HP 64700 Operating Environment

Symbolic Retrieval Utilities

User's Guide

HEWLETT PACKARD

Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1   B1471-97004, July 1991
Edition 2   B1471-97007, March 1992

What are Symbolic Retrieval Utilities (SRU)

SRU provides a mapping between symbols and addresses for some HP 64000 products, such as emulation. The symbols may be user-symbols (such as labels in assembly language, procedure names in C, and packages in ADA) or compiler generated symbols (such as loop-labels from the HP Advanced Cross Language System (HP-AxLS) C compiler).

SRU consists of three supported utilities, an unsupported utility, and a library that is loaded into products such as emulation. The three supported utilities provided by SRU are srubuild, sruclean, and sruprint. The unsupported utility included in the SRU software provided to the user is sruaccess.

Srubuild builds a symbol database and is a replacement for the edbuild utility for some emulators.

Sruprint prints the symbols in a symbol database and is a replacement for the edbprint utility.

Sruclean removes or cleans up the datebase that is created for each absolute file.

Sruaccess allows interactive examination of virtually all data within the database through the use of the SRU library.

The architecture of SRU requires several other executable and text files, but these files are not directly visible to the user.

In This Book

Chapter 1   **Using Srubuild.** This chapter how to use the srubuild utility to build a symbol database.

Chapter 2   **Using Sruclean.** This chapter how to use the sruclean utility to remove or clean up a symbol database.

Chapter 3   **Using Sruprint.** This chapter how to use the sruprint utility to convert the contents of a SRU symbolic datebase file to a readable format and print the contents to stdout.

Chapter 4   **Using Sruaccess.** This chapter how to use the sruaccess utility to interactively examine data in an SRU database.

Appendix A   **Finding More Information**. This appendix lists the online man pages for commands and shows you how to access them.

Contents

Contents

1

Using Srubuild

What Is In This Chapter?

The information in this chapter includes:

- What Srubuild Is.

- HP-AxLS and SRU.

- Symbols in SRU.

- File Names .

- Printing of Symbols in Trace Lists.

- How SRU File Name Mapping Works.

- Messages You May Get While Building an SRU Data Base.

- How To Use The Srubuild Command.

- Examples Using Srubuild.

What Is Srubuild?

Srubuild builds a symbol database from an object module. The object module is typically built by a compiler, assembler and linker. The symbol database consists of several files that are stored in a single subdirectory. The subdirectory name is based on the name of the absolute file. For example, the subdirectory name for an absolute file named "myprog.X" will be "myprog.Ys".

Database files with an extension ".GY" or ".LY" are considered private to SRU. Users should not remove or modify any of these files except as occurs as a result of action from one of the SRU utilities.

The srubuild utility has been designed to read in only part of the symbol or executable file at a time and write out the database as the symbols are read in. The database consists of a set of files that are stored in a single subdirectory. One file contains all the global symbols; this would be all the symbols in the linker symbol file (.L file) for HP 64000 format files.

There is also a database file for each set of the local symbols; this is all the symbols in a single assembler symbol file (.A file for HP 64000 format files).

Incremental Builds

Srubuild is capable of doing incremental builds for object formats that contain sufficient information to support this feature. If one source file in a multi-file program is modified and the program is then compiled and linked, the symbol information for the files that were not modified is retained from the previous srubuild; only the modified symbols need to be rebuilt. This feature reduces the amount of time required to build a symbol database when changes are made in the source files. The HP/MRI IEEE-695 format and HP 64000 format are the only currently supported formats that contain sufficient information for this feature.

The srubuild utility and SRU library benefit from previous builds by using a process called "incremental build". The SRU global symbol file and each local symbol file contains a creation date. When an absolute file is updated, the dates in the SRU database files are compared to dates in the absolute file. Any symbol files that were not modified when the absolute was updated can continue to be used without modification. The global symbol file

will be updated whenever the absolute file changes, along with any SRU local symbol files that are affected.

The ability of this incremental build methodology to work is dependent on having correct dates in the absolute file. The HP/MRI IEEE 695 file format contains a date for every module in the absolute file. Language tools which place this date correctly in the IEEE file (such as the HP-AxLS language tools) will work correctly with incremental builds. The HP 64000 absolute file formats do not contain creation dates. In this case, the .X, .L and .A file modification dates are used as creation dates. Incremental builds work as long as file modification dates are not changed. Changing the modification date will have the same effect as recompiling the module, and will cause the symbol information to be rebuilt for that file.

Partial Builds   The srubuild utility and SRU library are also capable of "partial builds"; that is, building only part of the database. If you choose to use the srubuild utility you can specify which modules to build or ignore. The edbuild utility had a similar option. An important extension available in SRU is the ability to build additional parts of the database at a later time. For example, you can ask the srubuild utility to create symbol information for only the modules of interest. If, during emulation, you need to access a symbol that is not in the SRU database, SRU will automatically create the missing symbol information.

These abilities cause some side effects in your emulation tools. For example, if you display (symbols on) a trace list that contains addresses from files that have not been built, there will be a pause in the trace display while the symbol information is added to the SRU database. There will be a message on the status line informing you that symbol information is being updated, and the status line will report progress every few seconds. Another side effect is that if you build the database from within emulation (rather than using the srubuild command in your makefile) only the global file will be built; local files will be built on demand. If srubuild is invoked from the command line, both local and global information is built.

HP-AxLS and SRU

Without SRU
Before SRU, it was necessary to compile and link with the -h option when using an HP-AxLS (Advanced Cross Language System) compiler (such as cc68000). When using the -h option, the HP-AxLS compiler would generate files (file.L, file.X, file.A) needed by EDB.

Note that the HP-AxLS compilers let you compile all the files with -h, then link either with -h or without, letting you create two different absolute files (with different OMFs) without having to recompile the source files.

With SRU
SRU will read the HP/MRI IEEE-695 OMF that the HP-AxLS generates without the -h option. Further, better symbol information will be available. Therefore, it is preferable to remove the -h option on your compile lines.

AxDB and Emulators With SRU
Note that the Advanced Cross Debug (AxDB) tools do not require the -h option, so if you used a debugger and an emulator using EDB, you had to do one link with the -h and another link without the -h option. Now that you have an emulator using SRU, you may use the same absolute file with the debugger and with the emulator.

SRU/EDB Compatibility
Since SRU and EDB create different databases, it is permissible to link with the -h option and without the -h option, and then use srubuild and edbuild. Although this is an option, there is probably no reason that you would ever need to use it.

Symbols in SRU

Arrangement of Symbols

Symbols are arranged in a 'tree' structure that mimics the natural scoping of the user's source language as much as possible. All emulation references to symbols - both input from the keyboard and output displays - make use of the tree structure to show the scoping of symbols and to disambiguate symbols that have the same name but different scopes.

Each absolute file has its own symbol tree. The exact entries in the tree depends on the language used, but in general the tree looks like:

```
                              | - child1_1
                   | - child1---| - child1_2
                   | - child2    | - child1_3
      root ----|
                   | - child3
```

Each entry in the symbol tree has a type and a name. The types define such attributes as "procedures", "tasks", etc. They do not indicate language types, e.g. "int" or "char *". See "Symbol Attributes", later in this chapter for more information.

The names are an ASCII string, such as "main" or "sub_program_1".

Sometimes a child symbol can appear to be in two places. For example, consider a global symbol; it is considered to be a child of "root" and a child of the file in which it is defined. In this case consider the symbol to be accessible in both places (for both input and output).

You can uniquely identify any entry in the symbol tree with a combination of type-name pairs, just as UNIX file names are uniquely identified by their paths from /. For example, consider the following tree, where each entry is shown as (< type> , < name>):

```
                              | - (static, "c")
            | - (procedure, "main")-----| - (static, "getopt_return")
            | - (procedure, "bonzoid")   | - (static, "i")
                                          | - (module, "getopt_return")
      root ----|
            | - (filename, "dumper")----- (static, "i")
```
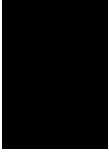
The type "static" refers to a symbol with a fixed address, and is not one of the attributes described above (procedure, module, task, file name, segment, procspecial (ENTRY/EXIT), or source-reference).

In this tree, there are two entries with the same name (static, "i"), but they are unique because their full paths are different:

>   symbol 1: (procedure, "main"), (static, "i")
>   symbol 2: (filename, "dumper"), (static, "i")

The full path of a symbol consists of the names of the entries separated by dots (or colons if the preceding symbol was a filename). The "root" is never displayed. The types are not usually displayed or entered as part of the name unless necessary; for example:

>   symbol 1: main.i
>   symbol 2: dumper:i

Note that this is equivalent to, but a lot faster to type in than:

>   symbol 1: main(procedure).i(static)
>   symbol 2: dumper(filename).i(static)

In the case where two symbols with the same parent have the same name, the type information is necessary, and is added as part of the ASCII string which represents the symbol. For example:

>   symbol 1: main.getopt_return(static)
>   symbol 2: main.getopt_return(module)

Language Dependencies

SRU attempts to reconstruct the user's view of user symbol space. However, it is limited by the information that the Object Module Format (OMF) can represent. For example, a common C language tree might look like:

```
                                          |-- (static, "j")
              |- (filename, "main.c")----|- (procedure, "main")---|
              |                          |- (static, "global")     |-- (static, "index")
       root- |
              |- (filename, "dump.c")----- (static, "i")
              |
              |- (static, "global")
```

However, for an OMF which is task-based (such as the 80386 OMF), the linker enforces the creation of "tasks". For the 80386, the above tree would look like:

```
                                          |-- (static, "j")
              |- (module, "main")----|- (procedure, "main")---|
         root- |                     |- (static, "global")    |-- (static, "index")
              |                      |- (filename, "main.c")
              |- (module, "dump")----|- (static, "i")
              |                      |- (filename, "dump.c")
              |- (static, "global")
```

Note that in the 80386 OMF the global symbols would be "main", "dump" and "global" instead of "main.c", "dump.c" and "global". Also note that if you want to refer to the variable "j" in the procedure main, you would enter a different command. (Refer to "Entering Symbols Using --EXPR--", later in this section for more information.)

main.c:main.j   # non-80386 OMF reference to "j"
main.main.j   # 80386 OMF reference to "j"

If you are unsure of what your "language tree" looks like, you can use the sruprint program to print out portions of your tree.

Procedure Special Symbols

All symbols of the type "procedure" have zero or more children of the type "procspecial". These symbols contain information about the entry and exit points of the procedure. They each contain the address of the exit or entry point of the procedure. In EDB, these symbols existed but were hidden from the user. In SRU, they will show up on a "display symbols" listing, and can be entered from the keyboard like any symbol, or you can continue to use the implied method from EDB.

They have names of the form:

ENTRY
if this is present, it means that the procedure had exactly one entry point. This is the most common case; FORTRAN and ADA can have multiple entry points as of this writing.

EXIT
if this is present, it means that the procedure had exactly one exit point. This is common when there is a large amount of code generated for an exit (for example, popping the stack or resetting an interrupt state), or the procedure has been compiled for debugging. Note that if a C function has multiple return statements it may or may not have multiple exit points; the compiler makes this decision.

ENTRY0,
ENTRY1, ...
These are present when a procedure has multiple entry points.

EXIT0,
EXIT1, ...
These are present when a procedure has multiple exit points.

TEXTRANGE
This symbol has the range of the code associated with the specified procedure. This may be different from the range of ENTRY through EXIT if the compiler generates code (such as a code-space optimized subroutine) which is after the EXIT.

DATARANGE   This symbol has the range of the code plus
                   data associated with this procedure. This is
                   different from TEXTRANGE in the case
                   where the compiler generates data
                   associated with a procedure that is after the
                   last section of code.

Consider the following two procedures:

```
procA(a, b)
{
   if (a == b)
      return a;
   return b;
}

procB(a,b)
{
   int rvalue;
   if (a == b)
      rvalue = a;
   else
      rvalue = b;
   return rvalue;
};
```

There will be the following symbols in the SRU (depending on the
compiler):

```
(procedure, "procA")----|--(procspecial, "ENTRY")
                        |--(procspecial, "EXIT0")
                        |--(procspecial, "EXIT1")

(procedure, "procB")----|--(procspecial, "ENTRY")
                        |--(procspecial, "EXIT")
```

Symbol Attributes   Each symbol in SRU has a "type" and a "level". The "types" are used
                         when necessary to disambiguate between two symbols of the same
                         name but of different type. You may specify the type of a symbol on
                         input, and the output routines will display the type when necessary.
                         You may also select the "level" of symbols that will be displayed.

Symbol Types

static
: Static symbols - includes global variables (global to a specific task and/or program). The static does not mean that the symbol declaration was prefaced with the C static keyword. It means that it has a logical address which will not change.

procedure
: Procedure (or function) symbols. Also used for code blocks in C. For example; when

```
m()
{
    static int alpha;

    if (alpha == 0)
    {
        static int o;
    }
}
```

occurs in a C file, the following tree is built:

```
                      |-(static, "alpha")
(procedure, "m")--    |
                      |-(procedure, "BLOCK_1")--|-(static, "o")
```
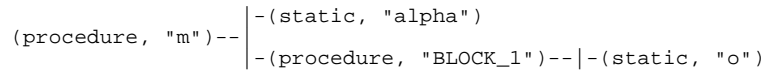
Refer to "Unnamed Block Symbols" later in this section for more information on block symbols.

filename
: Source file name symbols. These are "symbols" which actually define the name of a file (such as the ".c" files for C). Source references occur only under filename symbols.

module
: Module symbols. For 80386 C, these names are derived from the source file name. For Ada, they are packages. Other language systems may permit the user to explicitly name these.

task	Denotes task symbol. Task symbols are specific to a multi-tasking environment and therefore have a range defined by that task. The meaning of the range may vary with processor and language system used.
procspecial	Special symbol names are used to denote language-system symbols not defined by the user (refer to "Renaming of special Symbols" later in this section for more information). Any symbol of this type is always the direct child of a renaming of symbol.
fsegment	This is currently used only in the 80386 environment to hold code or data fsegments that are in the Global Descriptor Table (GDT). The names are made up and only information about their address range is available.

Symbol Levels

highlevel	This indicates that the symbol was emitted by a "high level" in the tool chain (that is; symbols that were available from the compiler).
lowlevel	This indicates that the symbol was generated by a compiler or is an assembler symbol.

The following truth table defines the symbol level conditions and their meanings.

hilevel	lowlevel	Meaning
0	0	Cannot happen
0	1	This symbol is a low level symbol
1	0	This symbol is a high level symbol
1	1	Insufficient information to determine what this is (that is, it might be high or low level)

Refer to "Selecting High/Lowlevel Symbols" later in this section for more information.

Segment Symbols

Depending on the OMF, "segment" symbols will be generated. The purpose of the "segment" symbols is to provide a convenient method of relating an assembler listing and a linker listing with what is displayed on the screen.

An assembler listing shows addresses as offsets from the start of a 'segment'. The linker listing shows the start of each segment. Without the SRU 'segments', you would have to subtract the linker's "start of segment" to obtain an offset that you could use with your assembler listing.

Language Sections Versus Segments

A language "section" consists of all identically-named segments which are contiguous in memory; a "segment" starts with each source file.

To show this difference, consider the examples on the following pages.

The set of tables for the linker's view and the physical view, that follow, all refer to the following four symbols:

fileA and fileB are files;
symA is a symbol which is defined within fileA (for example, it
    might be a variable outside of any renaming of scope);
symB is a symbol defined in fileB.

Further, the tables assume the following HP-AxLS linker commands:

```
SECT  PROG  $1000        # Start of full PROG section
SECT  DATA  $3000        # Start of full DATA section
LOAD  fileA,fileB        # fileB's PROG section is placed
                         # immediately after fileA's PROG
                         # section, and fileB's DATA section will
                         # immediately follow  fileA's DATA
                         # section
```

Symbol Tree. The symbol tree associated with this example, including segment information, is:

```
                              |-(static, "symA")-|-(segment, "PROG")
          |-(filename, "fileA")--|-(segment, "PROG")
          |                   |-(segment, "DATA")
root ---|
          |-(filename, "fileB"(--|-(static, "symB")-|-(segment[0], "PROG")
                              |-(segment, "PROG")
                              |-(segment, "DATA")
```

Note ☞ The "segments" are shown as being in the tree only for the sake of making this clearer; they are not in the tree, in that you cannot enter a symbol named

      fileA:PROG

and get anything back (other than "symbol not found"!), and they do not show up in the "display symbols" command.

Linker's View. This shows how the two segments (sections) defined above (PROG and DATA) are loaded into memory:

Full sections	PROG	DATA
Start address	1000	3000
	(fileA's PROG @ 1000	(fileA's DATA @ 3000
	fileB's PROG @ 1101)	(fileA's DATA @ 3053)
End address	1200	3098

Physical View. This shows how the symbols and files are loaded into memory. The "offset from start of segment" is useful to know when you are looking at a linker listing and at an assembler listing. Note that while the files have DATA segments, the symbols do not. (This is typically defined by each compiler/assembler/linker tool chain.)

symbols	fileA	symA	fileB	symB
	PROG			
absolute start address **offset from start of section**	1000 0	1030 30	1101 101	1154 154
absolute end address **offset from start of section**	1100 100	1036 36	1200 200	1167 167
	DATA			
start absolute address **offset from start of section**	3000 0	---- ----	3053 53	---- ----
absolute end address **offset from start of section**	3052 52	---- ----	3098 98	---- ----

Unnamed Block Symbols

Consider the following code:

```
main()
{
    static int outside_static;

    if (j == 0)
    {
        static int inside_static;
    }
}
```

You can easily get access to the variable "outside_static" by using the name:

main.outside_static

but you get access to the variable "inside_static" as follows:

SRU will create a name for each "unnamed code block", so that any variables declared in it are accessible. The format for the name depends on the OMF. Some OMFs will tell the line number where the code block starts. In that case, the name created for the unnamed code block will be in the form:

BLOCK_< line_number>

If the OMF tells only that you are entering an unnamed code block, the format will be:

< ordinal number> _BLOCK

Where < ordinal number> starts with "1" and increments for each encountered code block. To discover which code block a given < ordinal number> _BLOCK is related to, use "display symbols" to look at the addresses of source references, then the address associated with the < ordinal number> _BLOCK.

Some language tools will create their own name for "unnamed code blocks". In that case, SRU will pass on the created name. Do a "display symbols < procedure name> " to see what they look like when you encounter this case.
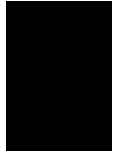
Renaming of Symbols

Some languages (notably Ada) allow multiple symbols with the same names.

For example, Ada allows a procedure named "multiply" with three parameters and a different procedure named "multiply" with two parameters; the correct procedure is called based on the number of parameters in the invocation.

However, SRU does not allow two symbols of the same "type" to have the same name (where "type" is "procedure", "module", "procspecial", etc.). So, SRU will rename the symbols by adding a unique suffix to each instance. To determine what symbols were renamed, use the "display symbols < symbol> " command and look at the address ranges.

Maximum Symbol Length

SRU does not place any limit on the number of characters that a symbol (a language symbol or a filename) may have. However, HP/MRI IEEE-695 has a maximum length of 127 characters, and OMF-386 has a limit of 40 characters. Further, each tool chain (typically the assembler or compiler and/or linker) places a constraint on the maximum number of characters a symbol may have.

For HP-AxLS, the assembler has a significance limit of 31 characters; the compiler 30 characters for external symbols, and 255 characters for non-external symbols. All SRU utilities are compatible with the UNIX "long filenames" system.

Entering Symbols Using --EXPR--

Entering symbols from the command line is done using the "--EXPR--" prompt softkey. The command syntax for entering a symbol is:

```
[.|:][<file:>|<ident.>[(<type>)]]*{<ident>[(<type>)]|<file:>}
```

optionally, this is followed by:

```
procedure {entry_exit_range | text_range}
```

Refer to "Renaming of special Symbols," later in this section for more information.

For comparison, EDB's command syntax was:

```
[:][<file:>][<ident.>]*<ident>
```

Note that this is expressed in a common UNIX notation:

```
<string>               - a non-literal string
[<anything>]           - anything between brackets can be omitted
{<string1>|<string2>}  - string1 or string2 is valid as input
[<string1>|<string2>]  - either string1, string2, or nothing is valid
[<anything>]*          - anything may be repeated indefinitely or omitted
```

Some examples:

```
globalvar        # a global variable - but see HP64KSYMBPATH later in
                 # this aection

:globalvar       # The leading colon forces "globalvar" to refer to a
                 # global variable instead of a local variable. A
                 # leading dot means the same thing.

main.c:index     # anything preceding a colon (up to a previous
                 # colon) is part of a filename

file1.c:file2.c: # file2.c has actual code but is an include file
                 # reference in file1.c

file1.c:"file2.c" # a variable named "file2.c" in file file1.c  Use
                 # quotes to surround any "special characters" (dots
                 # and colons) to prevent them from being used as
                 # separators.

"a:b.c":alpha    # the variable named alpha in the file named a:b.c -
                 #  again note the use of quotes to prevent inter-
                 # pretation of the colon after the 'a'

.procedure.block1.block2.static_variable  # a global procedure....

 :main(procedure)  # a global procedure - but there is another
                   # global symbol named 'main' which is of a
                   # different type

 :package1."file.c":  # the file "file.c" which is a child of the global
                      # symbol "package1". If a file is a child of a
                      # non-file symbol, the entire filename
                      # must be surrounded in quotes.
```

The Values Referred To

The following rules determine what address is referred to when
entered using either the --EXPR-- or --SYMB-- softkeys:

1. If a procedure is entered, the default range is the ENTRY/EXIT range. If the ENTRY/EXIT range does not exist, the TEXTRANGE is used (and you will receive a warning on the status line). If the TEXTRANGE does not exist, the range of the symbol itself is used (and you will receive a warning on the status line).

2. If a filename is entered, the default range is the segment named "PROG". If there is no such segment, the segment named "prog" will be used. If that does not exist, the first segment with attribute "code" will be returned (if the OMF supports it). Otherwise, no symbol is returned.

3. If a line number is specified, the default range is the first through last address associated with that line number.

4. If math is performed on a symbol (including the three outlined above), the starting address associated with that symbol is used unless the "end" softkey is present.

5. If the symbol is in a context that cannot use a range (such as "display memory at < symbol> "), the start address associated with the symbol is used.

Examples:

```
trace_on_range file.c:          # trace addresses between the start and end addresses
                                # of the "PROG" segment of file.c (or "prog" segment,
                                # or first segment with type "code" found)
trace_on_range proc(procedure)  # trace addresses between the start and end addresses
                                # of the ENTRY/EXIT range of proc (or, if the OMF does
                                # not support ENTRY/EXIT range, the TEXTRANGE)
trace_on_range symbolA          # trace addresses between the start and end addresses
                                # of the symbol "symbolA"
trigger on symbolA              # trigger on the start address of the symbol "symbolA"
trigger on symbolA end          # trigger on the end address of the symbol "symbolA"
trigger on symbolA + 4          # trigger on 4 plus the start address of the symbol
                                # "symbolA"
trace_on_range proc text_range  # trace addresses between the start and end addresses
                                # associated with the TEXTRANGE of procedure 'proc'
```

HP64KSYMBPATH and cws

SRU has symbol-searching capability. Further, it has the ability to explicitly set a "current working symbol" (cws), which allows you to refer to symbols relative to the cws (just like a UNIX 'cd' command allows you access to files below your current working directory).

When the shell variable HP64KSYMBPATH is set (and exported) to be a blank-separated list of symbols, a "search list" is set. When a symbol is entered without the leading colon or dot (which forces it to be global), the following happens:

1. The current working symbol (if there is one) is prefixed to the entered symbol; if the resulting symbol exists, that will be the symbol used.

2. For each entry in HP64KSYMBPATH:

    a. prefix the entry with the entered symbol. If the symbol exists, that is the symbol to use.

    b. Otherwise, remove the last entry in the HP64KSYMBPATH's symbol and repeat the previous step (refer to the following examples - they make the meaning of this statement clearer.)

Example:

```
set HP64KSYMBPATH=".file1:proc1 .file2:proc2:code_block_1"
cws :omega.c:  # cws is the file "omega.c"
hello.stat     # the entered symbol


look for symbol :omega.c:hello.stat        # prefix with cws (assume symbol not found)
look for symbol :file1:proc1.hello.stat    # prefix with first entry in HP64KSYMBPATH
look for symbol :file1:hello.stat          # remove one element & prefix
look for symbol :file2:proc2.code_block_1.hello.stat  # prefix with second entry in
                                           #  HP64KSYMBPATH
look for symbol :file2:proc2.hello.stat    # remove one element & prefix
look for symbol :file2:hello.stat          # remove one element & prefix
```

The current working symbol can always be changed with the "cws" command. Some emulation products may supply a softkey-method of changing it, but "cws < symbol> " is always available.

Further, the command "pws" (print working symbol) is also available. It displays the current working symbol on the status line.
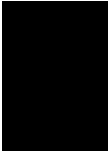
Procedure Special Symbols

When you have entered a procedure, SRU provides a special syntax to access the ENTRY and EXIT symbols that are associated with a procedure. The syntax is:

> < symbol> procedure {entry_exit_range | text_range}

Using the entry_exit_range will cause SRU to return all ENTRY and EXIT symbols related to the procedure < symbol> . For speed of prompting, the special syntax is available even when you have not entered a procedure; but, after you hit < RETURN> , if the < symbol> is not a procedure an error is generated.

If you use "text_range", the addresses associated with TEXTRANGE will be returned by the parser.

Segment Symbols

When you have entered a filename, you may use the syntax:

> < filename> segment < segment>

Where < segment> is a string (such as "axls_named_segment") with or without quotes. In EDB, there were only 3 segments: "PROG ", "DATA ", and "COMM ". For compatibility, if SRU encounters a segment named "PROG " but no segment named that exists, it will search for a segment named "PROG" or "prog".

If you enter a < filename> with no "segment < segment> ", the segment named "prog" will be used. For OMFs which support 'typing' of segments ("code", "data", or "unknown"), if no segment named "prog" exists, the first segment of type "code" will be used.

Printing of Symbols (in Trace Lists)

In general, when a symbol is displayed in the trace list it looks much like it would look if the user typed the symbol in.

SRU will display as much of the symbol as possible. If necessary, the following truncation rules (from last to be truncated to first to be truncated) apply:

1. Sign of the offset (+ /-) truncated on the right.

2. Last part of symbol name truncated on the right.

3. If the next to last part of the name exists and is not a filename (most closely enclosing scope), it is next, truncated on the right.

4. Basenames of parts of symbol name that are filenames truncated on the right.

5. Last part of segment name truncated on the right.

6. Remaining name parts that aren't file names truncated on the right as individuals but taking the most closely enclosing ones first (truncating left to right).

7. Remaining names from any parts that are filenames.

The basename is the last name on the right for UNIX filenames. UNIX filename components are separated by slashes. Remaining names are taken in right to left priority for UNIX filenames.

The segment name will be separated from the symbol name by a '| '. File names will be separated from the next name part by a ':'. The name parts that aren't filenames will be separated from each other by a '.'. A ':' or '| ' that occurs as the right most character will be output, but a '.' in this position will be suppressed. If NULL names are encountered, they are replaced by "??" and an appropriate separator is added.

Selecting High/Lowlevel Symbols

Each product will have a command to select either high, low, or both high and low level symbols for display. When this command is given, only the specified level of symbol will be displayed in trace lists, "display symbols" commands, etc. Note, however, that any level of symbol may be entered at any time.
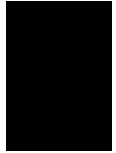
File Names

UNIX File Names
SRU (for OMF's that use file names) maintains a list of all files used in making an absolute. When you enter a filename without its path (e.g. "util.c" instead of "/users/mike/util.c"), a search is made of the filename list. If there is only one file named "util.c", it does not matter what directory you are in - you refer to that file. If there is more than one "util.c" used to make the absolute file, the one in your current directory will be chosen (if that was one of them), or neither will be chosen (yielding a "symbol not found") for the case where neither of the two files are in your directory.

The exact search algorithm used is described in detail at the end of this section.

A relative filename works the same way. Examples:

> Files in SRU's file list:
>   /users/mike/src1/util.c
>   /users/mike/src1/globs.c
>   /users/mike/src2/util.c

Current directory: /users/mike

```
Input filename      filename referenced
globs.c             /users/mike/src1/globs.c   # no conflict
util.c              - symbol not found -       # conflict: which 'util.c'?
src1/util.c         /users/mike/src1/util.c    # no conflict; the "src1" was sufficient
                                               # info to disambiguate between the two
```

Current directory: /users/mike/src2

```
Input filename      filename referenced
util.c              /users/mike/src2/util.c         # current directory used to disambiguate
src1/util.c         /users/mike/src1/util.c         # again "src1" sufficient
```

As a further convenience (especially for those uploading files from a system which only supports upper-case filenames), if the search for a filename fails, SRU will search again; this time case-insensitively.

Non-UNIX File Names

As long as the OMF permits, the filename that you wrote the file with will be used in SRU. For example, if you wrote and compiled the files on a VAX/VMS system, then transferred the files to UNIX, you could refer to your VMS filenames if you used the HP/MRI IEEE-695 file format.

Any filename may be entered as long as:

1. Any characters that are special to the parser (e.g. ":", "(", "+ ") must be escaped by quoting them.

2. the filename is followed by a non-escaped colon.

For example:

```
hello.c:dolly.c:              # two files: the file "dolly.c" as included by "hello.c"
"hello.c:dolly.c":            # one file named "hello.c:dolly.c"; the colon is escaped
                              # by the quotes
"vaxa::user$disk:[del.bozo]util.c":    # the colons are escaped
"dir1\dir2\file.c":                    # the backslashes are escaped
```

Search Algorithm for Locating File Name Symbols
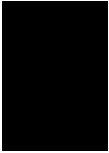
As previously mentioned, SRU maintains a list of all files used in making the absolute file. When you enter a file name, the list of files is searched, looking for a match. If a match cannot be found, SRU will cause an error message to be printed. The following steps describe the sequence of events for finding a match.

Step 1. Search for an exact match. Match found?

Yes	Search ends. (This will always work if the file name is typed exactly as it appears within the OMF file).
No	Go to step 2.

Step 2. Search again using a truncated search looking for any file name ending with the same file name entered on the command line. Exactly one match found?

Yes	Search ends.
No	Go to step 3.

Step 3. More than one file name found in step 2?

Yes   Go to step 4.

No   Go to step 5.

Step 4. Prefix the current working directory to the file name entered on the command line and search for an exact match. (This step provides backwards compatibility with EDB for disambiguating multiple file name symbols by using the current working directory.) Match found?

Yes   Search ends.

No   Go to step 8.

Step 5. Search for a match using a case-insensitive search. (This step provides more flexibility for entering case-insensitive VMS and MS-DOS file names.) Exactly one match found?

Yes   Search ends.

No   Go to step 6.

Step 6. More than one match found in step 5?

Yes   Go to step 8.

No   Go to step 7.

Step 7. Search for a match using a case-insensitive, truncated search looking for any file name ending with the same file name entered on the command line. Exactly one match found?

Yes   Search ends.

No   Go to step 8.

Step 8. Two or more files matching?

Yes   Message: Ambiguous file names.

No   Message: Symbol not found.

Note   👉   VMS version numbers and version delimiter (;) are only used in comparisons if the file name entered on the command line is specified with a version number and version delimiter.

How SRU File Name Mapping Works

SRU needs to access source and symbol (HP OMF .A) files in order to display source references and to build the local symbols database. The path names of these files are extracted from the absolute (OMF) file when the global symbols database file is created. Normally these file names reflect the correct paths to these files unless the files have been moved or the absolute file was created on a different host. For example, if a program is compiled and linked on a remote VAX/VMS system, the absolute (OMF) file will typically contain embedded VMS path names. In the past, when files are moved, the user was required to change the file names within the absolute (OMF) file to reflect the new paths. However, SRU has implemented a feature which allows the user to specify the location of user files and still retain the original path names within the absolute file.

Advantages

- Preserves the original file names within the OMF, allowing the user to see and reference files by their original name.

- Allows UNIX, VMS and/or MS-DOS file names with absolute or relative paths to be specified within the OMF.

- Facilitates software development on similar or diversified remote hosts.

- Supports file transfers from a remote host using standard file transfer file utilities instead of requiring a specialized tool which changes the file names within the absolute OMF) file.

- Supports the use of Remote File Access (RFA) or Network File System (NFS) to access source and symbol files across a network.

The HP64_DEBUG_PATH Variable.

SRU checks for the environment variable HP64_DEBUG_PATH to determine the location of source and symbol files at run-time. This variable is similar to the "PATH" variable in the shell but is used to find source and symbol files. If this variable is defined, searches will be performed on the path(s) it specifies. Multiple directories are are seperated by a colon (:). For example, in shell notation:

> HP64_DEBUG_PATH= /sources:/users/me/src:%
> export HP_DEBUG_PATH

will tell SRU to first look for files in the directory /sources, then in the directory /users/me/src, and finally by the path specified in the absolute (OMF) file (designated by the % character).

When using the HP64_DEBUG_PATH variable to search in a directory, SRU will strip off any directory path information from the original file name before prefixing the search path. File naming conventions are determined by the existence of their respective directory delimiters: slash (/) for UNIX, square or angle bracket pairs ([] or < >) for VMS and backslash (\) for MS-DOS. If the original file name is determined to be a VMS or MS-DOS file name, SRU will also strip off any VMS version number and semicolon delimiter (;), then look for the file using a case-insensitive search.

For example, if SRU needs to access a source file referred to in the absolute (OMF) file as USER$DISK:[BARB]MAIN.C;5, the search will be performed using the file name MAIN.C. Assuming this file was moved to /sources/main.c, SRU should find it correctly.

If the HP64_DEBUG_PATH variable is not defined, SRU will search for files in the following order:

1. Using a file name translation table, (described later in this section).

2. Using the path specified in the absolute file.

3. In the current working directory.

File Name Translation Table

If the HP64_DEBUG_PATH contains a double percent directive (%%) or the default search path is used, SRU will use a file name translation table (mapfile) to locate files. A mapfile is an ASCII file providing a one-to-one mapping of file names appearing within the absolute (OMF) file to corresponding UNIX files accessible on the local file system.

The mapfile should have the same name as the absolute (OMF) file with the extension ".MP". For example, if the name of your absolute file is /myproject/builder.X, the mapfile should be called /myproject/builder.MP. In addition to this mapfile (which is considered to be specific to a particular absolute) SRU will also look for a global mapfile in your login directory. The name of this file is $HOME/.mapfile.MP. If both files exist, both will be used.

The format of the mapfile is simple:

- Each line in the mapfile should contain a from-pattern (the original file name appearing within the absolute (OMF) file), a to-pattern (the alternate file name accessible on the local file system) and an optional comment, all separated by white space (blank(s) or tab(s)).

- White space before the first field will be ignored.

- Comments are delimited by a '#' and a new line.

- Either pattern may be optionally delimited by double quotes.

- A limited form of wildcarding is also supported:
    - A single asterisk may appear at the beginning of from-pattern (a suffix pattern), at the end of

from-pattern (a prefix pattern), or as the entire from-pattern.

– The asterisk matches any string , including the null string.

– The asterisk may appear anywhere in to-pattern.

■ If an input file name matches the from-pattern, the file name will be converted to the form defined by the to-pattern.

For example, consider the the following mapfile which contains two exact mapping patterns and three wildcard mapping patterns:

```
HP$DISK:[MYPROJECT]MAIN.C;2      /sources/main.c     #exact match
HP$DISK:[MYPROJECT]FILE1.C;5     /sources/file1.c    #exact match
/bozo/*                          /users/me/*         #prefix pattern
*.C                              *.c                 #suffix pattern
*                                /net/remotesys*     #add prefix
```

Given the file name HP$DISK:[MYPROJECT]FILE.C;5, SRU would map the file name to /sources/file1.c (by applying the exact match rule) before checking to see if the file exists. Similarly, a file named /bozo/glomp.c would be mapped to /users/me/glomp.c by applying the prefix-pattern rule. A file named /users/barb/main.c would be mapped to /net/remotesys/users/barb/init.c by applying the add prefix rule.

Algorithm for Mapping File Names

The following steps are performed to map the input file name using the file name translation table. When testing for matches using a wildcard, if there are multiple prefix- or suffix-patterns that match the input file name, the one matching the more explicit characters is used.

Step 1. Test for an exact match.
Does the input file name match a from-pattern without a wildcard?

Yes   Search ends. The associated to-pattern is the mapped file name to look for.

No   Go to step 2.

Step 2. Test for prefix-pattern match.
Does the input file name match a from-pattern ending in a
wildcard?

Yes   Transform the input file name as specified by
                            the associated to-pattern. Characters from
                            the input file name that match the wildcard
                            replace the wildcard in the to-pattern. Go to
                            step 3.

No   Go to step 3.

Step 3. Test for suffix-pattern match.
Does the input file name (or the transformed file name from step
2) match a from-pattern starting in a wildcard?

Yes   Transform the input file name as specified by
                            the associated to-pattern. Characters from
                            the input file name that match the wildcard
                            replace the wildcard in the to-pattern. Go to
                            step 4.

No   Go to step 4.

Step 4. Was the input file name transformed in step 2 and/or step 3?

Yes   Search ends. The transformed file name is
                            the mapped file name to look for.

No   Search ends. The input file name is the
                            mapped file name to look for.

Examples of Use

The following scenarios describe the various development environments and what the user must do to run emulation.

Scenario 1

Software development and debug are performed on the same host; source and symbol files are not moved.

1. The user compiles/links program on local HP 64000-UX host.

2. The user starts an emulation session. The user will see and reference the original file names. Files can be referenced by basename, relative path (must be in the correct directory) or absolute path. No mapping of file names is required to access source and/or symbol files.

Scenario 2

Software development and debug are performed on the same host; source and symbol files are moved to another directory.

1. The user compiles/links program on local HP 64000-UX host.

2. The user moves all source and symbol file(s) to a different directory (ie: /test).

3. The user creates a file name translation table so that source and symbol files will be accessed in test. For example, if the original files were located in the directory /users/me, the file name translation table could be created with the following commands:

   srubuild -ln /test/glomp.X | mapfn **-p**'/users/me/* /test/*' /test/glomp.MP

   The srubuild command will print out a list of file names appearing within the OMF file and the **mapfn** command will translate this list into a list of file name pairs, with the original file name first and the mapped file name second.

   Alternatively, the user defines a directory search path using the environment variable HP64_DEBUG_PATH.

For example, if the executable was originally created in directory /users/me, and all source and symbol files are now located in the subdirectories srcdir1 and srcdir2:

HP64_DEBUG_PATH= /test/srcdir1:/test/srcdir2

4. The user starts an emulation session. The user will see the original file names of /users/me. The user can reference files with relative or absolute paths under /users/me; files cannot be referenced with absolute paths under /test. The file name translation table and/or the HP64_DEBUG_PATH environment variable will force emulation to access the correct source and symbol files under /test.

Scenario 3

Software development and debug are performed on different hosts; HP 64000 format absolute files are created; source and symbol files are transferred to the debug host (original pathnames may or may not be retained).

1. The user compiles/links program on the remote host.

2. The user moves all source and symbol file(s) from the remote host to the local HP 64000-UX host.

3. If the pathnames are the same in both lists, the user simply starts an emulation session. Files can be referenced by basename, relative path (must be in the correct directory) or absolute path. No mapping of file names is required to access source and/or symbol files.

4. If the original pathnames are not retained, the user creates a file name translation table and/or defines a directory search path as described above.

5. The user starts an emulation session. The user will see the original file names. The user can reference files by original relative or absolute paths; files cannot be referenced using absolute paths on the local system. The file name translation table and/or the HP64_DEBUG_PATH environment variable will force emulation to access the correct source and symbol files on the local system.

Scenario 4

Software development and debug are performed on different Sun hosts; source and symbol files are remotely accessed using RFA (original pathnames are not retained).

1. The user compiles/links program on a remote host.

2. The user logs in to HP 64000-UX host and establishes a netunam connection back to the remote development host.

3. The user creates a file name translation table. For example, if the name of the remote system is **remotesys**, a file name translation table could be created in the user's home directory with a single wildcard pattern:

   echo "/* /net/remotesys/*" $HOME/.mapfile.MP

   Alternatively, the user defines a directory search path using the environment variable HP64_DEBUG_PATH. For example, if all source and symbol files are located in the subdirectories srcdir1 and srcdir2:

   HP64_DEBUG_PATH= /net/remotesys/users/me/srcdir1: /net/remotesys/users/me/srcdir2

4. The user starts an emulation session. The user will see the original file names without /net/remotesys. Files can be referenced by original, relative or absolute paths; files cannot be referenced by absolute paths beginning with /net/remotesys/.... . The file name translation table and/or the HP64_DEBUG_PATH environment variable will force emulation to access the correct source and symbol files under /net/remotesys/users/me on the remote system.

Scenario 5

Software development and debug are performed on different hosts;
source and symbol files are remotely accessed using NFS (original
pathnames may or may not be retained).

1. The user compiles/links program on a remote host.

2. The user logs in to 64000-UX host and has access to
   remote files using NFS set up by the system administrator.

3. If the original pathnames are retained, the user simply
   starts an emulation session. Files can be referenced by
   basename, relative path (must be in the correct directory)
   or absolute path. No mapping of file names is required to
   access source and/or symbol files.

4. If the original pathnames are not retained, the user creates
   a file name translation table or defines a directory search
   path as described above.

5. The user starts an emulation session. The user will see the
   original file names. Files can be referenced by original
   relative or absolute paths; files cannot be referenced by
   their alternate pathnames. The file name translation table
   and/or the HP64_DEBUG_PATH environment variable
   will force emulation to access the correct source and
   symbol files on the NFS mounted file system.

Note		NFS products on VAX computers vary. VAX source files will need to be converted to HP ASCII for successful emulation.

Messages
Generated By SRU

Error Messages   "Absolute file < file name> is newer than .GY file."

The absolute file has been modified since the SRU database files
were built.

What might have gotten you there:

> The modification date associated with your absolute file
> became newer than the dates on the SRU database files.

What to do:

> Rebuild the SRU database files.

"ABSTXT not found"

What might have gotten you there:

> Loading an OMF386 file that has no absolute data.

What to do:

> No absolute data can be loaded into memory since there is
> none in the file.

"Address error: < explanation> "

Internal software error.

What might have gotten you there:

> Users should not see this message. This implies a defect in
> the database software. Please try to remember and/or
> reproduce the steps you used to produce the problem.

What to do:

> Copy the exact text of the message and contact your
> nearest HP representative.

"Ambiguous name: < symbol name> "

More than one symbol referred to in the absolute file is named
< symbol name> .

What might have gotten you there:

> Using duplicate symbol names for symbols.

What to do:

> Rename the symbols that have duplicate names.

'Bad name passed to database (< name>)"

A call was made to the database with incorrect parameter data.

What might have gotten you there:

> Users should not see this message. This implies a defect in
> the database software. Please try to remember and/or
> reproduce the steps you used to produce the problem.

What to do:

> Copy the exact text of the message and contact your
> nearest HP representative.

'Bad processor id in < file name> : < processor-id name> "

A processor id read from the absolute file cannot be handled by the
version of SRU database software that you are using.

What might have gotten you there:

> Reference to a absolute file that is not the correct format
> for the processor tool you are trying to use.

What to do:

> Reference (or create) a different absolute file.

'Bad query made to database (< function name>)"

A call was made to the database with insufficient data or it was
made at a time when the database was not set up for the call.

What might have gotten you there:

> Users should not see this message. This implies a defect in
> the database software. Please try to remember and/or
> reproduce the steps you used to produce the problem.
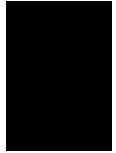
What to do:

> Copy the exact text of the message and contact your
> nearest HP representative.

'Builder cannot read < file name> ''

The software needed to read the absolute file < file name> is not
part of the SRU database software linked into your product.

What might have gotten you there:

> You are using the incorrect HP product to handle your
> absolute file.

What to do:

> Obtain the correct HP product.

'Cannot build database: < .GY file name> ''

The global symbol information associated with an absolute file
cannot be built.

What might have gotten you there:

> At the time that you were trying to build a database, there
> was another process also trying to build the database. This
> other process failed in its build and your process detected
> this.

What to do:

> Query the other process to see why the build failed or retry
> the build in your process to find out why it fails.

'Database < file name> cannot be used''

The .GY file for < file name> had inconsistent data within it as
compared to the information within the absolute file and therefore,
the .GY file cannot be used.

What might have gotten you there:

> Your absolute file might have been recompiled with a
> compiler for a different processor than it was compiled for
> when the .GY database file was generated.
>
> Or the global database (.GY) may have been built for one
> product with a older version of the data base software than
> the version that the product you are trying to use has

linked in. This will not usually be a problem, but it is possible for HP tools to exist that create/use incompatible data bases.

On initial release of the database software, all .GY's will be compatible for the same processor file formats. It is possible that future releases of the database software will result in incompatible databases among products and processor file formats.

What to do:

Regenerate the .GY file using the srubuild command (and the -p option) or regenerate it using the consumer tool (e.g. emulation).

'DEBTXT not found"

What might have gotten you there:

Loading an OMF386 file that has no debug information.

What to do:

You will have no symbols available until you load a file that has symbols (in a DEBTXT section).

'< file name> : absolute file is already open"

< file name> has already been opened by the SRU software (a database may only be opened once per database session).

What might have gotten you there:

Users should not see this message. This implies a defect in the database software. The database software should prevent multiple opens on the same absolute. Please try to remember and/or reproduce the steps you used to produce the problem.

What to do:

Copy the exact text of the message and contact your nearest HP representative.
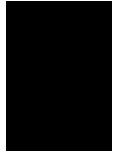
"< file name> : < strerror() explanation of problem> "- An I/O error was detected.

What might have gotten you there:

> You were trying to access a file that is not readable/writable or does not exist or has incorrect permissions.

What to do:

> Check file names, file permissions, etc.

"< file name> : unexpected end of file"

I/O error when reading < file name> .

What might have gotten you there:

> Refer to UNIX man pages read(2) and write(2)

What to do:

> Check the permissions on your files and their contextual validity. Perhaps regenerate files. Refer to UNIX man pages read(2) and write(2).

'Global symbols not found"

What might have gotten you there:

> Making a symbol-to-address request when there is no symbol database, e.g. "display memory MY_SYMBOL".

What to do:

> Load an absolute that contains symbols or don't try to use symbol names on the command line.

'Incorrect abs/sym mode when reopening file"

Internal software error.

What might have gotten you there:

> Please try to remember and/or reproduce the steps you used to produce the problem.

What to do:

> Contact your nearest HP representative.

'Incorrect file format in < file name> ''

The reader for an absolute file detected an error in the absolute file during reading.

What might have gotten you there:

> An absolute file was incorrectly modified and no longer meets the format specifications for its format.

What to do:

> Regenerate the absolute file or refer to a different absolute file.

'Internal error in < function name> ''- Database software problem.

What might have gotten you there:

> Users should never see this error message. This occurs when the software detects an unrecoverable fault condition. We can't predict what might get you there since it should not be possible to get this message. Please try to remember and/or reproduce the steps you used to produce the problem.

What to do:

> Copy the exact text of the message and contact your nearest HP representative.

'MMU Info not found''

What might have gotten you there:

> Loading an OMF386 file with no absolute data or no Global Descriptor Table/Local Descriptor Table (GDT/LDT) information.

What to do:

> All addresses will be interpreted as "Real Mode" addresses. If the file was supposed to be a "Protected Mode" file, it is possible that an error was made during the build phase. HP consumers of OMF386 rely on the presence of GDT/LDT information in the bootloadable file so the absolute file must be built with this information.

'No absolute file: < file name> "

The absolute file < file name> does not exist.

What might have gotten you there:

> Reference to an incorrect absolute file name or to an absolute file that was deleted or to an absolute file with no read permission.

What to do:

> Regenerate your absolute file or change its permissions.

'No absolute file, No database: < file name> "

Both the absolute file and the .GY SRU database file for < file name> do not exist.

What might have gotten you there:

> Use of an incorrect absolute file name.

What to do:

> Refer to a different absolute file name or regenerate your absolute file.

'No database: < file name> "

The SRU database file < file name> does not exist.

What might have gotten you there:

> In an attempt to read information from a previous .GY file, the file was found to not exist.

What to do:

> Regenerate the .GY file using the srubuild command or from within the HP product.

'No symbol file: < file name> "

The .GY SRU database file is corrupt or a HP 64000 format symbol file (.L or .A file) could not be found.

What might have gotten you there:

> The .GY file in your directory is corrupt or a HP 64000 format .L or .A file was deleted.

What to do:

>Regenerate the .GY file using the srubuild command or from within the HP product OR regenerate the appropriate .L or .A file.

'Reader called with wrong request: < function name> "

Internal database software error.

What might have gotten you there:

>Users should not see this message. This implies a defect in the database software. Please try to remember and/or reproduce the steps you used to produce the problem.

What to do:

>Copy the exact text of the message and contact your nearest HP representative.

'Reader error detected in < function name> "

The database builder detected an error in the data passed back by the file format reader software.

What might have gotten you there:

>Users should not see this message. This implies a defect in the database software. Please try to remember and/or reproduce the steps you used to produce the problem.

What to do:

>Copy the exact text of the message and contact your nearest HP representative.

'Segment not found"

What might have gotten you there:

>Asking for the segment of a symbol when that symbol does not contain the segment, e.g. "display memory file.c: segment DATA". If that file does not contain any memory in a segment named "DATA", then this message will be seen.

What to do:

>Don't ask for segments that are not associated with the symbol.

'Source reference not found''

What might have gotten you there:

> Requesting the address of a source line number that does not exist, e.g. "display memory file.c: Line 87".

What to do:

> Display symbols in the file to see which line numbers exist, e.g. "display local_symbols_in file.c:". Many language systems do not generate source references for every source line, only source lines that generate code.

'Subtree < .LY file name> cannot be accessed.''

The .LY with the local symbol information cannot be built (nor accessed) OR its information is out of date with respect to the absolute file.

What might have gotten you there:

> The SRU database files in the .Ys directory may have been deleted or permissions modified so that the database files cannot be correctly accessed. References to symbols that are not present in the absolute file might also generate this error. For example, library modules are often present in the absolute code, but symbols are usually stripped. If you use a library module called "math.c" and enter the command "display local_symbols_in math.c:", you may see this message.

What to do:

> You will be unable to refer to symbols in the module.

'Symbol not found''- Information for a symbol was not found in the database.

What might have gotten you there:

> Symbol was referenced whose information is not contained within the absolute file.

What to do:

> Check the symbol name or the valid symbols in your absolute file.

"Transfer address not found"

What might have gotten you there:

> "run from transfer_address" when there is no transfer address. Some file formats (e.g. OMF386) do not contain a transfer address. Most language systems that support the concept of a transfer address require the user to explicitly specify this address.

What to do:

> Check your language tools to see what you must do to ensure that there is a transfer address in the absolute file.

"Unrecognized file type - < file name> "

The file < file name> cannot be handled by the version of SRU database software linked into your product.

What might have gotten you there:

> The absolute file you are loading in is of a format that cannot be identified by the SRU database software.

What to do:

> Use a different absolute file or a different HP tool.

"Write error: exceeded allotted capacity of file < file name> "

The attempt to write to a file exceeded the allotted capacity of the file either because of the current ulimit() or because the physical end of the medium was detected.

What might have gotten you there:

> The current ulimit() is too low or you need a bigger disk.

What to do:

> Reset ulimit(); install a larger disk.

IEEE-695 Specific Error Messages

'File error: An address is not in the range of a BB11"

A BB11 is a part of an IEEE-695 file that describes the address ranges of a module. The BB11 must describe all the address ranges of all the symbols that appear in a module. It is an error for an IEEE-695 file to contain a symbol whose address is not encompased by an address range in the BB11 associated with the module.

What might have gotten you there:

> This message indicates a defect in the tool that produced the IEEE-695 file. In most cases fixing this defect will be beyond the users control.

What to do:

> Contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'File error: No BB11 found in module < module name> 's BB10 scope"

A module did not have a BB11 part. The BB11 part describes the address ranges of the module. It is an error for this part of the file to be absent.

What might have gotten you there:

> This message indicates a defect in the tool that produced the IEEE-695 file. In most cases fixing this defect will be beyond the users control.

What to do:

> Contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'File error: Type (TY) record has invalid index"

A type record is used to describe a user defined type such as a structure. Each type record contains an index so that data elements can simply refer to the type by its index. Some index numbers are reserved for pre-defined types. It is an error for an IEEE-695 file to contain a type record that uses one of these reserved indexes.

What might have gotten you there:

> This message indicates a defect in the tool that produced the IEEE-695 file. In most cases fixing this defect will be beyond the users control.

What to do:

> Contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'Form C LR/LT record in data part"

An LR or LT record can be used to specify data to be loaded into memory. There are three forms of an LR or LT record. Form A consists of an array of data to be loaded in memory. Form B and Form C are used for relocation. A Form B LR or LT record can safely be ignored, but a Form C record in an absolute file indicates a fatal error. It is pointless to continue to attempt to read the data part after encountering a Form C LR or LT record.

What might have gotten you there:

> This message indicates a defect in the tool that produced the IEEE-695 file. In most cases fixing this defect will be beyond the users control.

What to do:

> Contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'Module < module name> could not be read"

What might have gotten you there:

> The IEEE-695 file was modified between the time it was opened and the time of the access that produced this message.

What to do:

> Rebuild the IEEE-695 file and the SRU database, if the error persists contact your nearest HP representative.

Warning Messages

Note	Warnings that are generated by SRU are written to files to be referenced later. Warnings will usually be one line long. However, if a warning spans two lines, it will be followed by a blank line in the file to enhance readability.

"Adjacent source references in < file> at line < # > "

If an absolute contains source references that are adjacent (consecutive when sorted by line and column) aand have the same address, they are considered to be the same source reference and their source ranges are merged.

'Identical source references in < file> at line < # > "

If an absolute contains more than one source reference for a given line and column in a given module, those source references are considered to be identical. Identical source references are merged, that is; only one source reference will appear in the database but it will have multiple code ranges, one for each occurrence of that source reference in the absolute file.

'No symbols in < file> "

The builder attempted to build a subtree, but there were no symbols for that subtree. An empty subtree was created and there will be no further attempt to build this subtree until the entire database is rebuilt (that is; the next execution of srubuild).

'Symbol < name1> was renamed to < name2> to avoid name conflict"

Generated when symbols of the same type with the same name are detected in an absolute. They will be renamed by the SRU database software and the warnings generated will tell you the new name of the symbol. You should refer to the symbols using the new name assigned by SRU when using the HP product in order for the SRU database software to be able to derive information about the symbol.

IEEE-695 Specific Warning Messages

'HP/IEEE file offset (ASWx) was null. Part access not possible''

The IEEE-695 file consists of several parts, each with a set of related information. The first part of the file contains an index which specifies the locations of all the other parts. The ASWx record is used to specify the location of each part; if the ASWx record is null, the corresponding part of the file is missing or inaccessable.
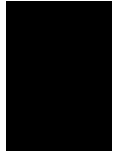
What might have gotten you there:

> The IEEE-695 file does not have an ASW record specifying the location of the debug part. Some language tools require the user to use a "debug" option on the command line in order to enable the generation of all the parts of the IEEE-695 file.

What to do:

> Rebuild your program using the "debug" option on all tools involved in translating and linking your code. If that fails, contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'Section with address < hex address> was renamed < section name> ''

The IEEE-695 file contains records for describing memory address regions, called sections, including the name of the sections. These section names are required for the incremental building feature of SRU. If section names are omitted in the IEEE-695 file, a name will be created by SRU.

What might have gotten you there:

> This message indicates that two sections in the IEEE-695 file were given the same name. This will most likely occur when absolute sections have been given a name consisting of a single blank i.e. " ".

What to do:

> You may use the absolute file and SRU symbol database. The message is intended only to make you aware of the section name that will be used. If you are concerned, contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'No source line information was found for module name"

The IEEE-695 file consists of several parts, each with a set of related information. The debug part contains symbol information for the program. Each module may have a set of language symbols and a set of source references.

What might have gotten you there:

> A module in the IEEE-695 file does not contain source reference information.

What to do:

> Review the documentation for the language tools that produced the IEEE-695 file. Some tools may require a special "debug" option to produce source references. Rebuild your program using the "debug" option on all tools involved in translating and linking your code. If that fails, contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.

'Form B LR/LT record in data part''

An LR or LT record can be used to specify data to be loaded into memory. There are three forms of an LR or LT record. Form A consists of an array of data to be loaded in memory. Form B and Form C are used for relocation. A Form B LR or LT record can safely be ignored, but a Form C record in an absolute file indicates a fatal error. It is pointless to continue to attempt to read the data part after encountering a Form C LR or LT record.

What might have gotten you there:

> This message indicates a defect in the tool that produced the IEEE-695 file. In most cases fixing this defect will be beyond the users control.

What to do:

> If you are concerned, contact your nearest HP representative. If the language tool that produced the IEEE-695 file was not HP, you may also need to contact their representative.
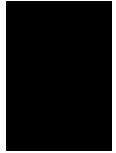
How To Use The Srubuild Command

Using the srubuild command requires the following syntax:

srubuild [options] filename [buildfile...]

Where:

> **filename** is the path name (full or relative to the current directory) of the program (i.e. absolute file) whose symbolic database is to be built. The files created for the database will be placed in the subdirectory **filename.Ys** located in the same directory as **filename**.

Options are:

-a addresstype

srubuild can usually determine the correct address-driver to use just by looking at the absolute file. However, in the event the absolute file does not contain the necessary information, this option can be used to instruct srubuild to use a specific address-driver when building the database. To get a listing of valid address types, enter an illegal value in the command; for example type in the command srubuild -a XXX.

-l

causes a list of the source file names whose source files were used to create **filename** to be copied to stdout (standard output file). This option is used to generate a list of source files that may need to be transferred from a remote system that the absolute was linked on, to the system which needs access to the source files (for displaying source-references). By default no list is produced.

-m messagefile

specifies the name of the file in which you want to append error and warning messages that may be produced during the building of

the database. If neither the **-m** or **-q** options are specified, the default message file **filename.Ys/bldmessages.WY** will be used.

Note that messages generated during subsequent building of local symbols, by either srubuild or HP 64000-UX products that perform "on demand" building, will continue to use the same message file that was in effect when the global symbols were built.

-p product   specifies the product that will be using the database that is built. If this option is not specified, srubuild will build the database in a manner it selects as appropriate for the absolute file.

-q   This option may be used instead of **-m** to completely suppress writing out of messages produced during the building of the database. If the **-m** option is be specified, the **-q** option will be ignored.

Note that srubuild and HP 64000-UX products that perform "on demand" building will continue to suppress build messages during all subsequent local symbol builds until the global symbols are rebuilt again.

-v   prints the results of each of the global and local builds to the stderr (standard error file).

By default, the entire database is built. This means that data on all source files will be created. If you are only interested in a subset of the source files, however, you can reduce the time and disk space required by using the following options:

-g   Only global symbols will be built; this means that no local symbols will be built at this time. However, local symbols will be
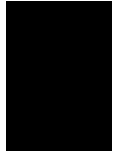
available because the local symbols will be built "on demand" when necessary.

This means that you can determine what subset of source files you are interested in when you are in an HP 64000-UX program instead of deciding ahead of time.

This option is ignored if either the **-f** option is used or one or more buildfile arguments are given.

-f buildfilelist   allows you to specify a subset of the program modules or source files to be processed. This means that the database for the local symbols defined by those program modules or source files is created by this command; when the local symbols are used in an HP 64000-UX program the data does not have to be built.

The **buildfilelist** consists of a number of lines. On each line is the name of a program module or the absolute path of a source file. Blank and comment lines beginning with the '# ' character will be ignored.

-n   requests srubuild to only scan the absolute file without building the database. This option is only useful in conjunction with the **-l** option to get a list of source file names from an absolute file without incurring the overhead of building the database.

Examples Using Srubuild

A convenient place to invoke srubuild is within a makefile. Part of an example makefile includes:

```
CC = cc68000
prog : program.o file1.o file2.o
  $(CC) -o prog program.0 file1.o file2.o
  srubuild prog
```

Sophisticated makefiles can be used to create the file list (for the **-f** flag). The following rule (used to make a relocatable file from a C source file) appends the assembly symbol file name to a file named "interesting", for example, only when the file is updated. This "interesting" file (file list) can then be used with the **-f** flag to specify "interesting" files. When this method is used, it is necessary to truncate this file on occasion since the "interesting" files are likely to change over time. For example:

```
CC = cc68000
prog : program.o file1.o file2.o
  $(CC) -o prog program.o file1.o file2.o
  srubuild -f interesting prog
```

2
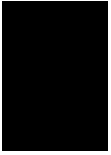
Using Sruclean

What Is In This Chapter?

The information in this chapter includes:

- What Sruclean Is.

- How To Use The Sruclean Command.

- Examples Using Sruclean.

What Is Sruclean?

HP 64000-UX products make use of the SRU database to get access to symbolic information produced by HP 64000-UX (or compatible) language systems (compilers, assemblers, and linkers). The database is placed in a subdirectory, which has the name of the absolute with ".Ys" appended.

The sruclean utility will remove or clean up the database that is created for each absolute file (created either by the srubuild command or by the HP 64000-UX products).

The directory or file arguments are interpreted as follows:

- If the argument ends in ".Ys" and is a directory, only the files in that directory will be modified as specified by the options.

- If the argument is a directory that does not end in ".Ys", the disk will be searched starting with that directory, and all files in subdirectories that end in .Ys will be modified as specified by the options.

- If the argument is not a directory, it is assumed to be an absolute file, and the SRU files associated with that absolute file will be modified as specified by the options.

You must specify at least one directory or file. If you want your entire file system to be searched (including network mounted file systems) use:

sruclean < options> /

How To Use The Sruclean Command

Using the sruclean command requires the following syntax:

sruclean [**-v**] [**-m**] [**-r**] [**-o**] [directory ...] [absolute file ...]

where:

-v   Prints actions (to stderr) as they happen. This will also print out how many bytes were deleted from the disk (if the **-r** option was used), or how many lines removed from the "bldmessages.WY" file (if the **-m** option was used).

-m   Cleans up the "bldmessages.WY" file. During srubuild or incremental builds, information about the build is appended to this file. Unless this option is used, the file will grow without bounds. So, you should consider using this in a crontab.

-r   Removes all SRU files and the ".Ys" directory. You could also use the

rm -fr < sru directory>

command, but -r is somewhat safer in that it guarantees that only directories ending in .Ys will be removed.

-o   This will remove any obsolete SRU files. Usually obsolete files are removed automatically when srubuild is run. Under certain conditions, however, they will have to be removed with this option.

If no options are given, it is equivalent to the command:

    sruclean -m -o

Examples Using Sruclean

Example 1:   **sruclean -r control race**

This command will remove all SRU files associated with the absolute files **control** and **race**.

Example 2:   **sruclean /users**

This command will remove all obsolete SRU files and clean up message files under the directory **/users**.

Example 3:   **sruclean -m *.Ys**

This command will clean up the message file in all SRU directories in the current directory.

Example 4:   **0 3 * * 6 usr/hp64000/bin/sruclean /**

This **crontab** entry will remove all obsolete SRU files and clean up message files every week starting at 3:00 AM on Saturday.

3

Using Sruprint

What Is In This Chapter?

The information in this chapter includes:

- What Sruprint Is.

- How To Use The Sruprint Command.

- Examples Using Sruprint.

What Is Sruprint?

Sruprint is a utility that converts the contents of the SRU symbolic database file to a readable format and causes the contents of the file to be printed to stdout (normally your terminal unless it is redirected to a file).

The sruprint utility will only print symbols that are already in the database; it will not build symbols for modules that have not been built.

How To Use The Sruprint Command

Using the sruprint command requires the following syntax:

sruprint [options] absolutefile [< symbol> ...]

Where:

absolutefile is the absolute file built by sruprint. If < symbol> does not follow absolutefile, the global symbols in the absolute file will be printed.

If one or more **< symbol>** arguments follow the absolutefile, they are assumed to be symbol names. This specifies that you also want symbols that are their children to be printed.

Options are:

-f symbolfile	allows you to specify a list of symbols in a file rather than on the command line.
	The **symbolfile** consists of a number of lines. On each line is the name of a symbol. Blank and comment lines beginning with '# ' character will be ignored.
-g	suppresses printing of the global symbols.

Examples Using Sruprint

Example 1:   **sruprint myexec.X**

This command only prints the global symbols associated with the absolute file **myexec.X**. Global symbols include global procedure names, global variables, and file names. When this command is used, the listing will include the symbol name, logical address, and the symbol's offset from the start of the segment which contains the symbol.

Example 2:   **sruprint -f my_sources myexec.X**

This command prints the global symbols and children of the sources files listed in **my_source**s command file.

my_sources looks like:

/users/me/controller/main.c:
/users/me/controller/sighandle.c:R/users/me/controller/except.c:
/users/projA/library/except.c:
/users/projA/library/margin.c:

Example 3:   **sruprint -g -f my_sources myexec.X**

This command prints only the symbols that are children of the source files listed in **my_sources**. See example 2 for what the **my_sources** command file looks like.

4
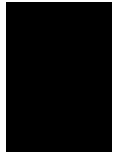
Using Sruaccess

What Is In This Chapter?

The information in this chapter includes:

- What Sruaccess Is.

- How To Use The Sruaccess Command.

What is Sruaccess?

Sruaccess is a utility that allows interactive examination of virtually all data within the database.

Note

Sruaccess is an unsupported utility program. It is not part of any Hewlett-Packard product and is provided at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose. However, for your convenience, its purpose and use are described in this chapter.

How To Use The Sruaccess Command

Using the sruaccess command requires the following syntax:

**sruaccess [-a < addrtype>] [-f < fmt-options>] [-p < product>]
[abs-filename]**

Where command line options are:

-a < addrtype>   Sets the default address type to **addrtype**; see
the command **setaddrtype** for more
information.

-f < fmt-options>   Sets the default print format to
< fmt-options> ; see the **fmt** command for
more information.

-p < product>   Use the builder process that is shipped with
< product> . If this option is not specified, a
builder will be selected from available
builders on the system when an **open**
command is executed. See the command
prodid.

Sruaccess Commands

The following commands are accepted by sruaccess:

absrec [startflag]   If **startflag** is not present or if it is "F", print
the next absolute record. If **startflag** is "T",
get the first absolute record.

childinfo   Give info about the current working
symbol's children.

close   Closes the current database.

cd < name> and **cs < name>**

> Change the current working symbol to the symbol named **< name>** .

create < name>   Create a database for the absolute file **< name>** .

fmt [options]   Defines the format that symbols are printed in. Use the command with no options to get a list of currently-supported formats.

fseg   Print fsegment information for the current symbol. An fsegment is a mechanism used to implement incremental builds. This command is useful to the SRU implementers and probably to no one else.

lc [-r] [-p < pattern>] [< type>] and
ls [-r] [-p < pattern>] [< type>]

> Print the current working symbol's children, using the format last defined with the **fmt** command. The **< type>** of the children to be listed can be specified.
>
> The **-r** option lists the children recursively (normally only the symbol's immediate children are printed).
>
> The **-p < pattern>** command requests that all children whose name matches **< pattern>** (an sh(1) style wildcard string) should be printed.

open < abs-filename> [symbolic]

> Attempts to open the database associated with the given absolute file. If a database is already open, this will close the previous one.

symbolic (which is either T or F) sets the value of the symbolic parameter to sruopen to either TRUE or FALSE. If not present on the command line, it is set to TRUE.

procid Print the processor id associated with the current database.

prodid < name> Set the product id to **< name>** .

pwd Print the name of the present working directory.

pws Print the name of the current working symbol.

range < start> < end> [< incr> [< type>]]

Print all symbols that are within the specified address range. This is essentially:

```
for (addr = start; addr <= end; addr+= incr)
{
    sym addr [<type>]
}
```

reopen Reopen an already open database. This command is useful to the SRU implementers and probably no one else.

sa Print the attributes associated with the current working symbol. These include the symbol type, name, address range, etc.

setaddrtype [< addrtype>]

This command affects how address are both printed and entered. Normally, addresses are up to 32-bits of physical address - that is; the libaddrII ADDRATPHYSICAL type. When this command is given, all subsequent input

is in the ADDRAT.... type you entered. If the specified addrtype has more than one libaddrII element, they will be printed out separated by colons.

When the command is given, a line will be printed telling what each of the colon-separated elements will be.

The command with no **< addrtype>** will print a list of currently implemented addrtypes.

sourceref < address>

Print the source reference for the specified address.

sym < address> [< type>]

Print the symbol of **< type>** that most closely encloses **< address>** .

xfraddr   Get the transfer address.

Data-entry Formats

< address>   See the command **setaddrtype**.

< name>   This is the name of a symbol. "| " means the root symbol. If a name does not begin with "| ", the specification is appended to the current working symbol to form the full symbol name. Other than that, the format of **< name>** is:

< ascii> [-< type>][| < ascii> [-< type>] ...]

Where **< ascii>** is the ascii portion of the SRUNAME, and **< type>** , if present, is the

type portion of the SRUNAME. See below for what **< type>** looks like.

< type>   **< type>** is one of the following strings:

high, low, fsegment, static, sourceref, procedure, filename, module, task, special, alltype. If **< type>** is not specified, it will default to **alltype**.

Interactive Versus Batch Mode

The sruaccess utility can be operated interactively or in batch mode. To use batch mode, simply create a file with the commands of your choice and use the shell to direct the file into sruaccess, for example:

      sruaccess < mycommands

Operation of the utility is slightly different in interactive and batch modes. A command line prompt is printed only in the interactive mode. In the interactive mode, commands may be abbreviated to any unique prefix. In batch mode all commands must be completely specified. The interactive mode will return to the command prompt on input of control-C. Both modes will exit on end of file.

Processor ID Names

Refer to the manual page given in appendix D for the list of processor ID names.

Address Types

Refer to the manual page given in appendix D for the list of supported address types.

A

Finding More Information

The Symbolic Retrieval Utilities software includes on-line "man" page information on the various related commands and files. This information is accessed by using the UNIX **man** command.

Because the "man" page files are installed under the $HP64000 directory, you must first modify the MANPATH environment variable. When using "sh" or "ksh":

```
$ MANPATH=$MANPATH:$HP64000/man:\
> $HP64000/contrib/man; export MANPATH
```

Or, when using "csh":

```
$ setenv MANPATH $MANPATH:$HP64000/man:\
> $HP64000/contrib/man
```

Once the MANPATH environement variable is set, you can access the on-line "man" page information. For example:

```
$ man srubuild <RETURN>
```

On-line "man" pages are included for the following commands:

> **sruaccess**
> **srubuild**
> **sruclean**
> **sruprint**
> **sruxlate**

Index

Certification and Warranty

Certification Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse,

operation outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.