DEBUGGER REFERENCE MANUAL

Manual Number: MN253

10 March 1986

Valid Logic Systems, Incorporated
2820 Orchard Parkway
San Jose, California 95134
(408) 945-9400 Telex 371 9004

TABLE OF CONTENTS

Introduction 1-1

Running the Debugger
Setting the Compiler Debug Flag 2-1
Obtaining a Debug Information File 2-2
Debugging a Target Program 2-4

Debugger Concepts and Commands
Concepts and Definitions 3-1
Debugger Commands 3-5
R - Run 3-6
Q - Quit 3-6
B - Breakpoints 3-7
T - Tracepoints 3-8
C - Clearing Break and Tracepoints 3-9
P - Print the Value of a Variable 3-11
S - Set the Value of a Variable 3-12
M - Memory Set/Print 3-14
W - Walkback 3-15
U - Move Environment Up 3-15
D - Move Environment Down 3-16
L - The List Command 3-16
< - Take Debug Commands from a File 3-19
> - Save Break and Trace Points 3-19
! - Execute Shell Command 3-20
I - Execute a Single Instruction 3-20
N - Execute Next Statement 3-21

.DBG File Format
The .DBG Header 4-1
The .DBG Link Map 4-2
VAriable and Type Descriptors 4-4
Type Descriptors 4-5
Variable Descriptors 4-9
Statement Offsets Descriptors 4-12

SECTION 1
INTRODUCTION

The SVS symbolic debugger allows the interactive execution and debugging of programs written in SVS **FORTRAN**, SVS **Pascal**, and SVS **C**.

Execution of a target program can be breakpointed or traced at the entry points or exits of subroutines, or at any statement boundary within a subroutine. After execution is suspended at a breakpoint, the values of variables and data structures can be examined and altered using their symbolic names as they are known in the environment in which the breakpoint occurred. When debugging **Pascal** and **C**, the subroutines which are active can be displayed as a calling sequence backtrace, and the debugger can be directed to change its current symbol naming environment to any of those which are active. Execution of the target program can be continued or terminated from a breakpoint, possibly after break and/or trace points are set and/or cleared.

Low level operations for displaying and setting memory locations by address are also available. Break and trace points may also be set at arbitrary addresses, although the debugger offers a more limited set of functions at these breakpoints if they occur in code which falls outside of the normal breakpointing areas.

The debugger is entirely driven by tables of information which describe the program being debugged. Thus, the executable image of a target program is identical whether or not it is run under the symbolic debugger. This allows for debugging sessions after programs are placed into production, eliminates problems related to finding program behavior which comes and goes with minor changes in the program or in the code generated for it, and allows for full speed execution under the debugger.

Debugger tables are created under control of compiler option flags which can be toggled on and off as desired. The size of *unlinked* object code will be larger based on the amount of symbolic information included. The linking step of the compile can be directed to consolidate the debugging information from the object files which are being linked and to place it in a file which is subsequently available for utilization by the debugger. It is possible to link object files created from more than one compile and from more than one source language, some with, and some without symbolic debugging information included. The symbolic debugger is designed to operate with partial information and with routines which originate from multiple source languages.

The command language for the debugger is terse so that an experienced user can operate it without overly cumbersome typing. The number of commands has been kept down and user friendly hints are provided by the system so that users can develop expertise in operating the debugger quickly.

SECTION 2
RUNNING THE DEBUGGER

In order to utilize the symbolic debugger, a source program should be compiled with the debugging flag set, the linker should create a file of information for the debugger to utilize, and the target program should be executed under the debugger.

2.1 SETTING THE COMPILER DEBUG FLAG

By default, the compilers do not place symbolic information into the generated unlinked object code. The SVS **Pascal**, SVS **FORTRAN**, and SVS **C** compilers will insert this information if they are invoked with the +d command line option. This option may appear anywhere on the command line. Alternatively, symbolic information can be turned on and off on a per procedure basis in each of the languages as follows:

Pascal	Comment toggle $D+ for debugger information on, $D- for debugger information off.
FORTRAN	Compiler control line beginning in column one $DEBUG for debugger information on, $NODEBUG for debugger information off.
C	Compiler control line beginning in column one #debug for debugger information on, #nodebug for debugger information off.

Setting the debugger flag has no effect on the generated object code except that certain additional tables of information are placed into the unlinked object code. This information is either consolidated or ignored by the linker, see below. The executable image produced by a program is identical regardless of whether or not the debug flag is set when the program is compiled.

2.2 OBTAINING A DEBUG INFORMATION FILE FROM THE LINKER

On the Motorola 68000 implementation, when creating an executable program, unlinked object code files (.obj files) are linked with each other and with appropriate run time libraries utilizing a linker program. The linker can be directed to consolidate all of the debugging information in the object files it is linking and create, in addition to its object code output file, another output file containing this information for later use by the debugger. By convention, this file will have the file name extension .dbg. Alternatively, the linker may ignore the debugging information in its input files.

The linker differs among the various environments in which the SVS languages operate. In most environments the linker accepts an optional command line argument:

+ sxxx.dbg

in any position on the command line as a directive to create the information file and name it xxx.dbg. This is the preferred name for the information file which is to be associated with an executable program named xxx (with possible system dependent extension indicating that the file is executable). If the linker is operated in prompting mode, the operation of the various linkers differs somewhat. Some of the linkers prompt:

Symbol file -

To this prompt, a plain carriage return indicates that no debugger information file is to be created. Any other input is considered a directive to create a debugger information file with the supplied name. The linker will supply the .dbg file name suffix if it is not specified in the file name. The other linkers do not prompt for the name of the debugger information file. These linkers accept the input:

+ sxxx.dbg

to any prompt to direct the linker to create an information file with the indicated name. No automatic coercion of the file name suffix is done by these linkers. This input can be specified to any prompt with identical results, regardless of whether input files have or have not already been processed.

On the National Semiconductor Series 32000, the linker automatically produces a .dbg file if the debug option had been set during the compile being processed. These .dbg files may be concatenated (for example using cat on UNIX) to create a debugger information file containing information from multiple compiles for use by the debugger.

Having created the .dbg debugger information file, the remaining steps of the compile should be carried out, resulting ultimately in an executable image associated with this symbolic information. It is essential that when debugging a given program, the matching .dbg file be provided to the debugger, since the symbolic information will otherwise not match the program being debugged. Under some operating systems, the debugger will issue a warning if incompatible symbolic and executable files are utilized, but under some environments it is not possible to detect mismatches.

The debugger can be used on programs with subroutines from non SVS languages. Of course, the debugger will not have any symbolic information about the portions of the program which are not compiled under SVS languages.

Note: when performing the final link under UNIX like operating systems, subroutines provided from non SVS languages should be linked into the image following the output of the SVS linker. If too much code is linked before the output of the SVS linker, the debugger may not be able to find the target image for debugging purposes.

2.3 DEBUGGING A TARGET PROGRAM

The debugger itself is a program named *dbg* (with appropriate file name extensions in some environments to indicate that the file is executable). In order to debug a program named 'x' (possibly with file name extension as above) and command line arguments to the program being debugged p1, p2, ... pn, use the command:

```
dbg [-sxxx[.dbg]] x p1 p2 ... pn
```

The optional -sxxx (or -sxxx.dbg) instructs the debugger to utilize the file xxx.dbg as the debugger information file. If this argument is omitted, the debugger will look for a file named x.dbg as the debugger information file. In certain environments, command line arguments are separated by commas.

Under operating systems which allow redirection of standard input and standard output, the debugger can cause a program to be executed with either, or both, redirected. This is accomplished by invoking the debugger with the -iinfile (input) or -ooutfile (output) options as follows:

```
dbg [-sxxx[.dbg]] [-iinfile] [-ooutfile] x p1 p2 ... pn
```

The -s, -i, and -o options may be specified in any order, but all must precede the program name and its command line arguments. An example of invoking the debugger with input redirected is:

```
dbg -idatafile processdata processdatacommandlineargument
```

The debugger will read the .dbg file and prepare the target program for execution. When this has occurred, the debugger prompt (a minus sign) appears and debugger commands, as described below, may be entered. At this point, the target program is not broken inside any environment, so no variables are available to be examined or set (see discussion of environments below). From this point the program may be run with break and/or trace points set, and various of the debugger tables may be examined.

Running the target program after setting a break point at the entry of the main program (named in the program statement in **Pascal** or **FORTRAN**, and named *main* or *_main* in **C**) will bring the debugger into a meaningful environment from which static (global) data areas are accessible.

SECTION 3
DEBUGGER CONCEPTS AND COMMANDS

3.1 CONCEPTS AND DEFINITIONS

Several concepts and definitions used in explaining debugger commands are listed below.

ENVIRONMENTS

When interacting with the debugger, the program is, in general, suspended at a break point. If this break point is within the bounds of a subroutine (used throughout to mean any procedure, function, or main program body), the symbolic names which are available to the target program within that subroutine are the current naming environment. The interpretation of data is based on the attributes of the name in that scope.

This concept also applies to static (global) data areas, in that the interpretation of static data areas may depend on the common and/or equivalence statements in the broken subroutine, whether or not the subroutine is within a **Pascal** unit, and whether or not the current environment belongs to a subroutine written in the **C** language.

At times the debugger is suspended outside of any environment. This is the case immediately after the debugging session is initiated. It also occurs when a breakpoint is encountered at an address which precedes the completion of the entry code or follows the start of the exit code of a subroutine, or at a breakpoint in a routine which was not compiled with the debugging option enabled. In these cases, no variables may be accessed symbolically.

The debugger allows display and manipulation of the current environment from certain breakpoints. In cases where it is possible, the debugger will allow interactive changing of the current environment to the environment from which the current subroutine was called, potentially

all the way back to the main program. As the current environment is changed, the debugger's perception of the data areas will reflect the local declarations and source language of the new environment.

Naming conventions which are altered by the scope of **Pascal** WITH statements do not affect the naming conventions used by the debugger.

BREAKPOINTS

Under interactive control, certain addresses in the target program's executable object code can be designated breakpoints. In the event that such an address is executed, except as noted below, control is returned to the debugger command level for further interactive debugging. The address must correspond to the first word of a 68000 instruction, although the debugger will automatically insure this for entry, exit, and statement breaks (i.e. all breakpoints except those set using the arbitrary address breakpoint specification).

The debugger will allow only one breakpoint per address, regardless of whether that address can be described in several different ways. For example, a given address which is the entry address of a subroutine is usually also the address of the first statement of that subroutine. In the case of null statements, several statements may begin at the same address. The debugger designates each breakpoint in all of the appropriate ways, relative to entries, exits, line numbers, and actual addresses, regardless of which method was utilized to set the particular breakpoint. In the event that an attempt is made to set more than one breakpoint on an address, the debugger will describe the situation and show the result of the conflicting commands.

An address on which a breakpoint is set can have skip counts associated with it, such that not every execution of the breakpoint returns control to the debugger. This is further described in the section which explains the breakpoint command.

TRACEPOINTS

As an alternative to setting a breakpoint on a given address, a tracepoint may be set. When executed, a tracepoint causes an informative message to be displayed, but, unlike the breakpoint, the program continues its execution. Tracepoints also may have skip counts which control the frequency with which the trace message is displayed.

In the event that both a trace and break are set on the same address, the breakpoint takes precedence, although the tracepoint becomes active again if the breakpoint is cleared. As described under breakpoints, it is the address which has the tracepoint properties, regardless of which method was used to create the tracepoint. It is also the address which has the skip count properties, so it is not possible to set the counts independently for a tracepoint and a breakpoint which are associated with the same address.

PNAME

A pname refers to any object code entry point that is resolved by the SVS linker. In general they are procedure, function or subroutine names. They can also refer to external or global entry points in assembly language modules.

A pname can be either a user name or a link name. A user name is the name of a procedure as it appears in a user's source program. The debugger treats the case of user names in the same manner that each programming language does. That is, user names ignore case when compared with **Pascal** or **FORTRAN** entry points, but preserve case for **C**. There can be several instances of the same user name in a single executable program, due to nested scopes, static functions, etc.

A pname can also be a link name, i.e. the name used by the linker to resolve code entry points. Link names come in two forms: local and global. Local link names are of the form dollar sign ($) followed by digits, as in "$12000". All others are global.

3.2 DEBUGGER COMMANDS

The debugger prompt character is the minus sign. To this prompt, any of the debugger commands may be entered, although the system may not accept certain commands in inappropriate environments.

In general, blanks (spaces) are significant in debugger commands, but not necessary unless omission of the blank causes the command to have a different interpretation. Thus, the debugger commands

```
B 32 E F
```

and

```
B32EF
```

(see breakpoint command below) are both acceptable to the debugger and are equivalent. The debugger would not accept

```
B 3 2 E F
```

since the syntax of the break command calls for at most one numeric value between the B and the E and the blank creates two such values. Where possible, the debugger accepts either upper or lower case letters. Exceptions are C names, link names, and string values, for which case is significant.

If a partial or incorrect command is encountered, the debugger responds with a short summary of the commands available at the point that the error is detected. For example, the incomplete command 'B' would generate the line:

```
?BE BLnnn BX BA
```

indicating that the debugger was expecting the B to be followed by an 'E' for breaking at an entry point, or an 'L' followed by a line number for the break, and so forth.

R — RUN

Control is transferred to the target program by issuing the run command to the debugger. It is simply:

R

The program executes as if it were not running under the debugger until a breakpoint is encountered, with the exception that messages are printed on each instance of execution at a tracepoint. When a breakpoint is encountered, control is returned to the debugger for further interactive command dialog. If the program terminates normally, an appropriate message is printed. In most instances in which fatal run time errors are detected in the program control returns to the debugger, although the debugger will, in general, not allow the program to be restarted in this situation.

Q — QUIT

A debugging session may be terminated by entering the quit command:

Q

This command prompts for confirmation:

Exit program (Y/N) ?

If the user really wishes to terminate the debugging session, enter a Y, followed by a carriage return. Any other input to the confirmation request cancels the quit command, and returns to the debugger for continuation of the debugging session.

B — BREAKPOINTS

The breakpoint command is used to set new breakpoints in a program being debugged. The general form of a break point command to set a controlled breakpoint is:

```
                    {E   }
B   [nnn[*]]        {X   }    pname
                    {Lnnn}
```

The general form of a break point command to set an uncontrolled breakpoint (set by address) is:

```
B   [nnn[*]]   A   {pname [+ hexconstant]}
                   {hexconstant [+ pname]}
```

The optional integer count which follows the B command, nnn, is interpreted to mean "break after nnn instances of the specified breakpoint have been encountered". If omitted, the effect is as if a count of 1 were specified. If the asterisk follows the count, the breakpoint is activated on every nnn'th encounter. If no asterisk is specified, the first nnn encounters of the breakpoint are skipped, but each subsequent encounter causes a break.

The pname in the command is the entry point name of the subroutine relative to which the breakpoint is set. The breakpoint may be set at the entry (E) of the subroutine, the exit (X) of the subroutine, or before the nnn statement (Lnnn) of the subroutine. Compiler generated program listings are very useful in determining the statement number of each source code line in the target program.

A given subroutine sets up its run time conditions (stack frame creation, code to copy value parameters, initialization of register pointers, etc.) in some code at the start of the routine in what is commonly called the subroutine entry code. Breaking on entry to a subroutine occurs after the run time conditions for the routine have been set up, which is the earliest that the environment of the subroutine is meaningful. Similarly, breaking on exit occurs just before the subroutines run time conditions are unwound. As a

consequence of this, it is only possible to break on entry and/or exit of a subroutine which was compiled with the debugging option on, since the debugger requires information as to the length of the subroutine entry and exit code in addition to the simple address at which the entry point occurs.

In the case of of uncontrolled breakpoints, the pname, if provided, refers to the address which is the actual start of the code for the subroutine. The hexconstant is specified without a leading dollar sign character. If the hexadecimal address immediately follows the "A", the debugger requires the constant to begin with a recognizable digit (which may be provided as a leading zero if necessary) to prevent ambiguity with the syntax of pname's.

Note that using uncontrolled breakpoints, any address may be specified and it is the user's responsibility to insure that the breakpoint is set at an address which is the first byte of an actual instruction. An incorrectly placed uncontrolled breakpoint will usually result in disaster!

T — TRACEPOINTS

The tracepoint command is used to set new tracepoints in a program being debugged. The general form to set a controlled tracepoint is:

```
                 {E   }
T   [nnn[*]]     {X   }    pname
                 {Lnnn}
```

The general form of a trace point command to set an uncontrolled tracepoint (set by address) is:

```
T   [nnn[*]]   A   {pname [+ hexconstant]}
                   {hexconstant [+ pname]}
```

The tracepoint command is exactly the same as the breakpoint command except that the effect of encountering a tracepoint while executing the target program is to print a message, and then to continue execution without breaking. The debugger allows breakpoints and tracepoints to be set to occur upon encountering the same address, although in this case the tracepoint is inactive until the breakpoint is cleared.

Like breakpoints, tracepoints must only be associated with the first byte of an actual instruction.

As explained in more detail in the concepts section of this manual, the counts are associated with the address on which the break or trace point occurs. Thus, it is not possible to set both break and trace points on the same address with differing skip counts.

C — CLEARING BREAK AND TRACE POINTS

The debugger command to clear a breakpoint specifying the address by entry, exit, and line number is:

```
C   [B]   [nnn[*]]   {E   }
                     {X   }   pname
                     {Lnnn}
```

Alternatively, a breakpoint may be cleared by referring to the address directly as follows:

```
C   [B]   [nnn[*]]   A   {pname [+ hexconstant]}
                        {hexconstant [+ pname]}
```

To clear tracepoints, the following commands are accepted:

```
C   T   [nnn[*]]   {E   }
                   {X   }   pname
                   {Lnnn}
```

or

```
C   T   [nnn[*]]   A   {pname [+ hexconstant]}
                        {hexconstant [+ pname]}
```

The optional count and asterisk is allowed by the debugger for consistency with the breakpoint and tracepoint command but is ignored by the clear break and trace point command. The optional B in the clear breakpoint command is provided for consistency with the clear tracepoint command and is ignored.

A break or trace point is associated with an address. The break or trace point can be cleared using any of the available methods of specifying the address, regardless of which description of the address was used when the break or trace point was set.

In the event that both a break and trace point are set on the same address, clearing one leaves the other set.

Short commands are accepted by the debugger to clear all break points, all trace points, or all break and trace points. These commands are:

CB* - Clear all break points.
CT* - Clear all trace points.
C* - Clear all break and trace points.

P — PRINT THE VALUE OF A VARIABLE

The print command is used to print either the value of a single variable, or the value of all variables in the present most local scope. The general form is:

P [var]

A solitary P prints the current value of all printable variables in the present local scope. If a variable name follows, then only the value of that variable is printed. Normally the only values that can be printed are simple variables. Structured values such as arrays, records, unions, etc. cannot be printed, but a single element of an array, field of a record, member of a union, etc. can be printed by using the normal syntax expected in the appropriate programming language to access subportions of structured types. Using the solitary P does not print all of the variables available in the environment, only those is the most local scope.

In the event that the target program is suspended outside of a known environment (see the concepts section of this manual), it is not possible to print variables symbolically.

S — SET THE VALUE OF A VARIABLE

Values of variables in the current environment can be set interactively using the debugger's S command. Its syntax is:

S var [=, :=] value

The debugger accepts either assignment operator between the variable to be set and the value it is to be set to, but does not require that an operator be present. The variable may be simple, or include indexing (by constant indices), field reference, indirection, etc. The value assigned to the variable is expected to be an integer or real constant, possibly preceded by a minus sign, or a string constant. The value must be appropriate for assignment into the variable. This implies that the variable, considered with all of its qualifications, must not be of a structured type. The possible variable type, value type combinations accepted are summarized in the following table:

Language	Var Type	Value Type	Notes
Pascal	integer	integer	
	scalar	integer	First=0, next=1, ...
	real	floating-point or integer	Includes double
	Boolean	integer or TRUE or FALSE	0=FALSE, 1=TRUE
	char	string	length must be 1
	string	string	
	pckd array of char	string	Trailing blank filled
	pointer	integer or NIL	
FORTRAN	integer	integer	
	real	floating-point or integer	incl. double precision
	logical	integer or TRUE or FALSE	0=.FALSE., 1=.TRUE
	char	string	Trailing blank filled
C	integer	integer	
	char	string	
	float	floating-point	includes double
	pointer	integer or NIL	

In the event that the target program is suspended outside of a known environment (see the concepts section of this manual), it is not possible to set variables symbolically.

M — MEMORY SET/PRINT

In addition to symbolic access to data areas, the debugger also allows the low level operations of printing and setting memory locations by address. On systems in which the debugger operates in a different address space than the target program, the address referred to will be interpreted in the target program's address space. All addresses and values input to or printed by the memory commands are in hexadecimal. The commands are:

M P xxx [xxx]

M S xxx xxx [xxx] ...

The memory print command expects the initial address to be specified. This is optionally followed by another hexadecimal number which is interpreted as the number of bytes to print if it is less than or equal to the initial address or as the final address to print if it is greater than the initial address. In the absence of a length, 16 bytes of memory are displayed.

The memory set command expects an initial address to be followed by one, or more, values to be placed into successive locations beginning at that address. Each value will be interpreted as a single byte, a pair of bytes, or four bytes, depending on the number of hex digits specified in the value. That is, one or two hex digits sets a single byte, 3 or 4 sets two bytes, and 5 or more sets 4 bytes. If more than 8 contiguous hex digits are specified only the last 8 are used.

W — WALKBACK: PRINT CALLING SEQUENCE

When a program is suspended at a breakpoint, it is often possible to determine where the current subroutine was called from, where that calling context was called from, etc. Where the information is available, this calling sequence walkback can be printed by the debugger using the following command:

W [nnn]

The optional integer argument limits the number of levels back that the debugger will show. The default number of levels shown is three. In the event that the debugger can not show the walkback, an appropriate message is printed. In some contexts the debugger attempts to walkback beyond the main program, resulting in an indication of an environment for which no information is known.

U — MOVE ENVIRONMENT UP

If a target program is suspended at a breakpoint for which it is possible to walk back through the calling environments, it is also possible to change the current environment of the debugger to be one of those calling environments. By doing this, it is possible to operate on the state of the target program in these other environments, including accessing data structures which are not visible in the environment of the breakpoint itself. Moving up an environment corresponds to changing to the calling context of the current environment. The debugger command to do this is:

U [nnn]

The optional count allows moving up through more than one environment at a time. Note: if the target program is restarted using the run command, execution continues from the breakpoint's environment, regardless of what environment has been set as current for the debugger.

L E — List Entry Point Attributes

L E [pname]

If a pname is specified, detailed information is displayed about the particular entry point. If pname is omitted, all entry point names and their attributes are shown (except those for those names beginning with the percent character which is heavily utilized by the run time libraries). Attributes always include the address at which the entry point is located and, where known by the debugger, the language which generated the entry point. For assembly and unknown languages, ??? is displayed as the language. In the event that the pname specified is the percent character, all entry points are displayed, including those which begin with the percent character.

L V — List Variable Attributes

L V var

This command may only be applied to variables which are accessible in the current environment. Attributes of variables include their type or type number and some indication of their storage location.

L F — List Record Fields / Struct or Union Members

L F var

For record variables which are accessible in the current environment, this command lists information relating to each field and its addressing attributes.

L T — List Type Description

L T [nnn]

If the optional integer is omitted, information is displayed about all numbered types. If the optional integer is specified, the debugger shows information only about the type with that number.

L S — List Segments Attributes

L S

A list of the program's segments and their attributes is displayed. This information is primarily meaningful under operating systems for which segments are usefully managed by programmers.

L D — List Data Areas Attributes

L D

A list of the program's static data areas and their attributes is displayed.

L R — List Registers Values

L R

This command displays the contents of the registers in the current environment whose values can be determined.

< — TAKE DEBUGGER COMMANDS FROM A FILE

A command of the form:

< filename

is interpreted by the debugger as a directive to accept commands from the named file instead of from the standard input. When the file has been fully processed, the debugger returns to the interactive debugging mode. Utilizing this feature, it is possible to conveniently set up break and trace points for repeated sessions with the debugger, although the commands accepted from the file are not limited to these commands.

> — SAVE BREAK AND TRACE POINTS IN A FILE

A command of the form:

> filename

creates a file with the current break and trace points. The file is suitable for reloading using the < command described above. The break and trace points are saved by address (as opposed to by entry, exit, or line number) so that care should be taken if the saved break and trace points are reloaded to debug binaries which are not identical to the one which created the command file.

The debugger places count directives into the created file which correspond approximately, but not always perfectly, to the counts associated with the break and trace points.

! — EXECUTE OPERATING SYSTEM SHELL COMMAND

Under certain operating systems, the debugger will pass commands to the operating system shell for execution. The form of the command is:

!anything, possibly including blanks and delimiters

The argument string is passed directly on without any interpretation or changes.

I — EXECUTE A SINGLE INSTRUCTION

Under certain operating systems, programs can be single stepped on an instruction by instruction basis. Debuggers operating under those conditions accept the following command:

I [nnn] [q]

This instructs the debugger to execute nnn machine instructions (one instruction if nnn omitted) and then return to the debugger as if a breakpoint had occurred. The optional quiet (q) argument is used to silence the default output that the debugger will produce on each instruction if the quiet option is omitted.

This debugger command is not implemented on any system in which trap instructions in the operating system interfaces accept parameters from the instruction stream (including UNIX and UNIX lookalike operating environments).

Notes: Using this instruction execution mode results in extremely slow target program execution. Also, using this mode it is possible to get the debugger breakpointed in program sections which were not compiled with the debug option set, are part of the language's run time system, or even inside the operating system. Potentially running in single instruction mode, code located in ROM and/or trap instructions with local parameter and return conventions may be encountered, causing the instruction mode to fail, possibly with ungracious behavior. The instruction mode should not be considered a supported feature, although it does sometimes provide a useful and working function.

N — EXECUTE NEXT STATEMENT

Under certain operating systems, it is possible to instruct the debugger to operate in single instruction mode until the next statement boundary is encountered. The command for this is:

N [nnn]

The optional count specifies the number of statements (default one) to process before the debugger breakpoints.

All of the notes mentioned in the section on single instruction execution apply to the next statement command.

SECTION 4
.DBG FILE FORMAT

All symbolic information required by the SVS debugger is contained in an auxiliary file. This file is created, upon request, by the linker and is given a suffix .dbg. The executable object code generated by SVS **Pascal**, **FORTRAN** and **C** compilers is not altered by the setting of the debug option. Thus, a production program can be debugged without recompilation if the .dbg file is saved.

A .dbg file consists of 3 major sections: link map information, variable and type definitions, and statement beginning offsets in the object code. Each major section is created by a different component in the SVS compilation system. The link map information is output by the linker, variable and type definitions are generated by the language front end, and the object code statement offset tables originate in the code generator.

Implementations on the National Semiconductor Series 32000 may produce a separate .dbg file from each run of the linker. These files may be concatenated in order to create a .dbg file for the program. This section describes one such file, although the debugger accepts file formats which are repeated instances of the following format.

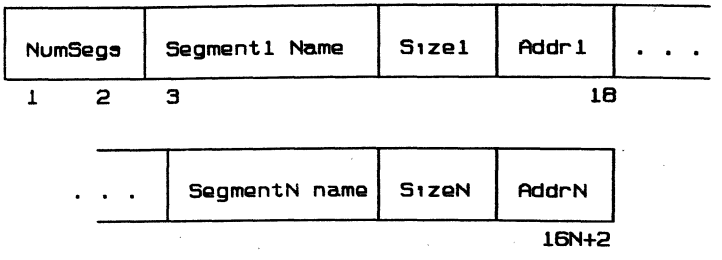
4.1 THE .DBG HEADER

The first 16 bytes of a .dbg file are the header. The contents of the header depends upon the target operating system.

4.2 THE .DBG LINK MAP

The first section of a .dbg file describes the link map of the associated program. This section contains three subsections: segment definitions, link name entry points, and data area names.

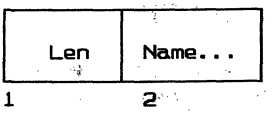
The form of the segment definition table is:



NumSegs	The number of segments
SegmentK Name	The 8 character name of segment K
SizeK	The size in bytes of segment K
AddrK	The load address of segment K

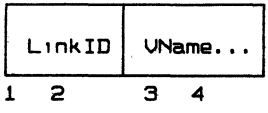
Several of the various tables which follow contain VNames. This is short for variable length names.

The format of a Vname is



Len	A byte containing the number if characters in this name.
Name...	The actual name. If Len is zero, there are no bytes in this field.

In addition, link name entries are used. They are of the form:



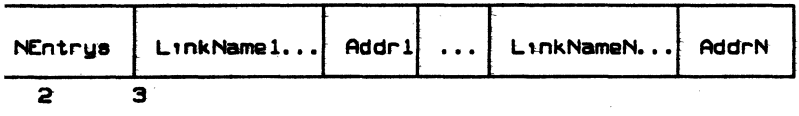
LinkID A two byte integer interpreted as follows:

If negative, the following name is to be used for linking purposes.

Otherwise, the value of the integer is to be used for linking purposes and the following name is the user name.

VName The name of this item, in VName format.

The form of the entry point table is:

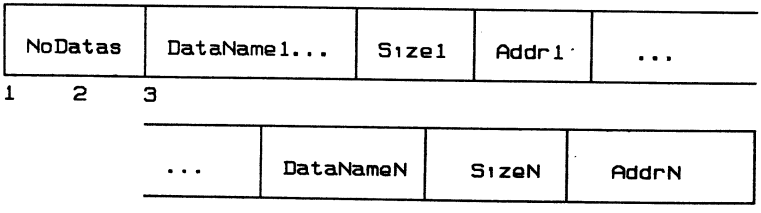


NEntrys **The number of entry points.**

LinkNameK **The link name entry (LinkID, Len, Name) for entry point K.**

AddrK **The address of entry point K. The first byte is the segment number, and the value in the last three bytes is the segment relative offset.**

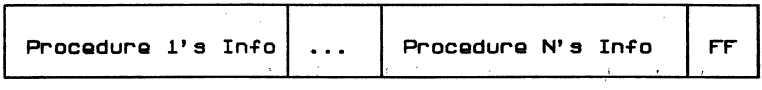
The form of the data area table is:



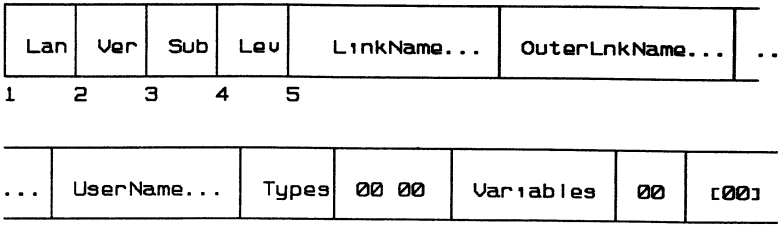
NoDatas	The number of data areas.
DataNameK	The link name entry (LinkID, Len, Name) of data area K.
SizeK	The size in bytes of data area K.
AddrK	The load address of data area K.

4.3 VARIABLE AND TYPE DESCRIPTIONS

The general form of this section is:



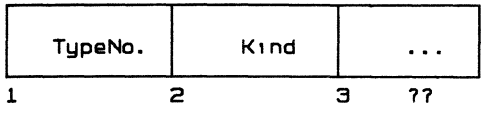
That is, it is a list of subsections, each describing the types and variables of a single procedure, terminated by a single byte of $FF. Each procedure's information is in the form:



Lan	The language in which this procedure is written. 0 = **Pascal** 1 = **FORTRAN** 2 = **BASIC** 3 = **C**
Ver	The language version number.
Sub	The language sub-version number.
Lev	The procedure's static level.
LinkName	The link name entry (LinkID, Len, Name) of this procedure.
OuterLnkname	The link name entry (LinkID, Len, Name) of an enclosing procedure. If none exists, this field's length is zero.
UserName	The user name of this procedure. It is in VName format.
Types	A description of any types defined by this procedure. Its format is given below.
Variables	A list of any locally defined variables. An extra terminating zero byte (Opt 00) is appended to the variables if needed to make the entire record even in length. The format for variables is given below.

TYPE DESCRIPTORS

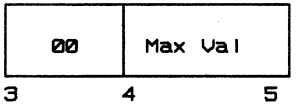
The form of a type descriptor is:



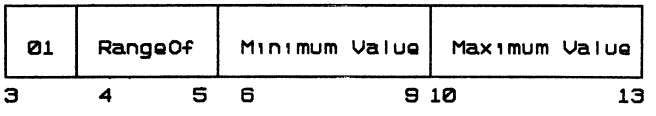
TypeNo. A 2 byte positive integer that is used to refer to this specific type.

Kind A byte containing a packed flag (bit 4) and a variant tag (bits 0..3). The format of the following information depends upon the value of the tag.

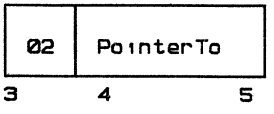
SCALAR:



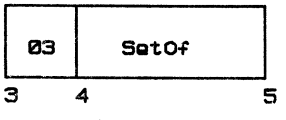
SUBRANGE:



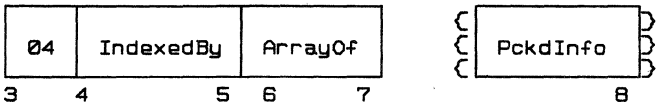
POINTER:



SET:

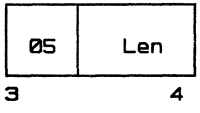


ARRAY:

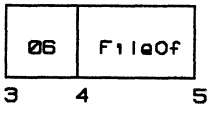


PckdInfo Signed bit is 1 if signed, bits 0..3 are size in bits of element. This field is only present for packed arrays.

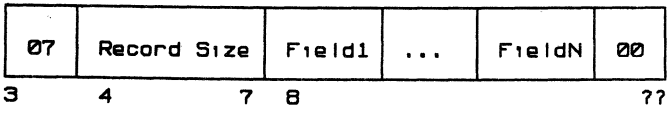
STRING:



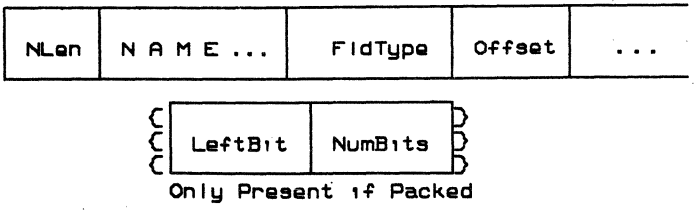
FILE:



RECORD:

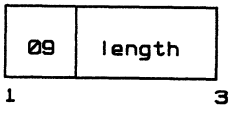


and the form of a Field descriptor is:

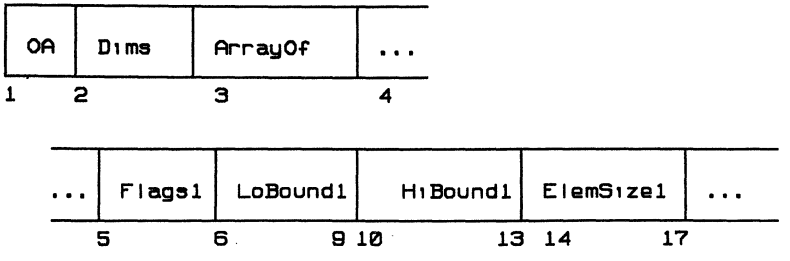


Note: If the sign bit of Offset is off, then offset is two bytes in length, otherwise it is 4 bytes long, and the sign bit is ignored. FieldName is in the form of a VName.

CHARACTER:



FORTRAN ARRAY:



FlagsK 00000000 = Constant Lo, Hi and ElSz
000000*1 = Lo computed, at LoBoundK(A6)
0000001* = Hi computed, at HiBoundK(A6)

If either Lo or Hi is computed, then ElemSize is also computed, at ElemSizeK(A6)

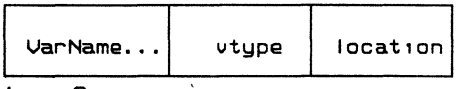
Predefined type numbers are:

-1: integer, 1 byte
-2: integer, 2 bytes
-3: integer, 4 bytes
-4: integer, 1 byte, unsigned
-5: integer, 2 bytes, unsigned
-6: integer, 4 bytes, unsigned
-7: character, 1 byte
-8: character, 2 bytes

-9: single precision floating point (4 bytes)
-10: double precision floating point (8 bytes)
-11: logical, 1 byte
-12: logical, 2 bytes
-13: logical, 4 bytes
-14: file;
-15: complex

VARIABLE DESCRIPTORS

The form of a variable descriptor is:

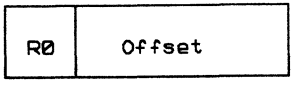


VarName A VName giving the variable's name.

vtype The type number of this variable, as above.

location A description of where the variable is stored, as described below:

REGISTER RELATIVE:

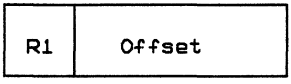


R 0 The upper 4 bits specify a register. The signed 2 byte Offset is added to the contents of that register to determine the variables address. The registers specified are:

68000: 0..7 is D0..D7, 8..15 is A0..A7

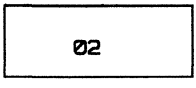
32000: 0..7 is R0..R7, 13 is FP, 14 is SB

REGISTER RELATIVE INDIRECT:



The fields are the same as R 0 above, but the location so specified contains a 4 byte pointer to the actual location of the variable.

REGISTER:

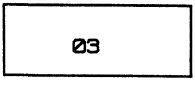


0 2 The upper 4 bits specify a register that contains the value of the variable. The registers specified are:

68000: 0..7 is D0..D7, 8..15 is A0..A7

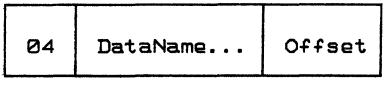
32000: 0..7 is R0..R7, 8..15 is F0..F7

REGISTER INDIRECT:



The register specified is as in 2 above, but the value of the register is the address of the variable.

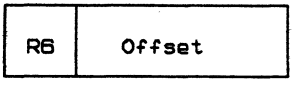
EXTERNAL DATA AREA:



DataName A link name entry (LinkID, Len, Name) of an external data area.

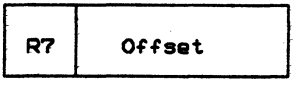
Offset A 4 byte offset in the above data area.

REGISTER RELATIVE LONG:



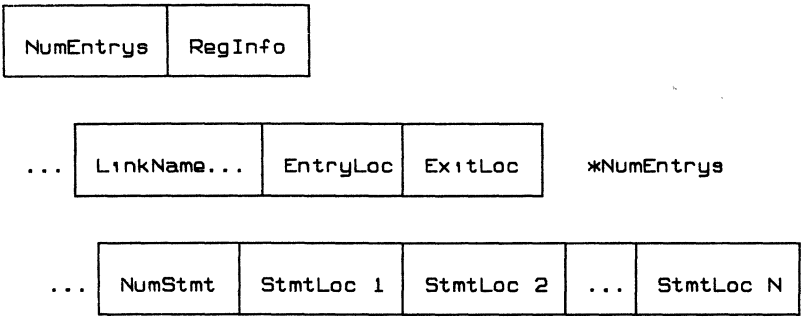
The fields are the same as R 0 above, but the offset is 4 bytes in length.

REGISTER RELATIVE INDIRECT LONG:



The fields are the save as R 1 above, but the offset if 4 bytes in length.

4.4 STATEMENT OFFSETS DESCRIPTIONS

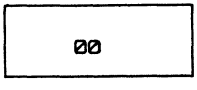


Reginfo — Describes which registers are saved by this routine, and where, if known. See below for a detailed description.

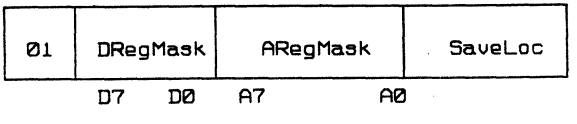
Linkname — A link name entry (LinkID, Len, Name).

Stmtloc N — A list of first instruction offsets for each statement. They are stored as deltas from previous statement. The first one is a delta from the first entry point for this subroutine.

The form of the register save information is one of the following:



00 Indicates no register save information is available.

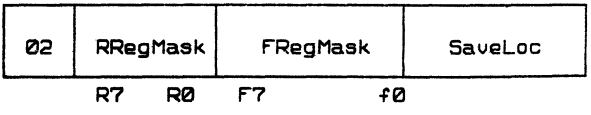


01 Indicates M68000 register save information is present.

DRegMask A set bit corresponds to a saved D register.

ARegMask A set bit corresponds to a saved A register.

SaveLoc The A6 relative location of the saved values.



02 Indicates N32000 register save information is present.

RRegMask A set bit corresponds to a saved R register.

FRegMask A set bit corresponds to a saved F register.

SaveLoc The FP relative location of the saved values.