PHILIPS

PHILIPS INTERACTIVE MEDIA

Technical Note #81

Using SrcDbg to Debug CD-I Software

Ken Ellinwood

October 2, 1992

Microware's source level debugger, SrcDbg, is a very useful tool for debugging CD-I software. This document offers many hints and tricks that make the debugger easier to use. This document is not intended as a tutorial on SrcDbg commands or how to use debuggers.

Using SrcDbg to Debug CD-I Software

Introduction and Caveats

The purpose of this document is to pass on knowledge gained through years of experience debugging CD-I titles with the Microware source level debugger, SrcDbg. Many hints and tricks described here make the debugger easier to use. However, this document is not intended to teach the specifics of SrcDbg commands or how to use debuggers. Please refer to the Microware SrcDbg manual for a detailed explanation of the SrcDbg commands.

To use SrcDbg, you must compile the application with the Microware C compiler. Other compilers, such as the GNU compiler and Macintosh compilers used with "Fix Code," do not generate the proper symbol tables required by the debugger.

Decreasing SrcDbg's Startup Time

To generate the symbol tables for debugging, compile and link your code with the -g option. When SrcDbg starts up, it reads and parses the symbol table. The time it takes SrcDbg to parse the symbol table is directly related to the size of the table. For large applications, this can involve a considerable amount of time (as much as a few minutes). The size of the symbol table—and the time required for SrcDbg to start up—can be reduced significantly by compiling a subset of the application's code modules with the -g option. In other words, compile only those files with the -g option that you need to debug. This method creates symbols for only those files, and the startup time for SrcDbg is greatly reduced.

Information about the structure and type of a particular global variable is generated only when the file in which that variable is declared is compiled with the -g option. The debugger still knows the name and address of the variable, even if the file has not been compiled with the -g option. However, it is not able to display the data in its correct form (for example, it will print a structure as an integer). So if you need to reference the value of particular global data element, make sure that the file in which it is declared is compiled with the -g option.

To compile a subset of the source code with the -g option, it is preferable to have a make file that is smart enough to do this for you. The following is a an example from a make file that does the trick:

```
CC = cc68
# RLIST specifies all .r files that link together to generate the executable
RLIST = file1.r file2.r file3.r file4.r file5.r

# DLIST specifies all .r files that are to be generated with the -g option
DLIST = file1.r file3.r file5.r

# CFLAGS specifies default compile options, do not put -g here
CFLAGS = -v=. -v=/usr/local/defs -r=./rels

# RFILES is the list of all .r's with the target path appended to the name.
RFILES = $(RLIST:%=rels/%)

# compile the .c files making a debug version if the .r file is in the DLIST
$(RFILES) : $$(@F:.r=.c)

    @found=false ; \
    for i in $(DLIST) ; do\
        if [ $$i = $(@F) ] ; then \
            found=true ; \
            echo "$(CC) $(CFLAGS) -g $(@F:.r=.c)" ; \
            $(CC) $(CFLAGS) -g $(@F:.r=.c) ; \
        fi; \
    done; \
    if [ $$found = false ]; then \
        echo "$(CC) $(CFLAGS) $(@F:.r=.c)" ; \
        $(CC) $(CFLAGS) $(@F:.r=.c) ; \
    fi
```

With this make file, the DLIST macro specifies the subset of source code that is compiled with the **-g** option. If you add a new file to this list without having modified its contents, it may be necessary to touch the source code file to force the file to compile so that the necessary symbols are generated for debugging purposes.

Translating the EOL Characters in the Source Code

Depending on the system on which the source code is developed, it may also be necessary to translate the end-of-line (EOL) character from the one used on the development platform to the one used by OS-9. For example, the Sun uses octal 12 as its EOL character, and OS-9 uses octal 15. The translation is necessary so that SrcDbg correctly displays the source code during the debugging session. Once again, the make file is the place to do this. The following line may be added to the make file just below the compile command. It uses the Sun **tr** command to translate the EOL characters in the source code to the correct value for OS-9. The translated files are kept separate from the original source files and are placed in the local **dbsrc** directory.

```
                tr '\012' '\015' <$(@F:.r=.c) >dbsrc/$(@F:.r=.c) ; \
```

The next step is to move the translated source, symbol tables, and executable to a local player device. This may be done by way of NFS, PC-Read, or by building the files into the emulated disc image. If you use PC-Read, keep in mind that the file names must remain the same in order for the debugger to work.

Setting the Environment for Debugging

Once the necessary files are accessible by the player, the debugger requires a specific set of environmental variables to be defined. The PORT environment variable must be set to the name of the terminal device. For 180 players this is /t0. For 605 players, this is **/term**. The TERM environment variable must be set to a value recognizable by the termcap library. The SOURCE variable is optional, but it should list the various locations that SrcDbg is to search to find the source code. The PATH variable is also optional, but it should be set to the locations that SrcDbg is to search for the executable and symbol tables. Here is an example:

```
setenv  PORT  /term         ; name  of  terminal
setenv  TERM  vt100         ; type  of  terminal
setenv  SOURCE  /h0/src     ; location  of  translated  source  code
setenv  SOURCE  /h0/bin     ; location  of  exe  and  symbols
```

It is important to remember that SrcDbg searches two locations before it starts down the SOURCE and PATH lists in its attempt to locate the required files. These locations are the current directory and the original location of the file. This "feature" is likely to cause the following problem. If, in its search, SrcDbg is able to locate the untranslated source before it finds the translated source (these files have the same name, of course), when it the source code from the program is displayed, SrcDbg simply displays **[End of file]**. For example, if the original source is compiled from **[serpentine]:/x/code/test.c** and **serpentine:/x** is mounted as **/x** on the player, SrcDbg will find the untranslated source before the translated source, because, from the player, the path **/x/code/test.c** is perfectly valid. This problem probably would not arise if relative path names were used when compiling the code because a similar relative path may not be valid on the player.

Common Problems

Here are a few things that cause SrcDbg to break and hints on how to avoid them.

- SrcDbg does not work if the keyboard signal is armed. The recommended workaround is to avoid arming the keyboard signal when you are debugging.
- Debugging is difficult when signals are being generated. If you have launched your program using the debugger, it runs normally until it reaches a breakpoint. If signals are being generated when the breakpoint is taken, they will be queued and not handled until the program continues. Unfortunately, if you type the **next** or **step** commands to get to the next line of code under these conditions, the debugger executes the signal intercept routine in response to the command and returns to the same line of code without advancing to the next one. This continues until all of the signals have been exhausted from OS-9's internal queue. If enough signals are generated while the program is stopped and waiting for commands, you may begin to lose signals. Or, if you are using Balboa, the dispatcher queue overflows when you finally allow the program to continue. The only practical way to deal with this problem is to use **#ifndef logic** to eliminate the source of the signals from the executable that requires debugging. Typically, only the **alm_cycle()** or **tmr_clock()**

calls are sources that need to be dealt with in this manner. Other sources, such as the video signal, must be re-armed so the breakpoint can usually be taken before the signal is enabled again. When debugging a problem related to a real-time play, **ss_pause()** and **ss_continue()** (or the play manager equivalent of these functions) may be used to bracket the area of code that is to be debugged. When the play is paused, play-related signals that would otherwise wreak havoc with any attempt to debug will not be generated.

- The symbol tables (files with .dbg and .stb suffixes) must have executable attributes for SrcDbg to work correctly. If not, SrcDbg will complain with a 214 error.

elpful Hints

rom the SrcDbg command line, it is possible to execute a function and display its eturn value. Simply type **print**, followed by the function call. For example, **rint foo(5)** will call the function **foo()**—passing it the parameter 5—and display the esult. It is also possible to use this technique to cancel a timer or to pause and continue h lay, for example.

arn how to use the scope features of the debugger. This is helpful when you want to et a breakpoint or examine a variable in another source file. For example, if the urrent source file you are in is **file1.c** and you are stopped at a line in the function **unction1()**, it is possible to set a breakpoint at an arbitrary line in **function3()** inside **ile3.c** by following the procedure listed here:

) Use the **list** command to determine the line number where the breakpoint is desired, for example, **list function3<ret>**.

) Once the line number has been determined, set the breakpoint by issuing a **break** command with a scope argument. For example, **break file3.c\88** sets a breakpoint at the 88th source code line in **file3.c**.

t is possible to determine the name of a function or variable if its address can be letermined. Use the **dump** command to display memory at the given address. SrcDbg lisplays the dump with memory addresses in symbolic form (in relation to the nearest n symbol). In the following example, a function address is given and it can be rmined that this address points to the function **EfxTerminate()**.

```
rcDbg:  dump  0x8cd822
fxTerminate              -  4E550000  48E7C080  4EAEB3E6  4EAEB3EC
fxTerminate+0x10         -  4AAE8136  670A72FF  7000206E  81364E90
fxTerminate+0x20         -  4CED0102  FFF84E5D  4E754E55  000048E7
adeDownAudVid+0x6        -  C0804AAE  85CE670C  221741FA  22172008
adeDownAudVid+0x16       -  4EAEB32C  4AAE8136  670A72FF  7000206E
adeDownAudVid+0x26       -  81364E90  70012D40  81784878  000142A7
adeDownAudVid+0x36       -  72FF202F  00086176  508F4CED  0102FFF8
```

n Advanced Technique for Debugging the DCP

he following is an especially powerful technique that has been used to effectively ebug DCP problems. A stand-alone program, called ViewDCP, was developed to run n the player. It displays the DCP for a program in familiar mnemonics by responding to commands from a command line interface. ViewDCP opens the video path for itself but retrieves the DCP identifiers from a data module that is created by the CD-I application program. To determine the contents of the DCP at any point in the application code, a breakpoint is set and taken using SrcDbg. The **shell** command is used from inside SrcDbg to temporarily escape to an OS-9 shell where ViewDCP can be started and then used to examine the current contents of the DCP. The ViewDCP program is somewhat title specific in regard to the format of the data module that contains the DCP identifiers. However, it may be easily modified to fit the architecture of another title. This method is recommended for any title that relies on intricate DCP operations. ViewDCP will soon be available in **/usr/local/cdi**.

Summary

The following is a summary of the things that have been elaborated on in this document.

1) Reduce the size of the symbol table by compiling the minimum subset of files required for debugging with the **-g** option.

2) If necessary, translate the EOL characters in the source code to the character that OS-9 expects for EOL.

3) As much as possible, automate the steps described above by creating a "smart" make file. The make file can even be used to generate scripts for copying the source, symbols, and executable to a player device.

4) Be aware of how SrcDbg searches for the source code files if you have translated the EOL characters in the source code.

5) Do not arm the keyboard signal.

6) Make sure that the symbol table files have exectuable attributes.

7) Develop a method to compile out timers that interfere with debugging for debug versions of the code.

When debugging play-related problems, place an **ss_pause()** call just before the breakpoint. Use **ss_continue()** when it is time to resume execution of the code.

9) Learn how to set breakpoints and examine variables using scopes.

10) Use the **dump** command to determine the name of a function or variable that would otherwise be identifiable only by its address.

11) Customize your code to use ViewDCP for debugging DCP related problems.