   0  0  1  0  0 "oblem........."
```

The :a format is useful for finding the address of any name expression. For example, given the array object MY_ARRAY, the command

```
MY_ARRAY(1):a
```

displays the address of the first element. Use the :a format for expanded names, selected names, and so forth.

Note - Even though they cannot be typed in directly by the user, the debugger can display bit addresses using the notation hex_byte_addr.bit_offset. For example, given the following declarations,type boo is:

```
array(1..8) of boolean;
 pragma pack (boo);
 abc: boo := (true, true, false, false, true, true, false, false);
```

the following can be displayed:

```
>abc:b
07fffc620: 11001100 01000001 10011011 10000100 00010000 00000101
>abc(0):b
07fffc620: 11001100 01000001 10011011 10000100 00010000 00000101
>abc(1):b
07fffc620.01: .1001100 01000001 10011011 10000100 00010000 00000101
>abc(2):b
07fffc620.02: ..001100 01000001 10011011 10000100 00010000 00000101
>abc(3):b
07fffc620.03: ...01100 01000001 10011011 10000100 00010000 00000101
```

User-defined Formats

In addition to the lengths and formats of display listed above, the debugger can display memory at a given address using a type declared in the user Ada or C program. For example, if TASK_BLOCK is a complex variant record type, p 080040:TASK_BLOCK displays the entire record at address 080040, including all the correct variants.

You can use the field of a record type as a type:

```
p (080040:TASK_BLOCK).NEXT_TASK
```

As another example, consider these type declarations:

```
type task_block_ptr is access task_block;
```

```
type queue;
type queue_ptr is access queue;
type queue is
record
     next: queue_ptr
     node: task_block_ptr;
end record
```

If the user is breakpointed in a subprogram where the register i4 holds a QUEUE_PTR value, the following expressions can be typed:

Debugger command	Value printed
> p $i4:queue_ptr	The access value in i4, i.e., an address
> p ($i4:queue_ptr).all	The object pointed to by the access in i4 as type QUEUE
> p ($i4:queue_ptr).node	The access value of the node field
> p ($i4:queue_ptr).node.all	The *task_block* object pointed to by node

After an object is created using the p *address:type* syntax, use it in any expression where such an object is legal.

Note – First, the debugger checks the specified type against the debugger basic display values. If a match exists, it uses the basic value, even if the Ada program contains a declared type of the same name. The debugger display values are listed above and are displayed as part of the diagnostic if an unrecognized value is used.

Name Expressions

The :a format is useful for finding the address of any name expression. For example, given the array object MY_ARRAY, the command

```
p MY_ARRAY:a
```

displays the address of the first element. Use the :a format for expanded names, selected names, etc.

Note – Typing a plain decimal number moves the current position to that line number. Memory displays require the number to be followed by a colon and a display letter. The `p` command symbolically displays the value of variables or name expressions; a command consisting of only a name expression is a syntax error.

Type Casting of Raw Memory to a Data Structure

The debugger also has the ability to display an object in memory when no visible variable points to that object.

There are times when debugging that you either have only the address of an object because you are debugging a machine code routine, or are in a location in your program where the debugger cannot determine with certainty where an object is. In the latter case, you can often determine the address of the object by examining the machine instructions and/or register contents. In addition, there is a convenient way for you to display your object using just its address and type mark.

The syntax of the debugger command is an extension of the syntax for examining memory. Instead of supplying a format specifier such as L or B, you supply the type mark (which must be visible according to the standard visibility rules). Consider the following example:

```
package dashboard is
   ...
   type speedometer_t is record
           speed:         speed_t;
           trip_counter:  miles_t;
           odometer:      miles_t;
   end record;
```

Suppose you know that the address of an object of this type is 16#100A48#. To display the object you would simply enter the command:

```
>0100a48:dashboard.speedometer_t
```

Further, if the object is located at an offset of 16 off of register `r0`, you could either get the address in register `r0`, add 16 then use the method above, or more simply type:

```
>*($r0+16):dashboard.speedometer_t
```

This expression is evaluated as follows:

`$r0`
value in register `r0`

`$rp+16`
add 16 to value in `r0`

`*($r0_16)`
The `*` means “indirect”, i.e., read the memory whose address is `$r0+16`. The 32 bits of memory at that location are the address that is used to display `dashboard.speedometer.t`.

To display an object as another type, use the same command as above but substitute the name of the object for the address on the left. For example, if there is an object declared in the program as

```
foo:speed_t
```

and you want to display `foo` as type `MILES_T`, simply enter the command

```
foo:dashboard.miles_t
```

and the debugger displays the object `foo` as type `MILES_T`.

Note that the "typecast" operation, `:`, binds most tightly. If you have an expression to the immediate left of the `:`, you must use parentheses to typecast the entire expression. For example, if you type:

```
> sym + foo:newtype
```

`foo` is recast to `newtype`. To recast `sym+foo`, you must surround the expression with parentheses.

```
> (aym + doo) :newtype
```

Typecasting is also available for C in Sun SPARC or Sun-3 self-hosted debuggers.

For example, in X-window applications, you may want to look at the fields of a widget. The type `WIDGET` is defined as:

```
typedef struct _WidgetRec {
       CorePart core;
} WidgetRec, *Widget;
```

Even though all the widget variables declared are used as though they are type WIDGET, in fact they are more complex structures.

For example, when a label widget is created, it returns a pointer to a LabelRec:

```
typedef struct _LabelRec {
        CorePart    core;
        SimplePart  simple;
        LabelPart   label;
} LabelRec, *LabelWidget;
```

If a variable LABEL_W is declared as

```
label_w: Widget
```

the debugger command

```
p *label_w
```

displays only the fields in CorePart of the widget. To display ALL the information about this label WIDGET, the variable can be typecast:

```
p *(label_w: LabelWidget)
```

or

```
p (*label_w): LabelRec
```

As with Ada typecasting, an address can be used instead of a variable name.

Some special restrictions with C typecasting are:

- The only types that can be used are the C basic types (e.g. int, double, char, etc.) or names defined by typedefs. No C symbolic information exists for #defines or struct <type>s. To use a structure, declare a typedef as shown above for Widget.
- By default, the debugger searches only for typedefs defined in the current file or in header (.h) files included in the current file. If you want the debugger to search all the symbolic information in the entire executable, use the debugger set command:

```
set c_types global
```

References

Section 3.1, “Summary,” on page 3-1
“expressions — arithmetic expressions in the debugger” on page 3-65
“p — (print) display the value of a variable or expression” on page 3-108
“In Screen Mode” on page 3-110
“obase number” on page 3-143
display raw memory, *SPARCompiler Ada User’s Guide*

3

e — *(enter) move to a new source file*

Syntax

e [*ada_entity*|*ada_source_file*]

Arguments

ada_entity
Name of an Ada entity such as a subprogram, package, task, variable, constant, etc.

ada_source_file
Name of an Ada source file. This file must in a directory on your ADAPATH.

Description

The e command provides a convenient way to move the current position to a new file or line within a file. If a file is specified, the current position becomes the beginning of the file. If an Ada entity is specified, the current position becomes the first line in the source file of the entity full definition. For example, for a subprogram, the current position becomes the first line of the subprogram body. For a type, the current position becomes the first line of the type full declaration.

For purposes of the e, edit, and vi commands, visibility rules for the *ada_entity* name are as follows.

If the *ada_entity* name given is a simple identifier, all subprograms, tasks, and packages (with an elaboration subprogram) in the program are visible. Other Ada entities (including packages with no elaboration subprogram) must be directly visible from the current context or must be library units.

If the *ada_entity* name has multiple definitions, it is overloaded. The debugger prints a diagnostic showing the alternatives. Retype the e command attaching a '1, '2, ... suffix (matching an alternative shown in the diagnostic) to the *ada_entity* name to disambiguate it. Alternatively, sometimes it is sufficient to use an expanded name to disambiguate it.

Filenames

Filenames that contain only alphanumeric characters, dots, and underscores do not need to enclosed in quotes, but those containing other characters must be enclosed in quotes. Enclosing a filename in quotes (`“file.a”`) often eliminates errors if, for example, part of the filename collides with a keyword or program variable.

In addition, the debugger interprets `csh(1)` tilde notation and shell environment (exported) variables if the filename is enclosed in quotes,

```
e “$al/foo.a”
```

or

```
e “~/tst/math.a”
```

In Screen Mode

Invoke the `e` command by positioning the cursor on any character of a Ada entity simple name and typing Control-]. This has the same effect as typing `e` *ada_entity* in line mode—the source window is rewritten with the source code corresponding to the named subprogram.

If the Ada entity name is overloaded, the debugger prints a diagnostic showing the alternatives. Retype the <Control-]> command preceding it with a number (matching an alternative shown in the diagnostic) to disambiguate it. That is, typing a number n before <Control-]> has the same effect as typing `e ada_entity'n` in line mode.

Use `e` in screen mode by preceding the command with `:` and following with Return.

References

“overloading — disambiguate overloaded names” on page 3-106
move current position, *SPARCompiler Ada User’s Guide*
expanded names, section 4.1.3(13) in *Ada Reference Manual*

3

`edit` — *edit a subprogram or a file*

Syntax

`edit [`*ada_entity*`|`*ada_source_file*`]`

Arguments

ada_entity
Name of an Ada entity such as a subprogram, package, task, variable, constant, etc.

ada_source_file
Name of an Ada source file. This file must in a directory on your ADAPATH.

Description

`edit` invokes the editor with the specified file or the file containing the specified Ada entity. If no parameter is given, the file containing the current position is used.

The debugger consults the environment (exported) variable `EDITOR` for the name of the editor. If `EDITOR` is not defined, the debugger uses `vi` as the editor.

The parameters are interpreted exactly as they are for the `e` command.

In Screen Mode

Precede this command with `:` and follow with Return.

References

enter editor, *SPARCompiler Ada User's Guide*

examine — display program elements and components

Description

The following table lists the available debugger display commands and their functions:

Command	Function
`p`	Variable values
`e`, `edit`, `l` or `w`	Files of source code
`li` or `wi`	Machine instructions
`lb`	Current breakpoints
`lt`	Active tasks
`cs`	Call stack of currently active subprograms
`set all`	Current debugger parameter settings
`set signal`	Current signal setting
`address:display`	Raw memory display (display memory)

References

“cs —. (call stack) display the call stack” on page 3-40
“display memory — display raw memory” on page 3-51
“e — (enter) move to a new source file” on page 3-60
“edit — edit a subprogram or a file” on page 3-62
“l — (list) display part of a source program” on page 3-81
“lb — (list breakpoints) list all currently set breakpoints” on page 3-83
“lt — (list tasks) list all active tasks” on page 3-93
“p — (print) display the value of a variable or expression” on page 3-108
“set — set debugger parameters” on page 3-142
“signals — set/ignore signals” on page 3-148
“w — (window) list a group of source lines” on page 3-161

`exit` — *terminate the debugger session*

Syntax

```
exit
```

Description

`exit` exits from the debugger. Also, use `quit` to leave `a.db`.

In Screen Mode

Precede this command with `:` and follow with Return.

expressions — arithmetic expressions in the debugger

Binary Operators

The debugger performs arithmetic using the following Ada binary operators:

```
+           >=
-           <
*           <=
/           /=
**          =
>           AND
            OR
```

The operands to the AND and OR operations must be integer or boolean; floating point operands are not allowed. If both operands to AND are non-zero, the result is non-zero. If either or both operands to AND are 0, the result is 0. If either or both operands to OR are 1, the result is 1. If both operands to OR are 0, the result is 0.

Unary Operators

The debugger supports the following unary operators:

```
+
-
```

For debugging C, the following unary operators are supported:

& — Returns the address of its operand
* — Dereferences pointers

Operands

Operands are numbers (integers or floating point), program variables or function calls. More than one function call can appear in the same expression; use functions calls as parameters to other function calls, and so forth.

In an expression, the debugger implicitly converts an integer number to a floating point number if the integer is one operand of a binary operation (+, –, *, /) and the other operand is a floating point number.

The comparison operators in the following list return an integer value of either 0 (FALSE) or 1 (TRUE).

```
>           <=
>=          /=
<           =
```

Currently, the final value of an expression must have `type INTEGER` or `type FLOAT`. Also, any operand to one of the above operators must be an integer or floating point value. Ada access values are currently converted to 32-bit integers.

Attributes

The Ada debugger supports the following attributes:

```
'ADDRESS                    'FIRST(N)
'BASE                       'LAST
'CALLABLE                   'LAST(N)
'COUNT                      'RANGE(N)
'DELTA                      'SIZE
'DIGITS                     'TERMINATED
'EMAX                       'WIDTH
```

`'BASE` must be the prefix to another attribute.

References

"Unary Operators for Debugging C" on page 3-109
each attribute, *Appendix A* in *Ada Reference Manual*

files — specify files to debug

Description

The Invocation section explains how to control which executable file is being debugged.

For example, assume *pathname* is the name of a Ada directory. When debugging a program generated from

pathname`/foo.a`

the debugger uses the net files in

pathname`/.nets/foo*`

produced by the Ada compiler to obtain most of the Ada symbolic information for `foo.a`. Other important files are in *pathname*`/.lines/foo*`. The line number files contain a mapping between line numbers and instruction addresses for all the object code generated from `foo.a`.

The commands `e`, `vi`, and `edit` accept a filename. The filename can be contained in double quotes. Simple filenames, containing only alphanumeric characters, dots, and underscores need not be surrounded by quotes — type them in directly, for example, `foo.a, baz.exp_cmd.a`. For the `e`, `vi`, and `edit` commands, the debugger interprets the shell tilde (~) notation and shell environment (exported) variables for filenames, but only if the filename is enclosed in double quotes. The named file must be in a directory on your ADAPATH.

The `r`, `set`, and `<` commands can contain filenames. Do not enclose them in quotes. The debugger interprets tilde and environment (exported) variables in filenames in these commands.

References

“invocation — invoking the debugger” on page 3-76
“r — run a program” on page 3-119
“read — read debugger commands from a file” on page 3-121
“set — set debugger parameters” on page 3-142

3

g — *(go) continue executing*

Syntax

```
g
```

Description

g continues executing the program from where it stops. If the process is breakpointed because of a UNIX signal, g continues the process, ignoring the signal. Use the command gx to continue with the signal. Use the command gw to continue the program while watching for a variable value to change. Use the command gw to continue the program while watching for a variable value to change. The single-stepping commands s, si, a, and ai, execute the program, but only for one source line or instruction.

If the program has not started execution, the g command runs the program, although no invocation processing (processing of I/O redirection, options and other parameters used by the program) is done. If the program exits or terminates, the g command reruns the program, using the invocation parameters used the last time the program was run. To rerun a program from the beginning at any time, use the r command. r accepts csh-like invocation parameters.

g cancels the Return key memory.

In Screen Mode

Typing g when in screen mode causes the program to continue; pressing Return is not required. With set safe on, the screen mode command becomes gg.

References

"gw — (go while) continue executing until a variable changes" on page 3-69
"Return — re-execute debugger command" on page 3-128
"set — set debugger parameters" on page 3-142
continue execution, *SPARCompiler Ada User's Guide*

gw — *(go while) continue executing until a variable changes*

Syntax

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

Arguments

address
Memory address. This is either decimal or hexadecimal. If hexadecimal, a leading 0 is required.

name
Name of a variable.

number
Number of bytes. This value is from 1 to 16 inclusive. [Default: 4].

Description

gw executes the program until the value of the named variable changes. If an address (*number*) is used, the specified number of bytes (*number*) at that address is polled for change (4 is the default).

This breakpoint is useful, but it can be slow since the value being polled is checked after the execution of each machine instruction.

In Screen Mode

Precede this command with : and follow with Return.

References

continue execution, *SPARCompiler Ada User's Guide*

gx — *(go signal) continue executing, pass the signal to the program*

Syntax

```
gx
```

Description

When a signal occurs in a program being debugged, the program is first presented to the debugger. The debugger announces the signal and the location at which it occurs and stops, waiting for commands. To continue execution of the program as though the signal has not occurred, use `g`, `a`, `ai`, `s`, or `si`. To continue execution and pass the signal to the program, use the `gx` command. This is useful in debugging programs that do explicit signal handling.

This command cancels the Return key memory.

Some UNIX signals are transferred into Ada exceptions by the Ada runtime system. If the program stops when it receives such a signal, type `gx` to raise the corresponding Ada exception.

In Screen Mode

Precede this command with `:` and follow with Return.

References

“Return — re-execute debugger command” on page 3-128
“signal [signal_list | all] [b | g | gx]” on page 3-144
“sx — (step signal) single step, pass the signal to the program” on page 3-155
continue execution, *SPARCompiler Ada User’s Guide*

help — *print help text*

Syntax

help [*subject*]

Arguments

subject
Debugger command, debugger concept or SC Ada tool.

Description

In a debugging session, on-line help is available for debugger commands and concepts as well as for other Ada tools.

At the debugger prompt (>) type:

>help [*subject***]**

If *subject* is omitted, the intro screen is printed listing the debugger commands and concepts. This screen is available by typing help intro.

The help text for *subject* entries is paged automatically and --More-- is displayed at the last line of the screen. Table 3-1 provides a list of responses to the --More-- prompt. The paging program is defined by the environment (exported) variable PAGER. If PAGER is not defined, the default is used.

At the end of each subject entry, a deb *subject*? prompt is displayed. Type a *subject* name:

deb *subject***?** *subject*

to access help for another *subject* or q, followed by Return, to exit help.

In Screen Mode

When using the debugger in screen mode, the command

:help [*subject*]

provides the facility described above, paged on the lower portion of the screen.

A separate additional help facility provides help for the pager commands. Type:

H

which prints a line of the pager commands at the bottom of the screen. Type **H** again for an additional line of commands.

Table 3-1 Summary of Responses in Debugger Help

Prompt	You type	Response
>	**help** [*subject*] **quit**	Print help screen(s) for *subject*. Exit the debugger.
--More--	Press Space Press Return **q** **H** **.** (dot)	Print the next screen of help text. Print next line of help text. Skip to the end of the help text for *subject*. Print pager help screen. Repeat the previous command.
deb *subject*?	*subject* **q** and press Return	Print the help screen(s) for *subject*. Exit help.

References

on-line debugger help, *SPARCompiler Ada User's Guide*

home position — execution point in current frame

Description

The home position is the execution point in the current stack frame, the next instruction to execute. The line containing the home position is marked with an asterisk.

When program execution moves to a new frame, the home position changes as well. For the topmost frame(s), (i.e., topmost non-inline frame plus any inline frames above it), the home position is the instruction that executes next. For all other frames, it is the call instruction that called the next higher non-inline frame.

The asterisk is recognized in line mode as being a valid line number, meaning "the line containing the home position." Use an asterisk as a command by itself to move the current position to the home position.

References

"inline expansions — debugging inline expansions" on page 3-74
"line numbers — move to a specified line" on page 3-91
home position, *SPARCompiler Ada User's Guide*

3

inline expansions — debugging inline expansions

Description

The Ada debugger supports the debugging of inline expansions. Inline expansions occur for each inline call to an `INLINE` or `INLINE_ONLY` subprogram (including auto-inlined calls to small subprograms controlled by the `AUTO_INLINE INFO` directive) and for each generic instantiation that is inlined, i.e., a generic package specification is copied inline.

It is possible to step into inline expansions using the step (`s`) command and step over inline expansions using the advance (`a`) command. After an inline expansion is entered, for example, by stepping into it with the step command, it is possible to display the parameters and local variables of the expansion and to set breakpoints in the expansion.

The call stack (`cs`) command displays inline frames. These frames are marked with a `+` character immediately after the frame number. The call down (`cd`) and call up (`cu`) commands enable you to navigate up and down through inline expansion frames. Inline frames do not correspond to an actual hardware frame in the program call stack. They are logical frames that simulate the existence of a real frame for the inline, i.e., as if the inline is called with a normal call instruction. An inline frame shares all the hardware characteristics of the next lower non-inline frame, i.e., it has the same PC, FP, and SP register values.

The debugger supports the debugging of inline expansions with no call site code (hereafter referred to as NCSC inlines). An example of an NCSC inline is a call to a parameterless procedure or function. No parameter binding code is generated, so the first instruction generated is due to a source line in the inline body. With optimized code, even calls to subprograms with parameters can be NCSC inlines.

The first instruction of an NCSC inline is, in effect, associated with multiple source lines, that is, the line that causes the code to generate in the inline body and all the call site lines. Potentially, more than one call site line exists for nested calls (e.g., A calls B who calls C who has first instruction).

The debugger supports setting breaks at NCSC inline call sites. Similarly, it supports stepping to such lines without stepping into the inline. At such a call site, advance over the inline call or step into it. To do the latter, the debugger simulates the step into. That is, your context changes to the inline context, but the program is not physically stepped, i.e., the PC does not change.

A breakpoint associated with multiple source lines is announced at the line where the breakpoint is set originally. The list breaks (`lb`) command lists all lines associated with an inline breakpoint. A + character marks the line where the breakpoint is announced. In screen mode, breakpoint marks (characters = and -) are displayed at all lines associated with an inline breakpoint.

Caution – The stepping support for NCSC inlines is based on heuristics that fail in some situations. However, they work in most common cases. In the worst case, you are inside an inline when you wanted to be at the call site (or vice versa). Stepping out of an inline can step several inline frames down (e.g. last line in A calls B, last line in B calls C, step from last line of C goes to caller of A). Similarly, a `bd` from an inline frame may break several inline frames down.

3

invocation — invoking the debugger

Syntax

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

Arguments

a.db_options
Options to the debugger are:

`-A`
(asynchronous) Invoke debugger in asynchronous mode.

`-a` *pid*
Invoke the debugger on the currently executing process (*pid*). The debugger does not join the process group of that process. Use the `ps(1)` or `jobs` command to get the *pid*.

`-ag` *pid*
Invoke the debugger on the currently executing process (*pid*). The debugger joins the process group of that process. Use the `ps(1)` or `jobs` command to get the *pid*. This enables Control-c to work.

`-c`
Debug C programs. This option avoids error messages relating to missing Ada libraries.

`-I` *argument_list*
(interface) Pass arguments defined in *argument_list* to `DB_IFACE` when the debugger interface process is invoked.

`-i` *filename*
(input) Read input from the specified file.

`-L` *library_name*
(library) Read program compilation information from the specified library, rather than the current directory, as though you are operating in the specified library. This option is for debugging Ada programs only.

`-r` *"executable_file [executable_file_options]"*
(run) Initialize `set run` with *executable_file* and *executable_file_options*.

`-sh`
(show) Display the name of the debugger executable but do not execute. This option is useful if multiple versions of Ada are on a system.

`-t` *filename*
(terminal) Read terminal state from *filename*.

When the debugger runs in the background, it cannot reliably get the state of the controlling terminal, as that state changes as you run other programs in the foreground. But the output of the debugged program depends on the set up of your terminal. To ensure that the output is displayed in a consistent way, we provide the program `tty_state` in *ada_location*`/sup/diag`. `tty_state` must be run in the foreground and it dumps the terminal state to a file. Invoke this program as follows:

```
tty_state -f filename -w
```

Then supply that same *filename* to the debugger with the `-t` option. Note that you can print the `tty` state that is written to filename by typing:

```
tty_state -f filename -r
```

`-v`
(visual) Invoke the screen-mode debugger directly.

executable_file
Name of file to execute and debug. If *executable_file* is not specified, the debugger searches for `a.out`. If only a root filename is given (`foo`, as opposed to `/vc/sbq/foo`), the debugger searches the directories on the `PATH` environment (exported) variable for an executable file `foo` just as the shell does. Note that if "." is not on your `PATH`, you must enter `a.db ./foo`.

executable_file_options
Command line options that pertain to the *executable_file* being debugged. All command line options that follow the name of the executable are assumed to belong to the program being debugged.

References

"screen mode — screen-oriented debugger interface" on page 3-132
command file input, display debugger executable, executable file, invoking the debugger, and screen mode, *SPARCompiler Ada User's Guide*

Description

a.db is a symbolic debugger for Ada and C programs. On the Solaris 2.1 operating system, C programs must be compiled with both the `-g` option and the `-xs` option to be compatible with the Ada debugger.

Invocation options to the program being debugged can be specified at debugger invocation. All command line options that follow the name of the executable are assumed to belong to the program being debugged.

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. The `-r` option initializes `set run` to a string made up of the *executable_file* and the *executable_file_options* enclosed in quotes. Enclosing shell commands that pertain to the executable file within the quotes, (output redirection for example) assures that they are not interpreted by the shell to apply to the debugger itself.

Any single unit or token on the command line can be up to 511 characters long.

Detailed descriptions of interactive `a.db` commands are provided in this reference, which is available also online using `a.help` or the debugger internal `help` command.

Use the `quit` command to leave the debugger and return to the shell.

References

debugging C programs, *SPARCompiler Ada User's Guide*

Invocation File

In addition to the invocation line, you can supply parameters to `a.db` using an invocation file named `.dbrc`. During debugger initialization, `a.db` checks for `./.dbrc`. If that does not exist, it checks for `$HOME/.dbrc`. The `.dbrc` file can contain only `set` commands. These commands execute before commands in an input file specified on the command line but not before command line options.

A `set source` command in the `.dbrc` file can specify the location of an `ada.lib` for the debugging session other than the default `ada.lib`. If a `set source` command is present, the debugger searches the directories specified in the `set source` command for the first directory that contains an `ada.lib`. The debugger uses that directory to obtain the `DIANA` net files and the line number files produced by the compiler.

References

"set — set debugger parameters" on page 3-142

Start-up Environment

The debugger establishes the debugging environment when it starts up. Certain key parameters are displayed on the screen to verify what and where it is debugging. For example:

```
    a.db /vc/atst/phl
    Debugging: /vc/atst/phl
    ada_library : /vc/atst
    library search list:
    /vc/atst
    /usr2/ada_2.1/self/verdixlib
    /usr2/ada_2.1/self/standard
    /vc/install/build/tasking
>
```

Figure 3-2 Example of Debugger Start-up Environment

The first line of the example is the invocation of the debugger on the file `phl`, the dining philosophers program copied from the examples directory and compiled.

The first line of output shows the full path and name of the program being debugged. set source is initialized automatically to this path. `set run` is initialized automatically to this path with the executable name. This facilitates subsequent invocations of the debugger on this executable file. Any options that follow the executable filename are assumed to be for the executable and are sent to set run unless the -r option is used.

The *ada_library* is the name of the Ada library directory for this debugging session.

The first line of the example is the invocation of the debugger on the file `phl`, the dining philosophers program, copied from the examples directory and compiled.

The library search list is derived from the `ada.lib` file in the *ada_library*. It shows the Ada library directories that are searched when the debugger looks for an Ada unit. The search list is displayed in the same order that the debugger searches it.

References

Ada library directory, *SPARCompiler Ada User's Guide*

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*.

The two restrictions to using redirection are:

1. You cannot use screen mode when debugging input is a file.
2. Your program cannot share the debugger input file. Use `set input` *filename* to set the input file for your program. A sample `debug.in` file is:

```
set input debug.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 debuggers `-r` option:

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

References

redirecting input/output, *SPARCompiler Ada User's Guide*

`l` — *(list) display part of a source program*

Syntax

`l` [*line*] [`,` *number*] Pressing Return repeats

Arguments

line
Line number at which to start the listing.

number
Number of lines to display. [Default: 10]

Description

`l` lists a specified number of lines of the current source file starting at the specified *line*. The default value for *number* is 10, but change this with the `set` command. The default line is the current line, which is marked with < to the left. In screen mode, the current line is the line under the cursor.

The possible forms of line are:

number	Move to specified line
+*number*	Move *number* lines forward
–*number*	Move *number* lines backward
*	Move to home position

After the `l` command executes, the current line is the last line displayed. Consequently, typing `l` without parameters continues listing where the last `l` command stopped.

If * is displayed after the line number, that line is the current home position for this file. If = appears after the line number, a breakpoint is set for that line.

If – appears after the line number, a breakpoint is set in the code generated for this line but not on the first instruction. If both = and – apply to the same line, the one set most recently is displayed.

If + appears after the line number, this represents an inline frame.

In Screen Mode

Precede this command with : and follow with Return.

References

"line numbers — move to a specified line" on page 3-91
"Instruction and Source Submodes" on page 3-134
display lines, *SPARCompiler Ada User's Guide*

lb — *(list breakpoints) list all currently set breakpoints*

Syntax

```
lb
```

Description

`lb` lists all currently set breakpoints. To the left of each breakpoint is a number in brackets. Use it to delete the breakpoint.

In Screen Mode

Precede this command with : and follow with Return.

References

"d — (delete) delete breakpoints" on page 3-47
display breakpoints, *SPARCompiler Ada User's Guide*

3

li — *(list instructions) list disassembled instructions*

Syntax

`li` [*expression*|*decimal_number*] [, *number*] Pressing Return repeats

Arguments

expression
Expression defining the instruction address where listing is starts.

decimal_number
Decimal number indicating the line number at which listing is to start.

number
Number of lines to display in the listing. [Default: 10]

Description

`li` lists the specified *number* of lines including disassembled machine instructions interspersed with source lines. If a decimal line number is given, disassembly starts with the first line of code generated by or after that line. *instruction* stands for an instruction address expressed as a hexadecimal number with a leading zero. If *expression* is specified, the listing starts with the instruction at that address.

If * appears next to the instruction address, that instruction is the home position for the current frame. Use * as the *expression*|*decimal_number* to start disassembly at the home position.

If an equals sign appears after the address, a breakpoint is at that instruction.

If [*expression*|*decimal_number*] is omitted, the listing begins with the line or instruction following the most recently `l`'d or `li`'d line or instruction or the home line or instruction if no `l` or `li` command is given for this file. Listing begins with the home position if *expression*|*decimal_number* is *.

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

The debugger does not, in one `li` command, disassemble across a source file boundary, no matter how many lines it is instructed to print. It stops the display at the source file boundary. The next `li` command with no parameters starts at the beginning of the next source file.

In Screen Mode

Precede this command with `:` and follow with Return. You can show disassembly in the upper window with *Instruction Submode*.

In screen mode, toggle the upper window (source window) to display either source code or disassembled machine instructions. This screen mode command, `I` (uppercase `i`), performs this toggling.

References

"Instruction and Source Submodes" on page 3-134
"wi — (window instruction) list disassembled and original code" on page 3-162
display instructions, *SPARCompiler Ada User's Guide*

3

line editing — command history and line editing functions

Description

The debugger supports line editing functions that enable the user to make simple changes to a command before transmitting it to the debugger. In addition, the debugger supports a command history mechanism that enables the user to recall a previous command for editing and execution.

The best way to learn about the line editing and command history features of the debugger is to try them. We recommend getting into a debugging session and typing a few commands. Then try out the features described in the following sections.

Line editing and command history apply only to line-oriented debugger commands. A line-oriented command is submitted to the debugger when the debugger is in line mode or when entering a command in response to a : prompt in screen mode. Window control commands and debugger commands that are entered directly in screen mode cannot be line edited and are not saved in the history buffer.

Command History

When you transmit a command to the debugger by typing Return, the debugger remembers it (unless it is exactly the same as the previous command). The debugger has a 2048 character circular buffer to save the most recent non-screen-mode commands. Since debugger commands are typically under 10 characters, this buffer holds about 200 of the most recent commands.

Two line editing commands are specific to command history:

k	Go backwards in history one command.
j	Go forward in history. Only use j after at least one k.

Enter these two commands when line editing a debugger command (but not when in insert mode). Every time you type a k you go back one command. That command is displayed at the current prompt with the cursor at the right of the command. If you then type a Return, the command is submitted to the debugger. Alternatively, use any of the line editing commands to change the command, before typing Return to submit it to the debugger.

Line editing

When you enter a command to the debugger, you are in one of two modes, insert or edit. Normally you are in insert mode with the cursor at the end of the command line. Each character you type is added to the command at the end. The two most common line editing commands used in insert mode are “erase” (usually Control-h) to backspace over the most recent character, and “kill” (typically Control-u) to erase the entire line. The debugger uses the user `stty` “erase” and “kill” characters to perform these functions. The operating system stty command displays the current erase and kill characters.

To leave insert mode and enter edit mode, the Escape key is used. This moves the cursor back one character, placing it over the last character inserted.

Regardless if you are in edit or insert mode, when you type a Return, the entire line is transmitted to the debugger, even if the cursor is positioned in the middle of the line. In other words, what you see is what is sent to the debugger. Here is a list of the available line editing commands:

Table 3-2 Line Editing Commands

Name	Command	M	Meaning
AHEAD	j	e	Go ahead (down) in history
APND_END	A	e	Go to end of line and enter insert mode
APPEND	a	e	Enter insert mode after character under cursor
BACKWARD	k	e	Go back (up) in history
BEGINSRT	I	e	Move cursor to beginning of line and enter insert mode
BEGLINE	0	e	Move cursor to beginning of line
CARET	^	e	Move to first nonwhite space on line.
CHANGE	c {motion}	e	Change text [see "motion"]
CHGROL	C	e	Change rest of line, from character under cursor
CHNG_SRCH	N	e	Search in the opposite direction in history for previous *string*.
COMPLETE	*string*Esc\	e	Complete string with name visible in current scope that begins with *string*. Beep sounds if 0 or >1 matches are found.

Table 3-2 Line Editing Commands *(Continued)*

Name	Command	M	Meaning
CONT_SRC	n	e	Search in same direction in history for previous *string.*
DELCHAR	x	e	Delete character underneath cursor
DELETE	d {motion}	e	Delete text [see "motion"]
DELROL	D	e	Delete rest of line, from character under cursor
DOIT	Return	b	Transmit the current line to the debugger
ENDCOM	Esc	e	End insert mode (or CHANGE mode) - switch to edit mode
ENDLINE	$	e	Move cursor to end of line
ENTERIN	i	e	Enter insert mode before character under cursor
ERASE	Control-h	i	Erase last character, prints Backspace-blank-backspace
KILL	Control-u	i	Kill entire line typed so far
LITNEXT	Control-v	i	Literalize next input character
LS_NAMES	*string*Esc=	i	List all names in current scope that start with *string.* Beep sounds if no matches are found
MATCH_LS	*string*Esc(#)	i	Complete the name beginning with *string* using # (as produced by LISTMATCHES command) for the name.
MV_BWORD	b	e	Move cursor to previous word
MV_LEFT	h	e	Move cursor one position left
MV_RIGHT	l	e	Move cursor one position right
MV_WORD	w	e	Move cursor to next word
REPLACE	r<*char*>	e	Replace one character underneath cursor
REPRINT	Control-f	i	Reprint most recently typed in line

Table 3-2 Line Editing Commands *(Continued)*

Name	Command	M	Meaning
SRCH_BWD	?*string*	e	Search backward in history for the next command that contains *string*.
SRCH_FWD	*/string*	e	Search forward in history for the most recent command that contains *string*.
WERASE	Control-w	i	Erase most recently typed word

The "M" [mode] column
i = Command available in insert mode
b = Command available in both modes
e = Command available in edit mode

Motion
Select one of the following characters as a motion character for the CHANGE or DELETE editing functions. The motion character determines what changes.
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 nonwhite space of the line to the cursor $ - From the cursor to the end of the line

Additional Notes
1. For the change commands (CHANGE and CHGROL), you enter change mode after typing the command. The change area is marked by a dollar sign ($) on the right end and the cursor on the left end. This entire area is replaced by what you type. If what you type is shorter than the change area, then the remainder of the change area is deleted. If what you type is longer, then the remainder of the line is pushed to the right to make room.

2. The h, j, k, and l characters give the directions left, down, up, and right respectively. If you view the history as a page of text, then the j key moves up the page and the k key moves down the page.

3. If you type ?p in edit mode, the last command stored in history that has a p in it is matched. Note that the command does not have to start with a p. If no match is found, a beep sounds. Use the / in edit mode to search forward in history. Note that N and n in edit mode must follow the completion of either a / or a ? command.

4. In addition to ERASE and KILL, WERASE, and REPRINT are character sequences that the debugger inherits from your current stty settings.
5. The command *string*Esc= provides the ability to find all names in the current scope that begin with the string entered before the Esc. For example, if you are debugging HELLO_WORLD, the following can be entered.

```
p p<ESC>= (
1) text_io.put_line'2 (string )
(2) text_io.put_line'1 (file_type, string )
```

```
(3) text_io.put'4 (string )
(4) text_io.put'3 (file_type, string )
(5) text_io.put'2 (character )
(6) text_io.put'1 (file_type, character )
(7) text_io.positive_count
(8) text_io.page_length'2 ( ) return count
(9) text_io.page_length'1 (file_type ) return count
(10) text_io.page'2 ( ) return positive_count
(11) text_io.page'1 (file_type ) return positive_count
(12) EXCEPTION.program_error
(13) positive >p p
```

An additional feature uses the numbers produced from the *string*Esc= command to quickly change the command line. The current string is completed to the string listed by the number chosen using the *string*Esc(#) command. For example, after issuing the `p pEsc=` command, you may want to continue the `p p` command with `put'4` (number 3 in the list). You can do this by entering the following at the spot where the cursor is located:

```
Esc(3)
```

The full command appears as `>p p<ESC>(3)`.

After entering the above and pressing Return, the command line changes to

```
>p put'4
```

and you are back in insert mode again.

Note – Matching is done on the names of the procedures or variables that are visible, not by the name of the package. In the example above, the list comes from matching the letter `p`. You do not see the entries in the list that match `t` for `TEXT_IO` if you typed `t`Esc=. The name of the package is printed in the list for your convenience.

---

line numbers — move to a specified line

Syntax

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

Arguments

number
The number of the line to move to or the number of lines to move.

Description

Some debugger commands accept *number* as a parameter (`b`, `l`, `w`, `li`, `wi`). In line mode, typing *number* by itself is the debugger command to move to that line.

Specify a line number as a decimal number, representing that line in the current file. Typing `+`*number* or `-`*number* adds or subtracts the specified number from the current position. For example, typing `+5` or `-5` as a command changes the current debugger line position by adding or subtracting 5, respectively. Typing `*` as a line number means "use the line number corresponding to the current home position." For the debugger commands `li` and `wi`, typing `*` means the home position instruction (not the line number).

When the debugger displays current home position (in response to an `l`, `li`, `w` or `wi` command), `*` appears left of the line corresponding to home position.

In line mode, the debugger displays the current position by showing `<` to the left of the line corresponding to the current position. The `e` command with no parameters lists out the current file, subprogram, and line corresponding to the current position.

In Screen Mode

Move to a new line in screen mode by using this command:

```
[number]G
```

The `G` must be uppercase. This command moves the debugger source window, so it displays the line in the current file whose number was specified. If *number* is not specified, the source window moves to the end of the file. In screen mode, both of the following commands move to line 102.

```
102G
:102
```

By default, the debugger displays the source with associated line numbers, (controlled with the `set number` command).

References

"number [on|off]" on page 3-143
specify new position, *SPARCompiler Ada User's Guide*

lt — *(list tasks) list all active tasks*

Syntax

```
lt [all|use|task]
```

Arguments

`all`
Provides a detailed display for all tasks up to the maximum of 300 tasks.

task
Name or hexadecimal address of task for which detailed information is displayed. The current task is the breakpointed task or the most recent task.

`use`
Display the location and usage of stacks for all tasks up to the maximum of 300 tasks.

Description

If `all`, `use` or *task* is not specified, `lt` lists all active tasks (up to a maximum of 300) and gives a brief status for each one.

Display Status for All Active Tasks

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

The `Q#` column (queue numbers) has one of several values: `R`*n* indicates that the task is on the *run queue* in the *n*th position. `R1` runs next. `D`*n* means that the task is on the delay queue in the *n*th position. The delay for `D1` expires next.

Note – The queue position numbers are applicable only to the VADS MICRO kernel. They are not displayed when Ada tasking is layered on Solaris threads.

An asterisk (`*`) indicates the current task (the breakpointed task or the most recent task).

An `ABNORMAL` in the `Q#` column indicates the task has been aborted.

The TASK column contains either the name of the task or a T, followed by the name of the task type that declared the task. Note that the main program's task has no name and is listed as `main program`. Also, the runtime defines a task used when all other tasks are suspended and the scheduler is waiting for an interrupt event. This task is listed as `idle task` if it is the breakpointed task. Otherwise, the idle task is not listed

If the breakpointed task is not an Ada task or the idle task, this task is listed as `non-Ada task`.

A special signal task is created for each interrupt entry. It is given the name `signal` *sig_num* where *sig_num* is its interrupt vector number.

For Solaris threads, a special interrupt task is created for each attached ISR. It is given the name `interrupt` *intr_num* where *intr_num* is its interrupt vector number.

The NUM column contains the sequence number assigned to the task. This number is always 1 for the main task and is incremented every time a task is created. This number is used when setting breakpoints for a particular task. The number can also be used with the `lt` command to specify the task, `lt5`. The `tcb` address of the task can be displayed by getting a full listing of the task using the `lt task` command.

The STATUS column shows the state of each task and additional information for some states. Times displayed are absolute times, which start with 0 unless the timer is reset via the `package CALENDAR` or `package XCALENDAR`. The time must match `CALENDAR.CLOCK`. The possible states for each task are shown in Table 3-3.

Table 3-3 Task State Conditions

State	Description
not yet active	The task is created but not yet activated. See section 9.3 in the *Ada Reference Manual.*
ready to start	The task is activated and ready to start its first execution.
awaiting activations	Parent task suspended while waiting for child tasks to complete their activation.
awaiting terminations	Parent task suspended while waiting for child tasks to terminate.
executing	The task is executing.
ready	The task is on the run queue.

Table 3-3 Task State Conditions *(Continued)*

State	Description
suspended at `accept`	The task has executed an `accept` on an entry that no task is currently calling, so is now waiting until a task calls it.
suspended at fast `accept`	The task has executed a "fast" `accept` on an entry that no task is currently calling, so is waiting until a task calls it.
suspended at trivial accept	The task executes an accept on a CIFO 'trivial' entry that no task is currently calling, so is waiting until a task calls it.
suspended at `select`	The task has executed a `task` but no *tasks* are calling open entries; statement has an `open terminate` alternative and is waiting until some event enables it to proceed. `(terminate possible)` Task termination conditions satisfied. `(terminate not possible)` Task terminate conditions not satisfied; waiting for child tasks to terminate. `OPEN ENTRIES:` *entry_name* Lists each open entry `of select`. `NO ENTRIES OPEN:` No entries currently open.
suspended at call	The task has executed an entry call and remains in this state until transition `in rendezvous` state. *task_name*[*task_addr*].*entry_name* Gives target task and entry
in rendezvous	The task has executed an entry call and remains in this state until transition `in rendezvous` state. *task_name*[*task_addr*].*entry_name* Gives target task and entry
attempting rendezvous	The task is attempting rendezvous with the called task. *task_name[task_addr].entry_name* Gives target task and entry
finished rendezvous	The task has just finished its rendezvous with the called task. *task_name*[*task_addr*].*entry_name* Gives target task and entry
suspended at delay	The task has executed a `delay` statement.
suspended at passive call or cond wait	The task is suspended calling a passive task's entry whose guard is closed or the task is waiting on an ABORT_SAFE condition variable.
finished passive call or cond wait	The task suspended at a passive call has been resumed. The guard for the called entry has changed from closed to open. Alternatively, if the task was waiting on an ABORT_SAFE condition variable, the condition variable has been signalled.

Table 3-3 Task State Conditions *(Continued)*

State	Description
suspended on semaphore	Task blocked waiting for its semaphore to be signalled. *[semaphore_addr]* Gives semaphore ID
suspended on mutex	Task blocked from entering critical region protected by a mutex. Another task has locked the mutex. *[mutex_addr]* Gives mutex ID
suspended on cond	Task blocked waiting for its condition variable to be signalled. *[cond_addr] using mutex [mutex_addr]* Gives condition variable and mutex IDs
waiting to exit	`<main program>` suspended, waiting for child tasks to terminate.
completed	The task has executed all its code body. Upon completion of all subtasks, it terminates. See section 9.4(5) of the *Ada Reference Manual.*
terminated	The task is terminated. See section 9.4(6) of the *Ada Reference Manual.*
destroyed	The task is has been terminated and is in the process of being destroyed.
waiting for signal	The task created for the interrupt entry is waiting to be signalled by its interrupt handler. After being signalled, this task does an entry call to the interrupt entry in the attached task. *attached to task_name[task_addr] at entry entry_name* Gives attached task and interrupt entry
waiting for interrupt	The task created for the interrupt vector is waiting to be signalled by its interrupt handler. After being signalled, this task calls the attached interrupt service routine (ISR). The task is blocked at a `sigwait()` for the attached UNIX signal. *handler at handler_addr]* Gives the address of the ISR

Some additional information can be appended to the status entry.

`in rendezvous with` *task_name*`[`*task_addr*`] at entry` *entry_name*

means that the task accepted the entry call from the named task at the named entry. This message can occur more than once. It repeats for each rendezvous resulting from outer nested `accept`s in order from innermost to outermost `accept`.

The message

`on delay queue, until day:` *number* `sec:` *number*

The `lt` command produces output like that in Figure 3-3:

```
>lt
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 3-3 Example of Output from the `lt` Command

When tasks are dynamically created, that is, anonymous tasks, only their identifiers (the address of the task control block) are listed. Use the task number with the `task` and `cs` (call stack) commands.

Using the `lt` command to display tasks using the fast rendezvous optimization has a few subtle differences.

References

Fast Rendezvous Optimizations, *SPARCompiler Ada Runtime System Guide*

.Display Single Task Status

The `lt` *task* form of this command provides additional information about a single task. For example, `lt dining_room` produces the following output in Figure 3-4:

```
>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 3-4 Example of Output for `lt dining_room`

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 `task`, and the ordered lists of the tasks waiting at that entry.

The thread ID gives the ID of the underlying OS thread. For the VADS MICRO kernel, it is the address of its micro kernel task control block

The `tcb` address line indicates the address the runtime system assigned to the task when it was created. You can use the address value when identifying instances of a task type, as the task type name is not a unique identifier. An address of `00000000` is displayed for `idle_task` or non-Ada task since they are not real Ada tasks.

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`.

For the VADS MICRO kernel, if the task's thread priority differs from the current priority, the thread priority is displayed on the line following the current priority. In the CIFO add-on product, the thread priority can differ from the current Ada priority when the task owns a priority inheritance or priority ceiling, or it is a sporadic task.

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. The task executing the master is not necessarily the task that creates it.

The system clock can continue to run while the debugger is suspended at a breakpoint waiting for input. This can cause delays to expire immediately when stepping away from the breakpoint and the flow of program control to relate to breakpoints and their timing in an unpredictable fashion.

If time slicing is enabled, a breakpoint is often followed by a time slicing transfer of control. This transfer can be pathologic if a breakpoint is set in the actual time slicing logic in the Ada kernel such as in `SWITCH` or `SWITCH_TO`. The time slice response calls `SWITCH` and reaches another breakpoint. The delay in handling the breakpoint uses up the time slice, and so another time slicing interrupt comes just as execution resumes from this breakpoint. For this reason, it is convenient when debugging tasks to configure the runtime system without time slicing enabled

```
v_usr_conf.configuration_table.time_slicing_enabled := false;
```

or to call the subprogram in `V_XTASKING` to turn off time slicing.

```
v_xtasking.set_time_slicing_enabled(false);
```

Warning – Breakpoints in the runtime system itself may leave the tasking data structures in an unpredictable state. Output from the `lt` command can then be questionable, particularly for the current task. For example, a common breakpoint is `SWITCH_TO`, which is called when control transfers to a new task. At this point, in the middle of an `accept`, rendezvous information may not be consistent.

Display CIFO Pragma Values

After providing a detailed display for all tasks, the `lt` all form of this command displays the values set using the CIFO pragmas in the main procedure. The CIFO add-on product supports the following pragmas that can appear in the main procedure:

```
pragma SET_PRIORITY_INHERITANCE_CRITERIA;
pragma SET_GLOBAL_ENTRY_CRITERIA
  (TO: in QUEUING_DISCIPLINE.DISCIPLINE);
pragma SET_GLOBAL_SELECT_CRITERIA
  (TO: in COMPLEX_DISCIPLINE.SELECT_CRITERIA);
```

When the main program doesn't have any of the above pragmas, the lt all command has the following output at the end of its display:

```
Priority inheritance enabled = false
Global entry criteria        = fifo queuing
Global select lexical order  = false
```

The CIFO add-on product allows the entry criteria and select criteria to be specified on a per task basis using the following pragmas.

```
pragma SET_ENTRY_CRITERIA
  (TO: in QUEUING_DISCIPLINE.DISCIPLINE);
pragma SET_SELECT_CRITERIA
  (TO: in COMPLEX_DISCIPLINE.SELECT_CRITERIA);
```

The following lines are displayed for a task when its value differs from the global pragma value:

```
entry criteria        = fifo queuing | priority queuing
select lexical order  = false | true
```

The CIFO add-on is available from Verdix Corporation (1-800-BUY-VADS)

Display Stack Usage and Location

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

The command, `lt use`, produces the abbreviated output in Figure 3-5

```
>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%]
   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%]
                                                                  (continued)
```

```
(continued)
   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%]
```

Figure 3-5 Example of Output from `lt use`

The `lt use` command displays the normal stack memory address range and the exception stack range of a task. The exception stack is located directly below the normal stack area of a task. This is needed for execution of the exception unwinding logic to handle the stack limit `STORAGE_ERROR` exception. Notice that the size of the exception stack is the same for all tasks. It is defined by the configuration table parameter, `EXCEPTION_STACK_SIZE` in the `v_usr_conf_b.a` file found in the `usr_conf` directory.

Also notice that the size of the task stacks has been increased by the configuration table parameter, `WAIT_STACK_SIZE`. The wait stack area is allocated at the top of the task stack. The wait stack is needed to support the fast rendezvous optimization.

For each task stack range, `lt use` calculates stack usage by starting at the `LOW_ADDRESS` memory location and searching upward for the first nonzero byte. The following equations are used:

```
stack_size := (high_address + 1) - low_address
bytes_used := (high_address + 1) - first_nonzero_address
[usage %] := (bytes_used / stack_size) * 100
```

Warning – Stack usage information may be incorrect for applications having dynamic task creation and completion. When a task completes, its stack area is returned to a stack-free list. Subsequently created tasks attempt to get their stack areas from this free list. We chose to optimize task creation and completion processing by not zeroing out the task stack areas.

For self-hosts, we assume that the underlying OS returns zeroed memory for allocation requests. The `v_krn_conf.zero_heap_stack` routine called by `v_krn_conf.v_start_program` zeroes the memory. However, the user can decrease the time spent doing kernel initialization by eliminating this zeroing operation.

We assume that nonzero values are pushed on the stack.

References

"b — (break) break at a line or beginning of a subprogram" on page 3-16
"bd — (break down) break after current subprogram" on page 3-20
"bx — (break exception) break when an Ada exception occurs" on page 3-28
"task — print current task or choose a new current task" on page 3-156
display tasks, *SPARCompiler Ada User's Guide*
time slice configuration parameters, *SPARCompiler Ada User's Guide*
master task, section 9.4 in *Ada Reference Manual*
Fast Rendezvous Optimizations, *SPARCompiler Ada Runtime System Guide*

3

lu — *(list processes) list UNIX processes*

Syntax

`lu [`*`PID`*`]`

Arguments

PID

Process identification number of the process to list.

Description

`lu` provides a description of the UNIX process(es) to which the debugger is attached.

If no arguments are listed, a brief description of all the processes the debugger is attached to is displayed. For example, see Figure 3-6:

```
>lu
24600: stopped by the debugger.
24599: stopped by the debugger.
24598: stopped on signal "trap" (5).
```

Figure 3-6 Example of Output from `lu` Command

If a *PID* is listed following the command, the debugger prints a detailed description of the process whose process id (*PID*) is given. For example, see Figure 3-7:

```
>lu 24600
pid 24600, utime 0 sec, stime 0 sec
   ppid 24598, group id: 24596, session id: 0
FLAGS:
     stopped [PR_STOPPED]
     stopped on an event of interest [PR_ISTOP]
     inherit-on-fork flag set [PR_FORK]
run-on-last-close flag set [PR_RLC]
stopped in response to stop directive, normally PIOCSTOP
```

Figure 3-7 Example of Output from `lu` *PID* Command

References

“lt — (list tasks) list all active tasks” on page 3-93
`proc` or `ptrace`, UNIX Reference Manuals

3

overloading — disambiguate overloaded names

Description

a.db supports a subprogram, task or package name as the object of the e (move to a new source file), edit (edit a file), vi (enter screen-oriented mode), and b (set breakpoint) commands. `e`, `edit` and `vi` support other Ada names such as names of types, variables, constants, etc.

When debugging Ada programs, the name supplied to these commands need not be fully qualified if it is the name of a subprogram, task or package (with an elaboration subprogram). For example, the simple name `sin` can be used in place of `standard.math.sin`).

When a simple name is used for one of these commands, the debugger searches "program-wide" for subprograms of this name (including subprograms that correspond to task bodies and package elaboration). This search enables you to set a breakpoint or enter any subprogram even if its name is not directly visible now.

The `e`, `edit` and `vi` commands search for entities that do not correspond to subprograms (e.g,. packages with no elaboration subprogram, types, variables, etc.) directly visible from the current context or that are library units.

If the search finds multiple definitions for the same name, the debugger issues a message displaying each of the possible alternatives. The notation is *simple_name'n*, where *n* is a number, for example, `sin'2`. An example of the debugger message follows in Figure 3-8. A breakpoint is set at a subprogram with the simple name `add`, which is overloaded.

```
> 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 3-8 Example of Overloading

The suffix ‘*n* must be appended to the subprogram name to indicate precisely the name to reference, 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, it lists all of the overloadings.

A task that is passive or has interrupt entries shows up as being overloaded. Its different cases are indicated with the `PASSIVE ACCEPT`, `PASSIVE ISR`, and `NON_PASSIVE IRS` labels.

References

overload resolution, *SPARCompiler Ada User’s Guide*

3

p — *(print) display the value of a variable or expression*

Syntax

p *expression*

Arguments

expression
Name of a program variable, subprogram name or arithmetic expression.

Description

p is the debugger command for displaying the value of program variables and for calling subprograms. It evaluates arithmetic expressions that may contain program variables and/or function calls. If the displayed result is an integer, it is printed in the current output base (default, 10). Change this default with the `set obase` command.

Name Expressions

For Ada variables, the debugger currently supports simple variable names (`MARK, R_A`), selected components and expanded names (`MARK.LINE`), strings (`DATE 1..4`), and indexed components (`X(Y), Z(1,2)`). Use them in combination (`X(M.Z), B.X(1)`). The debugger supports the evaluation and display of Ada array slices, for example, p date (1...3). It evaluates the `'FIRST`, `'LAST`, `'ADDRESS`, and `'RANGE` attributes. Entry calls are not supported yet.

Call Ada subprograms and functions. You can use the results of a function in the expression. Some restrictions exist. Only parameters of mode `in` are supported. `in out` or `out` parameters are not supported. Default parameter values are not supported yet. The debugger does not call a function that returns an array or a record (functions returning access values *are* supported). A parameterless function must be called using empty parentheses, `p foo()`. When referencing an Ada subprogram, it may be necessary to resolve overloading.

The debugger uses visibility rules similar — but not identical — to Ada for looking up variable names. The current frame determines what names are visible. The visible names are the same names that are visible if a statement is added to the program at the point in the source text corresponding to the current frame.

Expressions

The p command calculates the value of an expression and prints the result.

References

“display memory — display raw memory” on page 3-51
“expressions — arithmetic expressions in the debugger” on page 3-65

User Procedure Calls

Call procedures in the program being debugged, user procedures, with the p command.

References

“procedure calls — call subprograms from the program” on page 3-112

Unary Operators for Debugging C

The debugger supports the * and & unary operators.

The value of the * operand is the address of the memory to be accessed. The operand cannot be a structure, union, float or double data type.

The * operator dereferences pointers. The fields of structures are given when the operand is a pointer to a structure.

The & unary operator returns the address of its operand. Use the & operator in procedure calls and with variables of any type except registers.

References

debugging C programs, *SPARCompiler Ada User’s Guide*

Display Memory

Use the p command to display raw memory.

Some of the facilities described under *display memory* can be used to extract memory as a value and use it in an expression. For example,

```
p (013A770:L) + 01 A
```

reads the 32 bytes at address `013A770`, adds `01A` to it and prints the result.

References

“display memory — display raw memory” on page 3-51

Display Exceptions

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

```
%p $exception
```

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

In Screen Mode

The `p` command has additional support in screen mode. To print the value of a variable on the screen (either window), position the cursor on top of any letter of a variable name, and press `p`. In response, the name of the variable is displayed in the command window followed by the variable value. This is useful when single stepping since usually the cursor is near an instance of variables of interest.

To display a name expression, such as `foo(k).link`, position the cursor on top of `foo` and press uppercase `P`. This moves the cursor to the `f` of `foo` and overwrites the last letter of `foo` with `@`. The cursor and the `@` delimit the expression. Type `P` again to move the `@` right again to the last character of the next part of the name expression, the right parenthesis of `(k)`. Continue typing `P` in this way until the entire expression to be displayed is delimited by the cursor and the `@`. At this point, a `p` displays the value of the delimited expression. `a` causes the debugger to display `foo(k).link.all`, which prints the object that `foo(k).link` points to. `*` is for C debugging; it displays `*c_foo[k].link` which prints the object where `c_foo[k].link` points. In the following example, the position of the

cursor is indicated by the letter contained in the box.

`foo(k).link` -- Before you type `P`, the cursor is on the `o` of the `foo`.

`fo@(k).link` -- After you type `P`.

`f oo(k@.link` -- After you type a second `P`.

`foo(k).link@` -- After you type a third `P`.

Now you type:

- `p` to send `foo(k).link` to the debugger
- `a` to send `foo(k).link.all` to the debugger
- `*` to send `*c_foo[k].link` to the debugger (where `c_foo[k]` is a C array with the field `link`)
- `y` to yank the delimited expression to the command line and precede it with `:p`

For example, if you enter y, the following appears on the command line:

```
:p foo(k) .link
```

Use the `P..y` facility to print variables using any of the display options described in the *display memory* command. For example, to display a variable in hexadecimal notation, use `P..y` to yank the variable to the command line (it is automatically preceded by `a :p`) and type `:x` Return after it. The hexadecimal value of that variable is displayed.

```
:p variable_name:x Return
```

To display a variable in decimal notation, enter `:d` Return after yanking it to the command line.

```
:p variable_name:d Return
```

To display memory at a variable location, enter `:m` Return after yanking the variable to the command line.

```
:p variable_name:m<RETURN>
```

References

display variables, *SPARCompiler Ada User's Guide*

3

procedure calls — call subprograms from the program

Description

It is possible in the debugger to call a procedure that is part of the program being debugged.

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

The debugger treats user procedure calling as an expression, and the `p` command implements this capability. For example, to call the procedure dump, enter the following:

```
>p dump(foo)
```

With user procedure calling, build customized displays for key objects and access data structure routines. Write routines to display data structures, to verify that data structures have certain properties (for example, is the table sorted?) or to display and navigate through a complex data structure. Then access these routines in the debugger.

A procedure called by the debugger is not limited to displaying numbers and text on the screen. The procedure can prompt for input, read it, and act on it, that is, the procedure can be interactive. Display output from the procedure or capture it in a log file.

Since these procedures are written in Ada or C as part of the user program, they can be called from the program, for example, a procedure may display data structures when an internal error is detected or prior to a catastrophic failure. If these procedures are left in the program after it is deployed, they provide a method for debugging the program in the field at a customer site.

Note – If the procedure call hits a breakpoint, the program stops and the user can debug as normal. However, the debugger abandons evaluating the expression, if any. Here is an example:

Case 1: the user calls `cos()` and has no breakpoints set in `cos` or in any routines that `cos()` calls. The debugger prints the return value:

```
> p cos(45.0)
0.5
```

Case 2: The user calls `cos()` and has a breakpoint set in cos():

```
> p cos(45.0)
[7] stopped at "/u/sincos/cos.a":494 in cos
> ...  -- the user debugs here, inside cos (or some subprogram
> ...  -- that cos has called).
> g
Procedure returned normally.
```

Note that in this second case, the value is not printed. When the procedure returns from the user-generated call, i.e., from the debugger command `p cos(45.0)`, because expression evaluation is interrupted by the breakpoint, the debugger makes no attempt to continue with expression evaluation.

Case 3: The user calls `foo(cos(45.0))`, where `foo` is a subprogram and has a breakpoint in `cos()`:

```
> p foo(cos(45.0))
[7] stopped at "/u/sincos/cos.a":494 in cos
> ...  -- the user debugs here, inside cos (or some subprogram
> ...  -- that cos has called).
> g
Procedure returned normally.
```

In this third case, the procedure that returned normally is `cos`, not `foo()`. `Foo()` is never called because expression evaluation is abandoned during the call to `cos()` because of the breakpoint.

References

“p — (print) display the value of a variable or expression” on page 3-108
“return — return from all called subprograms” on page 3-129
“log [off | filename]” on page 3-143

put — *(put) send characters to program input*

Syntax

put *string*

Arguments

string
A series of characters or a quoted string.

Description

The put command is used when the debugger is operating in asynchronous mode. It is used to allow characters to be sent to the program's input. The put command takes either a quoted string or a series of characters as an argument and writes them to the program's standard input. A new-line character is not automatically appended to the end of the string. If you wish for a new-line character to be appended to the end of the string, use the put_line command.

In the case of a quoted string, all characters between the double quotes are transmitted. Otherwise, leading blanks are stripped and the rest of the line up to the new-line that terminates the command are sent to the program.

Whether or not the string is quoted, the \ character is used as an escape similar to its use in C strings. The only special character the debugger handles in the string is the new-line, which is indicated by \n. \" indicates a single double quote character and \\ indicates a single back slash character.

The following commands are equivalent:

```
put hello\n
put "hello\n"
put_line "hello"
put_line      hello
```

Although the put command writes characters to the program's standard input, there are no guarantees that the program will read them up. If there are unread characters when the program announces a breakpoint or signal, or when you switch the debugger to synchronous mode, the debugger flushes them and emits a warning message.

References

“asynchronous debugging — run the debugger in asynchronous mode” on page 3-11
“put_line — (put line) send characters to program input, append new line” on page 3-116

put_line — *(put line) send characters to program input, append new line*

Syntax

put *string*

Arguments

string
A series of characters or a quoted string.

Description

The put_line command is used when the debugger is operating in asynchronous mode. It is used to allow characters to be sent to the program's input. The put_line command takes either a quoted string or a series of characters as an argument and writes them to the program's standard input. A new-line character is automatically appended to the end of the string. If you do not wish for a new-line character to be appended to the end of the string, use the put command.

In the case of a quoted string, all characters between the double quotes are transmitted. Otherwise, leading blanks are stripped, and then the rest of the line up to the new line that terminates the command are sent to the program.

Whether or not the string is quoted, the \ character is used as an escape similarly to its use in C strings. The only special character the debugger handles in the string is new-line, which is indicated by \n. \" indicates a single double quote character and \\ indicates a single back slash character.

The following commands are equivalent:

```
put_line "hello"
put_line              hello
put hello\n
put "hello\n"
```

Although the put_line command writes characters to the program's standard input, there are no guarantees that the program will read them up. If there are unread characters when the program announces a breakpoint or signal, or when you switch the debugger to synchronous mode, the debugger flushes them and emits a warning message.

References

“asynchronous debugging — run the debugger in asynchronous mode” on page 3-11
“put — (put) send characters to program input” on page 3-114

`quit` — *terminate the debugger session*

Syntax

```
quit
```

Description

`quit` exits the debugger. Also, `exit` exits the debugger.

In Screen Mode

Precede this command with `:` and follow with Return.

r — *run a program*

Syntax

r [*shell_arguments*]

Arguments

shell_arguments

Shell command arguments. The debugger uses csh(1) or the shell defined in the environment (exported) variable SHELL.

Description

r runs or reruns the program. r must be followed by a space, tab, or new line. If it is followed immediately by a special character, the debugger assumes that r is the name of a variable and not the command. All the characters after the r command (and the required space, tab or new line) are interpreted as a single entity.

If *shell_arguments* is specified, r runs the program as if it is executing 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 csh(1) or the shell defined in the environment (exported) variable SHELL. 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.

This command starts or restarts the program from the beginning. To continue execution from a breakpoint or step, use g.

In Screen Mode

Typing r when in screen mode starts or restarts the program executing from the beginning. (Pressing Return is not required.) In “safe” mode, the command becomes rr.

To establish the invocation parameters in screen mode, first type a colon to get a line-prompt, and then a full `r` command with invocation parameters, I/O redirection, and so forth. Alternatively, use `set run` after the colon. The debugger remembers I/O redirection and invocation parameters, so subsequent `r` commands without parameters use the parameters of the previous `r` command.

Typing `set run` without any parameters causes the `r` command to "forget" its I/O redirection and invocation parameters.

References

"g — (go) continue executing" on page 3-68
"run shell_arguments" on page 3-144
"safe [on | off]" on page 3-144
run the program, *SPARCompiler Ada User's Guide*

read — *read debugger commands from a file*

Syntax

```
read filename
return read
return read all
```

Arguments

filename
Name of file that contains the debugger commands to execute

`return read`.
When executing a read *filename* command, `return read` terminates the reading of commands in *filename*. If the `read` *filename* command was typed at the terminal, the user now gets a prompt at the terminal. If the `read` *filename* command is a command inside another file, the next command to be read is the command in that file following the `read` *filename* command. This command pops out of nested `read`s by one level.

`return read all`
This option returns from all files and the user gets a prompt at the terminal. For example, file `a` includes the command `read b` and file `b` contains the command `read c`. If during the execution of the commands in `file c` a `return read all` command is encountered, all files are exited and a terminal prompt appears. This command pops out of all nested read commands.

Description

`read` switches the debugger input source from the keyboard to the named file. Additional `read` commands can occur in the file but only can be nested four levels. After executing the file, commands are again read from the keyboard (unless the command file contains an `exit` or `quit` command).

Warning – Commands occurring after a `read` command on the same line do not execute. For example, the `p foo` command in

```
read X_DUMP; p foo
```

does not execute.

In Screen Mode

Precede this command with : and follow with Return.

References

`read` debugger commands, *SPARCompiler Ada User's Guide*

reg — *list the current machine register contents*

Syntax

```
reg [all|f|s]
```

Arguments

`all`
Display the contents of all registers.

`f`
Display the contents of the floating point registers.

`s`
Display the contents of the special registers.

Description

`reg` lists the contents of registers as they are when the program stops. A sample output is illustrated in Figure 3-9.

```
   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 3-9 Example of Output from `reg`

The command

```
reg f
```

lists the floating-point coprocessor registers if the implementation supports a coprocessor. See Figure 3-10

```
: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: -4613.000000         f5: 1073757.125000    d4:
-125002963649582280053515878 4.000000
 f6: 1.000000             f7: -NaN              d6: 0.007813
 f8: -NaN                 f9: -NaN              d8: -NaN
f10: -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 3-10 Example Display of Floating Point Registers

Display individual registers using the `p` command with the name of the register preceded by a dollar sign (`$`). For example (`p $i1`, `p $fp`).

On the SPARC, display floating point registers as single-precision floats using `p $f`*number* (`$f2`), as double-precision floats using `p $d`*number*, and as extended reals with `p $e`*number*.

In Screen Mode

Precede this command with `:` and follow with Return.

References

display registers, *SPARCompiler Ada User's Guide*

register variables

Since most RISC architectures rely on register-to-register operations for speed, the Ada compiler tries to put variables in registers. Since machine registers are precious resources, the compiler tries to reuse them. The compiler analyzes the lifetime of each variable, that is, the first point in the program where the variable is set and the last point in the program where the variable is used. If a variable is dedicated to a register, the compiler can use the register if execution has not reached that variable lifetime, or if execution goes past that variable lifetime.

The following two code examples are taken from a debugging session to illustrate some of these points. In Figure 3-11, execution has reached statement 19, where variable `i` is going to be used as input to the `CALC` function. The result is stored in variable `j`. Statement 19 is the last use of variable `i` and the first use of variable `j`.

```
 13  begin
 14      i := 8;
 15      for k in 1..10 loop
 16          i := i + calc(i - 1);
 17      end loop;
 18
 19*=    j := calc(i);
 20
 21      while j < 0 loop
 22          j := calc(j);
 23      end loop;
 24
 25      val := j;
 26  end;
 27
-*-------------------------------------------------------
regopt.a--
 :p i
 6146
 :i:a
 i0
 :p j
 'j' is not yet active
```

Figure 3-11 Example of Debugging with Register Variables-1

Before executing statement 19, we can print the `i` value (`:pi`) and the `i` address is the SPARC register, `i0` (`:i:a`). However, we cannot print `j` yet, because it is not active, that is, execution has not entered lifetime of `j` yet.

Figure 3-12 shows execution after stepping past statement 19. Two things happened as a result of executing statement 19: we left lifetime of `i` and we entered lifetime of `j`.

```
 13  begin
 14      i := 8;
 15      for k in 1..10 loop
 16          i := i + calc(i - 1);
 17      end loop;
 18
 19=     j := calc(i);
 20
 21*     while j < 0 loop
 22          j := calc(j);
 23      end loop;
 24
 25      val := j;
 26  end;
 27
-*-------------------------------------------------regopt.a--
:p i
'i' was in register i0; no longer active
:p j
6145
:j:a
i0
```

Figure 3-12 Example of Debugging with Register Variables-2

When we try to print `i`, we see that it was in register `i0` but is no longer active. When we print `j`, we see that it now has a value; when we display its address, we see that `j` is now in register `i0`, the register occupied previously by `i`.

The following list of messages may be printed if the debugger has difficulty in finding the real value of a variable, which is assigned a logical register number.

In the following messages,

var is a variable name, for example, `foo`
reg is a register name, for example, `g2`
`r`*num* is a logical register, for example, `r120`
addr is an instruction (text) address, for example, `02BA4`

"*var* was register *reg*; no longer active"
At some previous point in the execution of this procedure, *var* was in *reg*. It is now "dead," that is, the variable is no longer needed, and probably, the register is being used for other values.

"*var* was on stack; no longer active"
At some previous point in the execution of this procedure, *var* was on the stack. It is now "dead," that is, the variable is no longer needed and probably that temporary location on the stack is being used for other values.

"*var* is not active"
The variable is not in a register, is not on the stack, and is now dead. We are only documenting this message for completeness; it is possible that this message can be printed; however, we are not aware of any scenarios that can cause it.

"*var* is not yet active"
The address of *var* is a logical register number. But the program is stopped at a point BEFORE the logical register is assigned to either a register or an address. Therefore, the variable has no real address at this point of execution.

"*var* has been optimized away"
This message is printed if a variable is optimized away (that is, it never gets a real address).

Return — re-execute debugger command

Syntax

```
(Press) Return
```

Description

Press the Return key to repeat the most recent `a`, `ai`, `s`, `si`, `l`, `li`, `/` or `?` command. Debugging a program with `r`, `g`, or `gx` causes the Return key to lose its memory until one of the repeatable commands is used again.

In safe mode, this feature is disabled for the `a`, `ai`, `s`, and `si` commands.

In Screen Mode

Screen mode disables the use of the Return key to repeat the functions listed above. However, the dot (`.`) repeats the previous debugger command line.

References

"Window Control Commands" on page 3-135
"safe [on | off]" on page 3-144

`return` — *return from all called subprograms*

Syntax

```
return
```

Description

`return` returns from all the user procedures that are called using the debugger `p` *proc(...)* command. After issuing a `return` command, you are back to where the original user procedure is called — `return` always returns from all of the nested user procedure calls from the call stack.

Frequently, it is convenient to write subprograms as part of a program that take a pointer to a complex data structure and display it on the screen. The debugger `p` *proc(...)* command calls any subprogram from the debugger. These data structures display subprograms that are a valuable tool for debugging, but you may need to debug them. If the procedure that is called using `p` p*roc(...)* faults, use the `return` command to get back to the original executing program that is being debugged.

Set a breakpoint in a subprogram prior to calling it using `p` *proc(...)*. After you hit this breakpoint, a different procedure can be called, that is, you can nest user procedure calls. The `return` command clears *all* procedure calls.

References

“p — (print) display the value of a variable or expression” on page 3-108
“procedure calls — call subprograms from the program” on page 3-112

s — *(step) single step source code into subprograms*

Syntax

s Pressing Return repeats

Description

s single steps one line of source code, stepping *into* called subprograms. If the debugger is currently on a source line containing subprogram calls, s executes the program up to the point where it calls one of the subprograms. The program then stops inside the called subprogram. The a command single steps one line of source code, but steps *over* subprogram calls.

Subprogram here means procedure, function, separate package body or instantiated generic package.

If the program has not yet started — for example, just after invoking the debugger — the s command starts the program, stepping one source line. For Ada programs, this steps *over* all the library unit elaborations. Other stepping commands are a, ai, si, and gw.

A frequent debugging mistake is to use the s command to step into a subprogram when the a command to step *over* the subprogram is intended. The bd command sets a breakpoint at the place where the current subprogram returns. To recover, use bd, and then g. Usually, execution stops very close to where the a command stops. The breakpoint is deleted automatically so it is not encountered again. When stopped at the bd breakpoint at the middle of a source statement, use a to go to the beginning of the next statement.

Use two debugger parameters, alert_freq and step_alert, to track the number of instructions that are stepped. Using the default settings, a message is displayed after the first 1000 instructions are stepped (step_alert). After that, every 100 additional instructions stepped (alert_freq), generates a new message). In line mode, these messages are periods — one after the initial number of instructions are stepped with a new period displayed for each 100 additional instructions stepped.

In Screen Mode

Type s in screen mode to single step the program one source line into called subprograms. Pressing Return is not required. With set safe on, the command becomes ss.

In instruction submode, `s` is interpreted as `si`.

The number of instructions stepped appears as a number on the dashed line separating the command and source window. This number is first displayed after the first 1000 instructions are stepped (`step_alert`). This number is incremented for every 100 instructions stepped after the initial display (`alert_freq`).

References

“bd — (break down) break after current subprogram” on page 3-20
“Instruction and Source Submodes” on page 3-134
“alert_freq number” on page 3-142
“safe [on | off]” on page 3-144
“step_alert number” on page 3-145
“si — (step instruction) single step machine code into program” on page 3-147
step one command, *SPARCompiler Ada User’s Guide*

3

screen mode — screen-oriented debugger interface

Description

The debugger supports a screen-oriented interface in addition to the more traditional line-oriented interface.

To use the screen interface, either invoke the debugger with the `-v` option,

`a.db -v` *myprogram*

or use the `vi` command from line mode.

To switch from screen mode to line mode, type the screen command `Q`.

Windows

In screen mode, the debugger divides the screen into three windows. The top window is called the source window. It goes from the top of the screen to the dashed line and is used for program source text. The window below the dashed line is called the command window. It displays command input to the debugger, text input to the program being debugged, and debugger / program output. The last line on the screen is called the error window and displays error messages from the screen-mode interface, patterns, and so forth. Output that appears in the error window does not appear in a log file.

The top two windows display a portion of a potentially larger area. For example, the source window displays part of a source file. This window can 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 text that have been displayed in the command window in an internal history buffer. Typically, the command window is positioned to display the most recent interactions (the end of the history buffer). However, this window behaves just like the source window in that it can display a different part of the history buffer.

The command window is paged automatically. If output from either a single debugger command or your program would scroll off the top of the window, the output is stopped and `--More--` is displayed on the last line. The available responses are listed in Table 3-4.

When the screen interface is invoked, the source window is 2/3 of the available screen size, the error window has 1 line and the command window has the remainder of the screen. Change the size of the source and command window at any time using the `C` command. The debugger remembers the most recent settings over subsequent `Q` and `vi` commands.

References

screen mode windows, *SPARCompiler Ada User's Guide*

Table 3-4 Paging: Responses to `--More--` in Screen Mode

Keystroke	Result
Space	Print the next window of text.
Return	Print next line.
`g`	Enlarge command window (at expense of source window). Windows are restored at next debugger command or by window control command, `C`.
`q`	Do not display remaining output but enter it into history buffer and log it to specified file
`p`	Turn off paging for both debugger commands and program output. Turn paging off when not responding to `--More--` with the `:set page off` command.
`x`	Abort debugger command. (Stop program output by stopping the program with Control–c).

Entering Commands

Screen-mode commands are those commands that are processed directly by the screen interface. These commands are *not* followed by Return — they execute immediately when the entire command is typed. Some screen commands can be preceded by an optional numeric argument.

Two classes of screen commands exist. The window control commands enable manipulation of the window interface. The screen interface recognizes the *immediate* debugger commands and passes them directly to the debugger.

The window control commands are applied to the window that contains the cursor. The `,` command moves the cursor from source window to command window, or the reverse.

Instruction and Source Submodes

In screen mode, the debugger supports two submodes, *instruction* and *source*. The `I` (uppercase `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, it operates 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 and source mode, *SPARCompiler Ada User's Guide*

Screen Prompt

Columns 1 and 2 of the dashed line separating the source window from the command window show the screen-mode prompt, `*`. The prompt alternates between these two columns, and is displayed only when the debugger is awaiting a command. Typing input at any other time is treated as type-ahead and/or input to the program under debug.

The `*` alternates between the two columns to indicate that a change takes place in certain, otherwise non-changing, situations. For example, if stopped at a breakpoint in a loop, typing `g` returns to the same breakpoint again. The only change on the screen is the position of the `*`.

Alert Frequency

The number of instructions stepped appears on the dashed line. This number appears after a step (`s`) or advance (`a`) command is entered. The default setting for the initial display is 1000 but change it with the `set step_alert` command. After the first 1000 machine instructions are stepped, the alert

frequency number on the dashed line increments each time 100 additional instructions are stepped. Change the default value of 100 with the `set alert_freq` command.

References

“alert_freq number” on page 3-142
“step_alert number” on page 3-145

Help

The screen mode `H` command displays a line of help text in the error window. This is one of several lines of help text that summarize all screen mode commands. The next help line displays each time you press the `H` key.

Window Control Commands

Window control commands control the screen interface. The pattern-matching commands, `/` and `?`, move the cursor to the bottom of the screen and prompt for the pattern. To terminate the pattern, press either Escape or Return.

You can precede some commands 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 following are the window control commands. Note that the window control commands are case-sensitive

Control-b	(backwards) Move backward one full window
[*number*]`C`	(change) Change the number of lines in the window; no *number* restores original sizes
[*number*]Control–d	(down) Scroll down 1/2 window or *number* lines
Control–f	(forward) Move forward one full window
Control–g	Print the name of file displayed in source window
[*number*]G	(go to) Move to bottom or specified line of file
[*number*]h or ◄–	Move left one or *number* columns

(Continued)

`H`	Display the next one-line help message
`I`	Toggle between instruction and source submodes
[*number*]`j` or ↓	Move down one or *number* lines
[*number*]`k` or ↑	Move up one or *number* lines
[*number*]`l` or →	Move right one or *number* columns
`mark` *letter*	Mark a location in either the command or source window. You can return to this location using the 'letter command. Note that letter must be lowercase. There is a separate set of marks for the command and source windows. The previous set of marks is invalidated by changing files in the source window or changing the disassembly-mode with the I command. Since a limited number of lines are stored, some marked lines may become lost and the associated mark invalidated.
`n`	(next) Repeat the previous / or ? search
Q	Leave screen mode; enter line mode
`w`	Move forward one word
`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
Control–r	(redraw) Redraw all windows (clean up display)
[*number*]Control–u	(up) Scroll up 1/2 window or *number* lines
/ *pattern*	Search forward for *pattern*
? *pattern*	Search backward for *pattern*
`:`	Enter a debugger command
`.`	(period) Repeat the previous debugger command line
`,`	Move to the other window
`0`	Move to beginning of line

(Continued)

^	Move to first character on line that is not a tab or blank
$	Move to end of line
%	Move forward or back to matching parenthesis or brace % finds the matching `end` when the cursor is placed on: `if`, `loop`, `for`, `while`, `case`, `record`, `select`, `function`, `procedure`, `package` or `task`. Likewise, % moves the cursor back from `end if` to the corresponding `if`, back from `end` *procedure_name* to the corresponding `procedure`, and so forth.
*	Move the cursor to the current home position
`'letter`	Return to location previously marked with `mark` *`letter`* command. Note that this command works across files.

"	Return to the previous destination of a G, /, ?, [[,]]. % or another '' command. The debugger motion commands reset the destination of the '' command. Thus, typing the following sequence of commands returns you to the line reached by the a command.
	`a`
	`/package`
	`''`
	Entering another '' returns you to the line reached by searching for the string "package".

References

window command syntax, *SPARCompiler Ada User's Guide*

Immediate Debugger Commands

The following list shows the immediate debugger commands. Enter these commands without a preceding : or a following Return in screen mode. In addition, you can enter any line-mode debugger command by typing a colon first. In response to the colon, the screen interface scrolls one line up the command window, positions the cursor on the bottom line of the command window, and prompts with a colon. Now enter any line-mode debugger command followed by Return. After each such debugger command, the cursor returns to the source window.

`a`	Step to next source line over call statements.
`b`	Set a breakpoint on line containing cursor.
[number]`B`	Set breakpoint at subprogram name (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[P...]a`	Print *variable*.`all`.
`P[P...]*`	Print * variable.

(Continued)

`P[P...]y`	Yank variable to command line. The variable appears on the command 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 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. Note that each time you `Control-]` in another procedure, your current location is pushed onto a tag stack for use with the `Control-t` command.
Control-c	Interrupt the current debugger operation.
Control-t	Return to the position where you last entered Control-] or the e (edit) command.

Screen Interface and `set output`

The `set output` option displays input and output from the program being debugged in screen mode without overwriting the source window. With `set output pty`, the debugger intercepts the program output and displays it in the lower window. Terminal input required by the program is taken from the lower window. With `set output tty`, the output from a program is presented, beginning wherever the cursor is located.

When debugging programs that send cursor-positioning strings to the terminal, it is often useful to alternate between screen and line mode, or to clear the bottom portion of the screen of all cursor-positioning strings by typing `:` followed by Return for each line in the bottom window, and then refreshing the display with Control-r.

References

“output [pty | tty | filename]” on page 3-143
screen mode debugging, *SPARCompiler Ada User’s Guide*

search — search for a pattern in the current file

Syntax

`/` *pattern*	Pressing Return repeats
`?` *pattern*	Pressing Return repeats

Arguments

pattern

Pattern to search for. If no pattern is given, the most recent pattern for either a `/` or `?` command is used. Press Return to execute the command again.

Description

The `/` and `?` commands search forward and backward, respectively, in the current file, starting at the line following the current line, for the specified pattern. If the pattern exists, the line it is on becomes the current line. The search wraps around; that is, if the pattern is not between the current line and the end of the file, the file is searched from the beginning to the current line. Control the case sensitivity of the search with the `set case` command.

The searching commands (`/` and `?`) use `regex(3)` for pattern matching. This is similar to `vi` and is described in the *Solaris Developer Documentation* under `ed(1)`.

The line containing the matching pattern becomes the new current position.

In Screen Mode

Use the searching commands (`/` and `?`) in either the source or command window. Typing a `/` or a `?` moves the cursor to the last line on the screen and prompts for the search pattern. The prompt is either a `/` or a `?`. Terminate the pattern by pressing Return or Escape. Repeat the command by pressing `n`.

If the pattern exists, the window displays the line containing the pattern.

The n command in screen mode repeats the last / or ? search, but the search starts from the cursor current location within the current line.

References

“current position — current position in a source file” on page 3-46
“case [on | off]” on page 3-142
ed(1), and regex(1), and vi, *Solaris Developer Documentation*

3

`set` — *set debugger parameters*

Syntax

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

Arguments

option and *value*
Change a debugger parameter. Permissible settings for *value* are shown in brackets following the option. The default setting for *value* is shown in brackets following the description of the option.

`alert_freq` *number*
Update the alert message every number of instructions after the first warning. In screen mode, the number on the dashed line is updated. In line mode, a period (`.`) is displayed after each number of steps. [Default: 100]

`all`
Print current settings of all `set` parameters (except `signal` and `run`). `set all` displays the *ada_library* that generates the executable program being debugged.

`async [on|off]`
Force debugger to operate in asynchronous mode. If you switch to synchronous mode while the program is running (`set async off`), the debugger stops the program. [Default: `off`]

`case [on|off]`
Make `/` and `?` searches case-sensitive. [Default: `off`]

`c_types [local|global]`
Determine the scope of the search for C-type declarations. `local` searches the current file and the `include` files for the type definition, `global` searches the entire executable and `include` files. While more comprehensive, use of `global` requires substantial additional time. [Default: `local`]

`except_stack [on|off]`
Save registers when performing exception unwinding. If `off`, the fast exception unwinding algorithms are in effect as it searches for an exception handler. However, if no handler is found, the debugger cannot determine

what exception occurred or where it occurred. If `on`, the runtime system goes more slowly, but leaves more information for the debugger. Default: [`off`]

If `except_stack` is `off` (not being used) and no exception handler is found, a message indicates the debugger does not have enough information to find the line where the exception occurred

If `except_stack` is set (`on`) and no exception handler is found, additional information about the exception is available to the debugger.

`input[pty|tty|`*filename*`]`
Set the input device for the program being debugged. If `pty`, the debugger passes input to your program. If `tty`, your program reads directly from the terminal. If *filename*, your program reads from that file. Default: [`pty`]

`lines` *number*
Change the number of lines in the display produced by the `l`, `li`, `w`, and `wi` commands. *number* is decimal. Default: [`10`]

`log [off|`*filename*`]`
Start logging to a file. If a file or `off` is not given, logging is restarted to the log file specified most recently. Default: [`off`]

`number [on|off]`
In screen mode, determine whether lines in the source window are numbered. Default: [`off`]

`obase` *number*
Set output number base for displaying commands to 8, 10, or 16. Default: [`10`]

`output [pty|tty|`*filename*`]`
Set output device for the program being debugged. If `pty`, output passes to the debugger. In screen mode, then the debugger can put the output in the lower window. Write output from your program in your log file. If `tty`, output writes directly to the screen, wherever the cursor is. If *filename*, output is written to that file. Default: [`pty`]

`page [on|off]`
Turn paging `off` or back `on` in the lower window of screen mode. [`on`]

`persist [on|off]`
The debugger retains the previous file in the source window when the current source file is not available. Default: [`off`]

`prompt` "*new_prompt*"
Change the debugger prompt to the specified *new_prompt*, which must be in double quotes. Default: [`>`]

`run` *shell_arguments*
Set up the invocation arguments for a program (I/O redirection, options, and so forth) but do not start it. Without arguments, `set run` resets the arguments list to have no arguments. The next `a`, `g`, `r`, or `s` command restarts the program.

`safe [on|off]`
Require certain single-letter commands (`a`, `g`, `r`, and `s`) to be typed twice in screen mode for safety when debugging difficult constructs or when excessive line noise is experienced. Default: [`off`]

`signal [`*signal_list*`|all] [b|g|gx]`
Set the listed signals to have the behavior specified by the last parameter. *signal_list* can be a list of the standard signal numbers or names. (The command, `kill -l`o, prints a list of signal names.

`set signal` without parameters shows the current setting for each signal. Default: [`all b`]

`b` When the signal occurs, it is announced and the program stops as if it reaches a breakpoint.

`g` The debugger does not announce the signal but continues the program execution without reasserting the signal.

`gx` On `/proc` systems, the process is configured so that, if possible, the signal is passed to the program without debugger notification (SIGTRAP cannot be handled this way). For `ptrace` systems, the signal stops the process, the debugger does not announce the signal and the program execution is continued with the signal passed to it. This option is useful when debugging programs with exception handlers for hardware exceptions.

Control the behavior of each signal separately, to ignore some and propagate others to the program being debugged.

The behavior of two signals, ALRM and CONT, cannot be changed.

Be careful in changing certain signal behavior, since `set signal INT g` prevents interruption of the program by Control-c.

`source` *path_list*
Establish the directory search path the debugger uses to find source files. *path_list* is a set of directory names, separated by spaces. Default: [current directory]

`step_alert` *number*
Print a message after a step (`s`) or advance (`a`) command steps number machine instructions. In screen-mode, the number of steps appears on the dashed line. In line mode, a period (`.`) is printed after number steps. [Default: 1000]

`tabs` *number*
Set the tab stop to *number* for listing source (`l`, `li`, `w`, `wi`, and screen mode). Default: [8]

`verbose [on|off]`
The dbx-style debugger cannot debug C files 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]

Note – This option must be set before symbolics are read for the C executable, i.e., in the `.dbrc` file before calling `a.db`.

`with` *unit_name*`|all [on|off]`
Initiate (`on`) or do not initiate (`off`) DIANA net reading for *unit_name*. If `[on|off]` is omitted, the default is `on`. `set with` without arguments returns a listing of the current settings.

These settings provide control over the debugger net reading functions. If execution halts in a unit and that unit `with`s an extraordinary number of packages, use this command to prevent the debugger from reading all the nets for all the packages.

The initial setting is `set with all on`. Then, for example:

`set with` *unit_name*
Indicates that only nets for *unit_name* are read in.

`set with` *unit_name* `off`
Indicates that all nets except those for *unit_name* are read in. Add other units to the `with` list with additional `set with` commands.

Limitation: After the debugger initiates net reading for an Ada unit, all of the nets for that unit and any unit `with`ed by that unit are read in, even if `set with` is `off` for the `with`ed unit.

xrate *number*
Set time interval for x command to display values of monitored data that has changed. [Default: 5 seconds]

Description

`set` establishes various debugger parameters. The simple command `set`, as well as the command `set all`, displays the current settings of all the parameters.

Use `set` commands on a command line or supply them to the debugger at invocation.

In Screen Mode

Precede this command with : and follow with Return.

Use `set` commands on a command line or supply to the debugger at invocation.

References

“invocation — invoking the debugger” on page 3-76
modify debugger configuration, *SPARCompiler Ada User’s Guide*

`si` — *(step instruction) single step machine code into program*

Syntax

`si` Pressing Return repeats

Description

`si` single steps the program one machine instruction and steps into a called subprogram. This is in contrast to the `ai` command, which single steps one machine instruction, but treats the machine call instruction as one instruction, stepping *over* it. Other stepping instructions are `s`, `a`, `ai`, and `gw`.

If the program has not executed yet, for example, right after invoking the debugger, `si` starts the program, stepping one instruction. This relocates the current position from the main subprogram to the actual starting subprogram preceding your current program.

In Screen Mode

Precede this command with `:` and follow with Return.

References

step instruction, *SPARCompiler Ada User's Guide*

3

signals — set/ignore signals

Syntax

```
set signal [signal_list|all] [b|g|gx]
```

Arguments

`all`
Set all signals.

`b`
When the signal occurs, it is announced and the program stops as if it reaches a breakpoint.

`g`
The signal is not announced and the debugger continues the program execution without passing the signal to the program.

`gx`

On `/proc` systems, the process is configured so that, if possible, the signal is passed to the program without debugger notification (`SIGTRAP` cannot be handled this way). For ptrace systems, the signal stops the process, the debugger does not announce the signal and the program execution is continued with the signal passed to it. This option is useful when debugging programs with exception handlers for hardware exceptions.

signal_list
List of signals to set.

Description

The debugger provides the facilities to handle signals in several manners. With the "`set`" switch, instruct the debugger to treat signals in one of three ways: `b`, `g`, and `gx`. These are described in the Arguments section above.

The default setting for most signals is `b` (break).

To display a list of signal names on UNIX-based systems, enter `kill -l`. For example:

```
>kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE BUS SEGV SYS PIPE ALRM TERM URG
STOP TSTP CONT CHLD TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH LOST
USR1 USR2
```

To display the current setting for each signal, enter `set signal` without parameters. For example:

```
>set signal
b:  hup..pipe, term..tstp, chld..USR2
g:  cont
gx: alrm
```

The debugger resets all signals to their default before starting the process being debugged. The process has the same initial settings whether it is run under the debugger or from the shell.

You cannot change the behavior of two signals, `ALRM` and `CONT`.

Be careful in changing certain signal behavior. For example, `set signal INT g` prevents interruption of the program.

References

“b — (break) break at a line or beginning of a subprogram” on page 3-16
“g — (go) continue executing” on page 3-68
“gx — (go signal) continue executing, pass the signal to the program” on page 3-70
“signal [signal_list | all] [b | g | gx]” on page 3-144
`kill(2)`, *Solaris Developer Documentation*

3

stop — stop the debugger or program

Description

Stop the program being debugged by setting a breakpoint. After the `r`, `g` or `gx` command, the program executes until it reaches the breakpoint or exits. If the program appears to be in an infinite loop, stop it by pressing Control-c.

To terminate the debugger session, use `exit` or `quit`.

References

"exit — terminate the debugger session" on page 3-64
"quit — terminate the debugger session" on page 3-118
halting the program, *SPARCompiler Ada User's Guide*

strings — string operations and support

Description

This section describes debugger support for strings.

Printing Strings

You can print or display strings with the debugger p command. This is useful for putting comments into a log file.

```
>p “this is a string”
```

You can use the p command if you want to set a breakpoint and display a message when it is reached.

```
>b 16 begin p “test first date” end
```

User Procedure Calling with Strings

Call a procedure in the program being debugged with an actual parameter that is a string constant.

```
>p lookup(“foo.baz”)
```

This works for both C and Ada. You can use multiple string parameters in any position. In Ada, the debugger builds a string array as required by the formal parameter declaration and the calling conventions of that target. In C, the debugger places a null-terminated string on the stack and passes a pointer to it as the parameter.

ssigning Strings to Memory or a Variable

You can modify memory with a string.

```
>010082 := “this is it”
```

In the following example, memory location 0F7FFEAAC is assigned the string, “this is it.” When you modify raw memory with a string, that is, when the destination is an address, the string is null terminated automatically.

```
>0F7FFEAAC:m (display one line of STORAGE_UNITS, first in hex then decimal)
 0F7FFEAAC: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
“..........”
> 0F7FFEAAC := “this is it”

> 0F7FFEAAC:m
  0F7FFEAAC: 74 68 69 73 20 69 73 20 69 74 0 1 1 1 1 1 “this is it.....”
```

You can assign a string to a variable in your program. In Ada, the variable being modified must be exactly the same length as the string constant or you get an error. Fortunately, you can use a slice as the destination of the assignment.

```
foo: string(1..10);

>foo := “1234567890”
>foo(1..5) := “12345”
>foo := “12345”
=>cannot assign to 1lhs, string sizes do not match
=> (lhs length: 10, rhs length: 5)
```

If the destination is an Ada variable, the string is not null terminated. In addition to simple variables, the destination can be any name expression, for example, `a.b(3)`, `a(3).c.d(4.9)`.

In C, the variable must be an array of char, that is, it cannot be a pointer to a string. For example, a C variable declared as

```
char good(40);
```

is correct but

```
char *bad;
```

does not work.

In C, no bounds checking is performed, and the string is `null terminated` automatically.

```
>good := “hello world”
>/hello world
>p good
“hello world”
```

String Comparison

The debugger supports relational expressions where both operands are strings. This includes the operations: <, <=, =, >=, >, and /=.

The result returned is 1 if the relation is TRUE and 0 if it is FALSE. The operands are variables, Ada slices or string constants.

For example, given the following Ada declarations in Figure 3-13:

```
foo: string(1..10);
foo2: string(1..20);

>foo := “1234567890”
/1234567890
>p foo = “1234567890”
1
>p foo = “12345678”
0
>p foo > “123456789”
1

>foo2 := “12345678901234567890”
/12345678901234567890
>p foo = foo2
0
>p foo /= foo2
1
>p foo < foo2
1
>p foo = foo2(1..10)
1
```

Figure 3-13 Example of String Comparisons

String comparison for C works in the same way, except that slices are not supported. The entire string is examined in each of the string compare commands.

String comparison is useful for conditional breakpoints.

```
>b 113 when foo(1..3) = “123”        (Ada)
```

or

```
>b 223 when name = “foo”             (C)
```

sx — *(step signal) single step, pass the signal to the program*

Syntax

```
sx
```

Description

When a signal occurs in a program being debugged, first it passes to the debugger. The debugger announces the signal and the location at which it occurs and stops, waiting for commands. To continue single stepping as though the signal has not occurred, use `s` or `si`. To continue single stepping and pass the signal to the program, use the `sx` command. This is useful in debugging programs that do explicit signal handling.

This command cancels the Return key memory.

Use two debugger parameters, `alert_freq` and `step_alert`, to track the number of instructions that are stepped. Using the default settings, a message is displayed after the first 1000 instructions are stepped (`step_alert`). After that, every 100 additional instructions stepped (`alert_freq`), generates a new message). In line mode, these messages are periods - one after the initial number of instructions are stepped with a new period displayed for each 100 additional instructions stepped.

In Screen Mode

Precede this command with `:` and follow with Return.

The number of instructions stepped appears as a number on the dashed line separating the command and source window. This number is first displayed after the first 1000 instructions are stepped (`step_alert`). This number is incremented for every 100 instructions stepped after the initial display (`alert_freq`).

References

“Return — re-execute debugger command” on page 3-128
“s — (step) single step source code into subprograms” on page 3-130
“alert_freq number” on page 3-142
“signals — set/ignore signals” on page 3-148
“step_alert number” on page 3-145

3

`task` — *print current task or choose a new current task*

Syntax

`task [`*new_task*`]`

Arguments

`new_task`

Task identifier. This identifier can be name of the task, the decimal task sequence number found in the `NUM` column of output from the simple `lt` command or the hexadecimal number indicating the task's `tcb` address. The `tcb` address can be found using the `lt task` command.

Description

The task command, without a specified new_task, displays the type, name and decimal sequence number of the current task.

```
>task
T output at 2
>task 3
2 line 200 in dining_room TASK BODY
>task
T output at 2
```

task new_task identifies a task to become the new current task. The call stack commands (`cs`, `cd`, `cu`, `cb` and `ct`) operate on the current task call stack.

When the `task` *new_task* command executes, the current task is deselected before *new_task* becomes the new current task. The state of the deselected task is stored so if it is reselected before the program executes further, the current frame is the same as it was when the task is deselected.

In Screen Mode

Precede this command with `:` and follow with Return.

Note – The `task` command contains all the capabilities of the `select` command. The `select` command is no longer supported by Ada.

References

“lt — (list tasks) list all active tasks” on page 3-93
display task number and move current position, *SPARCompiler Ada User’s Guide*

3

terminal control — catching program input/output

Description

The principal function of the `set output` command is to prevent the output of a program from overwriting the source window when debugging in screen mode. If you use `set output pty`, output from your program displays in the lower window. If you use `set output` *filename*, output from your program writes to *filename*.

The debugger uses the pseudo terminal drivers (`pty`s) to position itself between your terminal and the process being debugged. `set input` and `set output` control this. If `input` is set to `pty`, the default standard input (file descriptor 0) for your process is a pseudo terminal. Likewise, when output is set to `pty`, the default standard output (file descriptor 1) is the pseudo terminal, not the control terminal. The debugger does all I/O on behalf of the process being debugged. When `input` or `output` is set to `tty` or to a file, the process being debugged has file descriptors for the actual terminal or file (assuming no I/O redirection is given with the `r` command).

If you are logging your debugging session and `input` is set to `pty`, all input to your program appears in the log. And if `output` is set to `pty`, output from your program appears in the log.

The (default) use of `pty`s is transparent to most programs. However, when you debug a program that makes use of special video terminal features, this handling may not be desirable.

References

“input[pty | tty | filename]” on page 3-143
“log [off | filename]” on page 3-143
“output [pty | tty | filename]” on page 3-143
pseudo terminal drivers, *Solaris Developer Documentation*

vi — *(visual) switch the debugger to screen mode*

Syntax

vi [*ada_entity*|*ada_source_file*]

Arguments

ada_entity
Name of an Ada entity such as a subprogram, package, task, variable, constant, etc.

ada_source_file
Name of an Ada source file. This file must in a directory on your ADAPATH.

Description

The debugger supports two interactive interfaces: a conventional line-oriented interface, called “line mode” and a screen-oriented interface, known as “screen mode.” The vi command switches from line mode into screen mode. Once in screen mode, the Q command switches into line mode.

The debugger starts in line mode by default. Invoking the debugger with the -v option starts it in screen mode.

The parameters to vi are interpreted exactly like the parameters to the e command. When switching to screen mode, the top part of the screen displays source code. This shows the source code that surrounds the current position. If a file or subprogram name is specified, the corresponding source code displays instead of the current position.

References

“screen mode — screen-oriented debugger interface” on page 3-132
screen mode operation, *SPARCompiler Ada User’s Guide*

3

visibility rules — determine visible identifiers at a breakpoint

Description

The debugger visibility rules determine which identifiers are visible at a given point in the program execution. The current frame establishes the current visibility rules using the following model.

At any time during a debugging session, the visible program names are those that are visible upon adding a statement to the program at the point in the source text corresponding to the current home position in the current frame.

If the current frame is an Ada subprogram, package or task, Ada visibility rules are used. However, since debugging is not the same as programming, these rules are extended in two ways.

First, if a name is not found based on the normal Ada visibility rules, a search is made of the Ada library for a library unit with that name. Therefore, all library units are visible regardless of what units are withed by the current context.

Second, a program-wide search is made for subprograms when a simple name is given as the argument to the `b`, `e`, `edit`, and `vi` commands. That is, it looks at all the subprograms, tasks, and packages (with elaboration subprograms) in the entire program, for one whose simple name matches the name typed in for the command. The results of this search are merged with the results of searching using Ada visibility rules (including library search extension).

References

“current frame — current position on the call stack” on page 3-45
“current position — current position in a source file” on page 3-46
“home position — execution point in current frame” on page 3-73
names, section 4.1(2) in *Ada Reference Manual.*

w — *(window) list a group of source lines*

Syntax

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

Arguments

line
Line number of the line, used as the center of the window display.

number
Number of lines to display in the window. [Default: 10]

Description

The term "window" means a section of source text. Do not confuse it with the windows of the screen-mode debugger.

`w` lists a window of source text around the specified line in the current file. If *line* is not specified, the center of the window is the line portion of the current position. If the line specification is `*`, the line portion of the current home position is used as the center of the window. If *line* is given, the current position moves to that line, and the window surrounding that line is displayed.

The window is the specified number of lines large. (Default: 10.) Change the default with the `set lines` command.

The `w` command resets the Return key memory as if an `l` is typed instead.

In Screen Mode

Precede this command with `:` and follow with Return.

References

"l — (list) display part of a source program" on page 3-81
"lines number" on page 3-143
display lines, *SPARCompiler Ada User's Guide*

3

`wi` — *(window instruction) list disassembled and original code*

Syntax

`wi` [*expression*|*decimal_number*] [, *number*]

Arguments

expression
Expression defining the instruction address at the center of the listing.

decimal_number
Decimal number indicating the line number at the center of the listing.

number
Number of lines to display in the listing. [Default: 10]

Description

The term "window" means a section of source text. Do not confuse it with the windows of the screen-mode debugger.

`wi` prints a window containing the specific number of lines including machine instructions that are interspersed with source code. If a decimal line number is given (*decimal_number*), this line is at the center of the window. If *expression* is specified, the instruction at the address given by *expression* is at the center of the window.

If an equals sign appears after the address, a breakpoint is at that instruction.

If [*expression*|*decimal_number*] is omitted, the display is centered on the line or instruction following the most recent `w`'d or `wi`'d line or instruction or the home line or instruction if no `w` or `wi` command is given for this file. The home position is centered in the window if [*expression*|*decimal_number*] is `*`.

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

The debugger does not, in one `wi` command, disassemble across a source file boundary, no matter how many lines it is instructed to print. It stops the display at the source file boundary.

`wi` resets the Return key memory as if `li` is typed instead.

In Screen Mode

Precede this command with `:` and follow with Return.

In screen mode, the upper window (source window) toggles to display either source code or disassembled machine instructions. This screen mode command, `I` performs this toggling.

References

“li — (list instructions) list disassembled instructions” on page 3-84
“screen mode — screen-oriented debugger interface” on page 3-132
“Instruction and Source Submodes” on page 3-134
“lines number” on page 3-143
display instructions, *SPARCompiler Ada User’s Guide*

3

x — *(eXamine) monitor memory location(s)*

Syntax

```
x expr
xl
xd integer | all
xb xe
```

Arguments

`all`
(all) All memory locations. Used with `xd` (delete) command to remove all entries on the list of locations being monitored.

`b (begin)`
Begin monitoring.

`d (delete)`
Remove memory location from list of monitored locations.

`e`
(end) End monitoring.

expr
(expression) Expression to be monitored. Note that expr must resolve to a memory location. Variable names, record fields, and array elements are allowed, but not expressions that contain operators or user procedure calls. The debugger determines the address of the expression at the time the `x` command is entered. If this expression denotes a stack based memory location, and the associated stack frame is deallocated by the program, the debugger prints out erroneous data.

integer
(integer parameters) number (from `xl`) command of memory location that is no longer to be monitored.

`l  (list) List memory locations being monitored.`

Description

The `x` command is used to monitor user-selected memory locations. It can only be used with asynchronous debugging on systems that support the `/proc` interface.

The `x` *expr* command signals the debugger to do a periodic display (examination) of the designated expression, *expr*.

When the debugger starts the program running asynchronously, it examines the value of the named variable periodically. If it has changed, the debugger prints it out. The length of the time period that transpires between examinations is configurable using the `set xrate` command.

The following algorithm is used to display variables as they change. When the debugger first receives the `x` command, it allocates memory equal to the size of the variable being monitored and initializes it to all zeroes. This is the saved value. At every `xrate` seconds, the debugger reads up the current value of the variable and compares it to the saved value using a binary compare. If they are equal, it does nothing. If they are different, it dumps out the new value and transfers it to the saved value.

Note that since the debugger is examining the variables at discrete intervals, it is possible that a variable changes value and changes back to its saved value before it is examined again.

Several memory locations can be monitored by entering several `x` commands.

The `xd` command removes a memory location from the list of monitored locations. It takes a list of integer parameters which can be discovered using the `xl` command.

The `xb` command starts the monitoring process. If the program is running when the command is entered, monitoring begins immediately. Otherwise, monitoring starts whenever the program starts running asynchronously and stops when the program stops.

The `xe` command stops the monitoring process. The list of monitored memory locations is unmodified.

Example

Figure 3-15 beginning on page 3-169 is an example of the use of the data monitoring commands. Note that the debugger is operating in asynchronous mode. The example uses two files, `data.a` and `cs1.a`. Listings for these files are provided below in Figure 3-14.

```
-- data.a

with system; use system;
package data is
        type rt1 is
                record
                        f1: integer;
                end record;

        type rt2 is
                record
                        f2: rt1;
                end record;

        type rt3 is
                record
                        f3: rt2;
                end record;

        rec: rt3;

        type int_ptr is access integer;
        pointer: int_ptr := new integer;

        int: integer;
--      for int use at system.memory_address(343);
        STACK_MAX: constant := 10000;
        stack_data: array(1 .. STACK_MAX) of integer;
        stack_limit: system.address := stack_data'address;
                pragma external_name(stack_limit,
"DEBUG_STACK_LIMIT");
        stack: system.address
                        := stack_data'address + (STACK_MAX *
                                (integer'size/storage_unit));
                pragma external_name(stack, "DEBUG_STACK");
end data;
                                                        (Continued)
```

```
(Continued)
-- csl.a

with data;
with text_io;
use text_io;
procedure csl is

        delay_time: duration := 0.400;

        procedure increment (datum: in out integer) is
        begin
                datum := datum + 1;
                if datum >1000000000 then
                        datum := -1000000000;
                end if;
        end increment;

        procedure do_delay is
        begin
                --delay delay_time;
                delay_time := delay_time + 0.001;
                if delay_time > 0.75 then
                        delay_time := 0.400;
                end if;
        end do_delay;

        procedure p3 is
                local_p3: duration := delay_time * 4;

                procedure p2 is
                        local_p2: duration := delay_time * 3;

                        procedure p1 is
                            local_p1: duration := delay_time * 2;

                        begin --p1
                                local_p1 := local_p3 / 4;
                                increment(data.int);
                                increment(data.rec.f3.f2.f1);
                                increment(data.stack_data(343));
                                increment(data.pointer.all);
                                do_delay;
                                local_p1 := delay_time * 10;
                        end p1;
                                                        (Continued)
```

```
(Continued)
            begin --p2
                        local_p2 := local_p3 / 2;
                        do_delay;
                        p1;
                        local_p2 := delay_time * 20;
                end p2;

        begin --p3
                local_p3 := delay_time * 60;
                do_delay;
                p2;
                local_p3 := delay_time * 30;
        end p3;

begin --csl
        while TRUE
        loop
                do_delay;
                p3;
        end loop;
end csl;
```

Figure 3-14 `data.a` and `csl.a`

```
SC Ada Unix Debugger, Version development
     [Wed Jun 9 08:39:17 PDT 1993]
Host: SPARCY   Current directory: /vc/test
Debugging: /vc/test/csl
Wed Jun  9 08:47:13 1993

>>ada_library: /vc/test
library search list:
    /vc/test
    /usr2/ada2.1/self_thr/standard
>g
csl
Starting program running ...

-- At this point the program is running asynchronously.
-- The next command says to add the local variable "delay_time"
-- to the list (currently empty) of data begin monitored.
>x delay_time
Monitoring delay_time.
-- Now add a simple package level variable to the list.
>x data.int
Monitoring data.int.
-- A field of a record.
>x data.rec.f3
Monitoring data.rec.f3.
-- An array element.
>x data.stack_data(343)
Monitoring data.stack_data(integer(343)).
-- Another array element.
> data.stack_data(342)
Monitoring data.stack_data(integer(342)).
-- A dereferenced access value
> data.pointer.all
Monitoring data.pointer.all.
-- Now list all of the data being monitored.
>xl
[1]  delay_time
[2]  data.int
[3]  data.rec.f3
[4]  data.stack_data(integer(343))
[5]  data.stack_data(integer(342))
[6]  data.pointer.all
                                          (Continued)
```

```
(Continued)
    -- Now tell the debugger to begin the monitoring process.
    >xb
    -- Now every 5.0 seconds (the default since mrate was not
    -- changed using the set mrate command) the debugger dumps
    -- out the values of data that have changed.
    -- Note that data.stack_data(342) isn't being printed out,
    -- since it's not changing.
    Modified data:
      delay_time: 0.625
      data.int: 6946105
      data.rec.f3: f2: RECORD
        f1: 6947114
      data.stack_data(integer(343)): 6947577
      data.pointer.all: 6947835
    Modified data:
      delay_time: 0.748
      data.int: 7554033
      data.rec.f3: f2: RECORD
        f1: 7554784
      data.stack_data(integer(343)): 7555218
      data.pointer.all: 7555328
    Modified data:
      delay_time: 0.714
      data.int: 8160714
      data.rec.f3: f2: RECORD
        f1: 8161722
      data.stack_data(integer(343)): 8162265
      data.pointer.all: 8164389
    Modified data:
      delay_time: 0.443
      data.int: 8768462
      data.rec.f3: f2: RECORD
        f1: 8769242
      data.stack_data(integer(343)): 8769828
      data.pointer.all: 8770312
    Modified data:
      delay_time: 0.48
      data.int: 9376650
      data.rec.f3: f2: RECORD
        f1: 9377638
      data.stack_data(integer(343)): 9378197
      data.pointer.all: 9380123
    -- Now tell the debugger to stop monitoring, but the program
    -- is still running.
    >xe
                                                        (Continued)
```

```
-- List out the monitored data again.
>xl
[1]  delay_time
[2]  data.int
[3]  data.rec.f3
[4]  data.stack_data(integer(343))
[5]  data.stack_data(integer(342))
[6]  data.pointer.all
-- Delete the record field and dereferenced pointer from the
-- list of monitored data.
>xd 3
>xd 6
-- Start monitoring again.
>xb
-- Now the monitored output only checks 4 items.
-- data.stack_data(342) is still being monitored, but it's
-- still not changing, so it's not being printed out.
Modified data:
  delay_time: 0.567
  data.int: 16270142
  data.stack_data(integer(343)): 16439472
-- Change the debugger so that it now dumps out the values
-- every 7.5 seconds.
>set xrate 7.5
Modified data:
  delay_time: 0.712
  data.int: 18938363
  data.stack_data(integer(343)): 18927946
>xe
```

Figure 3-15 Use of x Commands

References

“asynchronous debugging — run the debugger in asynchronous mode” on page 3-11
`set xrate`, “set — set debugger parameters” on page 3-142

"I am strong as a bull moose and you can use me to the limit."

Theodore Roosevelt

Limits A

A.1 Compiler and Tool Limits

This section provides a list of limits for the Ada compiler and tools.

499	Characters in identifiers and literals
4,000,000	Storage units in a statically-sized record type
10,240	Default storage size for a task[1]
400	`STORAGE_SIZE` default collection size for access type[2]
no limit	Number of declared objects (except virtual space)
10,240	Characters in a rooted name (full path of an object)
256,000,000	Maximum size of an array (in bits)
4	Number of recursive inlines
50	Number of nested inlines[3]
400	Number of nested constructs
2048	Characters in a `WITH`*n* or `INFO` directive
200 MB	Memory use per compilation (other OS limits may apply)
50	Lexical errors before the front end exits
100	Syntax errors before the front end exits
10	Attempts to lock `GVAS_table`

(Continued)

10	Attempts to lock `ada.lib`
20	Attempts to lock `gnrx.lib`
64	Debugger breakpoints
32	Debugger array dimensions in a `p` command
100	Debugger ‘call parameters’
256	Debugger ‘run parameters’
512	Number of arguments the debugger can support in the program being debugged
8K	Total space available to hold arguments of program being debugged

1. If tasks need larger stack sizes, the `STORAGE_SIZE` attribute may be used with the task type declaration

2. If tasks need larger stack sizes, use the 'STORAGE_SIZE attribute with the task type declaration

3 The default is 4 but can be changed using the `MAX_INLINE_NESTING INFO` directive.

ADAPATH: The limit on *each component* of the ADAPATH is the operating system limit on filenames. The full ADAPATH, however, is unlimited.

A.2 Source File Limits

499	Characters per source line
1296	Ada units per source file
32767	Lines per source file

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y