Power User's Guide

ProDG for PlayStation®2 Debugger

SN Systems

SN Systems Ltd
Version 3.0
December 2002

ProDGPS2_PU(DBG)-E / Power User's Guide ProDG for PlayStation 2 Debugger / Dec. 2002

Document change control

Ver.	Date	Changes
3.0	Dec. 2002	Released with ProDG for PlayStation 2 v3.0. Changes since v2.2g: new chapter layout with pane descriptions moved to relevant chapters; new mixed-mode pane; new debugger scripting and scripting API chapters; new debugger switches -vs7 and -w<optional flags>; updated numerous debugger panes and shortcut menus; more info on hardware breakpoints; revised kernel pane.

Contents

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 1: Introduction

Overview of ProDG for PlayStation 2

ProDG for PlayStation 2 is a suite of development tools that enable you to build and debug your games for Sony's PlayStation 2. Using the ProDG tools you can build your application on a Win32 PC and debug it running directly on the Sony Computer Entertainment Inc. PlayStation 2 Development Tool DTL-T10000.

ProDG for PlayStation 2 consists of:

- **ProDG for PlayStation 2 Build Tools**: including Win32 port of the GNU PlayStation 2 tools; Microsoft Visual Studio integration option; ProDG compiler driver; ProDG linker and ProDG DLL linker; SN Systems utilities and library; ProDG assemblers. The Build Tools are described in *Power User's Guide to ProDG for PlayStation 2 Build Tools.*

- **ProDG Target Manager** that enables you to control connection to the PlayStation 2 targets in your network. See *User Guide to ProDG Target Manager for PlayStation 2.*

- **ProDG Debugger** for PlayStation 2, a fully featured Win32 debugger for debugging your PlayStation 2 applications.

This manual covers the ProDG Debugger as well as the advanced ProDG Debugger features of ProDG Plus (see "Upgrading to ProDG Plus" on page 1).

Upgrading to ProDG Plus

ProDG Plus for PlayStation 2 is a suite of advanced game development and debugging tools for the Sony PlayStation 2.

- **ProDG for PlayStation 2** — a suite of development tools for building and debugging PlayStation 2 games. It consists of a C/C++ compiler, assemblers, linker, debugger and target manager. An optional Visual Studio Integration provides App-Wizards for building executables and libraries, and launching the ProDG Debugger from within Visual Studio.

- **Advanced ProDG Debugger features** — debugger scripting, and other enhanced features to be announced, provide ProDG Plus customers with the 'gold standard' in PlayStation 2 debuggers.

- **Tuner for PlayStation 2** — lets you capture and visualize program behavior so that you can eliminate conflicts and bottlenecks in your code. High performance games can now be achieved with less guesswork. The Tuner captures data to a host PC in real-time while you play the game. The captured data can then be analyzed frame by frame and saved for later comparison with your optimized code.

- **NDK for PlayStation 2** — enables you to add networking capabilities to your PlayStation 2 game. The NDK TCP/IP stack is located on the IOP with a BSD like interface on the EE. We have added a fast EE API to significantly improve performance. NDK supports the Sony Network Adapter (Ethernet/modem) and the widest range of USB Ethernet adapters and USB modems.

For details about upgrading from ProDG to ProDG Plus, please contact **sales@snsys.com**.

Updates and technical support

First line support for all SN Systems products is provided by the Support areas of our website. To view these pages you must be a registered user with an SN Systems User ID and Password.

- If you have forgotten your User ID and Password, send an e-mail to **webmaster@snsys.com** and we will send you a reminder.

Once you have a valid User ID and Password you can visit our website Support areas at these URLs:

www.snsys.com/support (English)
www.snsys.jp/support (Japanese)

If the answer to your problem cannot be found on the Support areas of our website, you can also e-mail our support team at:

support@snsys.com (English)
j-support@snsys.com (Japanese)

Please make sure that you explain your problem clearly and include details of your software version and hardware setup. If you have been given an SN Systems support log number (LN number) then this should be quoted in all correspondence about the problem.

SN Systems Limited

4th Floor - Redcliff Quay
120 Redcliff Street
Bristol BS1 6HU
United Kingdom

Tel.: +44 (0)117 929 9733
Fax: +44 (0)117 929 9251

WWW: **www.snsys.com** (English)
www.snsys.jp (Japanese)

Updating ProDG over the web

Regular updates to ProDG for PlayStation 2 are posted on the SN Systems web site (see "Updates and technical support" on page 2). You need to select **Technical Support** > **Downloads**, to access the Downloads area. You will be required to give your SN Systems User ID and Password to access these pages.

- Full product updates of ProDG for PlayStation 2 are provided in the **Full Install** section.
- Fully tested component updates are found in the **Product Updates** section.
- Beta versions of components may be provided in the **Pre-Release Files** section.

If installing from a ZIP file, make sure when you extract the components that you select the drive that contains the \usr\local\sce directory you created when you installed the libraries. This way the Win32 executables will be installed in the correct locations in the directory structure.

- You must also ensure that the **Use Folder Names** option is selected in your zip tool.

Updating the ProDG Debugger and Target Manager

If you download the .zip archive for the ProDG Debugger and Target Manager then you should put the files where they were previously installed on your system. The default location is C:\Program Files\ProDG for PlayStation2.

If you choose to put the programs elsewhere, then you must configure your path to provide visibility to the commands. This involves adding a path to the ProDG Debugger and ProDG Target Manager executables, to the PATH= environment variable, either in your autoexec.bat file, or, if you have Windows 2000 or NT, via the Control Panel, in the **Advanced** tab of the **System** icon.

For example:

```
SET PATH=%PATH%;"C:\PS2\SN Programs"
```

To make this change apply globally you will need to restart your machine. You should now be able to type any of the application commands (ps2dbg or ps2tm) in any directory at the MS-DOS prompt, and the associated application will start.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 2: User interface

Overview of the Debugger

The ProDG Debugger for PlayStation 2 is a stand-alone source level Debugger for the PlayStation 2 console.

This Debugger has been designed and built specifically for the PlayStation 2 and is not just a MIPS Debugger adapted for PlayStation 2. It allows you to load, run and debug your application running on the different PlayStation 2 processors.

It includes the following main features:

- Custom support for additional R5900 MMI instructions and 128-bit data.
- The ability to create as many different Debugger pane types as you like, to display CPU registers (in 32/64/128-bit configurations), memory, disassembly, source, local variables, watch points, etc.
- A TTY pane to display printf streams from the PlayStation 2 processors.
- Multiple unit support - where appropriate, Debugger panes can be set to display the registers or memory of the different console processors including the I/O processor and two vector units.
- Source level debugging of main CPU provides unlimited software breakpoints, hardware breakpoints, single-step, step-over, run to cursor, etc., directly in your source code.
- Source code search paths allow source level debugging of anything you have source code for, regardless of who built it.
- Uses the industry standard ELF file format with STABS debug information so the Debugger is 100% compatible with ELF files built on Linux using the standard tools.
- IOP debugging.
- Integration with Microsoft Visual Studio.
- Fast update and display of target information.

- Full Windows local support for the PlayStation 2 sim: file serving device.
- Configurable Debugger pane and windows layout can be saved and restored.
- Updates with additional functionality will be regularly posted to our web site in response to user requests.

Launching the Debugger

The ProDG Debugger is launched via the command line or via a Windows shortcut. If you use the Windows shortcut then you will need to configure the shortcut properties to set command-line parameters to the Debugger where necessary and set the Debugger working directory.

Providing you have installed the ProDG for PlayStation 2 Microsoft Visual Studio Integration, the ProDG Debugger can also be started from Microsoft Visual Studio via a toolbar button.

When the Debugger starts it can either work with the ProDG Target Manager that is already running, or if the Target Manager has not already been started the Debugger will automatically start it (provided the ps2tm.exe executable is stored in a directory on your path).

Note: The ProDG Debugger will also fail to start if you have not set up at least one target in the ProDG Target Manager. You will need to start ProDG Target Manager and add a target.

It is important to set up a working directory for the Windows shortcut as otherwise the Debugger configuration files will be saved in different areas depending on the current directory at the time. For more information on Debugger configuration see "Configuring the user interface" on page 28.

ps2dbg command-line syntax

The ProDG Debugger for PlayStation 2 program is called ps2dbg.exe. This is the ps2dbg command-line syntax:

```
ps2dbg <switches> <file.elf> <app_params>
```

<switches>	Optional switches to specify Debugger options.
<file.elf>	Enables you to specify the name of the ELF application file to be loaded. If you do not specify this file on the command line you will need to load your ELF file from the menu later. **Note:** If the filename contains spaces then it must be enclosed in double quotes.
<app_params>	argc and argv[] parameters passed to main().

If you specify an ELF file but do not download it to the target, only the symbols will be loaded. This mode is suitable for post-mortem debugging of a crashed target.

The valid switches are:

-b	Specifies that breakpoints are not persisted in the configuration file when the Debugger exits.
-c	Makes the Debugger case insensitive for developers using SAMBA and Linux.
-d	Suppresses the Debugger behavior of "auto running to main" on loading a file. In this situation the target will not start running your application until you start it manually (using Debug > Go, the start toolbar button or <F9>).
-e	Loads the application executable that is contained in your ELF file, as well as the symbols.
-f	Resets the fileserver root directory, for the target connected to, to the directory of the ELF file loaded by the <file.elf> argument.
-h<filename>	Sets the home directory of the target.
-m	Allow multiple copies of the Debugger to run (i.e., to allow you to run two instances of the Debugger to debug code running on two different PlayStation 2s. If you do not specify this switch then starting the Debugger a second time will cause a currently running version to be brought to the front.
-ns	Stops the Debugger autoswitching to a pane on the unit that has just stopped.
-r	Resets the PlayStation 2 target before down loading the executable.
-s	Safe symbol loading. Since version 1.19 of the Debugger, symbol downloading with large ELF files is faster. This is because by default the Debugger assumes all types of the same name in different modules are identical. However, if you have different types in different modules with the same name and you want the Debugger to resolve them properly then you will need to specify this switch.
-S<filename>	Names the script file used by the Debugger at startup. See "What is Debugger scripting?" on page 145.
-t<name>	Specify the target to connect to using its name. If you enter a name that cannot be identified in the list of targets on the Target Manager, or leave the –t option blank, then a dialog appears asking you to select from the available targets when you launch the Debugger. Note that if the target name contains spaces you must enclose it in double quotes on the

	command line (e.g., ps2dbg /t"MikesT10K over there").
-nd	Do not disconnect from the target when the Debugger exits. This overrides the default Debugger behavior that is to leave the target in the state it was in when the Debugger started.
-da	Always disconnect from the target when the Debugger exits. This overrides the default Debugger behavior that is to leave the target in the state it was in when the Debugger started.

-v#	Select VU access method, where # is a digit:
	0=block VU access if EE running (safe);
	1=allow VU access if EE running (not safe - EE may restart VU and access clash may corrupt it);
	2= reserved for stopEE/accessVU/restartEE (not implemented yet).

-vs	Enables Microsoft Visual Studio compatibility features, such as the import and export of Visual Studio breakpoints at the start and end of a debug session.
-vs7	Same as -vs but used for Visual Studio .NET.
-w<optional flags>	Specifies a subset of windows to be updated on target exception (e.g. single step). See "To select which panes are updated" on page 18.
-x	Code is to be executed on the target after load.
-xi	Stops the download of SNDBGEXT.IRX to the target, thus disabling IOP kernel support. EE is unaffected.

ps2dbg command-line examples

An example of a command line is:

```
ps2dbg -tMikesT10K -r -e main.elf
```

This will start the Debugger, initiate a connection (if not already connected) with "MikesT10K", load symbols from the file main.elf, reset the PlayStation 2, load main.elf and auto-run the program to main().

Another example is:

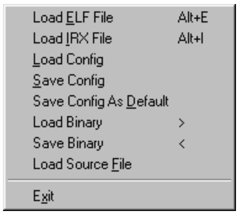
```
ps2dbg -t"MikesT10K over there" -rex main.elf param1 param2
```

This will start the Debugger, initiate a connection with "MikesT10K over there", load symbols from main.elf, reset the PlayStation 2 target, load the executable code from main.elf and start it running passing main() the parameters argc=3, argv[0]="main.elf", argv[1]="param1", and argv[2]="param2".

Menu options

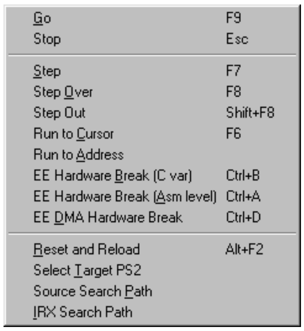
ProDG Debugger contains menu options labelled **File**, **Debug**, **Settings** and **Window**. The following sections describe their operation.

File menu



Load ELF File	Loads an ELF file to the PlayStation 2 EE processor. A dialog is presented to allow you to select a file and set file load options.
Load IRX File	Loads an IRX file to the PlayStation 2 IOP processor. A dialog is presented to allow you to select a file and set file load options.
Load Config	Reload the Debugger configuration file from the current working directory and overwrites your current configuration.
Save Config	Saves the current Debugger configuration. It is automatically saved in dbugps2.ps2 in the current working directory.
Save Config As Default	Saves the current Debugger configuration as the default configuration, which will be loaded if the dbugps2.ps2 file cannot be found in the current working directory.
Load Binary	Allows you to download a binary file to target memory.
Save Binary	Allows you to save target memory as a binary file.
Load Source File	Allows you to locate a source file and load it into a new source pane.
Exit	Shuts down ProDG Debugger.

Debug menu

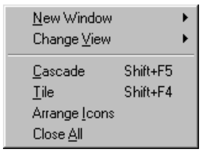


Go	Starts the application running.
Stop	Stops the application running.
Step	Single-steps the Debugger.
Step Over	Single-steps to the next line in the current function, stepping over code in any function on the current source line.
Step Out	Single-steps to the next line in the calling function, stepping out of the current function.
Run to Cursor	Sets the Program Counter to the line pointed to by the pane cursor.
Run to Address	Sets the Program Counter to the specified memory address.
EE Hardware Break (C var)	Allows you to set a C variable hardware break.
EE hardware Break (Asm level)	Allows you to set an assembler level hardware break.
EE DMA Hardware Break	Allows you to set a DMA hardware break.
Reset and Restart	Resets the Program Counter to the program entry point.
Select Target PS2	Allows you to select a target.
Source Search Path	Allows you to set or modify the search path for finding source files.
IRX Search Path	Allows you to set or modify the search path for IRX files.

Settings menu

Options	Allows you to set the ProDG Debugger pane colors and fonts, accelerator keys and DMA errors detected.

Window menu

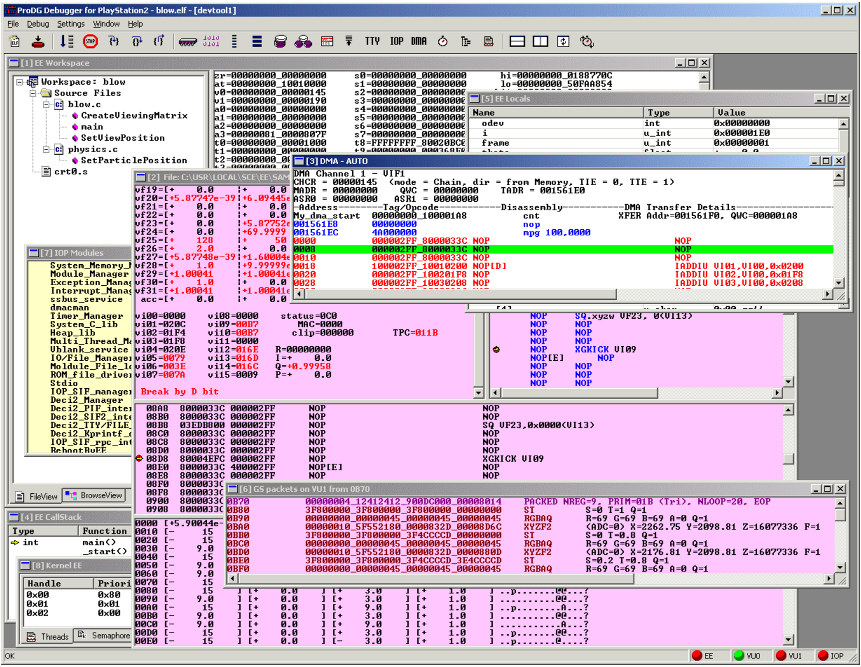


New Window	Allows you to create a new window in the Debugger.
Change View	Allows you to change the type of the current pane to another pane type.
Cascade	Arranges all windows in standard Windows cascade format (overlapping).
Tile	Arranges all Debugger windows in standard Windows tile format (non-overlapping).
Arrange Icons	Arranges all minimised windows.
Close All	Closes all open windows.

Windows and panes

The main functionality of the Debugger can be accessed through menus, toolbar buttons and shortcut keys. This section contains information on the available panes in the Debugger, and how to create and manipulate them using keyboard shortcuts, menus and toolbars.

ProDG Debugger is made up of one or more *windows* that contain *panes* for viewing the different types of information obtained from the target. Each window may contain one or more different pane types, as in this example:


ProDG Debugger for PlayStation2 - blow.elf - [devtool1]
File Debug Settings Window Help
TTY IOP DMA
[1] EE Workspace
[7] IOP Modules
FileView BrowseView
[4] EE CallStack
[8] Kernel EE
Threads Semaphore
[2] File: C:\USR\LOCAL\SCE\EE\SAM
[5] EE Locals
[3] DMA - AUTO
[6] GS packets on VU1 from 0B70
OK
EE VU0 VU1 IOP

```
Workspace: blow
  Source Files
    blow.c
      CreateViewingMatrix
      main
      SetViewPosition
    physics.c
      SetParticlePosition
  crt0.s
```

```
zr=00000000_00000000      s0=00000000_00000000      hi=00000000_0188770C
at=00000000_10010000      s1=00000000_00000000      lo=00000000_50FAA854
v0=00000000_00000145      s2=00000000_00000000
v1=00000000_00000190      s3=00000000_00000000
a0=00000000_00000000      s4=00000000_00000000
a1=00000000_00000000      s5=00000000_00000000
a2=00000000_00000000      s6=00000000_00000000
a3=00000081_0000807F      s7=00000000_00000000
t0=00000000_00001000      t8=FFFFFFFF_80020BC0
```

Name	Type	Value
odev	int	0x00000000
i	u_int	0x000001E0
frame	u_int	0x00000001

```
DMA Channel 1 - VIF1
CHCR = 00000145  (mode = Chain, dir = from Memory, TIE = 0, TTE = 1)
MADR = 00000000    QWC = 00000000    TADR = 001561E0
ASR0 = 00000000   ASR1 = 00000000
-Address--------Tag/Opcode-----------Disassembly-----------DMA Transfer Details-----------------
My_dma_start   00000000_100001A8         cnt              XFER Addr=001561F0, QWC=000001A8
001561E8       00000000                  nop
001561EC       4A000000                  mpg 100,0000
0000           000002FF_8000033C NOP                           NOP
0008           000002FF_8000033C NOP                           NOP
0010           000002FF_8000033C NOP                           NOP
0018           100002FF_10010200 NOP[D]                        IADDIU VI01,VI00,0x0200
0020           000002FF_100201F8 NOP                           IADDIU VI02,VI00,0x01F8
0028           000002FF_10030208 NOP                           IADDIU VI03,VI00,0x0208
```

```
vf19=[+     0.0     |+      0.0
vf20=[+5.87747e-39 |+6.09445e
vf21=[+     0.0     |+      0.0
vf22=[+     0.0     |+      0.0
vf23=[+     0.0     |+5.87752e
vf24=[+     0.0     |+69.9999
vf25=[+     128     |+       50
vf26=[+     2.0     |+      0.0
vf27=[+5.87748e-39 |+1.60004e
vf28=[+     1.0     |+9.99999e
vf29=[+1.00041      |+1.00041e
vf30=[+     1.0     |+      0.0
vf31=[+1.00041      |+1.00041e
 acc=[+     0.0     |+      0.0

vi00=0000     vi08=0000      status=0C0
vi01=020C     vi09=00B7        MAC=0000
vi02=01F4     vi10=00B7       clip=000000        TPC=011B
vi03=01F8     vi11=0000
vi04=020E     vi12=016E     R=00000000
vi05=0079     vi13=016D     I=+      0.0
vi06=003E     vi14=016C     Q=+0.99958
vi07=007A     vi15=0009     P=+      0.0

 Break by D bit
```

```
NOP       SQ.xyzw VF23, 0(VI13)
NOP       NOP
NOP       NOP
NOP       NOP
NOP       XGKICK VI09
NOP[E]       NOP
NOP       NOP
NOP       NOP
NOP       NOP
```

```
System_Memory_
Module_Manager
Exception_Manag
Interrupt_Manag
ssbus_service
dmacman
Timer_Manager
System_C_lib
Heap_lib
Multi_Thread_Ma
Vblank_service
IO/File_Manager
Moldule_File_lo
ROM_file_driver
Stdio
IOP_SIF_manager
Deci2_Manager
Deci2_PIF_inter
Deci2_SIF2_inte
Deci2_TTY/FILE_
Deci2_Kprintf_d
IOP_SIF_rpc_int
RebootByEE
```

```
 08A8  8000033C 000002FF       NOP                    NOP
 08B0  8000033C 000002FF       NOP                    NOP
 08B8  03EDB800 000002FF       NOP                    SQ VF23,0x0000(VI13)
 08C0  8000033C 000002FF       NOP                    NOP
 08C8  8000033C 000002FF       NOP                    NOP
 08D0  8000033C 000002FF       NOP                    NOP
 08D8  80004EFC 000002FF       NOP                    XGKICK VI09
 08E0  8000033C 400002FF       NOP[E]                 NOP
 08E8  8000033C 000002FF       NOP                    NOP
 08F0  8000033C
 08F8  8000033C
 0900  8000033C
 0908  8000033C
```

Type	Function
int	main()
	_start()

Handle	Priori
0x00	0x80
0x01	0x01
0x02	0x00

```
0B70     00000004_12412412_900DC000_00008014   PACKED NREG=9, PRIM=01B (Tri), NLOOP=20, EOP
0B80     3F800000_3F800000_3F800000_00000000   ST         S=0 T=1 Q=1
0B90     00000000_00000045_00000045_00000045   RGBAQ      R=69 G=69 B=69 A=0 Q=1
0BA0     00000010_5F552180_0000832D_00008D6C   XYZF2      (ADC=0) X=2262.75 Y=2098.81 Z=16077336 F=1
0BB0     3F800000_3F800000_3F4CCCCD_00000000   ST         S=0 T=0.8 Q=1
0BC0     00000000_00000045_00000045_00000045   RGBAQ      R=69 G=69 B=69 A=0 Q=1
0BD0     00000010_5F552180_0000832D_0000880D   XYZF2      (ADC=0) X=2176.81 Y=2098.81 Z=16077336 F=1
0BE0     3F800000_3F800000_3F4CCCCD_3E4CCCCD   ST         S=0.2 T=0.8 Q=1
0BF0     00000000_00000045_00000045_00000045   RGBAQ      R=69 G=69 B=69 A=0 Q=1
```

```
0000 [+5.90044e-
0010 [-     15
0020 [-     15
0030 [-    9.0
0040 [-     15
0050 [-    9.0
0060 [-    9.0
0070 [-     15
0080 [-     15   ] [+    0.0   ] [+    3.0   ] [+    1.0   ] ..p.......@@...?
0090 [-    9.0   ] [+    0.0   ] [+    3.0   ] [+    1.0   ] ..........@@...?
00A0 [-     15   ] [+    0.0   ] [+    9.0   ] [+    1.0   ] ..p........A...?
00B0 [-    9.0   ] [+    0.0   ] [+    3.0   ] [+    1.0   ] ..........@@...?
00C0 [-    9.0   ] [+    0.0   ] [+    9.0   ] [+    1.0   ] ...........A...?
00D0 [-     15   ] [+    0.0   ] [+    3.0   ] [+    1.0   ] ..p.......@@...?
00E0 [-     15   ] [+    0.0   ] [-    3.0   ] [+    1.0   ] ..p.......@....?
```

If there is more than one pane in a window it is referred to as a *split pane view*.

The layout of windows and panes is saved in the Debugger configuration file at the end of each debug session (see "Configuring the user interface" on page 28).

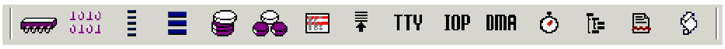
Creating new Debugger windows

A new window containing any of the Debugger panes can be opened at any time. These windows can either be opened using the main toolbar buttons or via the **Window > New Window** menu. In addition you can change the type of an existing pane using the **Change View** command in its shortcut menu.

A new pane can be created in an existing window by splitting it. For more information see "To split a pane horizontally or vertically" on page 15.

By default Debugger panes are updated with the current information whenever your application stops running on the target. There are some other update options that you can set. For more information see "Debugger pane update" on page 17.

Each pane type has its own shortcut menu that can be accessed via the right mouse button. The shortcut menus contain commands that are specific to each pane type. Some of the most frequently used commands can also be accessed by keyboard shortcuts. The types of pane available in the Debugger are briefly described below:



The buttons open the Debugger panes in the following order:

Registers view	Enables you to view the current register values on the PlayStation 2 unit being viewed. It also shows the program counter and the status of the target.
Memory view	Enables you to view memory on the PlayStation 2 unit being viewed.
Disassembly view	Enables you to view the disassembly that is currently running on the PlayStation 2 unit being viewed. It shows the instruction that the program counter is set on, and you can set breakpoints and single-step execution on the target unit.
Source file view	Enables you to view the source of the program that is currently running on the PlayStation 2 unit being viewed. The current program counter is shown and you can set breakpoints and single-step through your source running on the target unit.
Local variables view	Enables you to view the values of all the local variables in the current function. You can expand or close the display of any structures or arrays using the pane shortcut menu.
Watch view	Enables you to view a selected set of variables that you wish to track. You can add or remove watches, and expand or close the display of members of any watched structures or arrays.
Breakpoint view	Enables you to view a list of all the breakpoints that have been set in your application. They are indicated by the address of the line of source or disassembly in memory.
CallStack view	Enables you to view the function calls on the call stack that have been made to arrive at the current position in the program. The most recently called function is shown at the top of the call stack. You can change the Debugger context by selecting a different function call.
TTY console view	Enables you to view any standard output generated by the PlayStation 2 target.
IOP modules view	Enables you to view a list of IOP modules currently loaded.
DMA view	Enables you to view data sent to a DMA channel.
Profile view	Enables you to produce a basic profile of main CPU usage, so that you can see which processes are taking most time.
Workspace view	Creates a workspace for viewing in a tree view the ELF's source files, functions, variables and classes.
Kernel view	Enables you to view information about all the threads running on the target.

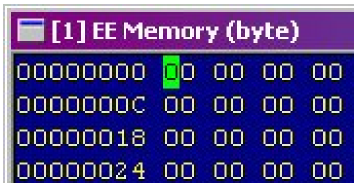
Script pane	Enables you to write 'C' style immediate statements and run pre-written script files.

To navigate between windows

If you have created more than one window, you can select it just by clicking the mouse on it. However it is also possible to cycle between windows using a keyboard shortcut. There are two ways to do this:

1. Press the standard Windows shortcut <Ctrl+Tab> and the active window will change to the next ProDG Debugger window in the list.

2. You can switch between windows via the <Ctrl+[0-9]> keys. The first 10 MDI windows created are assigned a number that appears in the title bar and this number is the index you use to bring it to the front of the pane layout.:



[1] EE Memory (byte)

```
00000000 00 00 00 00
0000000C 00 00 00 00
00000018 00 00 00 00
00000024 00 00 00 00
```

3. For example, to bring window [1] to the front, press <Ctrl+1>. Any windows above 10 are not assigned an index so there is no way of quickly switching to them. If you delete a window, then its index is reused next time you create a window, while the other windows keep their original indexes.

Note: The window indexes are persisted in the debugger's project-specific configuration file so when you load up the debugger again the window indexes will be as you left them.

Navigating between source/disassembly/ mixed mode panes

When a source, disassembly or mixed mode has the focus, you can switch between these types of pane by pressing the spacebar. The sequence is source to disassembly to mixed mode to source etc.

Note that the keystroke for this facility is configurable via the **Toggle src to dis** option in both the source and disassembly pane accelerator key groups (see "Accelerator keys" on page 31).

Pane shortcut menus

Each pane has a shortcut menu, accessed by right-clicking on the body of the pane. The format of this menu varies, but some menu items are common to all pane types:

Change View	Allows you to change the type of the current pane to another pane type.
Pane	Enables you to access the commands to

	manipulate split pane views (split, delete, etc.).
Delete Pane	Deletes the currently selected pane.
Set Font	Enables you to change the display font for information in the current pane only. If you wish to change the display font globally for all panes use the Application Settings dialog (see "Pane colors and fonts" on page 28) .

Descriptions of these options will not be repeated, when describing the individual pane shortcut menus.

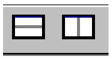
Creating split pane views

Once you have created a new window containing a single pane you can split it in a variety of different ways and specify the pane type that is to be put in the new pane site.

To split a pane horizontally or vertically

1. Select the pane you wish to split.

2. Right-click to obtain the pane shortcut menu and click **Pane > Split Horizontally** or **Split Vertically** or from the Debugger toolbar click the **Split View Horizontally** or **Split View Vertically** toolbar buttons.

   The pane is split horizontally or vertically, and the new pane will contain an identical pane type to the originally selected pane.

3. You now need to use the **Change View** option in the shortcut menu to indicate the required pane in the newly created pane site (see "To change the type of a pane" on page 15).

To change the type of a pane

The type of an existing pane can be changed at any time using the **Change View** option in the pane shortcut menu. This enables you to access a submenu of all the different pane types.

Alternatively you can change the pane type using the Change Pane Type accelerator key:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.

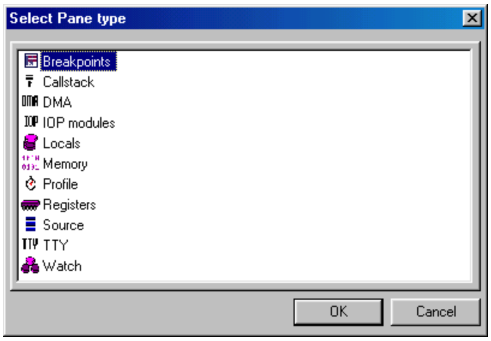
2. In the list box at the top right, select **Application**.

3. Scroll down the **Command Name** list until the accelerator key setting for **Change Pane Type** is shown and then note the appropriate **Key Sequence** (if any).

4. Press **Cancel** to close the Application Setting dialog.

5. Select the pane to be changed to a different pane type.

6. Press the key sequence and a dialog is displayed in which you can select the new pane type.



7. Use the up and down arrow keys to navigate in the list and then when the required pane is selected press <Return> or click **OK**.

To delete an existing pane

1. Select the pane that is to be deleted.

2. Click **Pane > Delete Pane**.

3. Once you have deleted the pane, the pane that it had been split from will expand to take up the space left by the deleted pane.

To move the focus between panes

At any one time in the Debugger one of the panes has the focus and you can work in this pane. This pane can be set just by clicking the mouse on it.

Alternatively you can move the focus between panes using the Move Focus accelerator keys:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.

2. In the list box at the top right, select **Application**.

3. Scroll down the **Command Name** list until the accelerator key settings for **Move Focus up / down / left / right** are shown and then note the appropriate **Key Sequence** (if any).

4. Press **Cancel** to close the Application Setting dialog.

5. Press the key sequence you need to move the focus to the pane that you would like to navigate to. If there is no pane in the arrow direction that you select, then the current pane will not change.

To move pane splitter bars

The splitter bars in a split pane view can be moved to allow more space for the panes on either side of it. The bars can simply be picked up and dragged using the mouse.

Alternatively, you can move the pane splitter bars using the Move bar accelerator keys:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.
2. In the list box at the top right, select **Application**.
3. Scroll down the **Command Name** list until the accelerator key settings for **Move top / bottom / left / right bar out / in** are shown and then note the appropriate **Key Sequence** (if any).
4. Press **Cancel** to close the Application Setting dialog.
5. Select the pane for which you wish to move the bordering bar.
6. Press the key sequence you need for moving the bar in the direction desired. Note that this will not affect the pane border if it is the border of the window.

Debugger pane update

By default all Debugger panes are updated when the target stops running. You can also update all the pane using the **Update all views** and **Toggle auto-update** buttons on the Debugger toolbar.

The **Update all views** button performs a one-time refresh of all pane contents, whereas if you enable auto-updating by pressing the **Toggle auto-update** button the panes are continually refreshed by polling the target for up-to-date information.

You can also limit which panes are automatically updated when the target stops, by using the -w command-line switch.

To update all panes manually

- Click the **Update all views** button on the toolbar:

  This causes the target to be polled once for up-to-date information, which is then redisplayed in the appropriate open panes.

To update all panes automatically

- Click the **Toggle auto-update** button on the toolbar.

  This causes the target to be polled continually for up-to-date information, which is then displayed in the appropriate open panes.

- The **Toggle auto-update** button will appear to be depressed when the auto-update feature is ON. Press the **Toggle auto-update** button again to turn OFF the auto-update feature.

To select which panes are updated

By default a single step or breakpoint exception will update register panes and disassembly/source panes. If there is time (i.e. if there is not another key press waiting to be handled) then the Debugger will go on to update more windows until it sees another key press. It is however possible to force the Debugger to update only a subset of panes by using the -w command line switch. For example, you can force a full update of watch panes whilst you are single stepping. The optional flags are:

b	breakpoint panes
c	callstack panes
d	disassembly panes
k	kernel panes
l	local panes
m	memory panes
r	register panes
s	source panes
t	TTY panes
w	watch panes

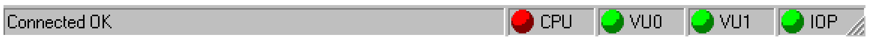
e.g. starting the debugger with the command line

```
ps2dbg -wclw
```

will force the debugger to always update callstack panes, locals panes, and watch panes in addition to the default register and disassembly/source panes.

Target and processor status

The ProDG Debugger window contains a status footer which includes the target connection status and a graphical representation of the status of the target processors, similar to the following:



Connected OK 🔴 CPU 🟢 VU0 🟢 VU1 🟢 IOP

When the processor is stopped its "Smarty" (=M&M) is red. The color changes from red to green when the processor becomes active.

Text selection and the clipboard

Text selections have been implemented in the Source and TTY panes to enable text to be selected and copied to the clipboard. These selections work in a similar way to Windows text editors and word processors such as Notepad or Microsoft Word.

Text may be selected using either the mouse or keyboard (or even a combination of both). However, the TTY pane only supports selections made using the mouse.

Memory addresses can also be selected using the mouse in a memory pane. See "Memory pane" on page 81 for further details.

Making selections using the mouse

To make a selection with the mouse, move the cursor to the first character that is to be selected and hold down the left mouse button. Then drag the mouse to the last character for the selection and release the mouse button. While dragging the mouse, the text within the selection will be displayed in inverted colors to show it has been selected. Once selected, the text may be copied to the clipboard using the context menu (while the mouse is over the selected text) or using the keyboard accelerator (user configurable from the Source Settings tab, but defaults to <Ctrl+C>) and then may be pasted into either the watch pane or another application.

Making the selection in this way will select horizontally across lines. If the mouse is moved down several lines, the first line will be selected from the selection start position up to the end of the line, and the last line will be selected from the start of the line up to the end position (where the mouse is). Lines between will be completely selected.

If the <Alt> key is held down (before the mouse button) when defining a selection, *Column mode* will be enabled. This allows a rectangular area to be selected rather than full lines spans. For example you could select the first six characters of each line in the window by starting from the top left of the source pane and dragging down to the bottom of the pane and six characters in. Ordinarily this would have selected the contents of the whole pane.

Making selections using the keyboard (source pane only)

The source pane contains a cursor – either a full-width bar or a vertical line cursor. This can be moved around the window using the arrow keys.

Holding the <Shift> key down while using the arrow keys makes a selection in the same way as holding down the mouse button. The arrow keys may then be used to select lines (up and down) and/or rows (left and right).

There are also several methods of making certain types of selection quickly.

To select text with the mouse

- Double-clicking over a word with the left mouse button will select the entire word.
- Clicking with the left mouse button in the left border (when within this area, the mouse cursor will change from a vertical bar to a right-pointing arrow) to select an entire line.
- Moving up and down after the above to select multiple lines.

To select text with the keyboard (source pane only)

- Holding the <Ctrl> key down will enable Word Selection. When this is held down (along with the selection key – <Shift>), the left and right arrow keys will select the previous or next word respectively.
- <Ctrl+Shift+Home> selects from the cursor position to the top of the file.
- <Ctrl+Shift+End> selects from the cursor position to the bottom of the file.
- <Shift+Home> selects from the cursor position to the start of the line.
- <Shift+End> selects from the cursor position to the end of the line.

Other

- The context menu contains **Select All** and **Clear Selection** menu items (these have user-configurable accelerators).

When defining a selection with the mouse or keyboard, the file will scroll in the appropriate direction when the edge of the pane is reached.

With the mouse, the speed of scrolling will be adjusted according to the distance of the mouse between the edge of the pane and the edge of the screen. The closer the mouse is to the edge of the screen, the faster the scroll.

Usage notes

- Clicking the mouse within the pane (or pressing one of the arrow keys) cancels the current selection.
- All selection keyboard accelerators are user-configurable from the **Accelerator** tab in the **Settings** dialog.
- When displaying the Find Text dialog box, any text selection will automatically be inserted into the text control when the dialog is displayed. If a multiple line selection was made, only the first line will be added.

- Double-clicking in the border area of the source pane will add a breakpoint at that line in the code.
- The watch pane now has a **Paste** function (which appears on the shortcut menu – default accelerator of <Ctrl+V>). So a variable name can be selected in the source pane, copied to the clipboard, and then pasted to the watch pane (only the first line is pasted for multi-line selections).

Expressions

As you use the Debugger you will find it necessary or useful to enter expressions. An expression might be as simple as a hexadecimal constant or a C variable or as complex as a typecast of a structure member found by de-referencing a pointer from an array.

The ProDG Debugger allows you to build any such expression, using the variables known to be in your program and a wide selection of C and C++ style operators.

You can also evaluate expressions on the spot to check that you have defined them properly.

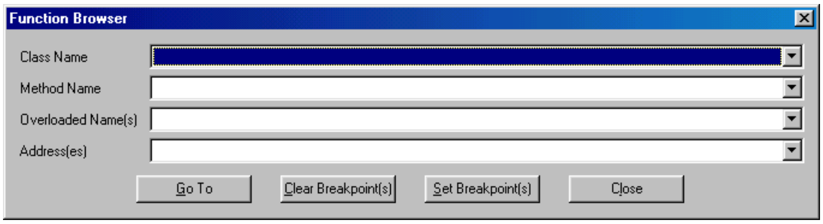
- Expression and address evaluation will now try all active symbol tables to evaluate an expression.

Expressions can be used wherever a Debugger dialog asks you for an address, expression or value. The watch and locals pane refer to it constantly, so it is essential that you understand its operation in order to exploit its full power.

Using the function browser

You can use the function browser in the source, disassembly or breakpoint panes to move through your application source or disassembly according to the functions in it. In addition you can set or remove breakpoints globally on a particular function.

To open the function browser you can use the Browse Functions command in the source, disassembly or breakpoint pane's shortcut menus.



Name demangling

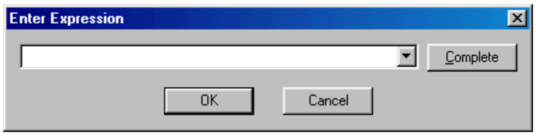
You can choose to display "demangled" C++ function names at various places in the Debugger, for example in a breakpoints or call stack frame.

To demangle C++ function names

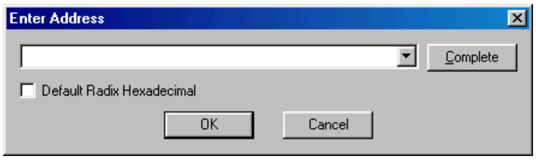
To demangle C++ function names you simply need to copy the file demangle.dll to your ProDG for PlayStation 2 program directory. When the Debugger detects this DLL it will automatically convert "mangled" function names into "demangled" ones.

Entering expressions and addresses

The Enter Expression dialog is used to enter an expression when adding a watch to the watch pane:



The Enter Address dialog is used to enter an address when setting a breakpoint in the Breakpoints pane, and this behaves very similarly to the Enter Expression dialog:



Both the Enter Expression and Enter Address dialogs provide the following facilities:

- name completion
- expression / address copying and pasting to/from the clipboard
- a history of most recently used expressions or addresses, accessible from a drop-down menu.

Hover evaluation in the source pane

The ProDG Debugger includes a 'hover evaluation' feature whereby holding the mouse cursor over the source pane for a short time attempts to evaluate the expression at that position in the source code.

Holding the mouse over 'main' in the code fragment below would display function type and address information in a ToolTip, while for the variables it would evaluate the whole 'front_end_game_data.selected_country' expression and display type and value information.

With the implementation of text selections, this has been enhanced to allow the evaluation of the selected text. This is very useful in the example given below for evaluating part of the variable name.

For example, selecting the 'front_end_game_data.selected_country' text below would work as before, however selecting just the 'front_end_game_data' part of the variable and holding the mouse over this, would simply evaluate the selection only and (in this case) display type and address information for the class:

```
main()
{
    front_end_game_data.selected_country = 0;
    front_end_game_data.selected_game_mode =
WWR_TIME_ATTACK_MODE;
    front_end_game_data.selected_class = NOVICE_CLASS;
    front_end_game_data.selected_car = WWR_DOUBLE_MOON;
    ........................
}
```

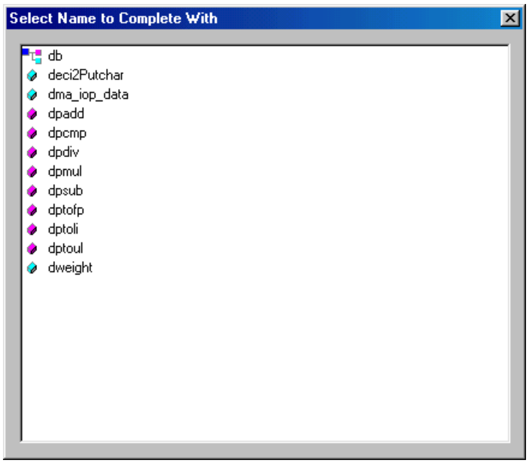
Name completion

The **Complete** button can be used to speed up entering a variable name, by taking a partly completed name and then displaying a list of all the variable names which begin with the partial string.

- Name completion searches all active symbol tables for names.

If the search string matches none of the available names, then the Complete option does nothing. If only one name matches the starting string, then the name is simply completed, without displaying a list box.

If the search string matches two or more names then the Select Name to Complete With window is shown. For example, if you entered "d" and then clicked **Complete**, you will be shown a list of names starting with the letter "d-", from which the one you need can be selected:



db
deci2Putchar
dma_iop_data
dpadd
dpcmp
dpdiv
dpmul
dpsub
dptofp
dptoli
dptoul
dweight

Note that each symbol is shown with an icon which reflects its type. A cyan diamond = variable; pink diamond = function; tree = class or structure.

Building expressions

The primary elements of expressions are numbers and variable names.

- you can enter numbers either in decimal or in hexadecimal format
- hexadecimal numbers must be prefixed with 0x, unless the **Default Radix Hexadecimal** checkbox is checked when you can safely omit the 0x (provided the number starts with a decimal digit, e.g., "0A0" not "A0").
- you can enter a variable by typing its name and can use the **Complete** button to save typing or ensure that you are referring to the right name (see "Name completion" on page 23).

More complex methods of building expressions include: C and C++ operators; label and function addresses; and typecasting.

Register names

You can use register names in an expression. Register names must be prefixed with a $ symbol, to distinguish register names from variables.

Registers have a C type of long128 if uncast and of the appropriate type if cast (see "Typecasts and typedefs" on page 24).

Typecasts and typedefs

You can typecast any expression just as you would in C. For example, if you entered (int*)$fp in a watch pane, you might see the following:

```
+ (int*)$fp   int *  -> 00000001
```

You can use structure tags to typecast but you are not required to enter the keyword struct when casting to a structure tag. You would expect to see the following when typecasting to a structure or class:

```
-Tester*  (Tester*)$fp = 0x807ff88
-Tester
+unsigned char* m_pName = 0x00000645
+unsigned char* mpLongName = 0xFFFFFFFF
```

You can also cast to typedefs; for example, entering (daddr_t)p might produce:

```
long (daddr_t)p = 0x00003024
```

Labels

You can use labels in an expression. The evaluator tries to match variable names first, then looks for labels.

Labels have a C type of int.

Functions

You can use a function name in an expression. The value of a function name is its address.

Functions appear in a watch window as follows:

```
main   int ()     @ 00201020
```

Functions have a C type of int.

The precedence for matching names

The search order for a name in an expression is as follows:

1. Escaped register names (names prefixed with $)
2. C names
3. Label names

Moving data between panes

Drag-and-drop has been implemented to facilitate the movement of data between different panes. For example, if you want to add a breakpoint at a particular function, but cannot remember the exact name/spelling of it, you can browse through the Workspace pane to find the function, and then 'drag' the function name across to the Breakpoint pane and 'drop' it; this adds a breakpoint at the address of the function.

Note: As in normal Windows usage, drag-and-drop is carried out by moving the mouse over the information to be dragged and holding down the left mouse button. The mouse is then moved (with the button still held down) over to the pane on which it is to be dropped. The mouse button is then released and the data is copied into the new pane.

The outcome of a drag-and-drop operation varies according to the pane from which the data was dragged, and that which it is dropped on. See the table below for a diagram of the possible combinations:

	Drag to						
Drag from	SOURCE	WRKSPC	MEMORY	WATCH	LOCALS	BRKPNT	DISASM
SOURCE	Goto function		Goto address	Add watch		Add bp	Goto
WRKSPC file	Load					Add bp to all funcs in file	
WRKSPC browse	Goto function		Goto address	Add to watch		Add Bp	Goto
MEMORY	Goto function		Goto address	Add to watch		Add bp	Goto function
WATCH	Goto function		Goto address	Duplicate/ split watch entry		Add bp	Goto function
LOCALS			Goto address	Add to watch			
BRKPNT	Goto function		Goto address	Add to watch			Goto function
DISASM							
CALLSTK	Goto function		Goto address	Add to watch		Add Bp	Goto function

When the mouse is dragging data, the cursor changes to indicate that data is being dragged.

Drag cursor— drag-and-drop in progress.

No Entry cursor— cannot drop here.

- Goto operations (dragging onto source pane) work on functions only at present
- Drag from the memory pane to a different type of pane only works from addresses at the start of an object. To see ly dragging from the address of the variable will be correctly evaluated and added to the watch pane. Dragging to another memory pane will work with any address.
- On the memory pane, dragging must be from the address at the left of the pane.
- On the source pane, text selections are dragged. If multiple lines are selected, only the first line is dragged. Leading spaces are removed.
- If a structure is added to the watch and is expanded, the members of the item may be dropped onto the watch pane to add a watch

expression for just that member. For example, if 'front_end_game_data' was added to the watch pane and expanded, then dragging the 'selected_country' member variable and dropping it back onto the watch pane would add a new watch expression for 'front_end_game_data.selected_country' (see "Drag-and-drop and the source pane" on page 27 for source fragment).

- In the watch pane, for dragging and dropping onto itself the cursor must move by at least the height of the line for the drop to be carried out. This is to avoid accidental drag operations (e.g. moving the mouse when clicking to expand a structure).

Drag-and-drop and the source pane

To drag information from the source pane, a text selection should be made. The mouse button should then be held down with the cursor over the selected data. This can then be dragged in the normal way. If the selection covers multiple lines, only the first line will be 'dropped'.

For example, selecting and dragging 'front_end_game_data. selected_country' in the code to the watch pane would add this class member to the watch pane. However, selecting just the 'front_end_game_data' part of the line and dropping this in the watch pane would add the actual class as a watch variable, allowing it to be expanded to show all member variables.

```
main()
{
    front_end_game_data.selected_country = 0;
    front_end_game_data.selected_game_mode =
WWR_TIME_ATTACK_MODE;
    front_end_game_data.selected_class = NOVICE_CLASS;
    front_end_game_data.selected_car = WWR_DOUBLE_MOON;
    ........................
}
```

Pane modifier keys

When dropping data onto a pane, it may be desirable to create a new pane rather than add to the existing one (e.g. load a source file into a new pane while leaving the exist source pane unmodified).

To enable this, three keyboard modifiers (<Ctrl>, <Shift> and <Alt>) have been utilized. These keys are not currently user-configurable. Holding one of these keys down while dropping data onto a pane activates the modifier.

- The <Ctrl> key splits the destination pane vertically and drops data into the new pane (beneath the existing one) .

- <Shift> splits horizontally and drops into the rightmost pane.

- Finally, the <Alt> key creates a new (floating) pane for the dropped data.

If data is dragged over a pane that cannot accept the data (or over an area of the screen that doesn't contain any panes), the cursor will change to show a No Entry sign (see "Moving data between panes" on page 25). Releasing the mouse button while the No Entry cursor is displayed will abort the drag operation and discard the data.

Configuring the user interface

The ProDG Debugger panes are now fully and individually configurable so that you can set the appearance of the user interface exactly how you like it. You can set the following parameters in the Application Settings dialog:

- pane colors and fonts, including syntax coloring in source panes
- accelerator keys
- types of DMA error detected in the DMA pane

In addition, the layout of panes, the name of the last target connected to, and other project settings can be saved and restored as a project configuration.

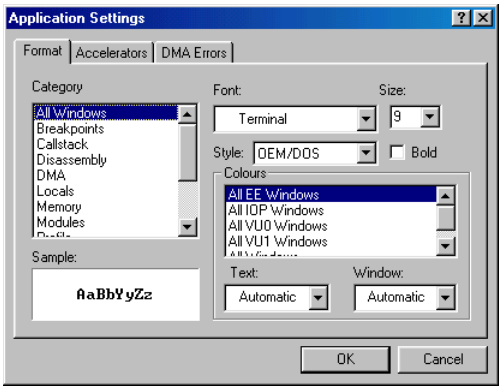
When the Debugger application exits it saves the current configuration in the working directory in a plain text file dbugps2.ps2. When the Debugger application starts up it tries to restore the current configuration from this file in the current working directory.

Pane colors and fonts

Pane colors and fonts are defined from the Application Settings dialog.

1. From the **Settings** menu option, select **Options**. Make sure that the **Format** tab is selected.

   The Application Settings dialog is displayed similar to the following:



Application Settings

Format | Accelerators | DMA Errors

Category: All Windows, Breakpoints, Callstack, Disassembly, DMA, Locals, Memory, Modules

Font: Terminal

Size: 9

Style: OEM/DOS

Bold

Colours: All EE Windows, All IOP Windows, All VU0 Windows, All VU1 Windows

Sample: AaBbYyZz

Text: Automatic

Window: Automatic

OK | Cancel

2. The **Category** list contains all of the available pane types. When you select a different pane type the sub-category list updates to show the CPU types available for that pane. However, for the

Source category the sub-category list displays some additional entries that allow you to define syntax coloring (see "To set syntax coloring in source panes" on page 29), and for the **All Windows** category the sub-category list displays all the available CPUs plus **All Windows**. Choose **All windows** if you wish to define colors and fonts that apply to all windows of a certain type (e.g. all EE windows), or else select a particular pane type (e.g. **Memory** for memory panes) from the list.

3. The sub-category section on the right changes according to the Category selection, so for example if you select **Memory**, you will be allowed to set colors and fonts individually for EE, IOP, VU0 and VU1 memory panes.

4. For each pane type (e.g. EE Memory) you can individually set the font face (**Font**) and point size (**Size**). You can also check the **Bold** checkbox if you would like to set the font to be displayed in bold face.

5. According to the choice of **Font**, you may be able to select a different character set from the **Style** list box.

6. Finally, you can choose the color of the text from the **Text** drop-down list box and of the window background from the **Window** drop-down list box. The **Automatic** option displays the text and background in the system colors.

7. Press **OK** to save your choice of pane colors and fonts. Settings will take effect for new panes. Your format settings will be saved in the configuration file ps2dbg.ini that is located in the same directory as the ps2dbg.exe executable.

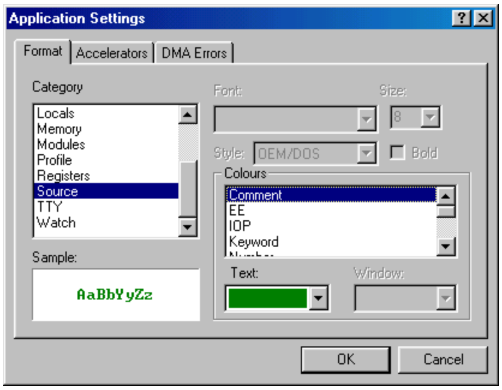
To set syntax coloring in source panes

Syntax coloring in the source pane allows you to display different syntactical components of your source in different colors. Source files written in C, C++ and VSM code all support syntax coloring.

The **Source** category allows you to change the font and color settings for the source pane, but it also includes options to allow you to define syntax coloring. The sub-category list contains entries for **Comment**, **Keyword**, **Number**, **Operator**, **String** and **User Defined Keywords**, which allow you to configure the colors used for these different elements in the source pane.

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Format** tab is selected.

2. Select the category **Source**. The Application Settings dialog should then look similar to the following:



3. In the sub-category section, you can now set the coloring individually for comments, keywords, numbers, operators, strings and user-defined keywords.

Note: <> markers used on a #include line are recognised as string identifiers.

Saving the project configuration

When the Debugger application exits it automatically saves the current configuration in the Debugger working directory in a plain text file dbugps2.ps2.

The working directory is the directory that you are in when you entered the ps2dbg command (or the directory specified in the **Start in** field in the Windows shortcut properties dialog).

When the Debugger application starts up it tries to restore the current configuration by locating the dbugps2.ps2 file in the following directories:

1. The current working directory.

2. If it cannot find a file in this directory, it will attempt to load a default configuration file from the directory in which the Debugger executable (ps2dbg.exe) is located.

3. If this file cannot be found then it will open with a built-in default configuration.

Note: If you run the Debugger from a Windows shortcut with no default working directory then your Debugger configuration files will be saved in different areas depending upon the current directory at the time and therefore your Debugger configuration may not persist as you might expect.

The Debugger configuration contains the following information:

- The size and position of the Debugger application pane on the desktop.

- The name of the last target that was connected to.
- The flags used in the Load ELF dialog.
- Any breakpoints that have been set in the source and disassembly (providing the -b command line option was not used on startup).
- The default font for new panes.
- All open Debugger windows and any set up information such as:

  - their size and location;

  - their type;

  - any extra display mode info (i.e. memory as bytes/words, bytes per line setting, the start address, cursor position, any watches that have been added to a watch window, name of the file loaded in source windows).
- The currently active Debugger window/pane.
- The source search path.

To save and reload your Debugger configuration

You can save the current Debugger configuration at any time using **Save Config** in the **File** menu. It is automatically saved in dbugps2.ps2 in the current working directory.

In addition you can also save a configuration as the default configuration that will be loaded if the dbugps2.ps2 file cannot be found in the current working directory. Use the **Save Config as Default** option in the **File** menu.

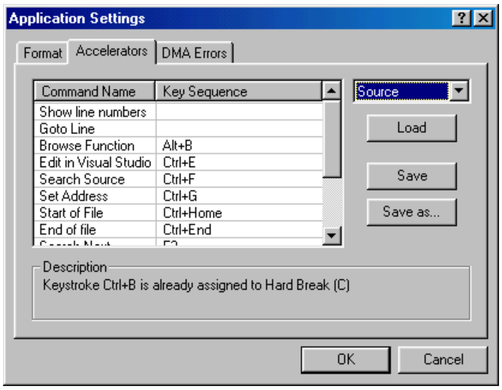
To reload the configuration file from the current working directory and overwrite your current configuration use **Load Config** in the **File** menu.

Accelerator keys

The Accelerators dialog box allows you to assign or change the keystrokes for any of the commands in the Debugger.

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.
2. In the list box at the top right, select the pane type for which you wish to set the shortcut menu accelerator keys, for example **Source** pane. There is also an **Application** pane type at the end of the list that allows you to set accelerator keys for ProDG Debugger application menu options. According to the pane type

chosen, the list of command names and key sequences will change in the main part of the dialog.



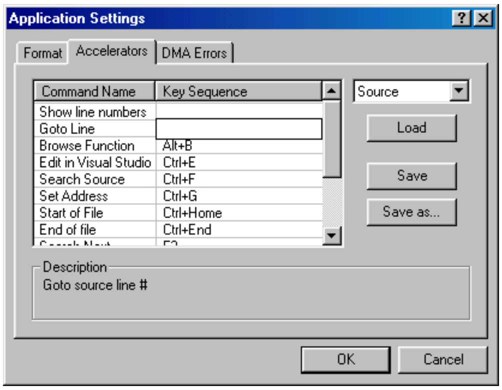
Command Name	Key Sequence
Show line numbers	
Goto Line	
Browse Function	Alt+B
Edit in Visual Studio	Ctrl+E
Search Source	Ctrl+F
Set Address	Ctrl+G
Start of File	Ctrl+Home
End of file	Ctrl+End

Description
Keystroke Ctrl+B is already assigned to Hard Break (C)

3. The **Description** field provides a slightly more detailed description for each command name, as each line is selected.

To set an accelerator key

1. Select the command name for which you wish to set an accelerator key, then click on the corresponding **Key Sequence** field so that an edit box is displayed:



Command Name	Key Sequence
Show line numbers	
Goto Line	
Browse Function	Alt+B
Edit in Visual Studio	Ctrl+E
Search Source	Ctrl+F
Set Address	Ctrl+G
Start of File	Ctrl+Home
End of file	Ctrl+End

Description
Goto source line #

2. Press the key combination you wish to be recorded as the accelerator key sequence for the chosen command name. The key sequence will be recorded in the edit box.

3. If the key sequence has already been defined for some other command in the same pane, then an error message will be displayed in the **Description** field: "<Key sequence> is already assigned to <Command Name>". You will then have to choose a different key sequence.

Note: Currently you cannot use the Accelerators dialog box to assign the following keys to an accelerator <Tab>, <Enter> and <Esc>. It is intended that this will be fixed in a future version.

To clear an accelerator key

Once an accelerator key has been set, it can be easily set back to <blank>:

1. Select the row containing the accelerator key to be cleared, so that both the command name and key sequence are highlighted.

2. Press the **Delete** key, or right-click and select the **Clear Accelerator** shortcut menu option.

Saving and loading accelerator key configurations

Accelerator keys are saved in Accelerator Key Mapping files (.akm files). The ps2dbg.ini file contains the full path and filename of the previously loaded .akm file. This file will then be reloaded the next time the Debugger is started.

The default keystrokes are stored in a default ps2dbg.akm file in the directory containing the ps2dbg.exe executable. However you can store your accelerator keystrokes to any filename.

1. Press **Save** to save the current accelerator keys to your default file ps2dbg.akm. You will be prompted to confirm or cancel the save.

2. Press **Save as** to save the current accelerator keys to another named .akm file. You will be presented with a browser to choose a directory and filename of your choice.

3. Press **Load** to cause a file browser to be displayed. You can then select a named .akm file to load accelerator keys from. The default file is called ps2dbg.akm.

A Visual Studio keymap is also provided as the file PS2VS.AKM. For a reference chart detailing the default SN Systems and Visual Studio keymaps, see "Keyboard shortcut reference" on page 33.

Keyboard shortcut reference

This section describes the shortcut keys for ProDG Debugger for PlayStation 2.

ProDG Debugger for PlayStation 2 shortcut keys

This section describes the default accelerator keys for ProDG Debugger. Accelerator key settings are saved in a keymap file.

Two keymap options are available in the Debugger. You can switch between them using the options in the **Settings** menu (see "Saving and loading accelerator key configurations" on page 33).

The following table shows the keyboard shortcuts that are available in each of the two keymaps provided (SN and Visual Studio).

SN keymap	**Visual Studio keymap**	**Description**	**Context**
Shift+,	Shift+,	Save binary file.	Any
Shift+.	Shift+.	Load binary file.	Any
Ctrl+a	Ctrl+a	Set Hardware Breakpoint dialog (assembly).	Any
Ctrl+b	Ctrl+b	Set Hardware Breakpoint dialog (C variable).	Any
Alt+b	Alt+b	C/C++ variable or function browser.	Source, disassembly and watch panes
Ctrl+d	Ctrl+d	Set Hardware Breakpoint dialog (DMA channel).	Any
Ctrl+e	Ctrl+e	Edits the current source line in Microsoft Visual Studio (providing it is already running).	Source pane
Alt+e	Alt+e	Load ELF file dialog.	Any
Ctrl+f	Ctrl+f	Enter search string dialog	Source pane
Ctrl+g	Ctrl+g	Enter address dialog.	Memory, source or disassembly panes
Ctrl+g	Ctrl+g	Set DMA disassembly start address	DMA pane
Alt+i	Alt+i	Load IOP module dialog.	Any
Ctrl+l	Ctrl+l	Lock to address.	Disassembly pane
Ctrl+n	Ctrl+n	Next DMA error	DMA pane
Ctrl+w	Ctrl+w	Toggle word size	Memory pane
Tab	Tab	Go to the line that the program counter is positioned on.	Source or disassembly panes
Shift+tab	Shift+tab	Set PC to current cursor position.	Source or disassembly panes
Ctrl+tab	Ctrl+tab	Cycle to next window.	Any
Return	Return	Enter new memory value.	Memory pane
Insert	Insert	Add a new watch.	Watch pane

Insert	Insert	Add a breakpoint.	Breakpoints pane
Delete	Delete	Remove selected watch.	Watch pane
Delete	Delete	Remove selected breakpoint.	Breakpoints pane
Ctrl+Home	Ctrl+Home	Position cursor on first line of source	Source pane
Ctrl+End	Ctrl+End	Position cursor on last line of source	Source pane
Left cursor arrow	Left cursor arrow	Decrement expanded array variable index to view array members.	Local or watch panes
Right cursor arrow	Right cursor arrow	Increment expanded array variable index to view array member.	Local or watch panes
Shift+numpad*	Shift+numpad*	Update all panes	Any
numpad +	numpad +	Expand local or watched variable.	Local or watch panes
numpad +	numpad +	Increment selected memory.	Memory pane
numpad +	numpad +	Show more detail	DMA pane
numpad -	numpad -	Collapse local or watched variable.	Local and watch panes
numpad -	numpad -	Decrement selected memory.	Memory pane
numpad -	numpad -	Show less detail	DMA pane
Esc	Esc	Stop your application running on the target.	Your application is running on the target.
Ctrl+Shift+1	Ctrl+Shift+1	Brings up a menu of valid pane types to change the current pane to. Can use the arrow keys to navigate up and down the menu.	Any
Alt+ arrow keys	Alt+ arrow keys	Navigate across panes in a split pane view in the direction of the arrow key.	Any pane in a split pane view
Alt+F2	Ctrl+Shift+F5	Reset the target and reload the application.	Any
Shift+ arrow keys	Shift+ arrow keys	Push pane splitter bar outwards (in direction of arrow key).	Any
Alt+Shift+ arrow keys	Alt+Shift+ arrow keys	Pull pane splitter bar inwards (in direction of arrow key).	Any

Ctrl+arrow keys	Ctrl+arrow keys	Enables you to specify a split in the current pane, using the arrow keys to specify how to split the current pane – left and right arrows will split the current pane vertically, and up and down horizontally.	Any
Ctrl+Shift+ arrow keys	Ctrl+Shift+ arrow keys	Enables you to specify that a split in a pane is deleted. The deletion will attempt to remove the splitter bar in the direction of the arrow.	Any
F2	F2	Next bookmark	Source pane
Shift+F2	Shift+F2	Previous bookmark	Source pane
F3	F3	Find search string again.	Source pane
Shift+F4	Shift+F4	Tile all visible panes in the main window.	Any
F5	F9	Toggle breakpoint on current line (D bit breakpoint at cursor for DMA pane)	Source. disassembly or DMA panes
Shift+F5	Shift+F5	Cascade all visible panes in the main window.	Any
F6	Ctrl+F10	Run to cursor.	Source or disassembly panes
F7	F11	Step Into – single step through your application stepping into any functions called.	Source or disassembly panes
F8	F10	Step Over – single step through your application stepping over any functions called.	Source or disassembly panes
Shift+F8	Shift+F11	Step Out – step out of a called function that you are stepping through.	Source or disassembly panes
F9	F5	Start your application running on the target.	Loaded an application on the target
Shift+F9	Shift+F9	Quick watch - add a watch to the topmost watch pane	Any
Ctrl+P	Ctrl+P	Previous Files menu	Source pane
Ctrl+F	Ctrl+F	Find text	Source pane

F3	F3	Find next	Source pane
Ctrl+Return	Ctrl+Return	Fill memory	Memory pane
Ctrl+Insert, Ctrl+C	Ctrl+Insert, Ctrl+C	Copy to clipboard	Any
Shift+Insert, Ctrl+V	Shift+Insert, Ctrl+V	Paste	Any

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 3: Building and loading your game

Building for debugging

You must follow certain steps if you wish to use your program with the ProDG Debugger:

1. Remove compiler optimizations, i.e. delete the -Ox entry from the CFLAGS= and/or CXXFLAGS= variables in your makefile.
2. Set the debug information flag. Add the -g switch to the CFLAGS= and/or CXXFLAGS= variables in your makefile.
3. If you wish to do profiling or VU debugging, you will need to include libsn.a, preferably as the first included library in your LIBS= variable.

Connecting to targets

Using the command-line options you can start the Debugger with an ELF file automatically loaded (just its symbols, or symbols + executable with an optional target reset first).

If you start the Debugger with no command-line parameters you can still access the Debugger start-up functionality, for example an ELF file can be loaded via the toolbar and menus.

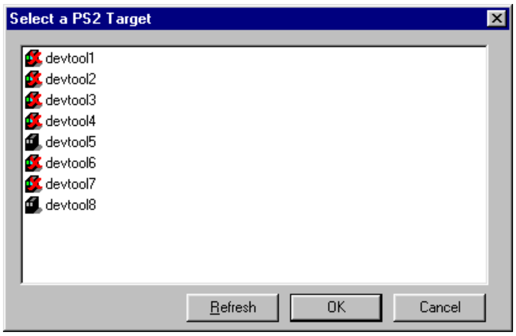
When you start the Debugger the Target Manager is automatically started, and a dialog appears in which any targets that you have set up are listed (depending on your Debugger command line options). You can also change the target that you are working on while using the Debugger using the **Select Target PS2** option from the **Debug** menu.

If you haven't yet set up your PlayStation 2 targets in the Target Manager you will need to start it separately and then set up the targets.

To connect to a target

You can change the target PlayStation 2 that the Debugger is connected to during an existing Debugger session. Once you have successfully connected to a new target you will need to load your ELF file (see "Loading and running ELF files" on page 41).

1. Click **Select Target PS2** in the **Debug** menu, and the following dialog is displayed:



   This is the same dialog that is displayed when you start the Debugger the first time, or with the –t option and no target name. It shows a list of the target sessions that you have set up in the Target Manager.

   Targets that are in use are shown with a red cross, whereas targets that are available are shown in black. Also, targets to which you are already connected have the LEDs highlighted on the target icon.

2. To update the display to reflect the latest connections and disconnections, click **Refresh**.

3. Select an available target from this list and click **OK**.

   The Debugger immediately reconnects to the new target (if possible). You will need to load the required ELF file (**Debug > Reset and Restart**). If the Debugger cannot successfully connect to the new target then a dialog appears telling you that reconnection failed and that you are still connected to the current target in the Debugger.

4. Once you have selected a target session, the next time the Debugger is started it will automatically connect to the target that you were working on when you quit the Debugger.

Note: When you change the PlayStation 2 target that you are connected to you may still remain connected to the previous target. You can see the connection status of your targets in the Target Manager, and disconnect any sessions that are no longer required.

Multiple users debugging on the PlayStation 2

Only one user can connect to the PlayStation 2 at any one time for debugging purposes. If someone else is connected to the PlayStation 2 that the Debugger is trying to connect to, a dialog showing possible target sessions is displayed allowing you to select another.

In ProDG Target Manager you should be able to view the identity of the user who is currently connected to a target. If you wish to continue using the same target PlayStation 2 you will need to negotiate directly with the other user.

Loading and running ELF files

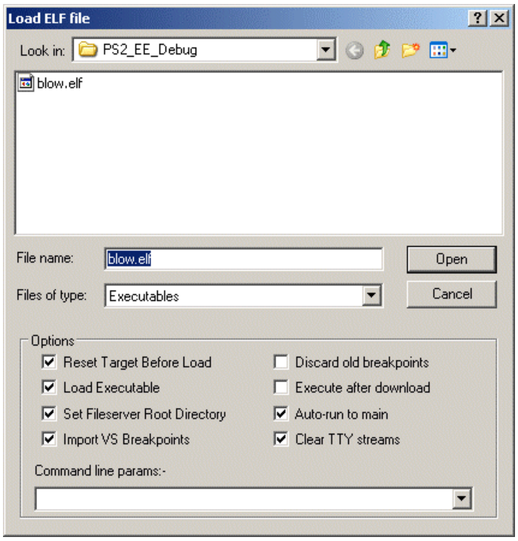
You can load and run ELF files on the target directly from the Debugger. You may be doing this after starting the Debugger if you did not specify an ELF file on the command line, or after having switched targets or just to load a newly built application.

Note: You can also load an ELF file using the Target Manager or using the ps2run command line utility.

To load your application ELF file manually

The Debugger must have been started.

1. Click **Load ELF** file in the **File** menu.
2. In the dialog that is displayed locate and select the ELF file that you wish to load and debug.



3. Set any of the options that you require for the file load:

Reset Target Before Load	Resets the target before your ELF file is loaded.
Load Executable	Indicates that you wish the application executable contained in the ELF file to be loaded as well as the symbol information.

Set Fileserver Root Directory	Sets the application file serving directory to the ELF file directory.
Import VS breakpoints	Specifies that any breakpoints that have been set in your source in Microsoft Visual Studio are imported into the Debugger when the ELF file is reloaded on the target.
Discard old breakpoints	Specifies that any breakpoints previously set in the program by the ProDG Debugger, will be discarded.
Execute after download	Specifies that the ELF file will be run on the target once it is loaded. This will override the default Debugger start-up behavior, which runs to the main routine.
Auto-run to main	Specifies that the ELF file will be run to the main routine and stopped, once it is loaded on the target.
Clear TTY streams	Empties all TTY streams before loading ELF file.
Command line params	Enables you to specify any command-line parameters for your PlayStation 2 application.

4. Click **Open** to activate the load.

   The file is loaded and "run to main" or executed on the target, depending on the option that you selected. You can now start debugging the newly loaded application ELF file.

Running your game on the PlayStation 2

Once you have started the Debugger and downloaded your ELF file, you can start it running on the target PlayStation 2. You can start and stop your application running at any time using the commands provided.

You may also wish to set some breakpoints in your source or disassembly, start the PlayStation 2 running again, or single-step through your application. The following sections describe how to just start and stop your application running using the target control commands in the **Debug** menu. However for information on setting breakpoints and stepping through your application see "Using breakpoints" on page 49.

To restart the target

You can restart the application at any time. This means that the target PlayStation 2 is reset and the ELF file is loaded again, and your application runs to main again.

- Click **Reset and Reload** in the **Debug** menu or on the toolbar.

This is useful if you change the PlayStation 2 target, or just need to reset the target to its initial load state.

To start your application running on the PlayStation 2

When the application has stopped running on the target (e.g., because it has reached a breakpoint, or stopped at the main subroutine, etc.), you can start it running again from the Debugger.

To restart the application from the program counter:

- Click **Go** in the **Debug** menu or Start the target on the toolbar.

Your application should start running on the PlayStation 2 and you will notice that the registers pane displays 'Running' to indicate this.

To stop your application running on the PlayStation 2

If your application has not already stopped on a breakpoint or exception, then you can stop it manually at any time:

- Click **Stop** in the **Debug** menu or Stop the target on the toolbar.

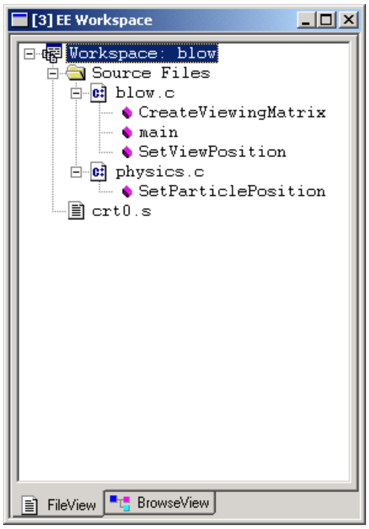
  You will notice that once your application has stopped running on the target, any other open panes will be updated to show the current information at the new program counter position.

Using the **Go to PC** commands in the source or disassembly panes you can view the current position of the program counter. Alternatively if you leave either the source or disassembly pane as the active pane, then it will automatically update to show the current program counter position when the target stops.

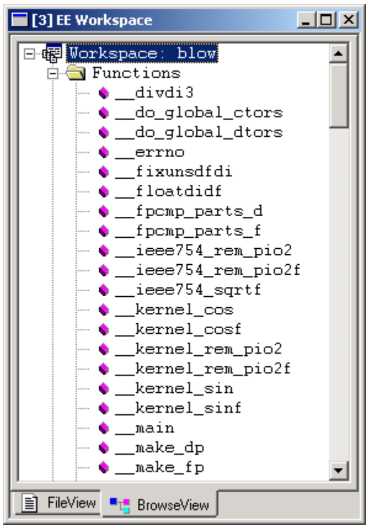
Note: The source pane will only show the program counter if it can successfully be mapped to the original source code.

Using the workspace pane

The workspace pane allows files, functions, global variables and class definitions of an ELF project to be displayed in a tree view. The workspace pane has two views, FileView and BrowseView, which are selectable from the tabs at the bottom of the pane.



Workspace pane with FileView tab selected



Workspace pane with BrowseView tab selected

FileView	This displays a tree view, containing the name of the project and a list of all files in the ELF project that contain executable code. The root of this tree shows a **Workspace:** folder that displays the name of the ELF project (e.g. "blow"); all project files are displayed within this folder. C++ and header files are split into separate sub-folders while files with other extensions are placed in the root of the Workspace folder.
BrowseView	This is used to browse through all class types, global functions and global variables within the project. Each class contains sub-items for

methods and static members. Browse view contains three folders named **Classes**, **Functions** and **Variables**, which contain all class types, global functions and global variables respectively.

Folders can be expanded by clicking on the [+] symbol, if you need to look at the folder contents; for example, to display a list of all functions in the ELF. The class folder, when expanded, displays all class definitions within the project; these may in turn be expanded to show a list of all methods and static member variables.

Folders can be contracted by clicking on the [-] symbol.

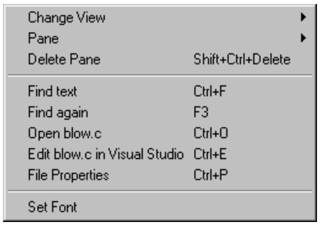
Workspace pane shortcut menus

Right-clicking on a **Workspace:** folder displays a shortcut menu containing a Workspace Properties option besides the standard pane management items. Right-clicking on any other folder displays a shortcut menu containing just the pane management items.

Right-clicking on any of the expanded folder list items displays a more detailed shortcut menu. In addition to the standard pane management menu items, there are also ones to carry out operations specific to the currently selected item.

Note: The group of menu options relating to files will not be shown if you right-click on the window background.

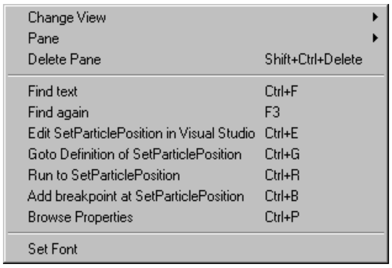
For file-type items, the shortcut menu contains operations for opening the file in a new source pane, or opening the file in Visual Studio. For BrowseView items, the shortcut menu changes according to the type of item.



FileView shortcut menu - file selected

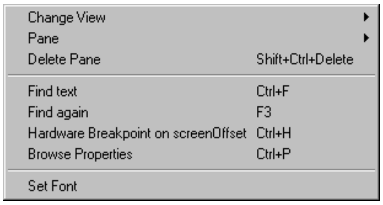
Find text	Opens a dialog that enables you to search for an item (file, class, function or variable) containing the specified text. If the text is found, the line containing the item becomes the current selection.
Find again	Repeats the previous **Find text** search command.
Open <file>	Opens the selected file in the last selected source pane. If a source pane is not available,

	then another pane may be changed to show the source.
Edit <file> in Visual Studio	Enables you to edit the selected file in Visual Studio.
File Properties	Displays a message window with information on properties of the selected file.


Change View
Pane
Delete Pane Shift+Ctrl+Delete
Find text Ctrl+F
Find again F3
Edit SetParticlePosition in Visual Studio Ctrl+E
Goto Definition of SetParticlePosition Ctrl+G
Run to SetParticlePosition Ctrl+R
Add breakpoint at SetParticlePosition Ctrl+B
Browse Properties Ctrl+P
Set Font

BrowseView shortcut menu - function selected

Find text	Opens a dialog that enables you to search for an item (file, class, function or variable) containing the specified text. If the text is found, the line containing the item becomes the current selection.
Find again	Repeats the previous **Find text** search command.
Edit <function> in Visual Studio	Works in the same way as **Goto Definition of <function>**, but loads the source file into Visual Studio instead.
GoTo Definition of <function>	Opens a new source pane and loads the source file containing the function and places the cursor at the start of the function, scrolling down the file if necessary.
Run to <function>	Starts executing the program, stopping inside the debugger when the program counter reaches the start of the function.
Add / Remove breakpoint at <function>	Adds a breakpoint at the selected function. If a breakpoint is set on the function then the menu text changes from **Add breakpoint at <function>** to **Remove breakpoint at <function>**, so that the breakpoint can be removed.
Browse Properties	Displays a message window with information on properties of the selected function.



BrowseView shortcut menu - variable selected

Find text	Opens a dialog that enables you to search for an item (file, class, function or variable) containing the specified text. If the text is found, the line containing the item becomes the current selection.
Find again	Repeats the previous **Find text** search command.
Hardware Breakpoint on <variable>	Displays the Hardware Breakpoint dialog box and fills in the **VarName** field with the name **<variable>**.
Browse Properties	Displays a message window with information on properties of the selected variable.

All of these context menu items have user-configurable keyboard accelerators, which can be set from the **Accelerators** tab in the **Settings** menu.

The File Properties and Browse Properties dialogs are toggled windows. That is, once the dialog has been opened, the context menu item will be ticked and the dialog will stay on the screen until either the context menu is selected again or the close button on the dialog is clicked. However, while the dialog is displayed the debugger continues to function normally and menus/toolbars are still accessible. When the selection changes in the workspace pane, the properties dialog will be updated to show the new selection.

In addition to the context menu, you can double-click on an item with the left mouse button. This will attempt to "Goto" the selected file or definition.

The keyboard may also be used to navigate around the workspace pane (up + down to change the items, left/right to collapse and expand, <Enter> to simulate a double-click).

Leaving the mouse over a Browse item in the workspace pane for a short time will attempt to evaluate the item. This will display the address and/or current value.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 4: Breakpoints and stepping

Using breakpoints

The key to tracing bugs in your code is to maintain fine control over its execution on the target. This section describes how to set breakpoints in your code, and the different ways to step through your code.

- You can at will start or stop your program running on the target (see "Running your game on the PlayStation 2" on page 42.
- You can issue a "break 1" assembler instruction if you want to be able to control the placing of breaks at compile time (see "Setting breakpoints at compile time" on page 49).
- By setting a breakpoint in the Debugger, either in a source or disassembly view of the program, you can interactively stop the program running at any point in its execution path.
- If you have manually halted your application on the target or if it has stopped for another reason (breakpoint, etc.) you can then step through the execution path one line of code at a time.
- You can set a memory-access or hardware breakpoint, which triggers every time an address (or address range) is accessed (see "Memory access or hardware breakpoints" on page 51).
- You can set a conditional breakpoint, which triggers whenever a condition is true (see "Conditional breakpoints" on page 55).

Setting breakpoints at compile time

You can embed illegal or breakpoint opcodes in your source code to cause the program to halt with a CPU exception that will be clearly visible in the debugger. With the program halted in this way you can use the debugger to examine the active scope at the time of the exception (i.e. callstack, local and global variables, memory etc).

If you try to continue execution of your program from this point you will first need to advance the PC past the breakpoint opcode otherwise your program will just immediately halt again. This is such a common requirement that rather than a general illegal instruction, we have reserved a specific "break" opcode for this purpose which automates this process. If your program is halted at a "break 1" opcode and you then hit **Run** in the debugger, then the debugger will automatically advance the PC past the break opcode and resume execution. You can embed a "break 1" in your C source code using a statement like:

```
asm("break 1");
```

Setting and viewing breakpoints

A breakpoint indicates that you want the execution of your program to stop when the program counter reaches the specified line of C, C++ or assembly code.

By default the position of breakpoints in your application code is saved between sessions in the dbugps2.ps2 configuration file (providing you start the Debugger with the same ELF file and that it is specified on the Debugger command line). If you do not wish breakpoints to be saved between subsequent sessions you must use the –b command line option (see "ps2dbg command-line syntax" on page 6).

In addition you can specify that Visual Studio breakpoints are imported or exported using the –vs option on the Debugger command line.

To set a breakpoint in source or disassembly

1. Ensure that the target is stopped and open a source or disassembly pane.
2. Ensure that you are viewing the source or disassembly on the required PlayStation 2 target unit (via the shortcut menu).
3. Scroll to the line of source or disassembly that you wish to set the breakpoint on.
4. Either double-click the line or click **Breakpoint** in the shortcut menu. If there is already a breakpoint there it will be toggled off. Otherwise the new breakpoint will be set on this line and indicated by the line being set in a different color.

Note: If you try to set a breakpoint on a line that the program counter cannot halt on (e.g., a comment), then the breakpoint will be set on the next valid line of code after the selected line. In addition if you cannot correctly set a breakpoint, or the application stops at the breakpoint in the wrong part of the program, it could be that optimization may have been used when compiling the source code. To get around this problem you will need to rebuild the application and remove the -Ox flag from the compiler arguments (in the makefile).

Now when you start the target, it will stop when the program counter reaches a breakpointed line, and any debug information panes will be updated. In addition if a disassembly or source pane is open and active it will be updated with the current program counter position indicated.

Stepping through your program

You can single-step through your program executing on the PlayStation 2 target. This can either be done in disassembly or source (if you can view the source at the current program counter).

To single-step through your program

1. Ensure that the target is stopped and open a source or disassembly pane.

2. Use the **Step** or **Step Over** commands in the pane shortcut menu to single-step your source or disassembly (depending on which pane you choose). The **Step** command will step in to any function calls and step through the lines of code in the called function.

Note: You are also able to use the Step and Step Over commands in the Debug menu. If the active pane is a disassembly pane then you will step through disassembly. Otherwise these commands will attempt to step through your application source (if there is any source information). If this is not available then they will step through the disassembly.

There are other **Run** possibilities in the shortcut menus of the disassembly and source panes. For more information on these see "Disassembly pane" on page 72, and "Source pane" on page 61. For example you can run to the current cursor position using the **Run to Cursor** command.

To step out of the current function

If you are currently single-stepping through a function you can return to the line of source that called the function without having to step through every remaining line of code in the function.

- Click Step Out in the Debug menu or Step out of current function on the toolbar.

The program counter will advance to the line of code following the function call.

However, if there are any breakpoints in the code up to the end of the function (or in any functions called before then), the pane may switch to halt on the next breakpointed line.

Memory access or hardware breakpoints

It is possible to set a single hardware breakpoint for the PlayStation 2 EE CPU. A hardware breakpoint enables you to specify that program execution is halted whenever a specified address or address range is accessed.

When a hardware breakpoint is triggered it behaves in a similar way to a software breakpoint, i.e. program execution will halt. A message box will also appear with a short description. You can then restart execution with any of the target control commands: **Step**, **Run to** or **Go**.

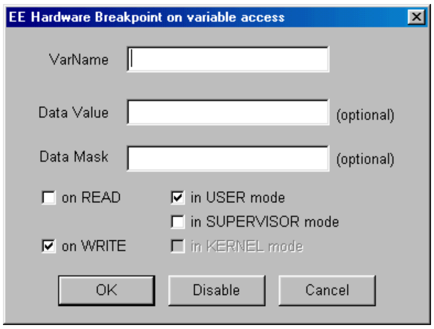
With the PlayStation 2 you can set a hardware breakpoint to trigger when a particular C variable is accessed, when an assembly address or address range is accessed, or when a DMA channel is started. You need to use the appropriate command **EE Hardware Break (C var)**, **EE Hardware Break (Asm level)** or **EE DMA Hardware Break** to set your hardware breakpoint. The difference between the **C var** and **Asm level** hardware breaks is that in C mode the Debugger does the thinking for you but the result is not quite as flexible. Therefore the assembly mode has been added for users who wish to have more control over the hardware breakpoint.

You can further refine when a hardware break is triggered, by specifying a mask or data value that must be present to cause the break to occur (rather than it being triggered each time the variable or address is read from or written to).

To set a C variable hardware breakpoint

You can either set a hardware breakpoint that is triggered when a C variable is accessed (C mode), or one that is triggered when a particular address or address range is written to (assembly mode).

1. Click **EE Hardware Break (C var)** in the **Debug** menu. The following dialog is displayed:


EE Hardware Breakpoint on variable access

VarName

Data Value (optional)

Data Mask (optional)

☐ on READ ☑ in USER mode

☐ in SUPERVISOR mode

☑ on WRITE ☐ in KERNEL mode

Hardware breakpoint set up dialog for C mode

2. Enter the name of a variable in your application C code in the **VarName** field.

3. You can specify that the hardware breakpoint is triggered whenever a particular value is read or written to the variable by entering the required value directly in the **Data Value** field. In addition you can enter a mask in the **Data Mask** field that is applied to the data value (if you entered one) that enables you to specify the part of the data value that will be compared with the hardware breakpoint variable or data. If you do not enter a data mask it is set to 0xFFFFFFFF meaning that the complete data value will be used.

4. Select the **on READ** or **on WRITE** options as necessary.

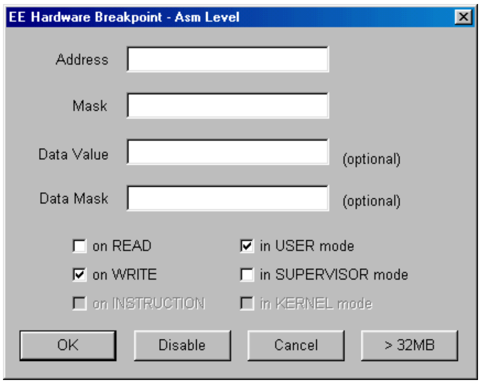
5. Select the mode that your application code will be running in, USER and/or SUPERVISOR.

6. Click **OK** to set the hardware breakpoint and enable it.

   Now when you run your program on the target PlayStation 2 execution will halt when the specified variable is accessed and satisfies all the criteria that you have set up in the breakpoint.

To set an assembly level hardware breakpoint

1. Click **EE Hardware Break (Asm level)** in the **Debug** menu. The following dialog is displayed:



   Hardware breakpoint set up dialog for assembly mode

2. Enter either just an absolute address, or the address bits to be checked in the **Address** field and a mask that is applied to the address bits in the **Mask** field. If you enter a mask, set the bits you wish to be taken into account to 1 and bits that you wish to mask to 0. For example, you could enter 0xFF to indicate that the breakpoint should be triggered by any access to the range of memory indicated by the bottom eight bits of the address value. If you do not enter a mask then it defaults to 0xFFFFFFFF meaning that the complete address bus will be checked.

3. There is an additional preset **>32MB** button that automatically enters the values in **Address** and **Mask** fields so that the hardware breakpoint will be triggered whenever memory access above 32MB is detected. The address value is set to 32 MB (in hex 0x02000000) and the mask is set to have 0s in the bottom bits and 1s for all the bits above and including the lowest bit set in the above number (i.e. 0xFE000000). That will cause any address with any of the top 7 bits set to generate the hardware break.

4. You can specify that the hardware breakpoint is triggered whenever a particular value is read or written to the variable or address by entering the required value directly in the **Data Value**

field. In addition you can enter a mask in the **Data Mask** field that is applied to the data value (if you entered one) that enables you to specify the part of the data value that will be compared with the hardware breakpoint variable or data. If you do not enter a data mask it is set to 0xFFFFFFFF meaning that the complete data value will be used.

5. Select the **on READ** or **on WRITE** options as necessary.

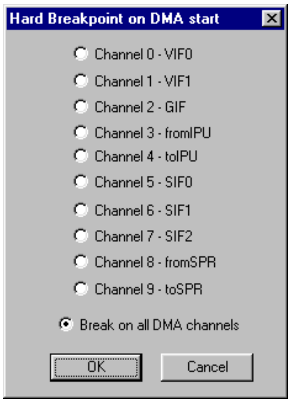
6. Select the mode that your application code will be running in, USER and/or SUPERVISOR.

7. Click **OK** to set the hardware breakpoint and enable it.

   Now when you run your program on the target PlayStation 2 execution will halt when the specified address is accessed and satisfies all the criteria that you have set up in the breakpoint.

To set a DMA channel hardware breakpoint

1. Click **EE DMA Hardware Break** in the **Debug** menu. The Hard Breakpoint on DMA start dialog is displayed:



Hard Breakpoint on DMA start

- Channel 0 - VIF0
- Channel 1 - VIF1
- Channel 2 - GIF
- Channel 3 - fromIPU
- Channel 4 - toIPU
- Channel 5 - SIF0
- Channel 6 - SIF1
- Channel 7 - SIF2
- Channel 8 - fromSPR
- Channel 9 - toSPR
- [✓ Break on all DMA channels]

OK Cancel

Hardware breakpoint set up dialog for DMA mode

2. Select the DMA **Channel** you want to set a breakpoint on, or select the **Break on all DMA channels** option (this option will be selected by default).

   Now when you run your program on the target PlayStation 2 execution will halt when the specified DMA channel(s) is/are accessed.

To disable a C variable or assembly level hardware breakpoint

Once you have set up a C variable (see "To set a C variable hardware breakpoint" on page 52) or assembly level (see "To set an assembly level hardware breakpoint" on page 53) hardware breakpoint you can disable it at any time. You should also make sure that you disable a hardware breakpoint before closing the Debugger as this might affect the next debug session that takes place on the target.

1. Open the required dialog to disable a hardware breakpoint that you have set up on a C variable or assembly level address, using the **EE Hardware Break (C var)** or **EE Hardware Break (Asm level)** commands in the **Debug** menu.

2. The dialog shows details of the hardware breakpoint that you have previously set up. To disable it, click on the **Disable** button.

3. The dialog is closed, and the hardware breakpoint is effectively disabled on the target. Should you wish to enable it again you can open the appropriate dialog and click the **OK** button again.

To disable a DMA hardware breakpoint

A DMA hardware breakpoint can only be disabled by performing a program reset.

- Click **Reset and Restart** in the **Debug** menu (or use the toolbar button).

  The program counter will be reset and the DMA hardware breakpoint will be disabled.

Conditional breakpoints

A conditional breakpoint is very similar to any other software breakpoint, i.e. when triggered it will stop the target. However conditional breakpoints have the advantage that the user can choose when and if the breakpoint stops the target based on a *condition* or *hit count*.

The *condition* can be any expression that can be qualified by the expression evaluator. Once set, the breakpoint will only fire when the condition evaluates to TRUE. If the condition evaluates to FALSE, then the target continues execution.

The *hit count* is another way of selecting when a breakpoint fires. Firstly you set the hit count to a number, which can then be checked against one of three possible scenarios:

- The breakpoint has been hit the same number of times as the hit count.

- The breakpoint has been hit a number of times which is a multiple of the hit count.

- The breakpoint has been hit an equal or greater number of times as the hit count.

The hit count and condition are not mutually exclusive, so you can set both a condition and hit count on any breakpoint. In this case both the hit count and the condition must evaluate to TRUE for the break to occur.

To set a conditional breakpoint

1. Set a normal software breakpoint. See "Setting and viewing breakpoints" on page 50 for details.
2. Open a breakpoints pane and select the breakpoint. See "Breakpoints pane" on page 56.
3. Right-click on the breakpointed line to display the shortcut menu.
4. Select the option **Breakpoint properties**, and using the Breakpoint properties dialog, set the condition and/or hit count for the breakpoint.

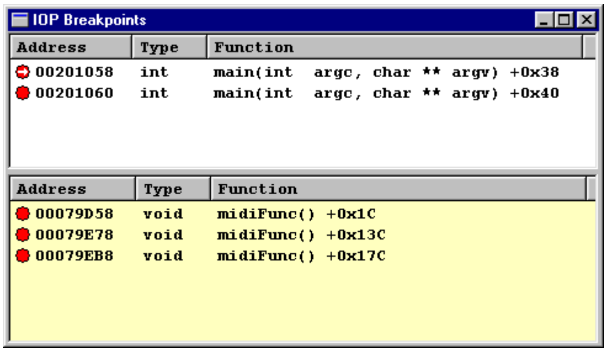
   The breakpoint will now be set as a conditional breakpoint.

Breakpoints pane

At any time all the breakpoints that have been set in your application source or disassembly can be viewed in the breakpoints pane. This lists the address at which the breakpoint has been set and the function that it is set in. You can add other breakpoints or delete the selected breakpoint, or all breakpoints in this pane.

When you first open a breakpoints pane it is empty, but is immediately updated as soon as breakpoints are set or unset. An arrow is used to show which breakpoint the program is currently stopped at.

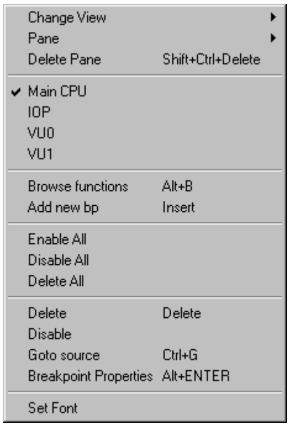
The following screenshot shows a pane split to show both EE (above) and IOP (below) breakpoints side-by-side. The PC is currently stopped at the first EE breakpoint, as indicated by the white arrow on the left.


IOP Breakpoints

Address	Type	Function
00201058	int	main(int argc, char ** argv) +0x38
00201060	int	main(int argc, char ** argv) +0x40

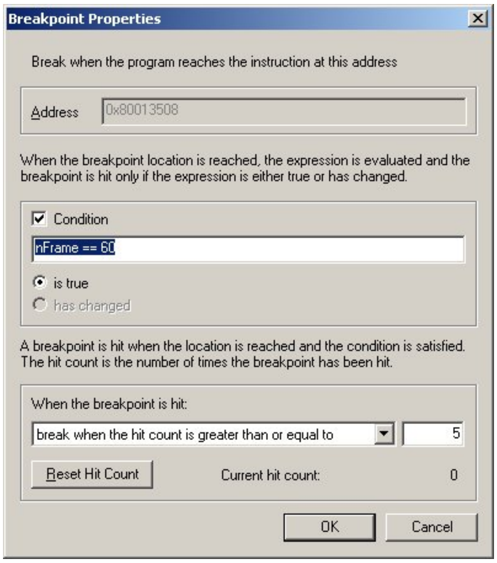
Address	Type	Function
00079D58	void	midiFunc() +0x1C
00079E78	void	midiFunc() +0x13C
00079EB8	void	midiFunc() +0x17C

Breakpoints pane shortcut menu



Main CPU	Changes to show breakpoints on the main CPU. This is reflected in the window title. The background color of the breakpoints pane is reset to white.
IOP	Changes to show breakpoints on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show breakpoints on the PlayStation 2 VU0 unit. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show breakpoints on the PlayStation 2 VU1 unit. The name of the unit being viewed is displayed in the window title.
Browse functions	Causes a Function Browser dialog to be displayed, in which you can select from dropdown list boxes contain alphabetic lists of classes, methods, overloaded names and addresses, for either clearing or setting breakpoints.
Add new bp	Causes the Enter Address dialog to be displayed so that you can specify an address on which a breakpoint is to be set.
Enable All	Enables all breakpoints.
Disable All	Disables all breakpoints.
Delete All	Causes all breakpoints to be removed for the selected processor.
Delete	Causes the selected breakpoint to be deleted.
Disable	Causes the selected breakpoint to be disabled.
Goto source	Goes to that breakpoint in a source/disassembly pane.
Breakpoint Properties	Displays the Breakpoint Properties dialog. See "Breakpoint Properties dialog" on page 58.

Breakpoint Properties dialog


Breakpoint Properties

Break when the program reaches the instruction at this address

Address: 0x80013508

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is either true or has changed.

☑ Condition

nFrame == 60

● is true
○ has changed

A breakpoint is hit when the location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit:

break when the hit count is greater than or equal to 5

Reset Hit Count   Current hit count: 0

OK   Cancel

This properties dialog appears when you select **Breakpoint Properties** from the breakpoint pane's shortcut menu. It enables you to enter a conditional expression, set a hit count condition or reset the current hit count to zero.

Address	Displays the breakpoint address.
Condition	The checkbox should be checked if setting a condition. In the expression box, enter the expression to be evaluated when the breakpoint is reached.
When the breakpoint is hit	If setting a hit count, enter the type of hit count evaluation and hit count target value (the hit count that will be compared against in order to trigger the breakpoint).
Reset Hit Count	Sets the current hit count back to zero.
Current hit count	Displays the current hit count.

Troubleshooting

DLL will not load

Sometimes a DLL will not load when debugging. This occurs when stepping or when particular break points are set. The call to snDllLoaded() returns an SN_DLL_INVALID_DEFINE_GLOBAL_FAILED error or similar. (See libsn.h for more detail.)

This occurs because the DLL loader cannot patch in the DLL if one or more of the locations it needs to patch have a breakpoint set on them. The problem also occurs when stepping through the code if code following the call to snDllLoaded() calls a function in the DLL.

To fix this problem, remove breakpoints on calls to functions declared in DLLs. You may wish to set a breakpoint just before the call and then step into the function.

If you experience problems stepping through the code add a call to:

```
asm("nop");
```

after the call to snDllLoaded(). You may wish to include this nop() conditionally in debug builds only.

Can I set a hardware breakpoint on an arbitrary block of memory?

When you use the assembly level hardware breakpoint dialog (see "To set an assembly level hardware breakpoint" on page 53) the values you enter are passed directly to the relevant COP0 registers. We provide the raw ASM style dialog interface to this feature.

The biggest restriction is that the hardware break can only detect a base address using a mask of "don't care" bits and a comparison address, i.e. you cannot set an arbitrary start address and length. By setting don't-care bits from the LSB upwards you can detect any access to a block which is a power of 2 long but only if it is on a multiple of its size (that same power of 2), i.e. you can spot access to a 4 byte long object only if it is aligned at a multiple of 4 bytes. You can spot access to a 1024-byte block only if it is aligned starting at a multiple of 1K, and so on. This is a limitation of the hardware.

If you want to detect access to an arbitrary range you have to set the nearest limits that include that range but then you will also trigger for access within the whole power-of-2-sized block.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 5: Viewing source and disassembly

Viewing source

Whenever the target stops running, if a source pane is open it updates to show the current position of the program counter in your application source (indicated by the ⇨ character). However in some instances the program counter cannot be viewed in source, in which case the source pane will not be updated.

The Debugger is able to locate the application source file that needs to be opened, if it is found in its default location (its location when your application ELF file was built). However if the file has subsequently been moved you will need to indicate additional search paths to the Debugger (**Source Search Path** in the **Debug** menu).

To view your program source

If the current program counter position can be viewed in the source code, you can see it simply by opening a source pane.

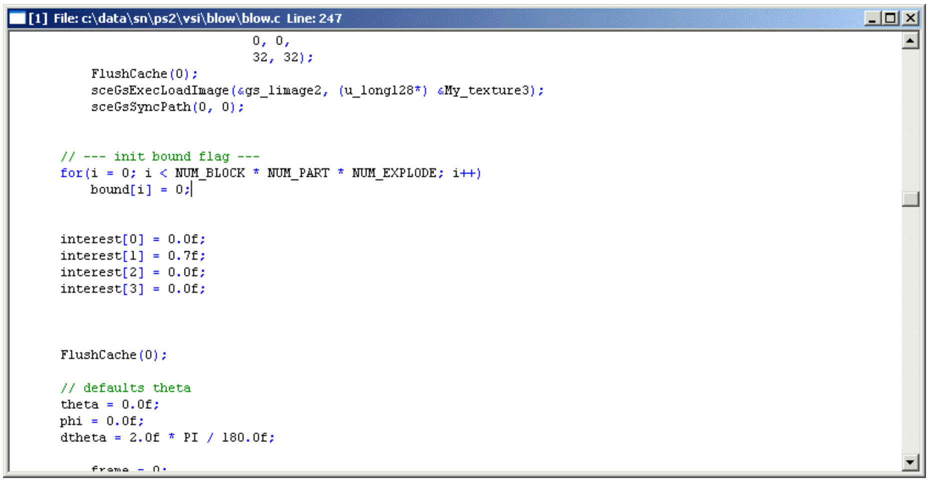
- Click **New Window > Source** in the **Window** menu, or click on the **Source file view** button in the toolbar.

  A new window containing a source pane will be opened and the current program counter will be indicated with a ⇨ character. If the program counter cannot currently be viewed in the source, or if the source file cannot be located, the pane will appear with the message "No Source File Loaded" showing.

Note: You can use the Go to PC option in the source pane shortcut menu to quickly move the display to show the new program counter position.

Source pane

The source pane enables you to view the source of your application that is currently being executed on the PlayStation 2.



```
                                0, 0,
                                32, 32);
            FlushCache(0);
            sceGsExecLoadImage(&gs_limage2, (u_long128*) &My_texture3);
            sceGsSyncPath(0, 0);


        // --- init bound flag ---
        for(i = 0; i < NUM_BLOCK * NUM_PART * NUM_EXPLODE; i++)
            bound[i] = 0;


        interest[0] = 0.0f;
        interest[1] = 0.7f;
        interest[2] = 0.0f;
        interest[3] = 0.0f;



        FlushCache(0);

        // defaults theta
        theta = 0.0f;
        phi = 0.0f;
        dtheta = 2.0f * PI / 180.0f;

            frame = 0;
```

The location of your application source files is contained in the ELF file. When the source pane is opened the Debugger will try to locate the source file in which the program counter is positioned in its original location. However, if this location has changed, or the ELF file was not built on your machine, you can enter directories that the Debugger can search to locate source files using the **Source Search Path** option in the **Debug** menu.

When you first open this pane it will normally show the current position of the program counter in your source file. However, if the program counter cannot be viewed in source then the pane shows a message saying that no source is available.

If the source pane is the active pane then the program counter will automatically be updated each time the target status changes, and you will always be able to view its current position in the source pane.

At any time you can position the cursor to the first line of source code by pressing the shortcut key <Ctrl+Home>, or to the last line of source by pressing <Ctrl+End>.

You can step through your application source executing one line of code at a time on the PlayStation 2, or set breakpoints to stop execution at certain points in your application.

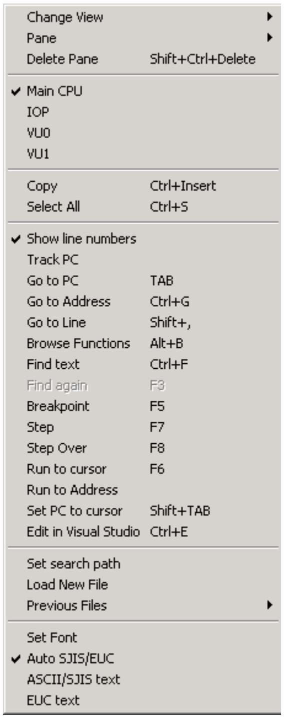
Note that multi-line comments (using the C style /* */ markers) are recognised and colored accordingly.

The border strip down the left side of the pane displays line numbers, bookmark and breakpoint markers etc. You can change the background / foreground colors for this strip via the "Settings menu" on page 10.

You can also use bookmarks to remember (and subsequently return to) locations in the source pane.

You can press the spacebar to switch between the source pane, the disassembly pane and the mixed mode pane (which shows the source code interleaved with the disassembly instructions associated with each line). The sequence followed is source pane, disassembly pane, mixed mode pane, source pane etc.

Source pane shortcut menu



```
  Change View                          ▸
  Pane                                 ▸
  Delete Pane          Shift+Ctrl+Delete
✔ Main CPU
  IOP
  VU0
  VU1
  Copy                 Ctrl+Insert
  Select All           Ctrl+S
✔ Show line numbers
  Track PC
  Go to PC             TAB
  Go to Address        Ctrl+G
  Go to Line           Shift+,
  Browse Functions     Alt+B
  Find text            Ctrl+F
  Find again           F3
  Breakpoint           F5
  Step                 F7
  Step Over            F8
  Run to cursor        F6
  Run to Address
  Set PC to cursor     Shift+TAB
  Edit in Visual Studio Ctrl+E
  Set search path
  Load New File
  Previous Files                       ▸
  Set Font
✔ Auto SJIS/EUC
  ASCII/SJIS text
  EUC text
```

Main CPU	Changes to show the source being executed on the main CPU. This is reflected in the window title. The background color of the source pane is reset to white.
IOP	Changes to show the source being executed on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the source on the PlayStation 2 VU0 unit. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the source on the PlayStation 2 VU1 unit. The name of the unit being viewed is displayed in the window title.
Copy	Copies selected item to clipboard
Select All	Selects all the text in the source pane.
Show line numbers	When checked, line numbers are displayed against each line of the source file.
Track PC	When checked, the position of the program counter in the disassembly pane is automatically

	tracked.
Go to PC	Changes the source being viewed in the pane to show the line on which the program counter is currently set. This is only useful when the PlayStation 2 is stopped.
Go to Address	Brings up a dialog that enables you to enter a particular address in the PlayStation 2 unit memory that you wish to view. You must prefix the address by 0x to indicate a hexadecimal address. You can either enter the required address or a symbol to specify the part of memory that you wish to view.
Go to Line	Changes the source so that it is centred on the selected line number.
Browse Functions	Opens a dialog that enables you to browse through the source according to the functions in your application. In addition you can set a breakpoint on a particular function in your application or remove it from this dialog.
Find text	Opens a dialog that enables you to search for some text in the source file. If the text is found, the line containing the text is marked with the cursor.
Find again	Repeats the previous **Find text** search command.
Breakpoint	Enables you to set a breakpoint on the currently selected line of code in the source pane. If you use this command again, the breakpoint will be removed. A breakpoint line is indicated by a red strip.
Step	Enables you to single-step through your application (stepping into any called subroutines). This command can be used once the target is stopped to step through the source code executing one instruction at a time on the PlayStation 2 unit. The program counter should move to the next instruction to be executed each time you step.
Step Over	Enables you to single-step through your application (stepping over any called subroutines). This command can be used once the target is stopped to step through the disassembly, executing one instruction at a time on the PlayStation 2 unit. The program counter will move to the next line of code to be executed each time you step.
Run to cursor	Enables you to specify that the application should be run on the PlayStation 2 unit until the currently selected line of code in the source

	pane, is reached.
Run to Address	Enables you to specify that the application should be run on the PlayStation 2 unit until the line of code at the specified address is reached. The address is specified in the dialog that appears, and must be prefixed by 0x if you enter it in hexadecimal.
Set PC to cursor	Sets the program counter to the currently selected line in the source pane. You should see the > character move to the selected line, indicating that the program counter has been reset. Note that none of the intermediate lines of the program are run to reach the new program counter position.
Edit in Visual Studio	Locates the current source line in Microsoft Visual Studio. The Visual Studio IDE must be already running for this to work. See "Editing source code" on page 68 for further details.
Set search path	Enables you to add directories to the search path for your application source files. If you add more than one path it must be separated by a semi-colon. This is useful if you do not have the source files stored in the same position as when you built the ELF file, or if you didn't build the ELF file. This command is equivalent to the **Source Search Path** option in the **Debug** menu.
Load New File	Enables you to specify a different source file to be loaded into the source pane that you access this command in. The File Open dialog appears allowing you to select the source file to open. Once you have located the file to be opened, it will replace any file that was originally displayed in the source pane. This differs slightly from the **Load Source File** command on the **File** menu that will always open the selected source file in a new source pane.
Previous Files	Enables you to access a sub-menu displaying the last 10 source files that were loaded. Click any of these to load the appropriate file. <Ctrl+P> will display this as a pop-up menu in the upper left corner of the source pane.

Searching through source

You can view different parts of the source much as you would navigate through the source file using a text editor.

To navigate through your program source

- Use the standard Windows shortcuts <PgUp> and <PgDn> to scroll through the source a page at a time.
- Use the <arrow keys> to scroll through the source a line at a time.

To quickly move to the first line of the source file, use the Start of file accelerator key:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.
2. In the list box at the top right, select **Source**.
3. Scroll down the **Command Name** list until the accelerator key setting for **Start of file** is shown and then note the appropriate **Key Sequence** (if any).
4. Press **Cancel** to close the Application Setting dialog.
5. Press the key sequence you need for moving to the start of the file.

To quickly move to the last line of the source file, use the End of file accelerator key:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.
2. In the list box at the top right, select **Source**.
3. Scroll down the **Command Name** list until the accelerator key setting for **End of file** is shown and then note the appropriate **Key Sequence** (if any).
4. Press **Cancel** to close the Application Setting dialog.
5. Press the key sequence you need for moving to the end of the file.

Using bookmarks to navigate through program source

You can also use bookmarks to remember (and subsequently return to) locations in the source pane.

- Use <Ctrl+F2> to add a bookmark. A blue marker will appear within the border on the line containing the cursor.
- Use <F2> to return to the line containing the bookmark when viewing a different part of the source file.

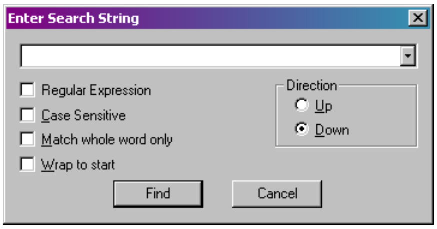
- More than one bookmark can be added to a source file. Each time you press <F2> the cursor will move to the next bookmark from the current location until the last bookmark is reached, at which point <F2> will begin the search at the beginning of the file.
- Use <Shift+F2> to search backwards from the cursor location.
- When the debugger is closed down, bookmarks for all open files are stored in the configuration file and restored next time the debugger is executed.

To find text in a source file

You can easily locate a search string in the source pane, using the **Find text** and **Find again** shortcut menu options. Searches can be case-sensitive if required.

1. To search for a text string, use the **Find text** shortcut menu option.

   The Enter Search String dialog is displayed, allowing you to define the search string.



2. The text entry field contains a dropdown history list of recently entered search strings (most recent first) so that you don't have to keep rekeying strings that you repeatedly search for.

   You can restrict or increase the number of matches found by checking the following:

Regular Expression	Indicates that the search string is a regular expression. See "Regular expressions" on page 68 for information on regular expression syntax.
Case Sensitive	Causes a match to fail unless the matched string is case-identical to the search string.
Match whole word only	Causes a match to fail unless the matched string is a whole word, i.e. matched substrings will not be detected.
Wrap to start	Causes a downward search to continue at the start of file, or an upward search to continue at the end of file.
Up and **Down**	These radio buttons determine the direction of the search, either upwards or downwards from the cursor position.

Note: These options are persistent only for the lifetime of the application, i.e. they are not stored in the configuration file on shutdown.

If the string is found, the source line containing the first occurrence will be highlighted in the source pane.

3. To repeat a text string search, use the **Find again** shortcut menu option.

Editing source code

When you invoke the **Edit in Visual Studio** option from the context menu of a Debugger source pane (or press <Ctrl+E>) - see "Source pane shortcut menu" on page 63 - the Debugger will try to launch an external editor to edit this source file at the same location. It will do this in the following order:

1. If you are using the ProDG Plus scripting feature (see "What is Debugger scripting?" on page 145) the Debugger will first look for a specific script function, and if it is found will call that function to invoke the editor.

2. If there was no suitable script function then the debugger will check to see whether there is an [opentemplate] variable in the debugger configuration file (dbugps2.ps2 - see "Saving the project configuration" on page 30). If a template was found it will use that template to determine how to launch an external editor, e.g.

   ```
   [opentemplate]
   c:\cw32\cw32.exe %f -G%l
   ```

   When this template is invoked the %f will be replaced with the pathname of the file to edit and the %l will be replaced with the line number.

   Note that if the debugger is launched without a suitable local configuration file in the current directory then it will attempt to load a default configuration file (of the same name) from the same directory as the debugger executable, so it is a good idea to add the two lines to that file also. When the debugger exits it save a local configuration file to the current directory, so the setting will be preserved.

3. If there was no script function and no [opentemplate] variable in the debugger configuration file then the debugger will attempt to invoke Microsoft Visual Studio to edit the file.

Regular expressions

A regular expression is a standard way to describe a string pattern, which can be used in string searches. To construct a regular expression, a prescribed syntax must be followed.

Regular expressions are comprised of characters, character sets and special symbols, similar to arithmetic expressions. Just as in arithmetic, regular expressions can themselves be delimited by enclosing them in parentheses, i.e.'(expr)', if it becomes necessary to separate them from other parts of an expression.

- *Character literal* This declares that a character, e.g. 'A', is part of the expression. Non-printing characters can be specified as escaped text formatting characters, i.e. '\ ', '\n', '\r', '\t' and '\v', or as hexadecimal numbers ('\xnn' where nn is a hexadecimal number). Note that the following punctuation and symbol characters must also be escaped because the symbols have a special meaning in regular expression syntax: '*', '\+', '\?', '\.', '\|', '\[', '\]', '\(', '\)', '\-', '\$' and '\^'.

- *Character sets* This declares that a match can be made from a range of possible values and is identified by enclosing the expression in square brackets "[set]" The expression set can itself be comprised of several subsets, e.g. the expression "[0-9A-Za-z]" matches any alphanumeric character. A character set can also be inverted by placing '^' (not) immediately after the opening square bracket, thus "[^0-9A-Za-z]" matches any non-alphanumeric character. Note that the special symbol '.' matches any character.

- *Concatenated expressions* Two expressions can be concatenated to form a larger expression, e.g. the expression "_[A-Za-z]" matches any string in which the first character is an underscore '_' and the second character is alphabetic. Two expressions can also be separated by the OR symbol '|', e.g.

  ```
  "[0-9]+|[A-Za-z_][A-Za-z_0-9]*"
  ```

  matches either an integer or a C identifier.

- *Operators* Various special symbols can be used to modify an immediately preceding expression, i.e.:

expr*	matches zero or more consecutive occurrences of expr
expr+	matches one or more consecutive occurrences of expr
expr?	matches at most one occurrence of expr
expr\i	matches expr while ignoring case

- *Named expressions* A regular expression can be named so that it can be used in a subsequent regular expression, e.g.:

  ```
  $Integer = "[0-9]+"
  $CIdntfr = "[A-Za-z_][A-Za-z_0-9]*"
  $MyExprn = "$Integer|$CIdntfr"
  ```

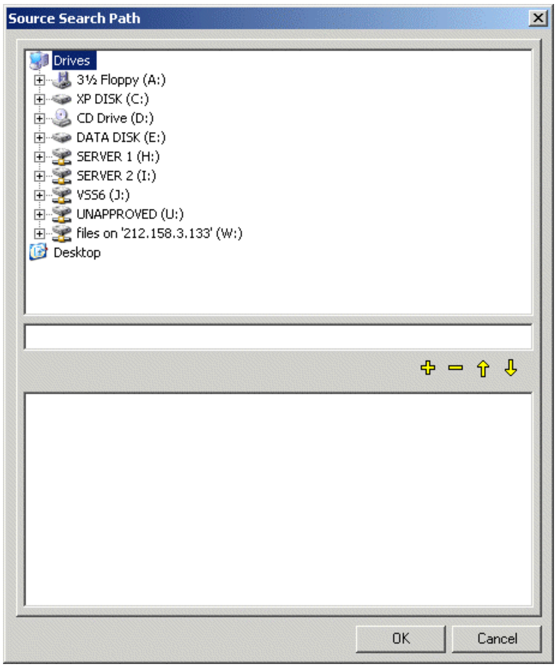
Locating source files

By default the Debugger will try to locate your application source files according to their location when the ELF file was built. However if the files have subsequently been moved, you may need to specify new paths for any required source files to be located.

To locate source files

1. Click Source Search Path in the Debug menu.

   The Source Search Path dialog appears similar to the following:



2. In the dialog that appears use the tree view to select the full path(s) of any directories where the Debugger should look for your application source files. You can add a selected directory to the search path by clicking on the [+] symbol, or remove a selected directory by clicking on the [-] symbol.

3. The order of directories in the search path can be altered by clicking on the up and down arrow symbols. To promote the selected directory click on the up arrow, to demote a directory click on the down arrow.

4. You may need to close any existing source file view and open a new one for the change to take effect, and the source file to be located and opened.

To view another source file

As well as viewing the line of source code that is currently being executed on the target you can also view any other source file.

1. Click **Load Source File** in the **File** menu.

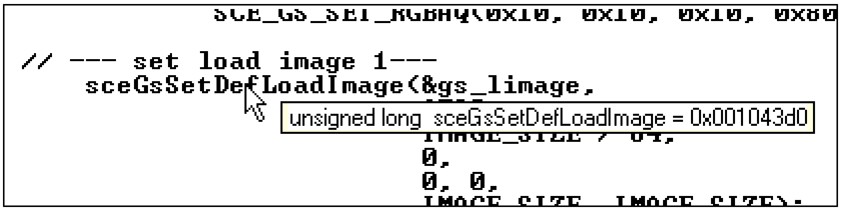
2. Locate the required source file in the dialog that is displayed.

   A new source pane is opened to display the contents of the selected source file. This can be used in the same way as the source pane that contains the program counter, and you can set breakpoints in this file and browse through it as required.

Note: If you are loading your file via source search paths from somewhere other than their original location, the Debugger is not able to assume the relationship to object code when you load a file using **Load Source File**. If this is the case it is better to go to the correct location in your source code using **Go to Address** (source pane shortcut menu) and entering a function name. In this way the file is located via the source search paths and the build information.

Evaluating expressions using floating ToolTips

You can evaluate expressions instantly by hovering the mouse over a line of code. If the expression under the code can be evaluated then the result is shown in a "floating ToolTip" style popup box, similar to this:



```
                SCE_GS_SET_RGBAQ(0x10, 0x10, 0x10, 0x80
// --- set load image 1---
    sceGsSetDefLoadImage(&gs_limage,
                            IMAGE_SIZE / 64,
                            0,
                            0, 0,
                            IMAGE_SIZE, IMAGE_SIZE);
```

unsigned long sceGsSetDefLoadImage = 0x001043d0

Viewing disassembly

Whenever the target stops running, if a disassembly pane is open it updates to show the current position of the program counter in your application disassembly (indicated by the ➡ character).

To view your program disassembly

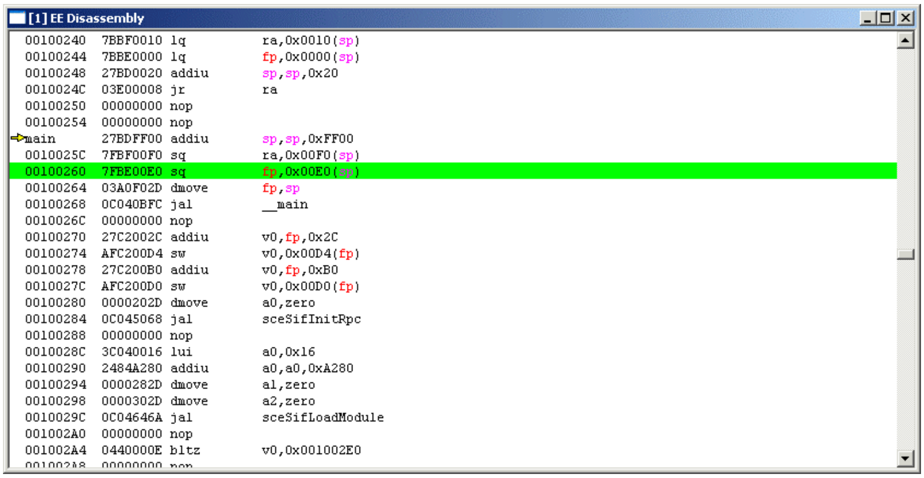
If you wish to open a new disassembly pane that will show the current program counter position:

1. Click **New Window > Disasm** in the **Window** menu or click the **Disassembly view** toolbar button.

2. A new window containing a disassembly pane is opened. This shows a disassembly of the code currently being run on the selected PlayStation 2 target unit. You can change the unit disassembly being viewed using the shortcut menu, and use the **Go to PC** command to view the program counter.

Disassembly pane

The disassembly pane enables you to view a disassembly of the program code on the PlayStation 2. This consists of addresses, opcodes and disassembly. You can view the disassembly running on the main CPU unit, the IOP unit or either of the vector units (VU0, VU1) on the PlayStation 2. To change the unit that you are currently viewing in the disassembly view, use the right-click shortcut menu.



```
 00100240  7BBF0010 lq          ra,0x0010(sp)
 00100244  7BBE0000 lq          fp,0x0000(sp)
 00100248  27BD0020 addiu       sp,sp,0x20
 0010024C  03E00008 jr          ra
 00100250  00000000 nop
 00100254  00000000 nop
main       27BDFF00 addiu       sp,sp,0xFF00
 0010025C  7FBF00F0 sq          ra,0x00F0(sp)
 00100260  7FBE00E0 sq          fp,0x00E0(sp)
 00100264  03A0F02D dmove       fp,sp
 00100268  0C040BFC jal         __main
 0010026C  00000000 nop
 00100270  27C2002C addiu       v0,fp,0x2C
 00100274  AFC200D4 sw          v0,0x00D4(fp)
 00100278  27C200B0 addiu       v0,fp,0xB0
 0010027C  AFC200D0 sw          v0,0x00D0(fp)
 00100280  0000202D dmove       a0,zero
 00100284  0C045068 jal         sceSifInitRpc
 00100288  00000000 nop
 0010028C  3C040016 lui         a0,0x16
 00100290  2484A280 addiu       a0,a0,0xA280
 00100294  0000282D dmove       a1,zero
 00100298  0000302D dmove       a2,zero
 0010029C  0C04646A jal         sceSifLoadModule
 001002A0  00000000 nop
 001002A4  0440000E bltz        v0,0x001002E0
```

The disassembly pane consists of four columns of information, of which the second is optional. These are as follows:

- the first column contains the address or label
- the second column displays the opcode in hexadecimal
- the third and fourth columns show the disassembled program instruction

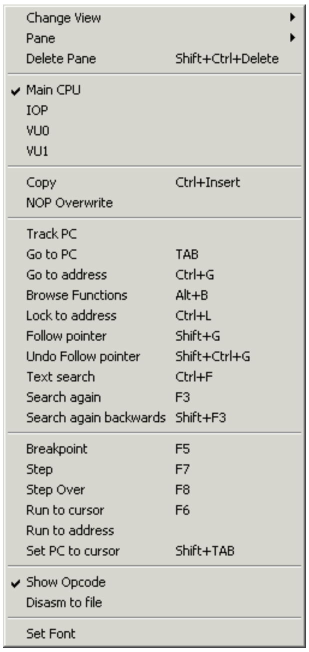
The current location of the program counter is indicated by a ➯ character in front of the first column.

Any lines that contain breakpoints are shown in a different color.

If the disassembly pane is the active pane then the pane will track the current program counter value.

You can press the spacebar to switch between the source pane, the disassembly pane and the mixed mode pane (which shows the source code interleaved with the disassembly instructions associated with each line). The sequence followed is source pane, disassembly pane, mixed mode pane, source pane etc.

Disassembly pane shortcut menu



Main CPU	Changes to show the disassembly on the main CPU. This is reflected in the window title.
IOP	Changes to show the disassembly on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the disassembly on the PlayStation 2 VU0 unit. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the disassembly on the PlayStation 2 VU1 unit. The name of the unit being viewed is displayed in the window title.
Copy	Copies selected item to clipboard.
NOP Overwrite	Replaces all code within the current selection with 'nop' instructions.
Track PC	When checked, the position of the program counter in the disassembly pane is automatically tracked.
Go to PC	Changes the disassembly being viewed in the pane to show the line on which the program counter is currently set. This is only useful when your application is not currently running on the target PlayStation 2.
Go to Address	Opens a dialog that enables you to enter a particular address in the PlayStation 2 memory that you wish to view. If you enter a

	hexadecimal address prefix it with 0x or check the **Default Radix Hexadecimal** checkbox. You can either enter the required address, symbol or expression to specify the part of memory that you wish to view. If you enter the start of the symbol name you can use the **Complete** button to provide you with a list of possible symbols in your application.
Browse Functions	Opens a dialog that enables you to browse through the disassembly according to the functions in your application. In addition you can set a breakpoint on a particular function in your application or remove it from this dialog.
Lock to Address	Enables you to specify an address expression that the display of disassembly will be locked to. You must enter the required address or expression in the dialog that is displayed. If you wish to unlock the display then you will need to access the Lock to Address dialog, but leave the expression field blank.
Follow pointer	'Jumps' the disassembly pane to the address on the current line, you can follow any number of pointers.
Undo follow pointer	Unwinds through the followed pointers and jumps the disassembly window to the previous location.
Text search	Allows you to enter a text string to search for. This can be straight text or a regular expression.
Search again	Will look for the next occurence of the text search string.
Search again backwards	Will look for the previous occurence of the text search string.
Breakpoint	Enables you to set a breakpoint on the currently selected line in the disassembly pane. If you repeat this, the breakpoint will be removed. A breakpoint line is indicated by a red highlight.
Step	Enables you to single-step through your application (stepping into any called subroutines). This command can be used once the target is stopped to step through the disassembly executing one instruction at a time on the PlayStation 2 unit. The program counter moves to the next instruction to be executed each time you step.
Step Over	Enables you to single-step through your application (stepping over any called subroutines). This command can be used once the target is stopped to step through the disassembly executing one instruction at a time

	on the PlayStation 2. The program counter will move to the next instruction to be executed each time you step.
Run to cursor	Enables you to specify that the target should be run until the instruction that the cursor is currently set on in the disassembly pane, is reached.
Run to Address	Enables you to specify that the application should be run on the PlayStation 2 until the instruction at the specified address is reached. The address is specified in the dialog that appears.
Set PC to cursor	Sets the program counter to the currently selected line in the disassembly pane. You should see the > character move to the selected line, indicating that the program counter has been reset. Note that none of the intermediate lines of the program are run to reach the new program counter position.
Show Opcode	When checked, displays the hexadecimal opcode in the second column of the disassembly pane
Disasm to file	Enables you to save a part of the PlayStation 2 disassembly to a text file. A dialog is displayed in which you can enter the **Start Address**, **End Address** and the **Filename** of the file in which the disassembly is to be saved. If you enter hexadecimal addresses, they should be prefixed with 0x.

Viewing source and disassembly together

The mixed mode pane displays the source code for the ELF program, interleaved with the disassembled instructions associated with each line.

You can use the standard step, step into and step out operations for either the source or disassembly. The context menu **Source Stepping** switch enables you to switch between these two options. The PC icon shows the current location alongside the appropriate source or disassembly line. You can also use the run to cursor operation.

You can add and remove breakpoints to and from the source code and/or the disassembly instructions. When source stepping mode is enabled, each breakpoint will be highlighted twice, once as a normal breakpoint icon alongside the source line for which it is associated, and secondly, as a dimmed breakpoint icon next to the specific disassembly instruction.

When this mode is disabled, only a single breakpoint icon will be displayed next to the disassembly instruction to which the breakpoint was added.

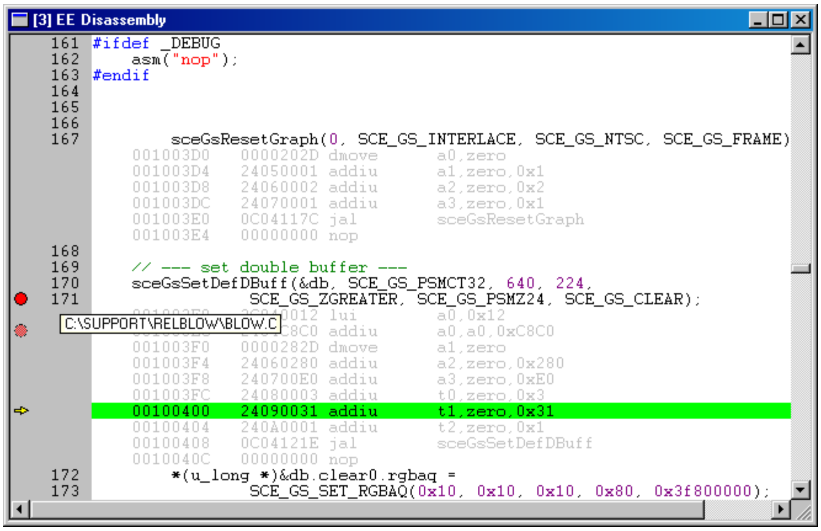
You can toggle the display of line numbers for the source code. When this option is enabled the border is enlarged and numbers are displayed alongside the source code lines. If you 'hover' the mouse over the border a tooltip will display the filename of the relevant source file.

Lines containing disassembly instructions remain blank.

You can also enable/disable the display of opcode on disassembly instruction lines.

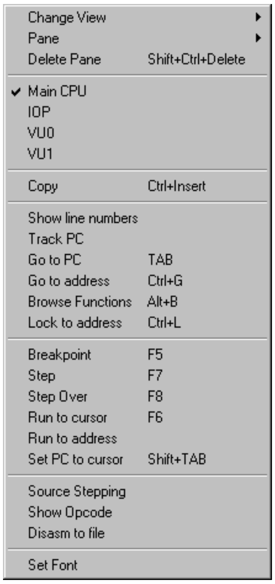
The source code syntax coloring will use the settings currently specified in the Application Settings dialog. You can use the **Mixed** category in the **Format** tab of the Application Settings dialog to set the color for the disassembly display. You can also use this category to set colors for the foreground, background and border for the mixed mode pane.

You can access the mixed mode pane by pressing the spacebar when a source or disassembly pane has the focus. This will take you to the next pane in a sequence. The order is source pane, disassembly pane, mixed mode pane, source pane etc.



```
161 #ifdef _DEBUG
162     asm("nop");
163 #endif
164
165
166
167         sceGsResetGraph(0, SCE_GS_INTERLACE, SCE_GS_NTSC, SCE_GS_FRAME)
        001003D0    0000202D dmove      a0,zero
        001003D4    24050001 addiu      a1,zero,0x1
        001003D8    24060002 addiu      a2,zero,0x2
        001003DC    24070001 addiu      a3,zero,0x1
        001003E0    0C04117C jal        sceGsResetGraph
        001003E4    00000000 nop
168
169     // --- set double buffer ---
170     sceGsSetDefDBuff(&db, SCE_GS_PSMCT32, 640, 224,
171             SCE_GS_ZGREATER, SCE_GS_PSMZ24, SCE_GS_CLEAR);
                        0012 lui        a0,0x12
  C:\SUPPORT\RELBLOW\BLOW.C C8C0 addiu  a0,a0,0xC8C0
        001003F0    0000282D dmove      a1,zero
        001003F4    24060280 addiu      a2,zero,0x280
        001003F8    240700E0 addiu      a3,zero,0xE0
        001003FC    24080003 addiu      t0,zero,0x3
        00100400    24090031 addiu      t1,zero,0x31
        00100404    240A0001 addiu      t2,zero,0x1
        00100408    0C04121E jal        sceGsSetDefDBuff
        0010040C    00000000 nop
172         *(u_long *)&db.clear0.rgbaq =
173             SCE_GS_SET_RGBAQ(0x10, 0x10, 0x10, 0x80, 0x3f800000);
```

Mixed Mode pane shortcut menu



Main CPU	Changes to show the disassembly on the main CPU. This is reflected in the window title.
IOP	Changes to show the disassembly on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the disassembly on the PlayStation 2 VU0 unit. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the disassembly on the PlayStation 2 VU1 unit. The name of the unit being viewed is displayed in the window title.
Copy	Copies selected item to clipboard.
Show line numbers	When checked, source code line numbers will be displayed. Lines containing disassembly instructions will remain blank.
Track PC	When checked, the position of the program counter in the disassembly display is automatically tracked.
Go to PC	Changes the disassembly being viewed in the pane to show the line on which the program counter is currently set. This is only useful when your application is not currently running on the target PlayStation 2.
Go to Address	Opens a dialog that enables you to enter a particular address in the PlayStation 2 memory that you wish to view. If you enter a hexadecimal address prefix it with 0x or check the **Default**

	Radix Hexadecimal checkbox. You can enter the required address, symbol or expression to specify the part of memory that you wish to view. If you enter the start of the symbol name you can use the **Complete** button to provide you with a list of possible symbols in your application.
Browse Functions	Opens a dialog that enables you to browse through the disassembly according to the functions in your application. In addition you can set a breakpoint on a particular function in your application or remove it from this dialog.
Lock to Address	Enables you to specify an address expression that the display of disassembly will be locked to. You must enter the required address or expression in the dialog that is displayed. If you wish to unlock the display then you will need to access the Lock to Address dialog, but leave the expression field blank.
Breakpoint	Enables you to set a breakpoint in the source code and/or disassembly instructions. If you repeat this, the breakpoint will be removed. If **Source Stepping** is checked, a red breakpoint icon will appear once on the appropriate source line and also as a dimmed version, on the specific disassembly instruction. If **Source Stepping** is *not* checked the icon will be displayed on the specific instruction line at which the instruction was added.
Step	Enables you to single-step through your application (stepping into any called subroutines). This command can be used once the target is stopped to step through the disassembly executing one instruction at a time on the PlayStation 2 unit. The program counter moves to the next instruction to be executed each time you step.
Step Over	Enables you to single-step through your application (stepping over any called subroutines). This command can be used once the target is stopped to step through the disassembly executing one instruction at a time on the PlayStation 2. The program counter will move to the next instruction to be executed each time you step.
Run to cursor	Enables you to specify that the target should be run until the instruction that the cursor is currently set on is reached.
Run to Address	Enables you to specify that the application should be run on the PlayStation 2 until the instruction at the specified address is reached. The address is

	specified in the dialog that appears.
Set PC to cursor	Sets the program counter to the currently selected disassembly line. You should see the > character move to the selected line, indicating that the program counter has been reset. Note that none of the intermediate lines of the program are run to reach the new program counter position.
Source Stepping	When checked, stepping will take place at the source level. If this option is unchecked, stepping will occur at the disassembly level.
Show Opcode	When checked, displays the hexadecimal opcode in the second column of the disassembly display.
Disasm to file	Enables you to save a part of the PlayStation 2 disassembly to a text file. A dialog is displayed in which you can enter the **Start Address**, **End Address** and the **Filename** of the file in which the disassembly is to be saved. If you enter hexadecimal addresses, they should be prefixed with 0x.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 6: Viewing and modifying memory and registers

Viewing and modifying memory

You can view memory in the memory pane. You can also modify the value of any memory address.

To view and modify memory

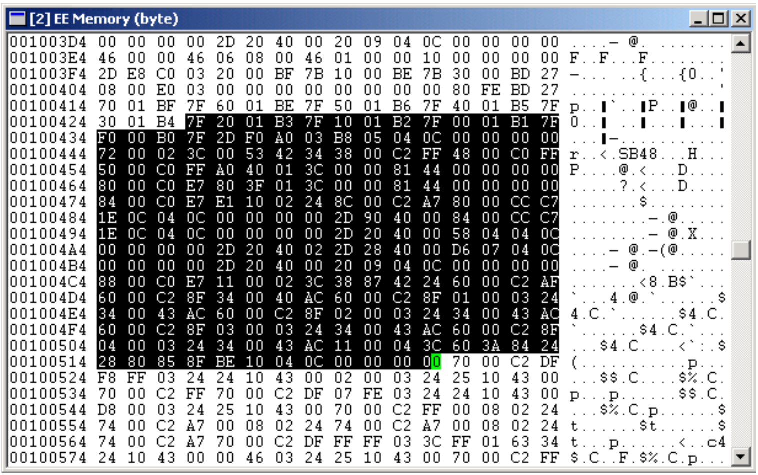
To view the value of memory addresses:

- Click **New Window > Memory** in the **Window** menu or click the **Memory view** toolbar button.

  This will open a new window containing a memory pane.

Memory pane

The memory pane enables you to view the contents of any part of the PlayStation 2 memory. You can view memory on the EE (main CPU), IOP, VU0 or VU1 processors, by selecting the required unit in the shortcut menu.



[2] EE Memory (byte)

```
001003D4 00 00 00 00 2D 20 40 00 20 09 04 0C 00 00 00 00 ....- @. .......
001003E4 46 00 00 46 06 08 00 46 01 00 00 10 00 00 00 00 F..F...F........
001003F4 2D E8 C0 03 20 00 BF 7B 10 00 BE 7B 30 00 BD 27 -.....{...{0..'
00100404 08 00 E0 03 00 00 00 00 00 00 00 00 80 FE BD 27 ...............'
00100414 70 01 BF 7F 60 01 BE 7F 50 01 B6 7F 40 01 B5 7F p..▌`..▌P..▌@..▌
00100424 30 01 B4 7F 20 01 B3 7F 10 01 B2 7F 00 01 B1 7F 0..▌ ..▌...▌...▌
00100434 F0 00 B0 7F 2D F0 A0 03 B8 05 04 0C 00 00 00 00 ...▌-...........
00100444 72 00 02 3C 00 53 42 34 38 00 C2 FF 48 00 C0 FF r..<.SB48...H...
00100454 50 00 C0 FF A0 40 01 3C 00 00 81 44 00 00 00 00 P....@.<...D....
00100464 80 00 C0 E7 80 3F 01 3C 00 00 81 44 00 00 00 00 .....?.<...D....
00100474 84 00 C0 E7 E1 10 02 24 8C 00 C2 A7 80 00 CC C7 .......$........
00100484 1E 0C 04 0C 00 00 00 00 2D 90 40 00 84 00 CC C7 ........-.@.....
00100494 1E 0C 04 0C 00 00 00 00 2D 20 40 00 58 04 04 0C ........- @.X...
001004A4 00 00 00 00 2D 20 40 02 2D 28 40 00 D6 07 04 0C ....- @.-(@.....
001004B4 00 00 00 00 2D 20 40 00 20 09 04 0C 00 00 00 00 ....- @. .......
001004C4 88 00 C0 E7 11 00 02 3C 38 87 42 24 60 00 C2 AF .......<8.B$`...
001004D4 60 00 C2 8F 34 00 40 AC 60 00 C2 8F 01 00 03 24 `...4.@.`......$
001004E4 34 00 43 AC 60 00 C2 8F 02 00 03 24 34 00 43 AC 4.C.`......$4.C.
001004F4 60 00 C2 8F 03 00 03 24 34 00 43 AC 60 00 C2 8F `......$4.C.`...
00100504 04 00 03 24 34 00 43 AC 11 00 04 3C 60 3A 84 24 ...$4.C....<`:.$
00100514 28 80 85 8F BE 10 04 0C 00 00 00 00 70 00 C2 DF (...........p...
00100524 F8 FF 03 24 24 10 43 00 02 00 03 24 25 10 43 00 ...$$.C....$%.C.
00100534 70 00 C2 FF 70 00 C2 DF 07 FE 03 24 24 10 43 00 p...p......$$.C.
00100544 D8 00 03 24 25 10 43 00 70 00 C2 FF 00 08 02 24 ...$%.C.p......$
00100554 74 00 C2 A7 00 08 02 24 74 00 C2 A7 00 08 02 24 t......$t......$
00100564 74 00 C2 A7 70 00 C2 DF FF FF 03 3C FF 01 63 34 t...p......<..c4
00100574 24 10 43 00 00 46 03 24 25 10 43 00 70 00 C2 FF $.C..F.$%.C.p...
```

The memory information is displayed in three distinct columns:

Address	This first column contains the address in memory on the target represented by the rest of the line. It is in hexadecimal format, and by default is 32 bits.
Data	The second column contains the actual data at the given memory address. The amount of data shown depends on the width of the pane. The data can be displayed in bytes, words, double words, floats or various fixed types.
ASCII	This third column shows the memory data in column two represented as ASCII characters. Any non-printing characters are shown as "." characters.

The memory data shown in the second **Data** column can be overtyped to directly change the memory on the target if required. It can also be quickly incremented or decremented using <numpad+> and <numpad->.

The memory pane now provides selection and copy/paste functionality. For example, a range of memory addresses can be selected and then copied to the clipboard, dragged to another pane etc.

The following selection methods are available:

- Double-click an address down the left-hand side of the pane to select that address. You can then drag this to another memory pane for **Goto Address** functionality or drag it to a watch pane in order to try and match the address with a variable that can then be added to the pane.

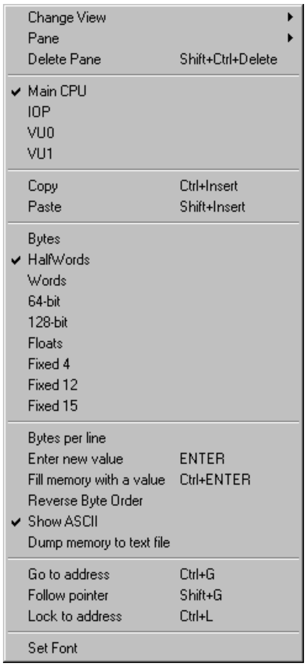
- Click and drag over the address column to select entire lines of memory.

- Click and drag on a memory value to perform free-form selections, in order to select individual ranges of memory.

Once a selection has been made it can be copied to the clipboard as text.

Once you have copied data to the clipboard, you can also paste it into a new memory location (in the same memory pane or a different one), by repositioning the cursor and selecting **Paste** from the shortcut menu. This will overwrite the existing contents of the memory location with the data from the clipboard.

Memory pane shortcut menu



Main CPU	Changes to show the memory on the main CPU. This is reflected in the window title.
IOP	Changes to show the memory on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the memory on the PlayStation 2 vector unit 0. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the memory on the PlayStation 2 vector unit 1. The name of the unit being viewed is displayed in the window title.
Copy	Copies selected item to clipboard.
Paste	Copies the (text-based) contents of the clipboard into memory beginning at the current cursor location.
Bytes	Displays memory as bytes (8 bits).
HalfWords	Displays memory as half words (16 bits).
Words	Displays memory as words (32 bits).
64-bit	Displays memory as 64-bit integer values.
128-bit	Displays memory as 128-bit integer values.

You can cycle through display as **Bytes**, **HalfWords**, **Words, 64-bit** and **128-bit** by repeatedly pressing the <Ctrl+w> shortcut key. The current setting will be displayed in the window title bar.

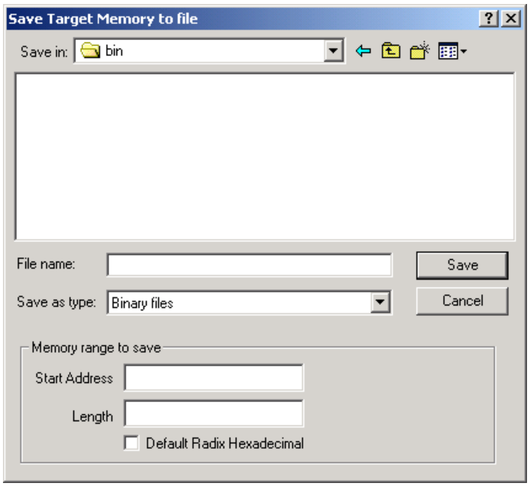
Floats	Changes the memory pane so that the memory is displayed as floating point values.
Fixed 4, Fixed 12, Fixed 15	Changes the memory pane so that the memory data is displayed in one of the fixed point formats used by the vector units (see Sony documentation).
Bytes per line	Enables you to specify the increment from one line in the memory pane to the next.
Enter new value	Enables you to enter an expression to specify a new value for an area of memory.
Fill memory with a value	Fills all data items within the current selection with a user-specified value.
Reverse Byte Order	Toggles between big- and little-endian display of multi-byte data items.
Show ASCII	When checked, the third column shows the memory data in column two represented as ASCII characters.
Dump memory to text file	Enables you to save a range of memory to disk in a text file. See "Saving memory to a text file" on page 85.
Go to address	Enables you to specify a particular address or symbol to be viewed in the memory pane. The address must be entered into the dialog that is displayed. If you enter a hexadecimal address prefix it with 0x or check the **Default Radix Hexadecimal** checkbox. If you enter the first part of a symbol name you can use the **Complete** button to provide a list of possible symbols to go to. Once you have indicated the required address, the memory pane updates to show this part of memory at the top of the pane.
Follow pointer	Jumps to the address stored within the currently selected data item. If the selected item is less than 32-bits, the neighboring item(s) are concatenated to form the address.
Lock to address	Enables you to specify an address expression that the display of memory will be locked to. You must enter the required address or expression in the dialog that is displayed. To unlock the display you must access the Lock to Address dialog again but leave the expression field blank.

Uploading and downloading target memory

You can save parts of target memory to a binary or text file, or alternatively download the contents of a binary file to the target memory.

To upload part of the target memory to a binary file

1. Click **Save Binary** in the **File** menu and the following dialog is displayed:

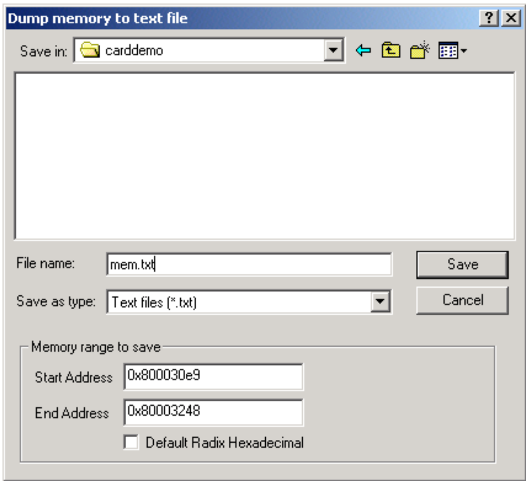

Save Target Memory to file
Save in: bin
File name:
Save as type: Binary files
Save
Cancel
Memory range to save
Start Address
Length
Default Radix Hexadecimal

2. Enter the **Start Address** and the **Length** of the memory range in the fields provided for this purpose. You can either use 0x as a prefix or if you leave this out and check the **Default Radix Hexadecimal** checkbox the fields will assume that you are entering hexadecimal values.

3. Select an existing file or enter a name for the binary file that you wish to save the contents of memory to and click **Save**.

Saving memory to a text file

1. Right-click in the memory pane and select **Dump memory to text file**. The following dialog is displayed.


Dump memory to text file

Save in: carddemo

File name: mem.txt

Save as type: Text files (*.txt)

Memory range to save

Start Address 0x800030e9

End Address 0x80003248

It enables you to save a range of memory to disk in a specified text file. The output looks exactly the same as in the memory pane display.

2. Navigate to the required text file and specify the start address of the memory range that will be saved in the **Start Address** field.

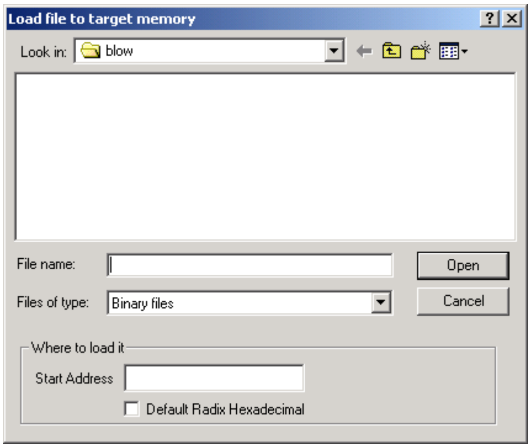
3. Specify the end address of the memory range that will be saved in the **End Address** field. If you have set the **Default Radix Hexadecimal** checkbox , there is no need to enter the 0x prefix to the **Start/End Address** fields as these values will be assumed to be hexadecimal.

4. Click **Save** and the memory range will be saved to the specified text file.

Note: If a memory range selection has already been made when this option is selected, the start and end addresses in the dialog will reflect this selection. Otherwise, the currently viewed address range will be used.

To download a binary file to target memory

1. Click **Load Binary** in the **File** menu, and the following dialog is displayed:



2. Navigate to the required binary file and specify the start address of the PlayStation 2 memory that you wish the binary file to be downloaded to in the **Start Address** field. You can either use 0x as a prefix or if you leave this out and check the **Default Radix Hexadecimal** checkbox the field will assume that you are entering hexadecimal values.

3. Click **Open** and the file contents are downloaded to the PlayStation 2 memory at the specified start address.

Note: The whole of the file is read into memory — there is no option to read only *n* bytes worth of the file into memory.

Viewing registers

You can view all of the DTL-T10000 registers in the registers pane. You can also modify the value of any register.

To view and modify registers

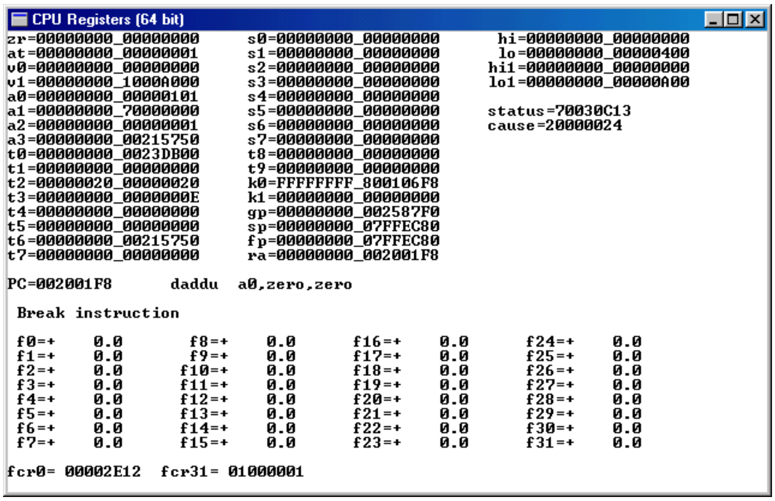
To view the values of registers:

- Click **New Window > Registers** in the **Window** menu or click the **Registers view** toolbar button.

  This will open a new window containing a registers pane that shows you all the registers and their current values.

Registers pane

The registers pane displays all the registers for which information is available in the PlayStation 2 unit that you are currently viewing. In addition it shows the disassembly instruction that the program counter is set on, and the current status of the target (shown in red).



```
zr=00000000_00000000      s0=00000000_00000000       hi=00000000_00000000
at=00000000_00000001      s1=00000000_00000000       lo=00000000_00000400
v0=00000000_00000000      s2=00000000_00000000      hi1=00000000_00000000
v1=00000000_1000A000      s3=00000000_00000000      lo1=00000000_00000A00
a0=00000000_00000101      s4=00000000_00000000
a1=00000000_70000000      s5=00000000_00000000      status=70030C13
a2=00000000_00000001      s6=00000000_00000000      cause=20000024
a3=00000000_00215750      s7=00000000_00000000
t0=00000000_0023DB00      t8=00000000_00000000
t1=00000000_00000000      t9=00000000_00000000
t2=00000020_00000020      k0=FFFFFFFF_800106F8
t3=00000000_0000000E      k1=00000000_00000000
t4=00000000_00000000      gp=00000000_002587F0
t5=00000000_00000000      sp=00000000_07FFEC80
t6=00000000_00215750      fp=00000000_07FFEC80
t7=00000000_00000000      ra=00000000_002001F8

PC=002001F8       daddu  a0,zero,zero

 Break instruction

 f0=+    0.0       f8=+    0.0      f16=+    0.0      f24=+    0.0
 f1=+    0.0       f9=+    0.0      f17=+    0.0      f25=+    0.0
 f2=+    0.0      f10=+    0.0      f18=+    0.0      f26=+    0.0
 f3=+    0.0      f11=+    0.0      f19=+    0.0      f27=+    0.0
 f4=+    0.0      f12=+    0.0      f20=+    0.0      f28=+    0.0
 f5=+    0.0      f13=+    0.0      f21=+    0.0      f29=+    0.0
 f6=+    0.0      f14=+    0.0      f22=+    0.0      f30=+    0.0
 f7=+    0.0      f15=+    0.0      f23=+    0.0      f31=+    0.0

fcr0= 00002E12  fcr31= 01000001
```

Registers pane default view

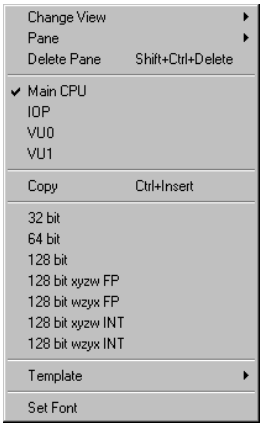
The registers pane is updated automatically whenever the target stops. If you click in the registers pane, the cursor will only be displayed on elements that can be modified.

To modify a register value you can directly overtype it in the registers pane. The new value is sent to the target as you are modifying it and there is no way of undoing this action.

Note: Changing the register values in this way may result in your application failing on the target!

In addition you can change the display of registers to 32-bit, 64-bit or 128-bit using the appropriate shortcut menu options.

Registers pane shortcut menu



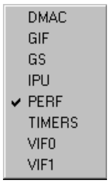
Main CPU	Changes to show the registers on the main CPU. This is reflected in the window title. The background color of the registers pane is reset to white.
IOP	Changes to show the registers on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the registers on the PlayStation 2 vector unit 0. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the registers on the PlayStation 2 vector unit 1. The name of the unit being viewed is displayed in the window title.
Copy	Copies selected item to clipboard.
32 bit, 64 bit, 128 bit, 128 bit xyzw FP, 128 bit wzyx FP, 128 bit xyzw INT, 128 bit wzyx INT	These options enable you to change the way in which the registers are displayed in the registers pane. The checked option is the one currently in effect. This option only applies to the main CPU registers pane.
Template	Allows you to select a templated registers view (see "Templated registers views" on page 89).

There is an additional shortcut menu that is activated just on the register values. You can only access it by right-clicking over an actual register value. This enables you to toggle between **decimal** and **hex** presentations (for the main registers) or between **float** and **hex** presentations (for the floating-point registers).

Templated registers views

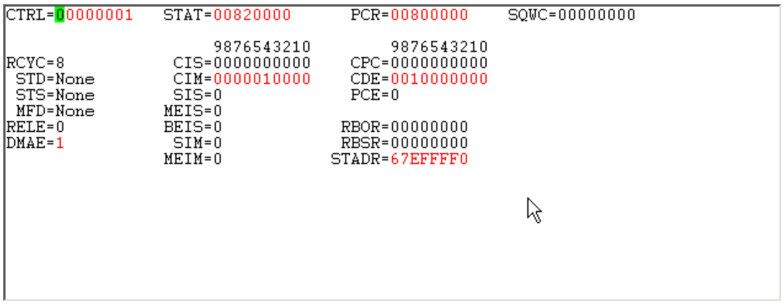
The registers pane supports external template (.RGT) files to display custom hardware register layouts. This feature extends the built-in **32 bit**, **64 bit** and **128 bit** layouts with customizable layouts displaying any of the available registers on all units within the PlayStation 2. When the Debugger starts, the folder in which the debugger is located is searched for any .RGT files.

The names of these files (minus the file extension) are then added to a Template sub-menu on the register panes shortcut menu:



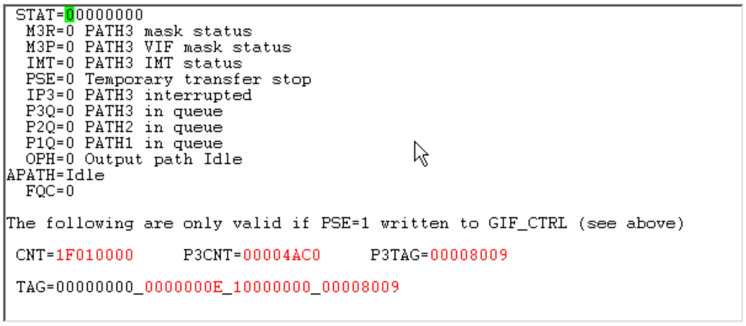
Alternate layouts are supplied for the following:

DMAC	Displays registers pertaining to the DMA controller.
GIF	Interface between the EE and GS processors.
GS	Displays GS signal register information.
IPU	Image Data Processor (MPEG playback, RGB conversion etc.)
PERF	Displays information pertaining to the performance counters.
TIMERS	Timers and counters within the EE.
VIF0/VIF1	VU0/VU1 DMA registers.



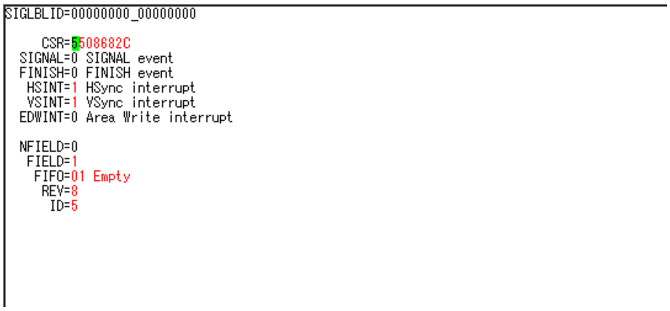
```
CTRL=00000001    STAT=00820000      PCR=00800000    SQWC=00000000

                      9876543210          9876543210
RCYC=8             CIS=0000000000      CPC=0000000000
 STD=None          CIM=0000010000      CDE=0010000000
 STS=None          SIS=0               PCE=0
 MFD=None         MEIS=0
RELE=0            BEIS=0              RBOR=00000000
DMAE=1             SIM=0              RBSR=00000000
                  MEIM=0             STADR=67EFFFF0
```

Registers pane DMAC view



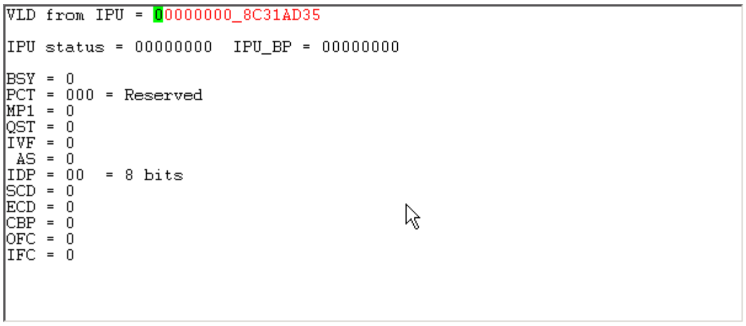
```
 STAT=00000000
  M3R=0 PATH3 mask status
  M3P=0 PATH3 VIF mask status
  IMT=0 PATH3 IMT status
  PSE=0 Temporary transfer stop
  IP3=0 PATH3 interrupted
  P3Q=0 PATH3 in queue
  P2Q=0 PATH2 in queue
  P1Q=0 PATH1 in queue
  OPH=0 Output path Idle
APATH=Idle
  FQC=0

The following are only valid if PSE=1 written to GIF_CTRL (see above)

 CNT=1F010000     P3CNT=00004AC0     P3TAG=00008009

 TAG=00000000_0000000E_10000000_00008009
```

Registers pane GIF view



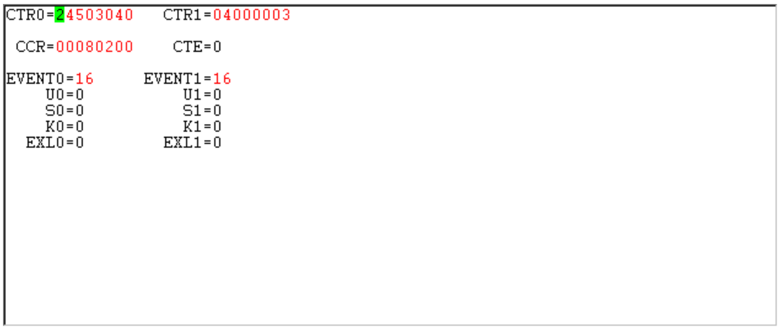
```
SIGLBLID=00000000_00000000

    CSR=5508682C
 SIGNAL=0 SIGNAL event
 FINISH=0 FINISH event
  HSINT=1 HSync interrupt
  VSINT=1 VSync interrupt
 EDWINT=0 Area Write interrupt

 NFIELD=0
  FIELD=1
   FIFO=01 Empty
    REV=8
     ID=5
```

Registers pane GS view



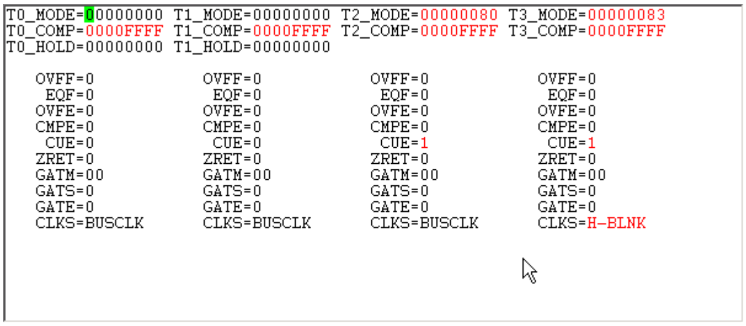
```
VLD from IPU = 00000000_8C31AD35

IPU status = 00000000  IPU_BP = 00000000

BSY = 0
PCT = 000 = Reserved
MP1 = 0
QST = 0
IVF = 0
 AS = 0
IDP = 00  = 8 bits
SCD = 0
ECD = 0
CBP = 0
OFC = 0
IFC = 0
```

Registers pane IPU view



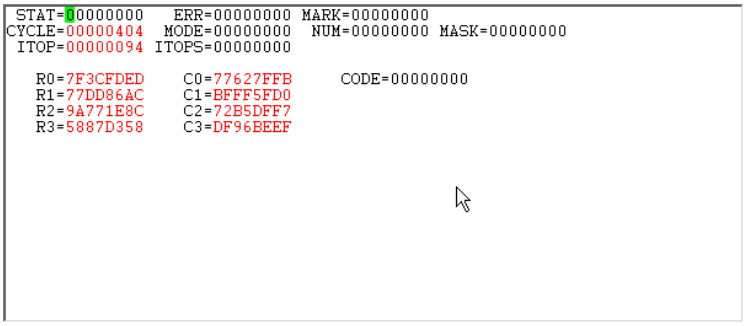
```
CTR0=24503040   CTR1=04000003

 CCR=00080200    CTE=0

EVENT0=16      EVENT1=16
    U0=0           U1=0
    S0=0           S1=0
    K0=0           K1=0
  EXL0=0         EXL1=0
```

Registers pane PERF view



```
T0_MODE=00000000 T1_MODE=00000000 T2_MODE=00000080 T3_MODE=00000083
T0_COMP=0000FFFF T1_COMP=0000FFFF T2_COMP=0000FFFF T3_COMP=0000FFFF
T0_HOLD=00000000 T1_HOLD=00000000

   OVFF=0           OVFF=0           OVFF=0           OVFF=0
    EQF=0            EQF=0            EQF=0            EQF=0
   OVFE=0           OVFE=0           OVFE=0           OVFE=0
   CMPE=0           CMPE=0           CMPE=0           CMPE=0
    CUE=0            CUE=0            CUE=1            CUE=1
   ZRET=0           ZRET=0           ZRET=0           ZRET=0
   GATM=00          GATM=00          GATM=00          GATM=00
   GATS=0           GATS=0           GATS=0           GATS=0
   GATE=0           GATE=0           GATE=0           GATE=0
   CLKS=BUSCLK      CLKS=BUSCLK      CLKS=BUSCLK      CLKS=H-BLNK
```

Registers pane TIMERS view



```
 STAT=00000000   ERR=00000000 MARK=00000000
CYCLE=00000404  MODE=00000000  NUM=00000000 MASK=00000000
 ITOP=00000094 ITOPS=00000000

   R0=7F3CFDED    C0=77627FFB     CODE=00000000
   R1=77DD86AC    C1=BFFF5FD0
   R2=9A771E8C    C2=72B5DFF7
   R3=5887D358    C3=DF96BEEF
```

Registers pane VIF0 view

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 7: Viewing local and watched variables

Viewing local variables

You can view the local variables of the function that the program counter is positioned in, in the local variables pane.

To view local variables

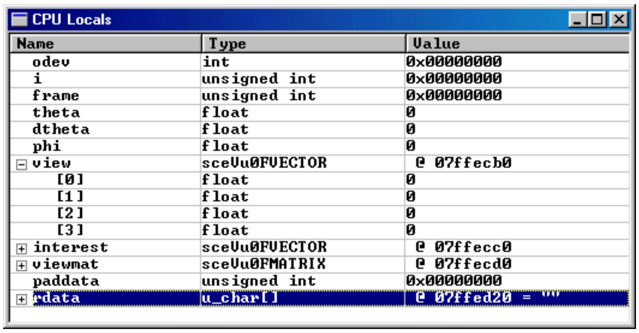
To view the values of local variables (variables in the current function):

- Click **New Window > Locals** in the **Window** menu or click the **Local variables view** button.

  This will open a new window containing a locals pane that shows you all the variables in the current function and their current values.

Locals pane

The locals pane is used to view the local variables of the current function scope on the PlayStation 2 unit. The display will change as you move between functions on the target unit and is automatically updated whenever the PlayStation 2 stops.



Name	Type	Value
odev	int	0x00000000
i	unsigned int	0x00000000
frame	unsigned int	0x00000000
theta	float	0
dtheta	float	0
phi	float	0
⊟ view	sceVu0FVECTOR	@ 07ffecb0
[0]	float	0
[1]	float	0
[2]	float	0
[3]	float	0
⊞ interest	sceVu0FVECTOR	@ 07ffecc0
⊞ viewmat	sceVu0FMATRIX	@ 07ffecd0
paddata	unsigned int	0x00000000
⊞ rdata	u_char[]	@ 07ffed20 = ""

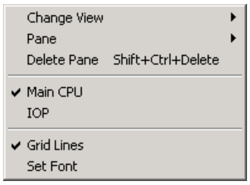
Each variable on the target is displayed with its current data Type and Value.

Any variables that are expandable, are indicated with a + sign in front of them. To expand out any of these variables you can double-click the variable. Any currently expanded variable can be collapsed again by double-clicking it. If you expand an array all array members are displayed.

To modify a variable, double-click on the value in the Value field and edit the value in place.

The local variables can be viewed and modified on the PlayStation 2 main CPU or the IOP processor.

Locals pane shortcut menu



Note: The Decimal/Hex menu option will only appear when local variables are being displayed in the locals pane.

Main CPU	Changes to show the local variables on the main CPU. This is reflected in the window title.
IOP	Changes to show the local variables being executed on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
Decimal / Hex	Enables you to toggle the display of the currently selected local variable between decimal and hexadecimal.
Grid Lines	When checked, displays a table grid separating rows and columns in the locals pane, to improve readability.

Monitoring variables

In addition you can set up variables that you would like to specifically monitor in the watch pane.

There is a C++ variable browser that helps you to select exactly the variable that you wish to add to a watch pane. In this browser you can view all the application variables grouped by class.

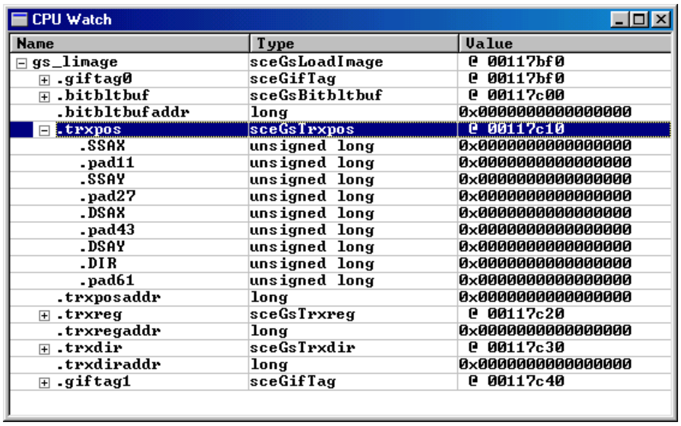
To monitor variables

- Click **New Window > Watch** in the **Window** menu or click the **Watch view** button.

  A new window containing an empty watch pane is displayed, in which you can add any watched variables you like. These are not limited to variables in the current function scope.

Watch pane

The watch pane enables you to monitor variables or expressions to see how they change as you step through the code. When you first open a watch pane it is empty, and you can add any watches that you require.


CPU Watch

Name	Type	Value
⊟ gs_limage	sceGsLoadImage	@ 00117bf0
⊞ .giftag0	sceGifTag	@ 00117bf0
⊞ .bitbltbuf	sceGsBitbltbuf	@ 00117c00
.bitbltbufaddr	long	0x0000000000000000
⊟ .trxpos	sceGsTrxpos	@ 00117c10
.SSAX	unsigned long	0x0000000000000000
.pad11	unsigned long	0x0000000000000000
.SSAY	unsigned long	0x0000000000000000
.pad27	unsigned long	0x0000000000000000
.DSAX	unsigned long	0x0000000000000000
.pad43	unsigned long	0x0000000000000000
.DSAY	unsigned long	0x0000000000000000
.DIR	unsigned long	0x0000000000000000
.pad61	unsigned long	0x0000000000000000
.trxposaddr	long	0x0000000000000000
⊞ .trxreg	sceGsTrxreg	@ 00117c20
.trxregaddr	long	0x0000000000000000
⊞ .trxdir	sceGsTrxdir	@ 00117c30
.trxdiraddr	long	0x0000000000000000
⊞ .giftag1	sceGifTag	@ 00117c40

As for the locals pane, any structures or arrays that you enter as watches can be expanded and collapsed by double-clicking them.

Watched variables can be added just by clicking **Add Watch** in the shortcut menu and typing the name of the required variable in the dialog that is displayed. Alternatively you can browse variables according to their C++ class to indicate exactly the variable that you wish to watch via the **Browse Vars** option in the shortcut menu.

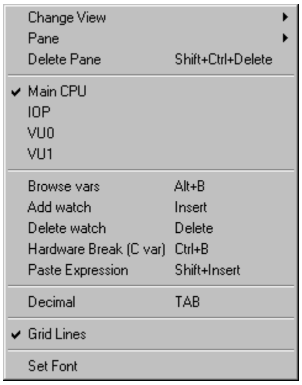
If you have more than one watch pane open, then each will only show the watched symbols that you have physically added to that pane.

The display of watched variable values is automatically updated whenever the target stops.

If you close the watch pane then any watched variables that you might have added to it are lost. However if you save your current configuration (**File > Save Config**) and it contains a watch pane with watches, then these are preserved between debug sessions.

To modify a variable, double-click on the value in the Value field and edit the value in place.

Watch pane shortcut menu



Note: In the third group of menu options, the options Delete watch and Decimal/Hex will only appear when a variable is currently being watched and is displayed in the watch pane.

Main CPU	Changes to show the values of the variables currently being watched on the main CPU. This is reflected in the window title. The background color of the watch pane is reset to white.
IOP	Changes to show the values of the variables currently being watched on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
VU0	Changes to show the values of the variables currently being watched on the PlayStation 2 VU0 unit. The name of the unit being viewed is displayed in the window title.
VU1	Changes to show the values of the variables currently being watched on the PlayStation 2 VU1 unit. The name of the unit being viewed is displayed in the window title.
Browse vars	Causes a Variable Browser to be displayed that enables you to view all the possible variables in your application, according to their C++ class, and select the exact variable you require to be added to the watch pane.
Add watch	Enables you to enter the name of a variable to be watched. A dialog is displayed into which you must type the variable to be watched.
Delete watch	Removes the selected watch variable from the watch pane.
Hardware Break (C var)	Adds a hardware breakpoint on the selected item.
Paste	Upon selection, a new watch expression will be

Expression	added to the pane using the contents of the clipboard. This menu item will only appear when the clipboard contains text information.
Decimal / Hex	Enables you to toggle the display of the currently selected watch variable between decimal and hexadecimal.
Grid Lines	When checked, displays a table grid separating rows and columns in the watch pane, to improve readability.

To add watched variables

1. To add a watch, click **Add Watch** in the shortcut menu.

2. In the dialog that appears enter the name of the variable that you wish to watch. The watch is added to the new watch pane, and a value is automatically displayed from the target (if it can be obtained).

Note: If you close the watch window you will lose any watches you have added. However if you save the configuration and restart the Debugger with a watch pane in it, the watches are maintained (see "Configuring the user interface" on page 28).

To quickly add a watch

To quickly add a watch to the topmost watch pane, for example while single-stepping in a source pane, use the Quick watch accelerator key:

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.

2. In the list box at the top right, select **Application**.

3. Scroll down the **Command Name** list until the accelerator key setting for **Quick watch** is shown and then note the appropriate **Key Sequence** (if any).

4. Press **Cancel** to close the Application Setting dialog.

5. Press the key sequence you need for quickly adding a watch.

To expand or collapse watches or local variables

If a local variable or watched variable has a + sign shown next to it, it is an array, structure or class which can be expanded to show its members and their values.

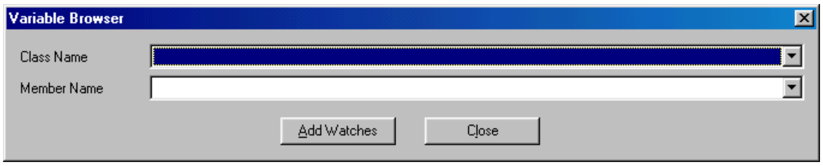
The quick way to expand the array or structure is to double-click on it in the locals or watch view. In addition you can quickly collapse an expanded variable by double-clicking it again.

Alternatively you can use the shortcut menu options **Expand Watch/Local** and **Collapse Watch/Local** to expand or collapse a variable that is an array or structure.

To add watches using the variable browser

You may not know the exact name of the variable that you wish to watch, or in addition you may wish to view a variable that belongs to a particular C++ class in your application (variables may have the same name, in different classes). If this is the case, then you can use the C++ variable browser to view all the variables in your application, grouped by class.

1. Click **Browse Vars** in the watch pane shortcut menu, and the following dialog is displayed:



2. If your application is programmed in C++ then you can select the class that contains your required variable in the **Class Name** field, or select Global to see all the variables in your application. If your application is programmed in C then there are no classes shown in this field.

3. In the **Member Name** field you can browse the drop-down menu to see a list of variables in the selected class, or all the variables if Global was selected.

   Select All to add all the static member variables in the selected class to your watch pane, or to add all the application variables if Global was selected.

4. Once you have specified the variable or set of variables that you wish to add click the **Add Watches** button. The dialog disappears and the selected variables should have been added to your watch pane. Use the **Exit** button to quit the dialog without adding any watches to the watch pane.

Chapter 8: Viewing the call stack and thread information

Viewing the call stack

You can view the current state of the program stack in a call stack pane. This displays the sequence of program functions that have been called by the application to arrive at the current program counter location.

To view the call stack

To view the call stack:

- Click **New Window > CallStack** in the **Window** menu or click the **CallStack view** button.

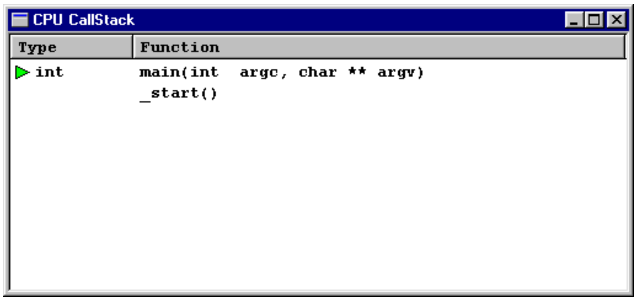
  This will open a new window containing a call stack pane that shows you current state of the program stack.

Call stack pane

The call stack pane enables you to view the contents of the program stack at any time, and change the current Debugger context by selecting a particular function call.

Whenever your application stops running, the call stack pane is updated with the functions that have been called by the application to arrive at the current program counter location.

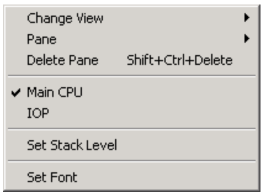
The topmost function listed in the call stack pane is the most recently called function.



Type	Function
int	main(int argc, char ** argv)
	_start()

To change the Debugger context you can either double-click on the required function, or select it and use **Set Stack Level** option in the shortcut menu. The > character indicates the context currently being viewed in the call stack. Once you have set the new context to a particular function call all the Debugger panes update to show the information that was on the target when that function was called.

Call stack pane shortcut menu



Main CPU	Changes to show the call stack on the main CPU. This is reflected in the window title.
IOP	Changes to show the call stack on the PlayStation 2 IOP unit. The name of the unit being viewed is displayed in the window title.
Set Stack Level	Enables you to change the Debugger context to the state of the target when the selected function was called. For this command to have any effect, a function call must have been selected in the call stack pane. Once the Debugger context has been changed, a > character is set in front of it to indicate the current stack context.

Viewing thread information

You can view thread information, for both the EE and IOP processors, in separate kernel panes. The panes are divided into various views that display information about all the threads running on the target.

To view thread information

To view thread information:

1. Click **New Window > Kernel** in the **Window** menu or click the **Kernel view** button.

   This will open a new window containing a kernel pane.

2. Select the tabbed view representing the type of thread information you wish to view.

3. Click **Main CPU** or **IOP** according to the type of processor you wish to view.

The kernel pane displays information about all the threads running on the target. You can view kernel information for both the EE and the IOP processors.

EE Kernel pane

Note: For this to work on the EE you must be using at least version 2.31.15 of ProDG Target Manager for PlayStation 2.

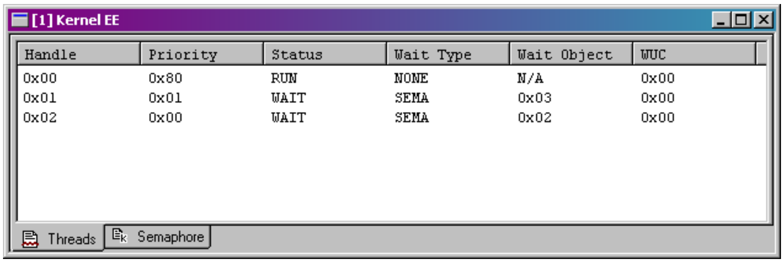
Each processor has a number of views and each view shows information about a certain type of kernel object. Each view has a set of headings appropriate for that object. You can click any heading to sort the displayed information according to the specified heading.

The EE processor has two views:

- Thread view
- Semaphore view

Thread view

The Thread view shows a list of the threads currently running on the EE processor and which semaphores (if any) they are waiting on.



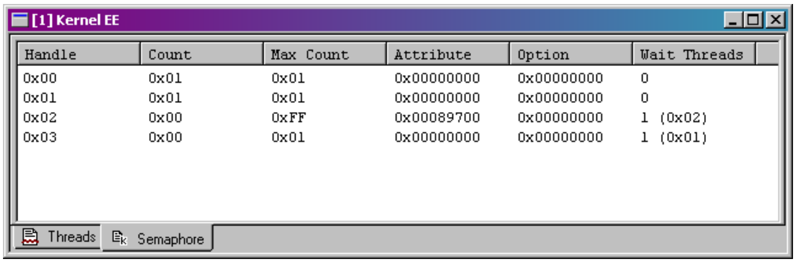
Handle	Priority	Status	Wait Type	Wait Object	WUC
0x00	0x80	RUN	NONE	N/A	0x00
0x01	0x01	WAIT	SEMA	0x03	0x00
0x02	0x00	WAIT	SEMA	0x02	0x00

Handle	The Thread ID returned by CreateThread().
Priority	The current priority of the thread.
Status	The status of the thread last time the Debugger had control of the EE. This can be:
	RUN - thread is running
	READY - ready for execution - thread is in stand-by as the CPU is executing another thread
	WAIT - thread is currently waiting (see **Wait Type** to determine why the thread is waiting)
	SUSPEND - thread was forced into a suspend state (e.g. SuspendThread)
	WAITSUSPEND - double-wait state - the thread was forced into a waiting state by another thread while it was in a WAIT state
	DORMANT - thread is dormant: the thread has been created but has not been started yet, or the thread

	has terminated but has not yet been deleted.
Wait Type	Only applicable if thread is in WAIT or WAITSUSPEND. **Wait Type** specifies the type of wait of the kernel object that the thread is waiting on: SLEEP - WAIT state due to SleepThread() SEMA - Semaphore WAIT state
Wait Object	If **Wait Type** is SEMA, the handle of the semaphore that the thread is waiting on.
WUC	Unprocessed WakeupThread() count.

Semaphore view

The Semaphore view shows a list of the semaphores on the EE processor and which threads (if any) are waiting on them.



Handle	Count	Max Count	Attribute	Option	Wait Threads
0x00	0x01	0x01	0x00000000	0x00000000	0
0x01	0x01	0x01	0x00000000	0x00000000	0
0x02	0x00	0xFF	0x00089700	0x00000000	1 (0x02)
0x03	0x00	0x01	0x00000000	0x00000000	1 (0x01)

Handle	The Semaphore ID returned by CreateSema().
Count	Semaphore current value.
Max Count	Maximum value for semaphore.
Attribute	The current semaphore attribute - either SA_THFIFO (Enqueue waiting threads using FIFO) or SA_THPRI (Enqueue waiting threads according to the thread priority).
Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object.

IOP kernel pane

Note: For this to work on the IOP, SNDBGEXT.IRX must be in the same directory as the debugger.

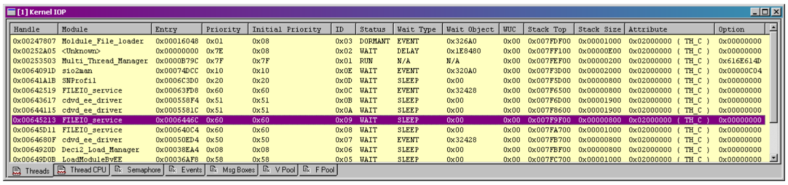
The IOP processor has seven views:

- Thread view
- Thread CPU view
- Semaphore view
- Events view

- Message Box view
- V Pool view
- F Pool view

Thread view

The Thread view shows a list of all the threads currently running on the IOP processor and which kernel objects (if any) they are waiting on.



[1] Kernel IOP

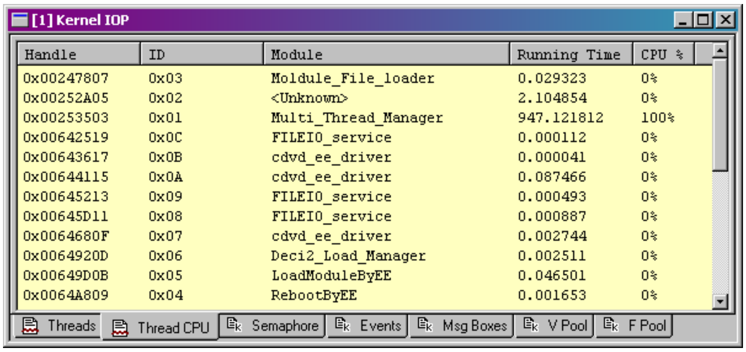
Handle	Module	Entry	Priority	Initial Priority	ID	Status	Wait Type	Wait Object	WUC	Stack Top	Stack Size	Attribute	Option
0x00247807	Moldule_File_loader	0x00016048	0x01	0x08	0x03	DORMANT	EVENT	0x326A0	0x00	0x007FDF00	0x00001000	0x02000000 (TH_C)	0x00000000
0x00252A05	<Unknown>	0x00000000	0x7E	0x08	0x02	WAIT	DELAY	0x1E8480	0x00	0x007FF100	0x00000E00	0x02000000 (TH_C)	0x00000000
0x00253503	Multi_Thread_Manager	0x0000B79C	0x7F	0x7F	0x01	RUN	N/A	N/A	0x00	0x007FEF00	0x00000200	0x02000000 (TH_C)	0x616E614D
0x0064091D	sio2man	0x00074DCC	0x10	0x10	0x0E	WAIT	EVENT	0x320A0	0x00	0x007F3D00	0x00002000	0x02000000 (TH_C)	0x00000C04
0x00641A1B	SNProfil	0x0006C3D0	0x20	0x20	0x0D	WAIT	SLEEP	0x00	0x00	0x007F5D00	0x00000800	0x02000000 (TH_C)	0x00000000
0x00642519	FILEIO_service	0x00063FD8	0x60	0x60	0x0C	WAIT	EVENT	0x32428	0x00	0x007F6500	0x00000800	0x02000000 (TH_C)	0x00000000
0x00643617	cdvd_ee_driver	0x000558F4	0x51	0x51	0x0B	WAIT	SLEEP	0x00	0x00	0x007F6D00	0x00001900	0x02000000 (TH_C)	0x00000000
0x00644115	cdvd_ee_driver	0x0005581C	0x51	0x51	0x0A	WAIT	SLEEP	0x00	0x00	0x007F8600	0x00001900	0x02000000 (TH_C)	0x00000000
0x00645213	FILEIO_service	0x0006446C	0x60	0x60	0x09	WAIT	SLEEP	0x00	0x00	0x007F9F00	0x00000800	0x02000000 (TH_C)	0x00000000
0x00645D11	FILEIO_service	0x000640C4	0x60	0x60	0x08	WAIT	SLEEP	0x00	0x00	0x007FA700	0x00001000	0x02000000 (TH_C)	0x00000000
0x0064680F	cdvd_ee_driver	0x00050ED4	0x50	0x50	0x07	WAIT	EVENT	0x32428	0x00	0x007FB700	0x00000800	0x02000000 (TH_C)	0x00000000
0x0064920D	Deci2_Load_Manager	0x00038EA4	0x08	0x08	0x06	WAIT	SLEEP	0x00	0x00	0x007FBF00	0x00000800	0x02000000 (TH_C)	0x00000000
0x00649D0B	LoadModuleByEE	0x00036AF8	0x58	0x58	0x05	WAIT	SLEEP	0x00	0x00	0x007FC700	0x00001000	0x02000000 (TH_C)	0x00000000

Handle	A unique identifier for the thread.
Module	The name of the module that contain's the thread's code.
Entry	Entry point of the thread function.
Priority	The current priority of the thread.
Initial Priority	The initial priority of the thread.
ID	The numerical ID of the thread (usually from creation order).
Status	The status of the thread last time the Debugger had control of the IOP. This can be: RUN - thread is running READY - ready for execution - thread is in stand-by as the CPU is executing another thread WAIT - thread is currently waiting (see **Wait Type** to determine why the thread is waiting) SUSPEND - thread was forced into a suspend state (e.g. SuspendThread) WAITSUSPEND - double-wait state - the thread was forced into a waiting state by another thread while it was in a WAIT state DORMANT - thread is dormant: the thread has been created but has not been started yet, or the thread has terminated but has not yet been deleted.
Wait Type	Only applicable if thread is in WAIT or WAITSUSPEND. **Wait type** specifies the type of the kernel object that the thread is waiting on. SLEEP - WAIT state due to SleepThread() DELAY - WAIT state due to DelayThread()

	SEMA - Semaphore WAIT state
	EVENTFLAG - Event flag WAIT state
	MBX - Message box WAIT state
	VPL - Variable-length memory pool acquisition WAIT state
	FPL - Fixed-length memory block acquisition WAIT state
Wait Object	The handle of the kernel object the thread is waiting on.
WUC	Unprocessed WakeupThread() count.
Stack Top	The start address of the area of memory used as stack space for this thread (they grow downwards).
Stack Size	The size in bytes of the area of memory allocated for this thread to use as its stack.
Attribute	The current thread attribute.
Option	User defined data.

Thread CPU view

The Thread CPU view is refreshed on a timer and shows the CPU usage and the thread running time.

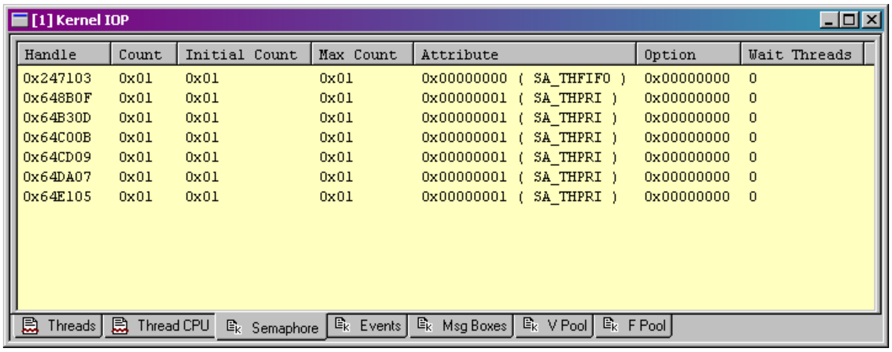


Handle	ID	Module	Running Time	CPU %
0x00247807	0x03	Moldule_File_loader	0.029323	0%
0x00252A05	0x02	<Unknown>	2.104854	0%
0x00253503	0x01	Multi_Thread_Manager	947.121812	100%
0x00642519	0x0C	FILEIO_service	0.000112	0%
0x00643617	0x0B	cdvd_ee_driver	0.000041	0%
0x00644115	0x0A	cdvd_ee_driver	0.087466	0%
0x00645213	0x09	FILEIO_service	0.000493	0%
0x00645D11	0x08	FILEIO_service	0.000887	0%
0x0064680F	0x07	cdvd_ee_driver	0.002744	0%
0x0064920D	0x06	Deci2_Load_Manager	0.002511	0%
0x00649D0B	0x05	LoadModuleByEE	0.046501	0%
0x0064A809	0x04	RebootByEE	0.001653	0%

Handle	A unique identifier for the thread.
ID	The numerical ID of the thread (usually from creation order).
Module	The name of the module that contain's the thread's code.
Running Time	Shows the total amount of time that this thread has spent in the RUN state since the target was last reset. The time is shown in seconds with decimal places down to microseconds.
CPU %	Shows the percentage of CPU time allocated to running this thread in this sample period.

Semaphore view

The Semaphore view displays a list of the semaphores on the IOP processor and which threads (if any) are waiting on them.



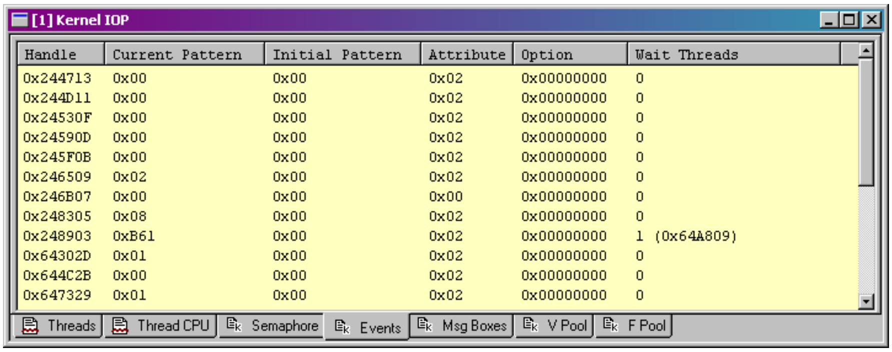
[1] Kernel IOP

Handle	Count	Initial Count	Max Count	Attribute	Option	Wait Threads
0x247103	0x01	0x01	0x01	0x00000000 (SA_THFIFO)	0x00000000	0
0x648B0F	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0
0x64B30D	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0
0x64C00B	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0
0x64CD09	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0
0x64DA07	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0
0x64E105	0x01	0x01	0x01	0x00000001 (SA_THPRI)	0x00000000	0

Handle	The Semaphore ID returned by CreateSema().
Count	Semaphore current value.
Initial Count	Initial value as specified in CreateSema().
Max Count	Maximum value for semaphore.
Attribute	The current semaphore attribute. This can either be SA_THFIFO (Enqueue waiting threads using FIFO) or SA_THPRI (Enqueue waiting threads according to the thread priority).
Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object.

Events view

The Events view shows a list of the events on the IOP processor and which threads (if any) are waiting on them.



[1] Kernel IOP

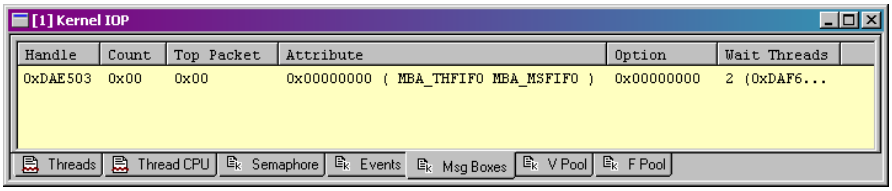
Handle	Current Pattern	Initial Pattern	Attribute	Option	Wait Threads
0x244713	0x00	0x00	0x02	0x00000000	0
0x244D11	0x00	0x00	0x02	0x00000000	0
0x24530F	0x00	0x00	0x02	0x00000000	0
0x24590D	0x00	0x00	0x02	0x00000000	0
0x245F0B	0x00	0x00	0x02	0x00000000	0
0x246509	0x02	0x00	0x02	0x00000000	0
0x246B07	0x00	0x00	0x00	0x00000000	0
0x248305	0x08	0x00	0x02	0x00000000	0
0x248903	0xB61	0x00	0x02	0x00000000	1 (0x64A809)
0x64302D	0x01	0x00	0x02	0x00000000	0
0x644C2B	0x00	0x00	0x02	0x00000000	0
0x647329	0x01	0x00	0x02	0x00000000	0

Handle	The Event Flag ID returned by CreateEventFlag().
Current Pattern	The current bit pattern of the Event Flag.

Initial Pattern	The initial bit pattern of the Event Flag (as specified in call to CreateEventFlag()).
Attribute	The current Event Flag attribute. This can be either EA_SINGLE (Multiple thread waits are not permitted) or EA_MULTI (Multiple thread waits are permitted).
Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object

Message box view

The Message Box view displays a list of the message boxes on the IOP processor and which threads (if any) are waiting on them.

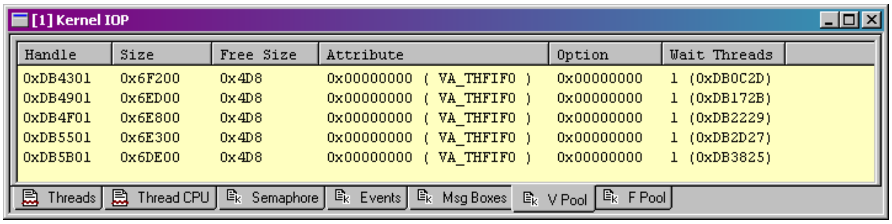


Handle	Count	Top Packet	Attribute	Option	Wait Threads
0xDAE503	0x00	0x00	0x00000000 (MBA_THFIFO MBA_MSFIFO)	0x00000000	2 (0xDAF6...

Handle	The Message Box ID returned by CreateMbx().
Count	The number of packets in msg box.
Top Packet	The address of top packet in msg box.
Attribute	The attributed specified in call to CreateMbx() MBA_THFIFO - Enqueue waiting threads using FIFO. MBA_THPRI - Enqueue waiting threads according to the thread priority. MBA_MSFIFO - Enqueue messages using FIFO. MBA_MSPRI - Enqueue messages according to message priority.
Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object

V Pool view

The V Pool view displays a list of the V pools on the IOP processor and which threads (if any) are waiting on them.

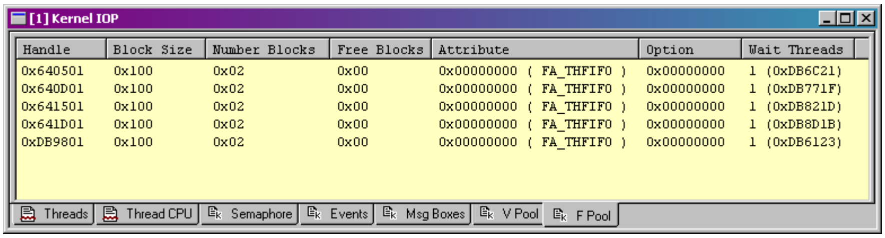


Handle	Size	Free Size	Attribute	Option	Wait Threads
0xDB4301	0x6F200	0x4D8	0x00000000 (VA_THFIFO)	0x00000000	1 (0xDB0C2D)
0xDB4901	0x6ED00	0x4D8	0x00000000 (VA_THFIFO)	0x00000000	1 (0xDB172B)
0xDB4F01	0x6E800	0x4D8	0x00000000 (VA_THFIFO)	0x00000000	1 (0xDB2229)
0xDB5501	0x6E300	0x4D8	0x00000000 (VA_THFIFO)	0x00000000	1 (0xDB2D27)
0xDB5B01	0x6DE00	0x4D8	0x00000000 (VA_THFIFO)	0x00000000	1 (0xDB3825)

Handle	The Pool ID returned by CreateVpl().
Size	Maximum number of bytes that can be allocated from the memory pool. This is the value obtained by subtracting the memory pool management area size from the memory pool size that was specified in CreateVpl().
Free Size	The number of unused bytes of memory in the memory pool.
Attribute	The attributed specified in call to CreateVpl() VA_THFIFO - Enqueue waiting threads using FIFO. VA_THPRI - Enqueue waiting threads according to the thread priority. VA_MEMBTM - Allocate the memory pool in the direction from the bottom of memory (high addresses).
Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object.

F Pool view

The F Pool view shows a list of F pools on the IOP processor and which threads (if any) are waiting on them.



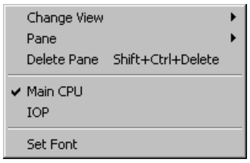
Handle	Block Size	Number Blocks	Free Blocks	Attribute	Option	Wait Threads
0x640501	0x100	0x02	0x00	0x00000000 (FA_THFIFO)	0x00000000	1 (0xDB6C21)
0x640D01	0x100	0x02	0x00	0x00000000 (FA_THFIFO)	0x00000000	1 (0xDB771F)
0x641501	0x100	0x02	0x00	0x00000000 (FA_THFIFO)	0x00000000	1 (0xDB821D)
0x641D01	0x100	0x02	0x00	0x00000000 (FA_THFIFO)	0x00000000	1 (0xDB8D1B)
0xDB9801	0x100	0x02	0x00	0x00000000 (FA_THFIFO)	0x00000000	1 (0xDB6123)

Handle	The Pool ID returned by CreateFpl().
Block Size	The memory block size (in bytes) that was set by CreateFpl().
Number Blocks	Number of memory blocks that was set by CreateFpl().
Free Blocks	Number of unused memory blocks within the memory pool.
Attribute	The attribute specified in call to CreateFpl(). FA_THFIFO - Enqueue waiting threads using FIFO. FA_THPRI - Enqueue waiting threads according to the thread priority. FA_MEMBTM - Allocate the memory pool in the direction from the bottom of memory (high addresses).

Option	User defined data.
Wait Threads	The list of handles of threads currently waiting on this object.

Kernel pane shortcut menus

Right-clicking on a **Kernel** pane displays the following shortcut menu:



Main CPU	Changes to show kernel information for the main CPU. This is reflected in the window title
IOP	Changes to show kernel information for the IOP unit. This is reflected in the window title

Chapter 9: Viewing TTY output

Viewing TTY output

You can view TTY output from your program in a TTY pane. This displays debug and other printf-type output. Using the shortcut menu, the TTY pane can be configured to show all TTY output, or just EE, IOP, DBG and LOG output.

Buffer sizes can be set independently for each of the TTY buffers (ALL, EE, IOP, DBG and LOG). The new buffer sizes are saved in the configuration file and are restored each time the debugger is executed. The 'All TTY' buffer defaults to 64KB and all the other TTY buffers are 32KB.

To view TTY output

To view TTY output:

- Click **New Window > TTY** in the **Window** menu or click the **TTY console view** button.

  This will open a new window containing a TTY pane that can then be configured to filter certain types of TTY stream, using shortcut menu options.

TTY pane

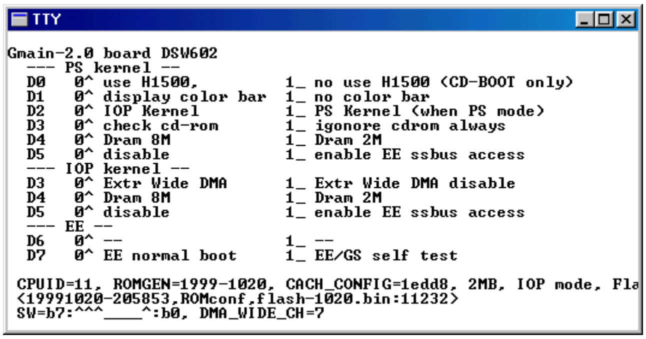
The TTY pane shows any standard output generated by the PlayStation 2. If you have inserted any printf commands in your source code, the output from these can be viewed in the TTY pane.

The TTY pane output is shown in a different color for each TTY channel .

By default the TTY pane shows the output from the PlayStation 2 EE CPU and the IOP. However you can change the TTY pane properties to view one or the other output stream if required.

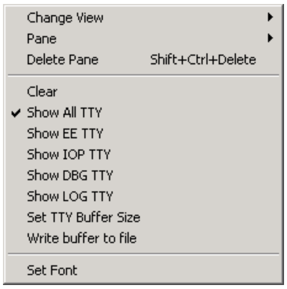
You can scroll up and down in the pane to view previous output.

Note: ProDG Target Manager also enables you to view the different TTY output streams.



```
TTY
Gmain-2.0 board DSW602
  --- PS kernel --
  D0    0^ use H1500,         1_ no use H1500 (CD-BOOT only)
  D1    0^ display color bar  1_ no color bar
  D2    0^ IOP Kernel         1_ PS Kernel (when PS mode)
  D3    0^ check cd-rom       1_ igonore cdrom always
  D4    0^ Dram 8M            1_ Dram 2M
  D5    0^ disable            1_ enable EE ssbus access
  --- IOP kernel --
  D3    0^ Extr Wide DMA      1_ Extr Wide DMA disable
  D4    0^ Dram 8M            1_ Dram 2M
  D5    0^ disable            1_ enable EE ssbus access
  --- EE --
  D6    0^ --                 1_ --
  D7    0^ EE normal boot     1_ EE/GS self test

 CPUID=11, ROMGEN=1999-1020, CACH_CONFIG=1edd8, 2MB, IOP mode, Fla
 <19991020-205853,ROMconf,flash-1020.bin:11232>
 SW=b7:^^^____^:b0, DMA_WIDE_CH=7
```

TTY pane shortcut menu



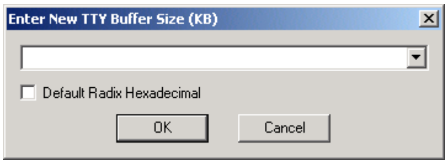
Clear	Clears the TTY buffers completely.
Show All TTY	When checked, displays all the PlayStation 2 output streams in the TTY pane.
Show EE TTY	When checked, filters the current contents of the TTY pane to show just the output generated by the PlayStation 2 EE CPU output stream.
Show IOP TTY	When checked, filters the current contents of the TTY pane to show just the output generated by the PlayStation 2 IOP output stream.
Show DBG TTY	When checked, filters to display ProDG Debugger status bar messages.
Show LOG TTY	When checked, filters to display a history of ProDG Debugger events, for example when a step has taken place, at what address, whether a breakpoint has been hit, and so on.
Set TTY buffer size	When selected a dialog appears prompting for the new buffer size (in KB). When **OK** is clicked, the new buffer size is applied to the currently selected TTY buffer. The buffer will be cleared if it is reduced in size, otherwise it will retain its contents.
Write buffer to file	Allows you to write the contents of the TTY pane (including history), to a file.

To resize the TTY buffers

You can resize a TTY buffer as follows:

1. Right-click on the TTY pane to access its shortcut menu. One of the TTY buffer options will be selected (**ALL**, **EE**, **IOP**, **DBG** or **LOG**).

2. Click on the option **Set TTY buffer size**. The Enter New TTY Buffer Size dialog box is displayed:



3. In the edit box provided, enter a value in KB. Any value will be accepted, but values <= 32 will set the buffer size to the minimum value of 32KB, while values >= 4096 will set the buffer size to the maximum value of 4096KB.

4. When **OK** is clicked, the new buffer size is applied to the currently selected TTY buffer. The buffer will be cleared if it is reduced in size, otherwise it will retain its contents.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 10: Basic EE profiling

Overview of EE profiling

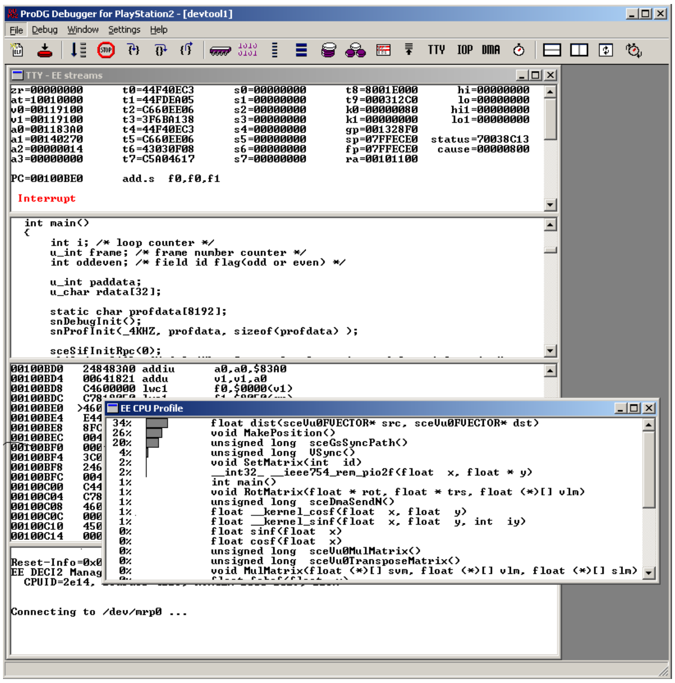
ProDG Debugger for PlayStation 2 v1.42 and later includes a new Debugger window type – the profile pane. This pane allows you to easily get a quick real-time view of what is consuming EE core CPU time in your PlayStation 2 application, with minimal impact on EE performance. The pane includes control features to allow you to restrict the profile data collection and accumulate results over time, while the PlayStation 2 library provides a simple API to allow your application to control its own profiling.

This profiler is provided in response to developer requests and is intended for those "why did that slow down then?" moments. It is not meant to provide a detailed system-wide analysis of your entire game run. Such detailed profiling is largely beyond the scope of this simple Debugger add-on because PlayStation 2 system performance is dependent upon a lot more than just EE code. There is usually little point fine-tuning EE code beyond this level as EE cache misses and pipeline usage are not usually significant PlayStation 2 performance bottlenecks.

The profile pane collects and analyses data produced by a small efficient interrupt handler that runs on the PS2 EE core. It does not use Timer0 or Timer1 so they are still completely free for your own use. The profiler runs a regular interrupt that effectively random samples your code to see where it is spending most time. This data is collected on the EE and occasionally shipped back to the Debugger to be analysed against symbol file data for the running application.

Profile pane

The profile pane will update in real time to show where the CPU is spending time during the last profile interval. By default the display will be textual and sorted into descending order (hungriest functions at the top). The standard display is a bit dynamic so the Debugger provides you with a few options to smooth out that display and control the update.



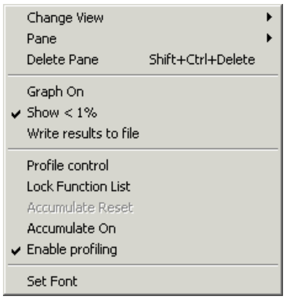
```
zr=00000000      t0=44F40EC3      s0=00000000      t8=8001E000       hi=00000000
at=10010000      t1=44FDEA05      s1=00000000      t9=000312C0       lo=00000000
v0=00119100      t2=C660EE06      s2=00000000      k0=00000080      hi1=00000000
v1=00119100      t3=3F6BA138      s3=00000000      k1=00000000      lo1=00000000
a0=001183A0      t4=44F40EC3      s4=00000000      gp=001328F0
a1=00140270      t5=C660EE06      s5=00000000      sp=07FFECE0  status=70038C13
a2=00000014      t6=43030F08      s6=00000000      fp=07FFECE0   cause=00000800
a3=00000000      t7=C5A04617      s7=00000000      ra=00101100

PC=00100BE0      add.s  f0,f0,f1

 Interrupt
```

```
  int main()
  {
      int i; /* loop counter */
      u_int frame; /* frame number counter */
      int oddeven; /* field id flag(odd or even) */

      u_int paddata;
      u_char rdata[32];

      static char profdata[8192];
      snDebugInit();
      snProfInit(_4KHZ, profdata, sizeof(profdata) );

      sceSifInitRpc(0);
```

```
00100BD0   248483A0 addiu      a0,a0,$83A0
00100BD4   00641821 addu       v1,v1,a0
00100BD8   C4600000 lwc1       f0,$0000(v1)
00100BDC   C78
00100BE0  >460
00100BE4   E44
00100BE8   8FC
00100BEC   004
00100BF0   000
00100BF4   3C0
00100BF8   246
00100BFC   004
00100C00   C44
00100C04   C78
00100C08   460
00100C0C   000
00100C10   450
00100C14   000

Reset-Info=0x0
EE DECI2 Manag
  CPUID=2e14,

Connecting to /dev/mrp0 ...
```

```
 34%      float dist(sceVu0FVECTOR* src, sceVu0FVECTOR* dst)
 26%      void MakePosition()
 20%      unsigned long  sceGsSyncPath()
  4%      unsigned long  VSync()
  2%      void SetMatrix(int  id)
  2%      __int32_ __ieee754_rem_pio2f(float  x, float * y)
  1%      int main()
  1%      void RotMatrix(float * rot, float * trs, float (*)[] vlm)
  1%      unsigned long  sceDmaSendN()
  1%      float __kernel_cosf(float  x, float  y)
  1%      float __kernel_sinf(float  x, float  y, int  iy)
  0%      float sinf(float  x)
  0%      float cosf(float  x)
  0%      unsigned long  sceVu0MulMatrix()
  0%      unsigned long  sceVu0TransposeMatrix()
  0%      void MulMatrix(float (*)[] svm, float (*)[] vlm, float (*)[] slm)
```

Profile pane shortcut menu



Note: The **Accumulate Reset** option is only available during profiling.

Graph On	When checked, the second column of the window shows a graphical display of the percentage CPU time *vs.* a decimal total sample count. The graphical representation is sometimes a clearer indicator especially where profile data is very dynamic.
Show < 1%	Allows you to selectively display just items consuming 1% or more of the CPU time. Particularly with small sample sets the lower percentages are not statistically significant and should really be ignored.
Write results to file	Allows you to save the current profile data to disk as comma separated text. Each entry in the file contains the CPU time taken, the number of calls

	and the function name.
Profile Control	Brings up a dialog that allows you to set PlayStation 2 target-side profile options including the sample rate as well as profile address range, and masks which determine which profile samples are collected and which are rejected. If you leave these fields empty they will assume sensible defaults (accepted PC range 0 to 0xFFFFFFFF, mask = 0xFFFFFFFF).

Note: For more details of how to use the mask value to selectively collect data from specific parts of your program that you can select at runtime see details of the PlayStation 2 EE-side API (see *Power User guide ProDG for PlayStation 2 Build Tools*).

The sample rate directly affects the performance impact of profiling on your application and the rate of data collection and subsequent update of the profile display. If you raise the sample rate from 4KHz to say 20KHz then your display will update 5 times more often but the performance impact will be slightly greater. Higher rates allow you to better use larger profile buffers without losing the real-time responsiveness.

Lock Function List	The default is a fully dynamic function list i.e. new functions are added to the list as they are spotted. The down-side of this is that the list can be quite dynamic and more difficult to read. You can remove that problem by locking the current function list (so extra functions will all be filed under "Unknown") or by setting **Accumulate On**.
Accumulate Reset	Resets the accumulated counts if you feel you are losing the real-time nature as data is piling up.
Accumulate On	By default the Debugger throws away the results of each profile interval and starts again. This is handy when your PlayStation 2 application is moving through different game sections with different functions. If the set of functions is pretty constant whilst you are profiling though you can smooth it out some more by setting **Accumulate On**. In Accumulate mode the Debugger profile samples are cumulative – just added to previous intervals' data. The result is a much larger growing and more slowly changing sample set.
Enable Profiling	When checked, allows you to freeze the profile display for closer examination.

Troubleshooting

If your profile display shows significant amounts of time being spent in blocking waits like Vsync() or sceGsSyncPath() then you must understand that this represents wasted or spare CPU time. This is time when the CPU is idle waiting for external events and this is time that you could be putting to other use. If you profile most of the SCE sample programs you will find that most of the demonstrations actually consume very little EE core CPU time, spending most of their time in Vsync() or sceGsSyncPath().

[One exception is the VU1/IGA demonstration which actually spends a significant amount of time in collision detection, especially if you increase the number of shapes, e.g. edit the line in sample.c to "#define NBALLS 240"].

Your code must never write the count register [COP0, r9] during profiling. The profiler relies on it reaching certain calculated values in order to cause the next profiler interrupt. If you read it and just take the difference from the last time you read it you will get similar results without rewriting it. However, if you zero it then you will effectively exclude large sections of your code following that point (up to 14 seconds at a time) from profiling.

Targeting profiling to specific situations

You can define selected regions within your code where profiling will be active. You can specify the regions in terms of an address range passed to the snProfSetRange() function but in addition and more usefully, with careful use of the flags value you can selectively limit that profiling to particular intervals or times during the execution of your program.

For example, here is a very simple hypothetical situation where you set the flag values to allow you to see the CPU usage of code related to different player characters in a game. Imagine that the following sequence of code occurs inside your game main loop on the EE CPU:-

```
snProfSetRange( -1, 0, -1);
// set profiler to accept all flag values
snProfSetFlagValue(0x01);
 // do AI calculations for this one
ProcessCharacterAI( Player1Object);
snProfSetFlagValue(0x02);
 // and process the other character
ProcessCharacterAI( Player2Object);
 // set flags to "none of the above"
snProfSetRange( 4, 0, -1);
```

Now, whilst your game is running, without disturbing your executing program at all, from within the Debugger you can set the mask value to selectively show you profile data just for the processing of Player1 or just for Player2, or for both combined, or for everything else except Player 1 and Player 2. That would require you to set the mask value to 1,2,3, ~3 (i.e. 0xFFFFFFFC) respectively for those four conditions.

Note the next logical extension of the above is to set the flags value to 4…. **not** to set the value to 3. If you were to set the value to 3 the profiler would not be able to distinguish those samples from those with just bit 0 or bit 1 set. For example taken further you could do something like this to allow you to select, from the Debugger at runtime, the sample data for any one of 32 different objects:

```
// Process all 32 non-player bots
for(bot=0; bot<32; bot++)
{
snProfSetFlagValue( 1<<bot );
ProcessBotAI( BotData[bot] );
}
```

Within the Debugger, from the profile control dialog, you would set the mask value to 1 to profile just the first bot, to 2 to profile the second, to 4 to profile the third etc.

Note: The flag settings will be applied only to profile data collected after the call to set the flags. Therefore if the target is halted due to a breakpoint etc., and no more profile data is being collected, changing the flag will not affect the sample data already collected.

Known problems

There are currently some problems with PlayStation 2 SIF usage in that certain combinations of SIF access on the PlayStation 2 can cause the SIF library or operating system code to completely deadlock or cause wrong data to be passed across the SIF. This can happen at almost any time but is particularly likely to happen if there is a mixture of user SIF access, printf or host: file access, debug communications (this includes profiling or having continual update enabled in the Debugger).

If you suspect such problems then:

1. delete all profile panes
2. turn off continual update
3. restart the Debugger without creating any new profile panes

If the problem goes away then you are seeing SIF-related problems. Hopefully a future SCE library release will fix these issues. In the meantime SN Systems are researching a workaround.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 11: IOP debugging

Overview of IOP debugging

The PlayStation 2 IOP (I/O processor) is dedicated to handling low-level, often I/O-related, functions which can be safely offloaded from being the main (EE) processor's responsibility. In addition to EE code debugging, the ProDG Debugger provides the means of source-level debugging of IOP modules.

An IOP module is loaded onto the IOP in the form of an IRX file. This section describes the different ways to debug an IOP module.

Usually IOP modules are loaded by an application running on the EE that makes a call to one of the module load functions (e.g. sceSifLoadModule) with the filename of the IRX file as an argument:

```
if (sceSifLoadModule("host0:ezmidi.irx",0,0) < 0)
{
    printf("Failed to load ezmidi.irx\n");
}
```

It is also possible to download IRX modules via the Debugger or Target Manager.

To view the IOP modules

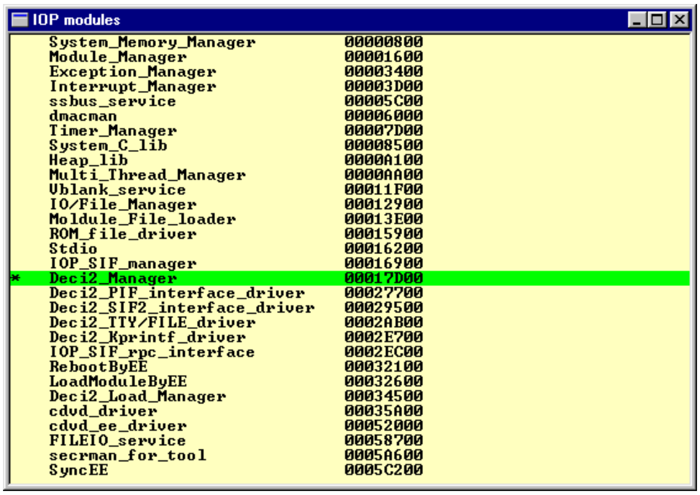
If you wish to view the IOP modules:

- Click **New Window > IOP modules** in the **Window** menu or click the **IOP modules view** button in the toolbar.

  A new window containing an IOP modules pane is opened.

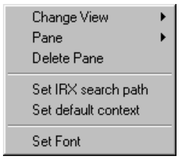
IOP Modules pane

The IOP modules pane shows a list of IOP modules currently loaded on the PlayStation 2. If you select a particular IOP module it becomes set as the default scope for expression evaluation (marked with an asterisk '*' so that the ProDG Debugger will know that if you specify a symbol like "start" or "main" you mean the one in that module). The IOP module which the PC is currently in is shown with a '>' marker.



```
   System_Memory_Manager          00000800
   Module_Manager                 00001600
   Exception_Manager              00003400
   Interrupt_Manager              00003D00
   ssbus_service                  00005C00
   dmacman                        00006000
   Timer_Manager                  00007D00
   System_C_lib                   00008500
   Heap_lib                       0000A100
   Multi_Thread_Manager           0000AA00
   Vblank_service                 00011F00
   IO/File_Manager                00012900
   Moldule_File_loader            00013E00
   ROM_file_driver                00015900
   Stdio                          00016200
   IOP_SIF_manager                00016900
*  Deci2_Manager                  00017D00
   Deci2_PIF_interface_driver     00027700
   Deci2_SIF2_interface_driver    00029500
   Deci2_TTY/FILE_driver          0002AB00
   Deci2_Kprintf_driver           0002E700
   IOP_SIF_rpc_interface          0002EC00
   RebootByEE                     00032100
   LoadModuleByEE                 00032600
   Deci2_Load_Manager             00034500
   cdvd_driver                    00035A00
   cdvd_ee_driver                 00052000
   FILEIO_service                 00058700
   secrman_for_tool               0005A600
   SyncEE                         0005C200
```

IOP modules pane shortcut menu



Set IRX search path	Enables you to set the search path for IRX files.
Set default context	Sets the selected IOP module (marked by an asterisk) as the default scope for expression evaluation.

Setting a module as the default scope

Double-clicking on an IOP module, or selecting a module and then using the shortcut menu option **Set default context**, causes the selected IOP module to be set as the default scope for expression evaluation.

This means that local variable names will be sought by the IOP Debugger within the selected module.

IOP Debugging

There are two different IOP debugging strategies:

- You can either debug an IOP module when it is loaded by stepping through code on the EE until you encounter the line of code that would load the required IOP module. At this point you can load the IRX file that corresponds to your IOP module and set a breakpoint at the start. This allows you to control execution from the point at which the IOP module is loaded. You can either step through the code or set breakpoints and browse locals or watches.

- Alternatively you can let your EE application load the module or you wish to debug an already loaded IRX module. In both cases you can debug the module by setting a breakpoint at the entry point of the IOP code, for example, the entry point of a function that is set to handle RPC requests originating from the EE. When you do this the IOP will stop each time a request is sent to the IOP allowing you to debug from that point.

More information on this is contained in the worked example at the end of this section (see "Debugging IOP modules in the Sony ezmidi sample" on page 124).

Loading and running IRX files

It is possible to load an IRX file onto the target at any time during Debugger use. You can only do this either after starting the Debugger, as there is no command-line option to load an IRX file.

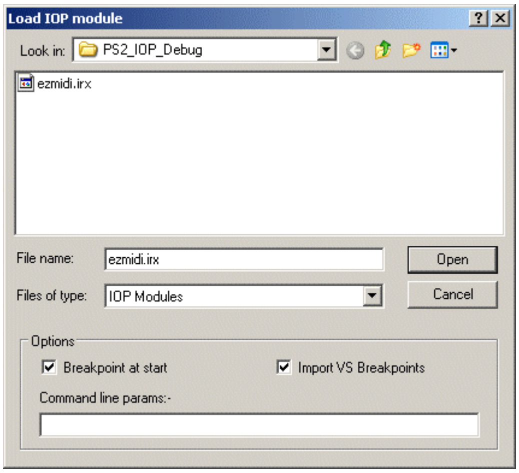
The main reason to load an IRX file during debugging is so that you can debug it directly from the point it is loaded onto the IOP processor. If you do this then you will need to select the **Breakpoint at Start** option in the Load IOP Module dialog.

Note: You can also load an IRX file using the Target Manager.

To load your application IRX file manually

The Debugger must have been started.

1. Click **Load IRX File** in the **File** menu.
2. In the dialog that is displayed locate and select the IOP module file that you wish to load and debug.



Look in: PS2_IOP_Debug
ezmidi.irx
File name: ezmidi.irx
Files of type: IOP Modules
☑ Breakpoint at start
☑ Import VS Breakpoints

3. Set any of the options that you require for the file load:

Breakpoint at start to set a breakpoint at the module entry point;

Command line params enables you to specify any command-line parameters for your IOP module.

4. Click **Open** to activate the load.

   You can now start debugging the newly loaded IOP module.

Setting the IRX search path

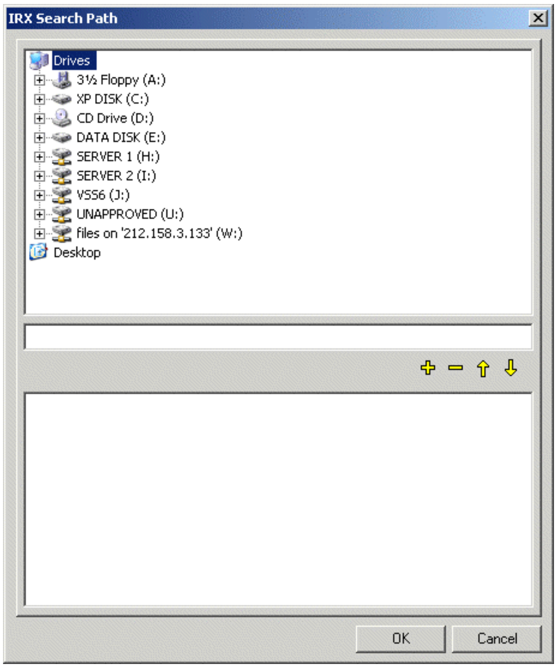
If you load the IRX while you are debugging then you do not need to set the IRX search path. However, if the game loads the IRX then this step will be necessary in order that the debugger can find the IRX symbols.

The Debugger looks for the IRX application source file by interrogating its IRX search path environment variable. The default value is ";", or the directory from which the ProDG Debugger was launched. In practice, you will need to set the IRX search path to the directory where the IRX source files are actually located.

To set the IRX search path

1. From the **Debug** menu, select **IRX Search Path**.

   The IRX Search Path dialog appears similar to the following:



2. In the dialog that appears use the tree view to select the full path(s) of any directories where the Debugger should look for your

IRX source files. You can add a selected directory to the search path by clicking on the [+] symbol, or remove a selected directory by clicking on the [-] symbol.

3. The order of directories in the search path can be altered by clicking on the up and down arrow symbols. To promote the selected directory click on the up arrow, to demote a directory click on the down arrow.

Creating new IOP Debugger windows

IOP debugging takes place by creating a split-pane container with separate Debugger panes, just as you did for EE debugging (see "Creating new Debugger windows" on page 12). Since the IOP modules are loaded by an application running on the EE, this would normally be set up in addition to an EE split-pane container.

To create a pane for monitoring the IOP, you create a pane as you would normally and then convert it to point to the IOP rather than the EE. Currently the following pane types can be switched to IOP debugging: Registers, Memory, Disassembly, Source, Breakpoints and CallStack.

To change a pane from EE to IOP monitoring

1. Right-click on the pane to display its shortcut menu.
2. If the option is available, click **IOP** to cause the pane to monitor IOP processes. Note that changing a TTY pane is rather different - you have to select **Show IOP TTY** from the shortcut menu instead.

The pane can be set back to EE monitoring by repeating the above steps only clicking **Main CPU** from the shortcut menu (or **Show EE TTY** for a TTY pane).

Quick IOP debugging

Here is a quick start guide to IOP debugging.

1. Allow your module to be loaded using sceSifLoadModule().
2. Open an IOP modules pane (see "To view the IOP modules" on page 119). If your module does not appear in the IOP Modules list then press the **Update all views** toolbar button.
3. Set the debugging context to the module you want to debug by double-clicking on it. An asterisk (*) will appear if the context has been set correctly.
4. Use the **Debug > IRX Search Path** menu option to add the directory of the IRX you wish to debug. The debugger should now have access to the debug symbols.

5. Now load a source file you want to debug and set a breakpoint. Alternatively, use the Go to address dialog box from an IOP source or disassembly window to jump to a function you wish to debug.

Debugging IOP modules in the Sony ezmidi sample

This section contains a worked example that uses the Sony ezmidi sample to illustrate how to debug IOP modules in the Debugger. It is described in steps that you can carry out once you have built the ezmidi EE and IOP sample code contained in \usr\local\sce\ee\sample\sound\ezmidi and \usr\local\sce\iop\sample\sound\ezmidi.

Before you start this example you will need to ensure that the code is built with no optimization and to generate debug information.

- For the ezmidi ELF build (in the EE tree), this means removing the -O2 and -G0 and adding –O0 and –g to the CFLAGS variable in your makefile:

  ```
  CFLAGS = -O0 -g -Wa,-al -fno-common
  ```

- For the ezmidi IRX build (in the IOP tree), this means replacing -G0 by –O0:

  ```
  CFLAGS = $(INCDIR) -I. -Wall -O0 -g
  ```

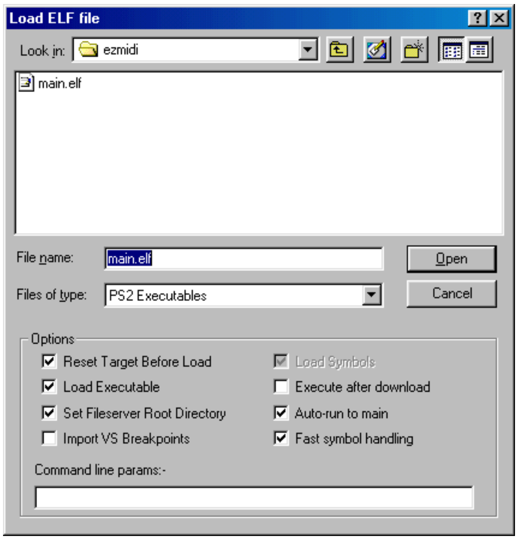
  In addition, you need to change the line which reads:
  ```
         $(LINK.o)  -o $@ \
  ```
  to
  ```
         $(LINK.o)  -r $@ \
  ```

  Both -r and -o are switches that are passed to IOPFIXUP; -o means 'convert without debug symbols' while -r means 'convert with debug symbol info'.

1. Start the ProDG Debugger from command line, windows shortcut, etc.

2. Set the IRX search path to the directory containing your newly built ezmidi.irx file using the **IRX Search Path** option in the **Debug** menu. Alternatively you can just move ezmidi.irx to the Debugger current working directory, which the IRX search path is set to by default to (;).

3. Load your newly built main.elf file, being sure to select the options shown in the following dialog:


Load ELF file

Look in: ezmidi

main.elf

File name: main.elf

Files of type: PS2 Executables

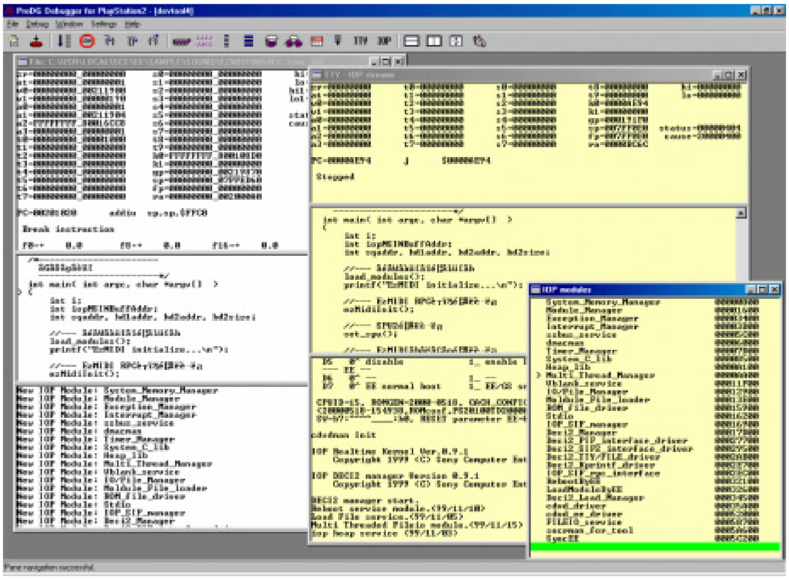
Options

- [✓] Reset Target Before Load
- [✓] Load Executable
- [✓] Set Fileserver Root Directory
- [ ] Import VS Breakpoints
- [✓] Load Symbols
- [ ] Execute after download
- [✓] Auto-run to main
- [✓] Fast symbol handling

Command line params:-

4. Set up some views in the Debugger. You will need to be able to see the following panes to effectively debug IOP modules: main CPU registers, main CPU source, IOP registers, IOP source, DBG TTY, IOP TTY and IOP modules.

   Your Debugger main window should now resemble the following:


ProDG Debugger for PlayStation2 - [devtool4]

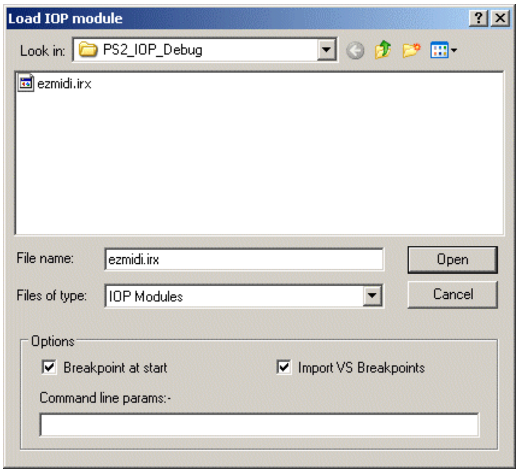
Pane navigation successful.

5. Now make the main CPU source pane the active pane and start to step through your code executing on the EE. Step into the load_modules function.

6. Continue stepping through the load_modules function. As each module is loaded you will notice that it appears in the IOP modules pane.

7. When you get to the line of code that is going to load ezmidi.irx, you have two choices: you can either debug it when it is loaded, or let the EE application load it and debug it later. The latter is the technique you would use if you were debugging an already loaded module. We will now describe debugging from the module start-up. If you wish to see how to debug the module after it has already been downloaded from the EE application, step through the code to load the ezmidi.irx and jump to step 13.

8. Click **Load IRX File** in the **File** menu, and select the ezmidi.irx file to download. Ensure that you select the **Breakpoint At Start** check box so that IOP is halted before it executes.



Look in: PS2_IOP_Debug

ezmidi.irx

File name: ezmidi.irx

Files of type: IOP Modules

☑ Breakpoint at start ☑ Import VS Breakpoints

The Debugger should switch to the IOP source pane to show the program counter entry point of the ezmidi module midi_ent.c. You will also notice the ezmidi module appear in the IOP modules pane, and in the DBG TTY pane you will see messages that explain what the Debugger is doing. You will also notice that "* >" appears next to ezmidi module in the IOP modules pane. The * indicates that the module is set as the default scope for expression evaluation and the > indicates that the program counter is currently in the module.

9. You can now start debugging the IOP module either by continuing stepping, setting breakpoints directly in the source or disassembly. You can also view the watches on the IOP as you step, by creating a new watch pane and setting it to view IOP. (By default if your active pane is viewing the IOP the new watch pane will automatically be set to view the IOP when created.)

10. As you step through the IOP source code, be careful not to step out of the module as the multi-threading on the IOP may cause you to lose the program counter.

11. Once you have finished debugging the ezmidi module in this way, you will need to return to the EE source so that we can look at the

second way of debugging IOP modules. To do this start the IOP running using the **Go** command in the **Debug** menu.

12. Once you have done this, return to the EE source pane. Because the ezmidi module was downloaded manually, you will need to move the program counter past the line of code that calls sceSifLoadModule by selecting the function return line and clicking **Set PC to cursor** in the pane shortcut menu. This is so that the module is not loaded twice.

13. Next, you need to single-step twice to get back to main(). You should now be stopped in the main() application with all the required IOP modules loaded. Note that the IOP modules had to be loaded in the correct order as they have dependencies on each other, so it would not have been possible to just load the ezmidi module at the start of debugging.

14. Before you continue you will need to set a breakpoint at the entry point of the ezmidi driver. This is so that when you set the target running again it will halt at the point at which it enters the IOP module that we wish to debug.

15. Go to the IOP source pane and click **Go to Address** in the shortcut menu. Enter midiFunc as the symbol to go to. This is the command entry point for the driver. When this function is shown set a new breakpoint there. If the Debugger cannot locate this function ensure that there is an asterisk next to the module in the list and that the IRX search path is set.

16. Now return to the EE source pane and step through the code until the program counter reaches the line of code: iopMSINBuffAddr = ezMidi(…).

17. Step over this line of code, and the EE will make an RPC call to the IOP. You will now notice that the EE will show that it is running because the line cannot complete yet. And if you look at the IOP source pane you will notice that it has halted at the breakpoint that we set. You are now able to switch to the IOP source pane and step through the code debugging as required (single step, browse locals, watches, etc.).

18. When you have finished debugging the IOP module click **Go** on the **Debug** menu to start your code running on the IOP processor and switch back to the EE source pane and you will notice that the step has been completed, and the program counter is on the next line of code.

19. If you continue stepping the EE it will send further commands to the ezmidi driver that will cause the entry point breakpoint to be hit again.

20. If you now remove all breakpoints and run both the EE and IOP processors, the application will run the midi demo and play music. At any time you can select the IOP source or disassembly pane and

put a breakpoint at the midiFunc entry point, which will cause the IOP to be halted whenever commands are sent to the IOP.

Chapter 12: VU and DMA debugging

VU and DMA debugging

On the PlayStation 2 there are two vector units, VU0 and VU1, for handling low-level vector graphic calculations.

VU debugging is carried out in a similar manner to IOP debugging. Various build options are needed in order to do VU debugging successfully, and these are described first.

Building for VU debugging

To make use of VU debugging you need to do the following:

1. Copy the files libsn.h and libsn.a files to your include and library directories respectively.

2. Include the header file libsn.h in your main program. e.g.

   ```
   #include <libsn.h>
   ```

3. Change your make file to link in the library libsn.a and ensure that libsn.a is the first item in your list of libraries so that the SN versions of some functions are used in preference to the libgraph ones.

   The new LIBSN contains replacement versions of sceGsResetPath() and sceGsSyncPath() which will not upset VU debugging. It also contains a workaround to allow the Debugger to access the accumulator registers in the EE, VU0 and VU1 processors.

Note: If your program uses a SCE-released sceGsResetPath() then that resets the FBRST bits that enable D-bit breakpoints. If that happens then your VU breakpoints will not be triggered.

4. In previous versions of ProDG for PlayStation 2 we required that you call the snDebugInit() function at the start of the function main() in your program and check the result is not 0, e.g.

   ```
   if (!snDebugInit())
     return 1;
   ```

   The call to snDebugInit() installed debug extensions which enabled the Debugger to access the accumulator registers in the EE, VU0 and VU1 floating point units, making it possible to see these register values in the register windows. However from v2.2 of ProDG for PlayStation 2 onwards this call is not necessary.

5. We strongly recommend that you use the SN Systems VU assembler ps2dvpas, rather than the GNU assembler ee-dvp-as (DVPASM variable). This is because ee-dvp-as's debug information is flawed and does not contain full path information. Be sure to use the -g switch on your ps2dvpas command line (if you don't see symbols in a VU disassembly after the MPG is transferred then this is probably the reason).

Creating new VU Debugger windows

VU debugging takes place by creating a split-pane container with separate Debugger panes, just as you did for EE debugging (see "Creating new Debugger windows" on page 12). Just as with IOP debugging, this would normally be set up in addition to an EE split-pane container.

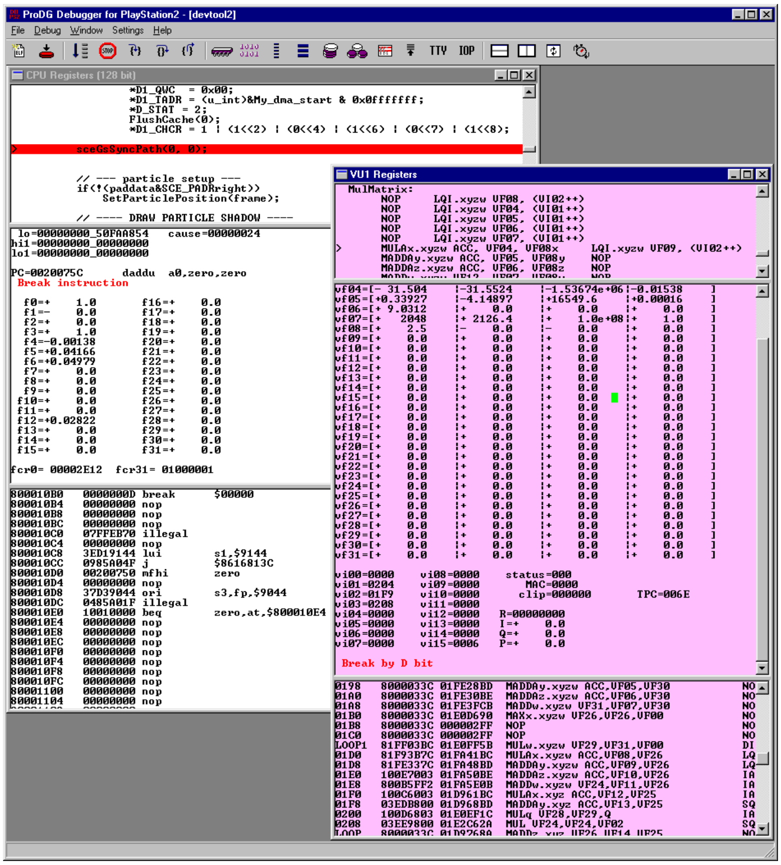
To create a pane for monitoring a VU processor, you create a pane as you would normally and then convert it to point to either VU0 or VU1 rather than the EE. Currently the following pane types can be switched to VU debugging: Registers, Memory, Disassembly, Source, and Breakpoints.

To change a pane from EE to VU monitoring

1. Right-click on the pane to display its shortcut menu.

2. If the option is available, click **VU0** or **VU1** to cause the pane to monitor VU0 or VU1 processes respectively. Note that there is no TTY output from the VU processors.

The following example shows a VU1 routine during debugging:


ProDG Debugger for PlayStation2 - [devtool2]
File Debug Window Settings Help
TTY IOP
CPU Registers (128 bit)
VU1 Registers

```
                 *D1_QWC  = 0x00;
                 *D1_TADR = (u_int)&My_dma_start & 0x0fffffff;
                 *D_STAT = 2;
                 FlushCache(0);
                 *D1_CHCR = 1 | (1<<2) | (0<<4) | (1<<6) | (0<<7) | (1<<8);

>          sceGsSyncPath(0, 0);


          // --- particle setup ---
          if(!(paddata&SCE_PADRright))
              SetParticlePosition(frame);

          // ---- DRAW PARTICLE SHADOW ----
```

```
 lo =00000000_50FAA854   cause=00000024
hi1 =00000000_00000000
lo1 =00000000_00000000

PC=0020075C      daddu  a0,zero,zero
 Break instruction

  f0=+    1.0       f16=+    0.0
  f1=-    0.0       f17=+    0.0
  f2=+    0.0       f18=+    0.0
  f3=+    1.0       f19=+    0.0
  f4=-0.00138       f20=+    0.0
  f5=+0.04166       f21=+    0.0
  f6=+0.04979       f22=+    0.0
  f7=+    0.0       f23=+    0.0
  f8=+    0.0       f24=+    0.0
  f9=+    0.0       f25=+    0.0
 f10=+    0.0       f26=+    0.0
 f11=+    0.0       f27=+    0.0
 f12=+0.02822       f28=+    0.0
 f13=+    0.0       f29=+    0.0
 f14=+    0.0       f30=+    0.0
 f15=+    0.0       f31=+    0.0

fcr0= 00002E12  fcr31= 01000001
```

```
800010B0   0000000D break      $00000
800010B4   00000000 nop
800010B8   00000000 nop
800010BC   00000000 nop
800010C0   07FFEB70 illegal
800010C4   00000000 nop
800010C8   3ED19144 lui        s1,$9144
800010CC   0985A04F j          $8616813C
800010D0   00200750 mfhi       zero
800010D4   00000000 nop
800010D8   37D39044 ori        s3,fp,$9044
800010DC   0485A01F illegal
800010E0   10010000 beq        zero,at,$800010E4
800010E4   00000000 nop
800010E8   00000000 nop
800010EC   00000000 nop
800010F0   00000000 nop
800010F4   00000000 nop
800010F8   00000000 nop
800010FC   00000000 nop
80001100   00000000 nop
80001104   00000000 nop
```

```
 MulMatrix:
       NOP      LQI.xyzw VF08, (VI02++)
       NOP      LQI.xyzw VF04, (VI01++)
       NOP      LQI.xyzw VF05, (VI01++)
       NOP      LQI.xyzw VF06, (VI01++)
       NOP      LQI.xyzw VF07, (VI01++)
>      MULAx.xyzw ACC, VF04, VF08x     LQI.xyzw VF09, (VI02++)
       MADDAy.xyzw ACC, VF05, VF08y    NOP
       MADDAz.xyzw ACC, VF06, VF08z    NOP
```

```
vf04=[- 31.504  |-31.5524  |-1.53674e+06|-0.01538   ]
vf05=[+0.33927  |-4.14897  |+16549.6    |+0.00016   ]
vf06=[+ 9.0312  |+    0.0  |+    0.0    |+    0.0   ]
vf07=[+   2048  |+ 2126.4  |+    1.0e+08|+    1.0   ]
vf08=[+    2.5  |-    0.0  |-    0.0    |+    0.0   ]
vf09=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf10=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf11=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf12=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf13=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf14=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf15=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf16=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf17=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf18=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf19=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf20=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf21=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf22=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf23=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf24=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf25=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf26=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf27=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf28=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf29=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf30=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]
vf31=[+    0.0  |+    0.0  |+    0.0    |+    0.0   ]

vi00=0000    vi08=0000    status=000
vi01=0204    vi09=0000       MAC=0000
vi02=01F9    vi10=0000      clip=000000       TPC=006E
vi03=0208    vi11=0000
vi04=0000    vi12=0000   R=00000000
vi05=0000    vi13=0000   I=+    0.0
vi06=0000    vi14=0000   Q=+    0.0
vi07=0000    vi15=0006   P=+    0.0

 Break by D bit
```

```
0198    8000033C 01FE28BD  MADDAy.xyzw ACC,VF05,VF30        NO
01A0    8000033C 01FE30BE  MADDAz.xyzw ACC,VF06,VF30        NO
01A8    8000033C 01FE3FCB  MADDw.xyzw VF31,VF07,VF30        NO
01B0    8000033C 01E0D690  MAXx.xyzw VF26,VF26,VF00         NO
01B8    8000033C 000002FF  NOP                              NO
01C0    8000033C 000002FF  NOP                              NO
LOOP1   81FF03BC 01E0FF5B  MULw.xyzw VF29,VF31,VF00         DI
01D0    81F93B7C 01FA41BC  MULAx.xyzw ACC,VF08,VF26         LQ
01D8    81FE337C 01FA48BD  MADDAy.xyzw ACC,VF09,VF26        LQ
01E0    100E7003 01FA50BE  MADDAz.xyzw ACC,VF10,VF26        IA
01E8    800B5FF2 01FA5E0B  MADDw.xyzw VF24,VF11,VF26        IA
01F0    100C6003 01D961BC  MULAx.xyz ACC,VF12,VF25          IA
01F8    03EDB800 01D968BD  MADDAy.xyz ACC,VF13,VF25         SQ
0200    100D6803 01E0EF1C  MULq VF28,VF29,Q                 IA
0208    03EE9800 01E2C62A  MUL VF24,VF24,VF02               SQ
```

The pane can be set back to EE monitoring by repeating the above steps only clicking **Main CPU** from the shortcut menu.

Debugging a VU module

1. Edit your code so the D bit is set on the first VU instruction. If you have multiple calls to the same microcode it can be difficult to choose which call to debug with. A good solution is to introduce a nop instruction just before the normal entry point, for example:

```
Microcode_Entry_DEBUG:
   nop[D] nop
Microcode_Entry:
   ; normal code goes here
```

   and then set up the DMA list to MSCAL() into the debug version.

2. The VU will halt just after the D bit instruction.

Note: VU breakpoints halt execution after the instruction they're on rather than at the current instruction.

3. From then on you can single-step and add breakpoints.

You will notice that the VU disassembly pane displays markers such as "S" where an instruction causes a stall and "X" on instructions which cannot be safely breakpointed because the pipeline will not be restartable.

You may notice that if you have a VU1 disassembly or source window open in the Debugger then EE single-stepping is slower than usual. This is because the PlayStation 2 DECI protocol does not send notification when VU execution stops so we must analyse the VU situation when all EE updates occur.

VU code is normally started as part of a DMA operation; see "DMA channel debugging" on page 132 for details.

DMA channel debugging

DMA channel debugging allows you to inspect the code about to be sent to a DMA channel. A DMA hardware break for the target channel (or for all DMA channels) needs to be set in order to cause a program execution break to occur when the DMA channel is accessed (see "To set a DMA channel hardware breakpoint" on page 54).

Viewing a DMA channel

Whenever the target stops running, if a DMA pane is open it updates to show the DMA channel registers. You can also choose to view tags and VIF packets.

To view a DMA channel

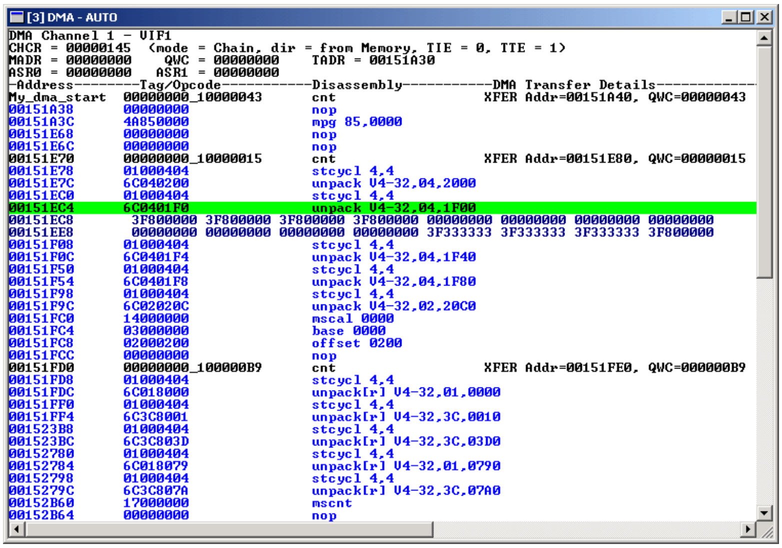
If you wish to open a new DMA pane that will show the DMA channel settings:

- Click **New Window > DMA** in the **Window** menu or click the **DMA channel view** toolbar button.

  A new window containing a DMA pane is opened. This shows a disassembly of the code about to be sent to the DMA channel. You can change the view to show the tags and VIF packets, using the shortcut menu.

DMA pane

The DMA pane shows the current DMA channel registers, and optionally tags and VIF packets, for the currently accessed DMA channel (listed at the top of the window).



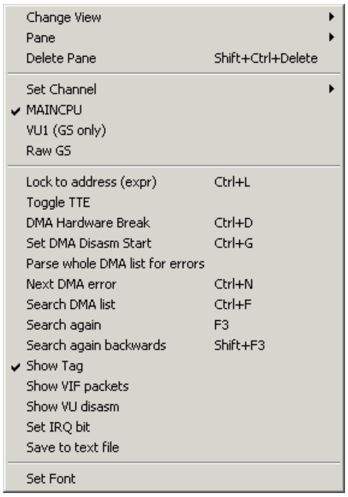
```
DMA Channel 1 - VIF1
CHCR = 00000145  (mode = Chain, dir = from Memory, TIE = 0, TTE = 1)
MADR = 00000000    QWC = 00000000    TADR = 00151A30
ASR0 = 00000000   ASR1 = 00000000
-Address--------Tag/Opcode-----------Disassembly-----------DMA Transfer Details------------
My_dma_start  00000000_10000043      cnt                   XFER Addr=00151A40, QWC=00000043
00151A38      00000000               nop
00151A3C      4A850000               mpg 85,0000
00151E68      00000000               nop
00151E6C      00000000               nop
00151E70      00000000_10000015      cnt                   XFER Addr=00151E80, QWC=00000015
00151E78      01000404               stcycl 4,4
00151E7C      6C040200               unpack V4-32,04,2000
00151EC0      01000404               stcycl 4,4
00151EC4      6C0401F0               unpack V4-32,04,1F00
00151EC8       3F800000 3F800000 3F800000 3F800000 00000000 00000000 00000000 00000000
00151EE8       00000000 00000000 00000000 00000000 3F333333 3F333333 3F333333 3F800000
00151F08      01000404               stcycl 4,4
00151F0C      6C0401F4               unpack V4-32,04,1F40
00151F50      01000404               stcycl 4,4
00151F54      6C0401F8               unpack V4-32,04,1F80
00151F98      01000404               stcycl 4,4
00151F9C      6C02020C               unpack V4-32,02,20C0
00151FC0      14000000               mscal 0000
00151FC4      03000000               base 0000
00151FC8      02000200               offset 0200
00151FCC      00000000               nop
00151FD0      00000000_100000B9      cnt                   XFER Addr=00151FE0, QWC=000000B9
00151FD8      01000404               stcycl 4,4
00151FDC      6C018000               unpack[r] V4-32,01,0000
00151FF0      01000404               stcycl 4,4
00151FF4      6C3C8001               unpack[r] V4-32,3C,0010
001523B8      01000404               stcycl 4,4
001523BC      6C3C803D               unpack[r] V4-32,3C,03D0
00152780      01000404               stcycl 4,4
00152784      6C018079               unpack[r] V4-32,01,0790
00152798      01000404               stcycl 4,4
0015279C      6C3C807A               unpack[r] V4-32,3C,07A0
00152B60      17000000               mscnt
00152B64      00000000               nop
```

DMA pane with tags and VIF packets shown. One unpack instruction has been expanded.

The top part displays the current DMA register settings for that DMA channel. The lower part displays a list of DMA memory transfers. By default the DMA pane is created in AUTO mode. In this mode the pane automatically tracks any DMA related hardware breaks and auto-switches to display that active channel.

DMA pane shortcut menu

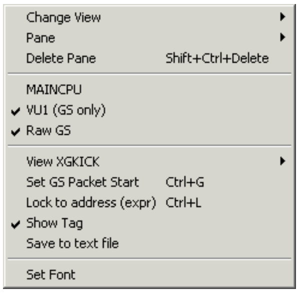
The DMA pane shortcut menu is different according to whether the option Main CPU or VU1 (GS only) is checked in the shortcut menu. The Main CPU shortcut menu is described first, followed by the VU1 (GS only) shortcut menu.



Main CPU DMA pane shortcut menu

Set Channel	Enables you to set the DMA channel to one of the following options: **AUTO**, **VIF0**, **VIF1**, **GIF**, **fromIPU**, **toIPU**, **SIF0**, **SIF1**, **SIF2**, **fromSPR**, and **toSPR**.
Main CPU	When checked, the DMA pane displays DMA chains in EE RAM.
VU1 (GS only)	When checked, the DMA pane displays GS packets in VU1 RAM and the shortcut menu changes to the VU1 (GS only) shortcut menu (see below).
Raw GS	Show data in raw GS format.
Lock to address (expr)	Locks the pane to a valid memory address.
Toggle TTE	Sets the default state of the "Tag Transfer Enable" bit in the DMA channel control register (Dn_CHCR).
DMA Hardware Break	Brings up the Hard breakpoint on DMA start dialog (see "To set a DMA channel hardware breakpoint" on page 54) which enables you to set a hardware breakpoint on a particular DMA channel.
Set DMA Disasm Start	Causes the disassembly to be shown from the selected start address.
Parse whole DMA list for errors	Progresses through the whole DMA chain down to VIF packet level checking it for errors. If it finds an error it will locate to that line and place the cursor on it.
Next DMA error	If the above parse found more than one error this will take you to the next error.
Search DMA list	Allows you to search your DMA chain (from the current cursor position onwards) for a specified text string. Note that regular expressions are supported.
Search again	Repeats the **Search DMA list** search from after the current cursor position to find the next occurrence of that same string.
Search again backwards	Search the DMA chain backwards for the last given search keyword.
Show Tag	When checked, tags are displayed.
Show VIF packets	When checked, VIF packets are displayed by expanding the view to include disassembly of the VIF codes within the DMA packets.
Show VU disasm	When checked, VU disassembly is displayed by expanding the view to include VU disassembly of code within MPG VIF packets.
Set IRQ bit	Sets the [I] bit in the selected DMA packet.

Save to text file	Saves the current DMA chain to a text file.



VU1 (GS only) DMA pane shortcut menu

Main CPU	When checked, displays DMA chains in EE RAM and the shortcut menu changes to the Main CPU shortcut menu.
VU1 (GS only)	When checked, the DMA pane displays GS packets in VU1 RAM.
Raw GS	Show data in raw GS format.
View XGKICK	Shortcut to view the XGKICK opcode. See "Using the XGKICK shortcut" on page 138.
Set GS Packet Start	Brings up the Enter address dialog allowing you to specify the GS packet start address.
Lock to address (expr)	Brings up the Enter expression dialog allowing you to specify an address or expression to lock the display to.
Show Tag	When checked, tags are displayed.
Save to text file	Enables you to save the disassembly to a DMA disassembly (.dma) file of your choice.

Debugging a DMA channel

This section describes how to debug a DMA channel.

To debug a DMA channel

1. Load up the ELF for a Sony demo such as blow.elf and set a hardware break on all DMA channels (see "To set a DMA channel hardware breakpoint" on page 54).

2. Create a DMA pane; see "To view a DMA channel" on page 132.

3. Hit **Step** a few times and you will see in the EE source pane that execution stops as each DMA is about to be triggered on the SyncPath() call.

Note: The execution halts before the DMA actually happens but the Debugger still figures out what the resulting Dn_CHCR register will be by looking at the breakpointed instruction and bases its interpretation on that.

4. Switch to the DMA pane to view the DMA chains; these are shown in black.

5. Using the shortcut menu from the DMA pane, select **Show VIF packets** which expands each DMA chain to show its component VIF packets; these are shown in blue.

6. Still using the shortcut menu from the DMA pane, select **Show VU Disasm** (disassembly) to show the VU disassembly; this is shown in red.

   Note that if you set the cursor on a VU disassembly line you can toggle the D bit of that instruction on or off using an accelerator key. If you set the D bit of a VU instruction like this you are changing the original copy in EE memory so all subsequent VIF transfers of that packet will send down a VU MPG with the D bit set.

7. Select an unpack instruction from the expanded VIF packet, then using the shortcut menu from the DMA pane, select **Expand Unpack** to show the expanded instruction; this is shown in navy blue.

When that VU code then executes it will automatically break *after* the instruction with the D bit executes and the Debugger will automatically switch focus to a VU disassembly or source pane if there is one open and will allow you to single-step and breakpoint the VU code from there.

Your program must not call sceGsSyncPath(0,0) afterwards because the VU will be halted. This means that the VU MPG will not complete and the EE function will timeout instead and typically then continues as if the VU had finished, whereas of course it has not.

The Debugger has to be pretty clever to generate VIF disassemblies on the fly but if you use **Set DMA Disasm Start** to point a DMA disassembly at a DMA chain which is "under construction" in memory and there is an old DMA chain already in that buffer then the Debugger might get confused. If you have such problems then you should try clearing out the buffer before generating your new chain.

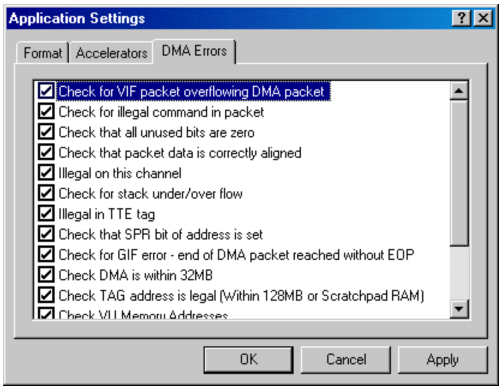
If you are building your chain piece by piece note that the "refresh" button will cause the entire DMA chain disassembly to be regenerated afresh.

The **Toggle TTE** shortcut menu option in the DMA pane allows you to set the default state of the "Tag Transfer Enable" bit in the DMA channel control register (Dn_CHCR). The setting of this bit will affect the way your DMA chain will be interpreted so it is important that the debugger know the state of this bit if it is to be able to disassemble your DMA chain correctly. Normally the debugger will read the state of this flag directly from the Dn_CHCR hardware register but if that setting is not correct for the DMA transfer you are examining then you will need to change it using this feature.

Detecting DMA errors

You can configure which DMA errors are detected by the Debugger from the Application Settings dialog.

1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **DMA Errors** tab is selected.



- ☑ Check for VIF packet overflowing DMA packet
- ☑ Check for illegal command in packet
- ☑ Check that all unused bits are zero
- ☑ Check that packet data is correctly aligned
- ☑ Illegal on this channel
- ☑ Check for stack under/over flow
- ☑ Illegal in TTE tag
- ☑ Check that SPR bit of address is set
- ☑ Check for GIF error - end of DMA packet reached without EOP
- ☑ Check DMA is within 32MB
- ☑ Check TAG address is legal (Within 128MB or Scratchpad RAM)
- ☑ Check VU Memory Addresses

2. A complete list of DMA errors handled by the DMA pane is displayed. You can check or uncheck individual errors to enable or disable their detection by the DMA pane.

From the DMA pane, shortcut menu options are available for detecting DMA chain errors:

Parse DMA list for errors	Progresses through the whole DMA chain down to VIF packet level checking it for errors. If it finds an error it will locate to that line and place the cursor on it.
Next DMA error	If the above parse found more than one error this will take you to the next error.
Search DMA list	Allows you to search your DMA chain (from the current cursor position onwards) for a specified text string. Note that regular expressions are supported.
Search again	Repeats the **Search DMA list** search from after the current cursor position to find the next occurrence of that same string

The following table lists the DMA errors detected by the DMA error parser.

Note that the first one defaults to OFF (i.e. not checked) but the rest currently default to ON (checked). If you change these settings the changes are preserved in the dbugps2.ps2 configuration file when you exit the debugger (see "Configuring the user interface" on page 28).

Check for VIF packet overflowing DMA packet
Check for illegal command in packet
Check that all unused bits are zero
Check that packet data is correctly aligned
Check if the operation is legal on this DMA channel
Check for DMA callstack overflow or underflow
Check that operation is legal in TTE tag
Check that SPR bit is set for scratchpad addresses
Check for GIF error - if end of DMA packet reached without EOP
Check DMA is within bottom 32MB address range
Check TAG address is legal (within 128MB address range)
Check VU memory addresses are legal
Check VIF IRQ interval is > 24 qwords
Check VIF IRQ to FLUSHA interval
Check PATH3 open interval > 24 qwords

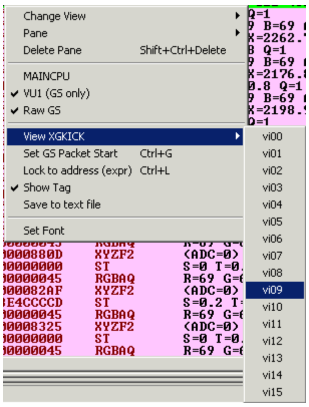
Note that some of these are not necessarily errors, e.g. the "size" error is flagged if you intentionally break a VIF or GIF op (such as UNPACK or IMAGE) across multiple DMA packets. Developers would typically do this if they wish to provide all or part of the data for the op using a separate REF DMA transfer. The "unused bits are zero" error is also not necessarily an error since non-zero unused bits do not affect the hardware operation and some developers have taken to using unused bits for other purposes. However, these are also common errors in runtime-generated DMA lists so it is flagged as an error for the developer to check.

The last three checks arise from recent Sony Technote issues regarding VIF1 interrupts and using FLUSHA VIF code too soon after a VIF interrupt. This does not apply if you use the workaround of having a stall in your interrupt handler until the GIF starts accepting data.

Using the XGKICK shortcut

XGKICK is the VU1 opcode which transfers GS primitives from VU1 memory directly to the GS to be drawn.

Typically your VU1 microprogram will build these primitives in VU1 memory based upon 3D geometry information it has been processing. If you halt the VU before this transfer happens (see "To halt the VU before the GS transfer" on page 139), you can inspect the GS primitives your VU microprogram has generated before it transfers them to the GS for drawing. The XGKICK opcode accepts a VI## integer register to specify the location of the GS packet in VU1 memory. Rather than require you to look up the register used and the value of this register and then enter that into a VU1 DMA pane with the **Set GS Packet Start** shortcut menu option, we have provided the **View XGKICK** menu option as a shortcut. Just select **View XGKICK** and then select the integer register which points to your GS packet data.



```
0000880D   XYZF2    <ADC=0>
00000000   ST       S=0 T=0.
00000045   RGBAQ    R=69 G=6
000082AF   XYZF2    <ADC=0>
E4CCCCD    ST       S=0.2 T=
00000045   RGBAQ    R=69 G=6
00008325   XYZF2    <ADC=0>
00000000   ST       S=0 T=0.
00000045   RGBAQ    R=69 G=6
```

The DMA pane will then display the XGKICK packet which is about to be transferred to the GS.

To halt the VU before the GS transfer

There are several ways to halt your VU microprogram just before the GS transfer:

- You can hard-code a D-bit breakpoint into your VU1 microprogram.
- You can use the debugger to set a D-bit breakpoint in the DMA chain in EE memory. To do this you would typically use a DMA hardware breakpoint to halt the EE before the DMA transfer occurs, then open a DMA pane on the EE, then expand the display to include VIF and MPG disassembly, then double-click (or press the breakpoint hotkey) on the opcode before the XGKICK. Next you can single step or run the EE to perform the transfer. The VU microprogram will halt at the XGKICK.
- You can use the methods above to halt your VU microprogram at an earlier point (like the beginning) and then use VU single-step and breakpoints in the VU1 disassembly/source pane to step the program to the point you are interested in.

DMA and VU debugging with the Sony blow sample

There is no one right way to use the tools so explaining all the debug possibilities is not always a good way to show someone how the tools can be used. The following walk though may be handy as a "getting started" guide for DMA and VU debugging.

This example assumes you have the SCE blow sample installed on your host PC. Although this example is based on working from a command line with a makefile you can equally well work from Visual Studio and/or ps2cc if you prefer. If this is your first time using VU debug support then it is probably best to work through this command line demo first before attempting the same things from Visual Studio just to make sure you have the correct debug options turned on and are building using the correct tools.

1. Open a command line window and change directory to the standard SCE blow demo (at /usr/local/sce/ee/sample/vu1/blow).

2. Edit the makefile to ensure that:

   - The compiler generates full debug info

   - You use SN's ps2dvpas rather than the GNU dvpasm.

   - You link libsn.a before the other libraries. This will provide alternate versions of some SCE library functions (the SCE versions upset D bit breakpointing). Note that this is for debug builds only and that for final release you will need to remove libsn.a from your build.

   The changed lines in your makefile will look like this:

   ```
   LIBS = $(LIBDIR)/libsn.a \
   $(LIBDIR)/libgraph.a \
   $(LIBDIR)/libdma.a \
   $(LIBDIR)/libdev.a \
   $(LIBDIR)/libpkt.a \
   $(LIBDIR)/libpad.a \
   $(LIBDIR)/libvu0.a
   DVPASM = PS2DVPAS
   CFLAGS = -g -Wall -Werror -Wa,-al -fno-common
   ```

   You should also check that your DVPASMFLAGS is set to produce VU debug info:

   ```
   DVPASMFLAGS = -g
   ```

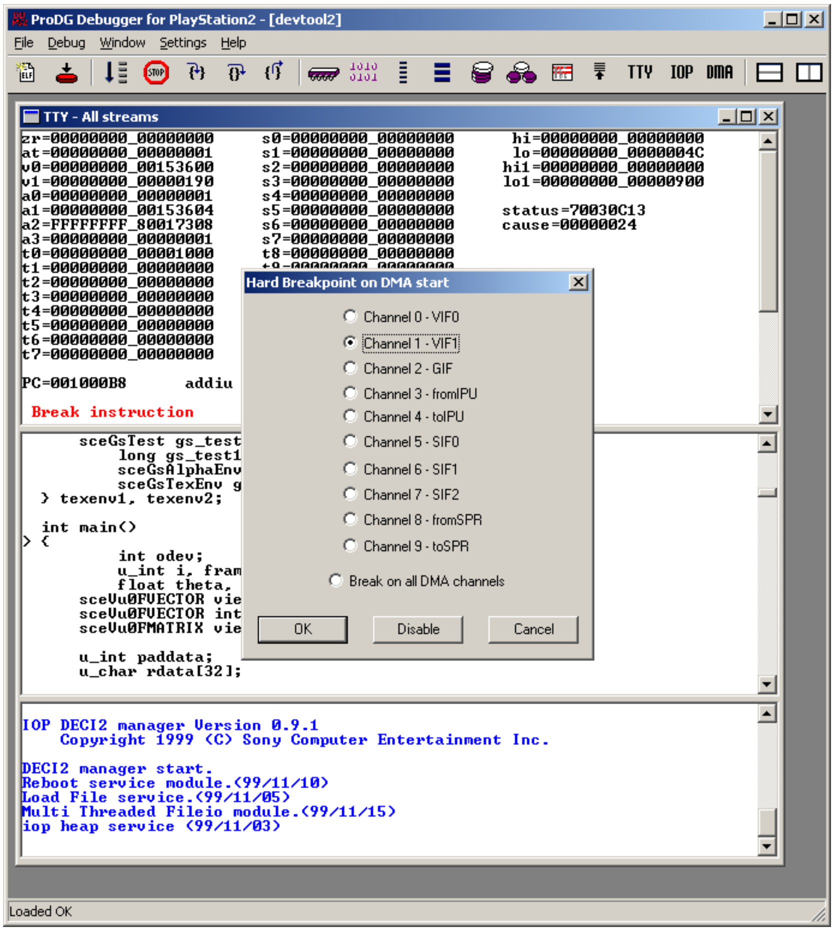
3. Delete all object and ELF files in that directory to force a full build.

4. Run make.

5. Launch the Debugger. Either launch it with no parameters and use the **Load Elf File** option (see "Loading and running ELF files" on page 41) to reset the target, load the blow.elf code and symbols, and run to main OR specify all that on the command line:

   ```
   ps2dbg -ref blow.elf
   ```

6. From the Debug menu, select EE DMA Hardware Break to bring up the Hard Breakpoint on DMA start dialog (see "To set a DMA channel hardware breakpoint" on page 54) and select channel 1 (VIF1) like this:


ProDG Debugger for PlayStation2 - [devtool2]
File Debug Window Settings Help
TTY IOP DMA
TTY - All streams

```
zr=00000000_00000000     s0=00000000_00000000       hi=00000000_00000000
at=00000000_00000001     s1=00000000_00000000       lo=00000000_0000004C
v0=00000000_00153600     s2=00000000_00000000      hi1=00000000_00000000
v1=00000000_00000190     s3=00000000_00000000      lo1=00000000_00000900
a0=00000000_00000001     s4=00000000_00000000
a1=00000000_00153604     s5=00000000_00000000      status=70030C13
a2=FFFFFFFF_80017308     s6=00000000_00000000      cause=00000024
a3=00000000_00000001     s7=00000000_00000000
t0=00000000_00001000     t8=00000000_00000000
t1=00000000_00000000
t2=00000000_00000000
t3=00000000_00000000
t4=00000000_00000000
t5=00000000_00000000
t6=00000000_00000000
t7=00000000_00000000

PC=001000B8        addiu

 Break instruction
```

Hard Breakpoint on DMA start

- ( ) Channel 0 - VIF0
- (●) Channel 1 - VIF1
- ( ) Channel 2 - GIF
- ( ) Channel 3 - fromIPU
- ( ) Channel 4 - toIPU
- ( ) Channel 5 - SIF0
- ( ) Channel 6 - SIF1
- ( ) Channel 7 - SIF2
- ( ) Channel 8 - fromSPR
- ( ) Channel 9 - toSPR
- ( ) Break on all DMA channels

OK Disable Cancel

```
      sceGsTest gs_test
          long gs_test1
          sceGsAlphaEnv
          sceGsTexEnv g
  } texenv1, texenv2;

  int main()
> {
          int odev;
          u_int i, fram
          float theta,
      sceVu0FVECTOR vie
      sceVu0FVECTOR int
      sceVu0FMATRIX vie

      u_int paddata;
      u_char rdata[32];
```

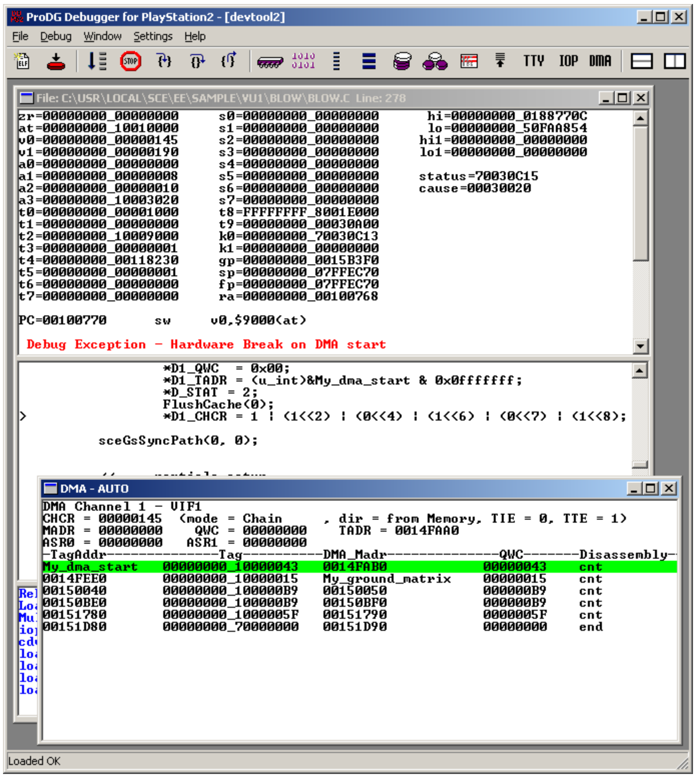
```
IOP DECI2 manager Version 0.9.1
    Copyright 1999 (C) Sony Computer Entertainment Inc.

DECI2 manager start.
Reboot service module.(99/11/10)
Load File service.(99/11/05)
Multi Threaded Fileio module.(99/11/15)
iop heap service (99/11/03)
```

Loaded OK

7. Start the target PlayStation 2 running. It will stop at the next attempt to trigger the VIF1 DMA channel. Note that the application has actually halted on the instruction that will trigger the DMA, i.e. the DMA has not happened yet.

   If you now open a new DMA pane - see "DMA pane" on page 132 for usage details - the Debugger will automatically recognise that a DMA hardware break has happened and will auto-locate to the DMA on the about-to-start DMA channel.

   You should see something like this:


ProDG Debugger for PlayStation2 - [devtool2]
File Debug Window Settings Help
TTY IOP DMA
File: C:\USR\LOCAL\SCE\EE\SAMPLE\VU1\BLOW\BLOW.C Line: 278
DMA - AUTO
Loaded OK

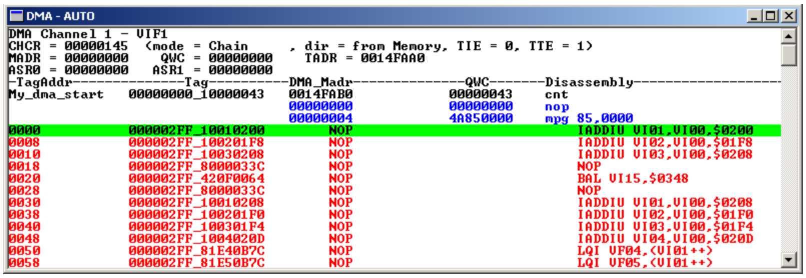
```
zr=00000000_00000000      s0=00000000_00000000       hi=00000000_0188770C
at=00000000_10010000      s1=00000000_00000000       lo=00000000_50FAA854
v0=00000000_00000145      s2=00000000_00000000      hi1=00000000_00000000
v1=00000000_00000190      s3=00000000_00000000      lo1=00000000_00000000
a0=00000000_00000000      s4=00000000_00000000
a1=00000000_00000008      s5=00000000_00000000      status=70030C15
a2=00000000_00000010      s6=00000000_00000000      cause=00030020
a3=00000000_10003020      s7=00000000_00000000
t0=00000000_00001000      t8=FFFFFFFF_8001E000
t1=00000000_00000000      t9=00000000_00030A00
t2=00000000_10009000      k0=00000000_70030C13
t3=00000000_00000001      k1=00000000_00000000
t4=00000000_00118230      gp=00000000_0015B3F0
t5=00000000_00000001      sp=00000000_07FFEC70
t6=00000000_00000000      fp=00000000_07FFEC70
t7=00000000_00000000      ra=00000000_00100768

PC=00100770      sw     v0,$9000(at)

 Debug Exception - Hardware Break on DMA start
```

```
                  *D1_QWC  = 0x00;
                  *D1_TADR = (u_int)&My_dma_start & 0x0fffffff;
                  *D_STAT = 2;
                  FlushCache(0);
>                 *D1_CHCR = 1 | (1<<2) | (0<<4) | (1<<6) | (0<<7) | (1<<8);

          sceGsSyncPath(0, 0);
```

```
DMA Channel 1 - VIF1
CHCR = 00000145  (mode = Chain      , dir = from Memory, TIE = 0, TTE = 1)
MADR = 00000000    QWC = 00000000    TADR = 0014FAA0
ASR0 = 00000000   ASR1 = 00000000
-TagAddr--------------Tag----------DMA_Madr-------------QWC-------Disassembly-
My_dma_start   00000000_10000043  0014FAB0             00000043    cnt
0014FEE0       00000000_10000015  My_ground_matrix     00000015    cnt
00150040       00000000_100000B9  00150050             000000B9    cnt
00150BE0       00000000_100000B9  00150BF0             000000B9    cnt
00151780       00000000_1000005F  00151790             0000005F    cnt
00151D80       00000000_70000000  00151D90             00000000    end
```

8. The context menu for the DMA pane provides several useful options. See "DMA pane shortcut menu" on page 133.

   Now activate the context menu and select **Show VU disasm** to expand to the deepest display level. Your display will now show a detailed disassembly of the beginning of the DMA chain like so:



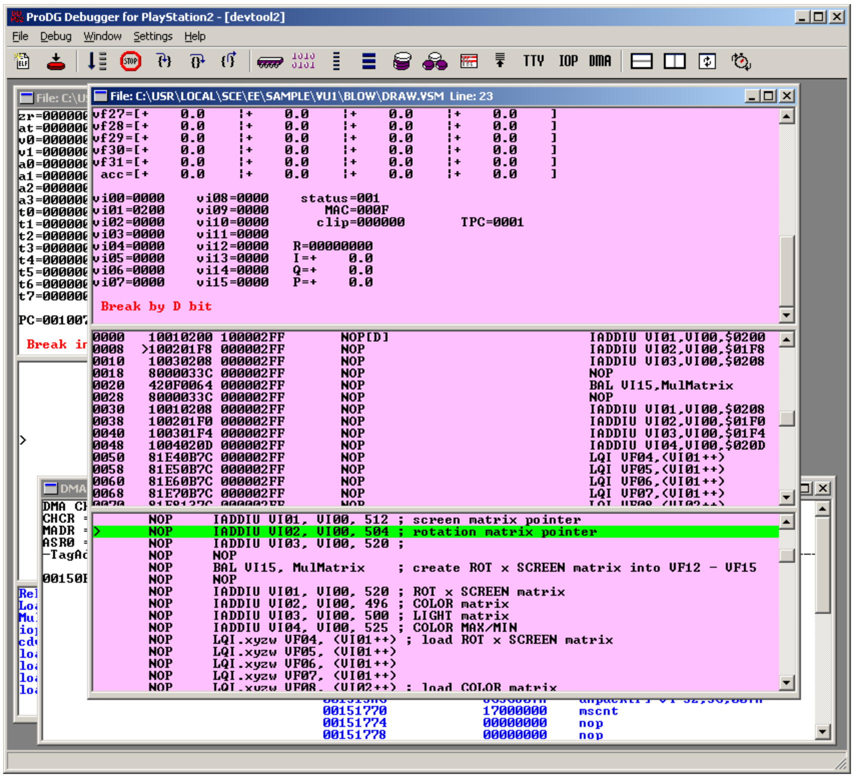
```
DMA Channel 1 - VIF1
CHCR = 00000145  (mode = Chain      , dir = from Memory, TIE = 0, TTE = 1)
MADR = 00000000    QWC = 00000000    TADR = 0014FAA0
ASR0 = 00000000   ASR1 = 00000000
-TagAddr--------------Tag----------DMA_Madr-------------QWC-------Disassembly------------------
My_dma_start   00000000_10000043  0014FAB0             00000043    cnt
                                  00000000             00000000    nop
                                  00000004             4A850000    mpg 85,0000
0000           000002FF_10010200     NOP                               IADDIU VI01,VI00,$0200
0008           000002FF_100201F8     NOP                               IADDIU VI02,VI00,$01F8
0010           000002FF_10030208     NOP                               IADDIU VI03,VI00,$0208
0018           000002FF_8000033C     NOP                               NOP
0020           000002FF_420F0064     NOP                               BAL VI15,$0348
0028           000002FF_8000033C     NOP                               NOP
0030           000002FF_10010208     NOP                               IADDIU VI01,VI00,$0208
0038           000002FF_100201F0     NOP                               IADDIU VI02,VI00,$01F0
0040           000002FF_100301F4     NOP                               IADDIU VI03,VI00,$01F4
0048           000002FF_1004020D     NOP                               IADDIU VI04,VI00,$020D
0050           000002FF_81E40B7C     NOP                               LQI VF04,(VI01++)
0058           000002FF_81E50B7C     NOP                               LQI VF05,(VI01++)
```

9. Select the first line of VU microcode (as above) and right-click to bring back the context menu and select **Set D bit breakpoint** – or just press the breakpoint button set-break accelerator key. This will set the [D] bit of that instruction so that when it executes it will cause the VU to halt.

Note: This D-bit is set in the original copy of the VU code in EE memory so any further MPG transfers of this code will also send the D bit breakpoint.

10. Now bring up a Debugger VU1 Window. You will normally want something with at least a VU1 register pane and a disassembly pane (and you may later want to add VU1 memory and source panes if you have a big enough monitor).

11. Switch back to the main CPU source pane and click **Step** in the **Debug** menu to single-step the instruction that will actually trigger the DMA to start. The DMA will start, the VU program will be downloaded to VU1 by the MPG packet. The following VIF packets which you saw as disassembly earlier will then copy data to the VU memory and call the VU program to process that data. The VU program will halt after executing the D bit breakpoint that you just set. If you select the VU window now you will see something like this:


ProDG Debugger for PlayStation2 - [devtool2]
File Debug Window Settings Help
TTY IOP DMA
File: C:\USR\LOCAL\SCE\EE\SAMPLE\VU1\BLOW\DRAW.VSM Line: 23

```
vf27=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]
vf28=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]
vf29=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]
vf30=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]
vf31=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]
 acc=[+    0.0    ¦+    0.0    ¦+    0.0    ¦+    0.0    ]

vi00=0000    vi08=0000    status=001
vi01=0200    vi09=0000       MAC=000F
vi02=0000    vi10=0000      clip=000000       TPC=0001
vi03=0000    vi11=0000
vi04=0000    vi12=0000   R=00000000
vi05=0000    vi13=0000   I=+     0.0
vi06=0000    vi14=0000   Q=+     0.0
vi07=0000    vi15=0000   P=+     0.0

 Break by D bit
```

```
0000   10010200 100002FF       NOP[D]            IADDIU VI01,VI00,$0200
0008  >100201F8 000002FF       NOP               IADDIU VI02,VI00,$01F8
0010   10030208 000002FF       NOP               IADDIU VI03,VI00,$0208
0018   8000033C 000002FF       NOP               NOP
0020   420F0064 000002FF       NOP               BAL VI15,MulMatrix
0028   8000033C 000002FF       NOP               NOP
0030   10010208 000002FF       NOP               IADDIU VI01,VI00,$0208
0038   100201F0 000002FF       NOP               IADDIU VI02,VI00,$01F0
0040   100301F4 000002FF       NOP               IADDIU VI03,VI00,$01F4
0048   1004020D 000002FF       NOP               IADDIU VI04,VI00,$020D
0050   81E40B7C 000002FF       NOP               LQI VF04,(VI01++)
0058   81E50B7C 000002FF       NOP               LQI VF05,(VI01++)
0060   81E60B7C 000002FF       NOP               LQI VF06,(VI01++)
0068   81E70B7C 000002FF       NOP               LQI VF07,(VI01++)
```

```
       NOP     IADDIU VI01, VI00, 512 ; screen matrix pointer
>      NOP     IADDIU VI02, VI00, 504 ; rotation matrix pointer
       NOP     IADDIU VI03, VI00, 520 ;
       NOP     NOP
       NOP     BAL VI15, MulMatrix    ; create ROT x SCREEN matrix into VF12 - VF15
       NOP     NOP
       NOP     IADDIU VI01, VI00, 520 ; ROT x SCREEN matrix
       NOP     IADDIU VI02, VI00, 496 ; COLOR matrix
       NOP     IADDIU VI03, VI00, 500 ; LIGHT matrix
       NOP     IADDIU VI04, VI00, 525 ; COLOR MAX/MIN
       NOP     LQI.xyzw VF04, (VI01++) ; load ROT x SCREEN matrix
       NOP     LQI.xyzw VF05, (VI01++)
       NOP     LQI.xyzw VF06, (VI01++)
       NOP     LQI.xyzw VF07, (VI01++)
       NOP     LQI.xyzw VF08, (VI02++) ; load COLOR matrix
```

```
00151770        17000000     mscnt
00151774        00000000     nop
00151778        00000000     nop
```

You can now single-step and set breakpoints in your VU1 code. You can even open a VU1 source pane and debug your VU program at source level.

Note: If you used the GNU DVP assembler then you may need to "set source search path" for the Debugger to be able to locate the source files correctly. This is because the SCE/GNU dvpasm does not output full path info in its debug symbols. If you use SN Systems' ps2dvpas instead you will get complete debug output and will not need to use search paths.

12. If you were to look at the DMA disassembly pane now you will see that the DMA has stalled having transferred the first bit of data and is now waiting for VU1 to complete before it can call it to process subsequent data. The DMA pane automatically tracks and displays the current state of the DMA.

13. If you allow the VU1 program to continue (press the **Run** button whilst the VU1 window is selected) then when the VU1 code completes the DMA will continue.

14. Every time this DMA chain is processed the VU microcode is re-downloaded to the VU and the D bit breakpoint is sent along with it. If you wish the whole program to continue without D bit breaks you will first need to go back to the DMA pane and remove the D bit in that MPG download (the same way you set it).

Chapter 13: Debugger Scripting

What is Debugger scripting?

Note: Debugger scripting is only available with ProDG Plus for PlayStation 2.

The C script interpreter is basically a single C program that you can interactively add to, and update the functions of, as the ProDG Debugger is running. Using a new *script pane* type it allows you to quickly and interactively enter programs to perform different tasks. These can be one-off 'execute until completion' type scripts or they can be bound to target and debugger events (like exceptions, window updates, keystrokes etc). These scripts can access the target console e.g. to fetch memory contents, and they can do console output to a debugger pane.

This product comprises a C interpreter, built into the Debugger, which supports most ANSI C, with access to a number of special built-in functions to allow the scripts to interface with the Debugger and the target console. When scripts are loaded (or immediate mode script functions are executed) the script is compiled to a byte-code form that is then interpreted.

Execution is pretty fast and typically of the order of one million C statements per second. The ProDG Plus script interpreter has many advantages over normal Windows scripting or compiled-code approaches. It is *much* faster than trying to use COM interfaces to control an application so a lot more things are achievable and user-friendly. Being written in C, the scripting interface is very familiar to C programmers.

The script interpreter core is EiC. For more information on EiC see http://www.kd-dev.com/~eic/. SN System's use of the EiC package to produce an aggregation for commercial distribution is covered by the standard artistic license. The original standard form of the EiC Package, including source code, is available from the above web site.

Installation notes

Note: If you wish to use scripting you must first obtain a license to ProDG Plus for PlayStation 2.

Scripting functionality is installed to the \TestScripts subdirectory. For detailed information on how to install and configure scripting, please follow the instructions in the file \autoexec.eic in this subdirectory.

Troubleshooting tip

If you have any problems, look at the DBG TTY stream because command-line scripts will output load-time and run-time errors there.

Using scripts

A script file is a plain text file containing C source code that can be loaded and executed by the Debugger. You can create and edit these text files using your preferred programmer's editor.

Note: You cannot currently edit script files in the Debugger itself.

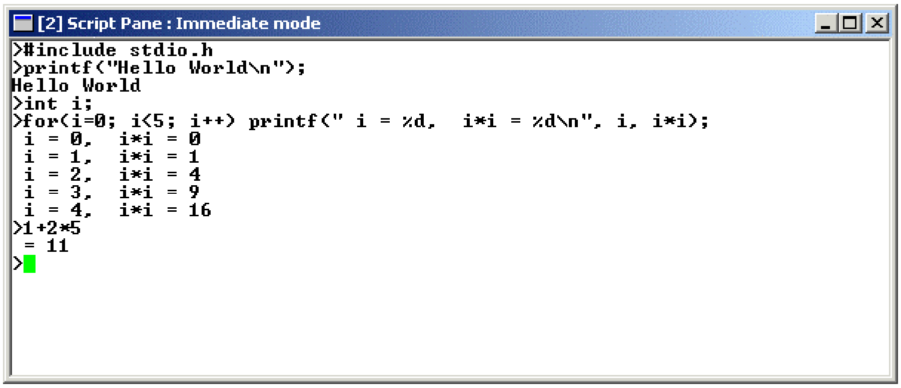
There is no specific extension for script files although since they are pretty much ANSI C you may wish to use the extension .c so that your editor knows how to format and syntax color them. You may also wish to use a different extension so that the script files are not confused with any C project files you may be using for source code in your console project.

Scripts can be used to perform the following:

- Interpret ANSI C including standard C library support (including file access)
- Output stdout to a Debugger pane or to a Debugger TTY buffer
- Supports C preprocessor directives
- Hook Debugger events such as "target exception"
- Handle Debugger key presses
- Reset the target and load/run ELF files
- Send/receive memory to/from the target console
- Access the symbol table to evaluate expressions with symbols from your project
- Set custom display formats for types in the watch and locals pane
- Provide custom debugger panes and hook into debugger context menus
- Auto-run at debugger startup to provide custom functions and new debugger panes
- Control the interfacing of an external editor to the debugger source pane

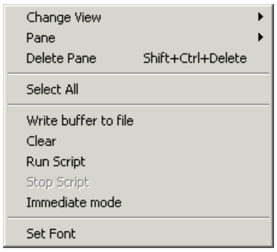
Script pane

The script pane allows you to run scripts on the PlayStation 2. The pane resembles a TTY pane but includes basic editing features to allow you to enter and run scripts immediately.



```
>#include stdio.h
>printf("Hello World\n");
Hello World
>int i;
>for(i=0; i<5; i++) printf(" i = %d,  i*i = %d\n", i, i*i);
 i = 0,  i*i = 0
 i = 1,  i*i = 1
 i = 2,  i*i = 4
 i = 3,  i*i = 9
 i = 4,  i*i = 16
>1+2*5
 = 11
>
```

Script pane shortcut menu



Select All	Selects all the text in the script pane.
Write buffer to file	Allows you to write the contents of the script pane to a file.
Clear	Clears the contents of the script pane.
Run script	Select a script file and run it. See "Run script dialog from shortcut menu" on page 149.
Stop script	Stops a script from running.
Immediate mode	When this option is checked, text typed into the script pane is available for immediate execution. See "Immediate mode statements" on page 147.

Immediate mode statements

Whenever the script engine is not busy you can select the script pane and type in statements to be executed immediately. These statements should follow the usual C syntax. This feature is normally used to call functions that you have previously loaded but have not hooked to keystrokes or callbacks.

All the usual C statements or preprocessor directives are supported but note that you must fit a complete statement or set of statements onto one line, i.e. you cannot break a C construct across two lines so it may be necessary to condense several C statements onto one line.

You could enter the following two lines:

```
#include stdio.h // note that quotes are not required
printf("Hello World\n");
```

but these two lines are not legal in immediate mode:

```
for(i=0; i<10; i++)
    printf("i=%d\n", i);
```

so they could be correctly and usefully entered as one line:

```
for(i=0; i<10; i++) printf("i=%d\n", i);
```

Note that the trailing semicolon (";") is not required for immediate mode statements:

- If you do not specify a trailing semicolon then one will be automatically added if required and when the statement has completed, the return value will be output.
- If you do enter a trailing semicolon then the display of the return value will be suppressed.

Note that the immediate mode script pane supports some limited line-editing capability including:

- <left-arrow>, <right arrow>, in-place insert/delete of characters on current line.
- <home> and <end> keys move to start or end of current line
- <esc> to abort the current line (also forces GraphMode to default visible combine mode)
- <up-arrow>, <down-arrow> to recover previous lines from the command history

Loading and/or running scripts

The following three methods are available for loading and/or running scripts:

Using autoexec.eic

When the Debugger starts up it will look for the file autoexec.eic in the same directory as the Debugger executable (i.e. in the directory where your ps2dbg.exe resides). If it finds this file it will automatically execute it.

At this point you can set any EiC interpreter settings and include any files you wish pre-included. Scripts run in this mode have their stdout directed to the DBG TTY stream.

Note: In a manual install you will need to set the default include path in the autoexec.eic script file before you can include any header files.

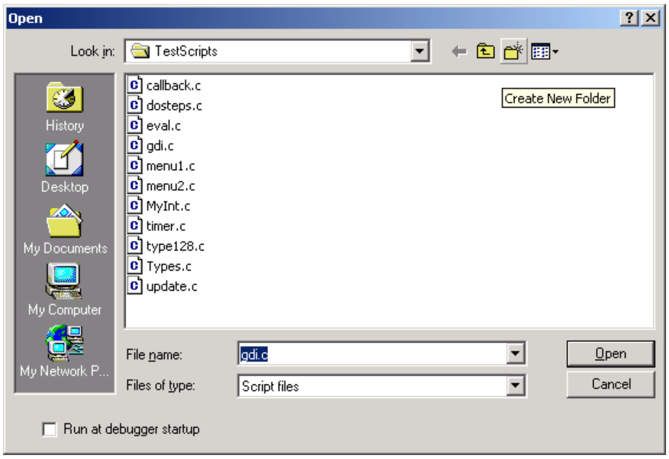
From ps2dbg command line

By using the ps2dbg –Sscriptname command line switch you can specify scripts to be run immediately after the Debugger has started and connected to the target console. Just as with autoexec.eic, scripts run in this mode have their stdout directed to the DBG TTY stream. They will be run in left to right order as specified on the command line.

Run script dialog from shortcut menu

If you have a script pane open you can run a script in that context by selecting **Run script** from the pane's shortcut menu. See "Script pane shortcut menu" on page 147.

The Run script dialog will be displayed allowing you to specify the name of a script file to be run, similar to the following:



This dialog has a **Run at debugger startup** option that you can set to cause that script pane to be loaded and run with a particular script at startup. Unlike the command-line scripts these scripts would default to using the script pane for their output and so have the advantage that they can hook callbacks and do GDI output.

Loading scripts at debugger startup

The order of scripts run at debugger startup is:

1. autoexec.eic from the directory containing the debugger executable file.

2. scripts specified on the ps2dbg command line using the -Sscriptname switch.

3. scripts started with **Run at debugger startup** set on the Run Script dialog.

Supported library functions

Standard C library functions

Although standard C library functions are effectively built in to the interpreter you will still need to #include the correct header files to declare their types before you can use them. If a header file has been previously included you do not need to include it again.

To become familiar with the supported set of standard library functions then please study the header files.

SN Systems functions

To enable scripts to do useful things in the ProDG Debugger, SN Systems have added a built-in library of functions that expose some of the Debugger functionality. These functions are declared in the header file snscript.h:

```
// For hooking scripts into the debugger
int   SNSetCallback(char* pFuncName);
int   SNHookMessage(int message);
int   SNUnhookMessage(int message);
int   SNBindKey(int keycode, int keydata, char* pstatement);
int   SNSetstdout(int ttychannel);

// For text output
int   SNTxtSetPos(int xpos, int ypos);
int   SNTxtHome(void);
int   SNTxtClrEol(void);
int   SNTxtClrEop(void);
int   SNTxtSetConsole(int width, int height);
int   SNTxtSetTTY(int buffersize);

// For using debugger functions
char* SNEvaluate(snval_t* result, int unitnum, char* pexpstr);
int   SNGetMemory(void* buffaddr, int unitnum, unsigned int
addr, unsigned int count);
int   SNSetMemory(void* buffaddr, int unitnum, unsigned int
addr, unsigned int count);
int   SNLoadElf(int unit, char* pfilename, char* pcmdline, int
flags);
int   SNLoadBin(int unit, char* pfname, unsigned int address);
int   SNSaveBin(int unit, char* pfname, unsigned int address,
unsigned int count);
void  SNUpdateWindows(int mask, int unitnum);
void  SNRefreshAll(void);
int   SNSetTypeDisplay(char* ptypestring, char* ptemplate);
int   SNGetTypeDisplay(char* pdest, char* ptype, int index);
int   SNDelTypeHandlers(char* ptype);

// execution control
int   SNSetBP(int unit, unsigned int addr);
int   SNClrBP(int unit, unsigned int addr);
int   SNIsRunning(int unit);
int   SNStart(int unit);
int   SNStop(int unit);
int   SNStep(int unit, int flags);
int   SNStepOver(int unit, int flags);
int   SNAddr2Line(int unit, unsigned int addr, char**
pnamebuff, unsigned int* pline);
int   SNLine2Addr(int unit, unsigned int* paddr, char* pname,
unsigned int line);
int   SNAddr2Label(int unit, unsigned int addr, char* pout,
int* plen);
int   SNRun2Addr(int unit, unsigned int addr);

// For extending debugger context menus
int   SNAddMenu(int panetype, char* menutext, char*
submenutext, char* script);
int   SNGetMenuInfo(int panetype, int command, char* menutext,
char* submenutext);
int   SNGetPaneInfo(unsigned int hwnd, struct paneinfo * pout);
int   SNGetPaneTitle(unsigned int hwnd, char* pout);

//Other Things that are useful
char* getcwd( char* buffer, int maxlen);
int   chdir( char* dirname);
int   ShellExecute(char* pverb, char* pfname, char* pparms, int
showcmd);
int   SNExit(int retval);  // abort the debugger (parameter
currently ignored)

// GDI functions for graphical output to a script pane:-
int    SNGetWindowSize(int* width, int* height);
int    SNCreateBmp(int width, int height);
int    SNClearBmp(void);
int    SNSetPixel(int x, int y);
int    SNSetLine(int x0, int y0, int x1, int y1);
int    SNSetColor(int red, int green, int blue);
int    SNRectangle(int x0, int y0, int x1, int y1);
int    SNEllipse(int x0, int y0, int x1, int y1);
int    SNPolygon(int count, int* points);
int    SNTextOut(int x, int y, char* pstr);
int    SNSetFont(int size, int flags, char* pname);
int    SNEnumFonts(void);
int    SNSetGraphMode(int flags, int mode);
```

The prototypes and return values for these functions are fully described in "*Appendix: Debugger Scripting API*".

Example scripts

Example scripts can be found in the \TestScripts directory. The following example scripts are currently available:

autoexec.eic	Autoexec startup script. Must modify this to install scripting support.
callback.c	Use of SNSetCallback. See "Setting up a timer callback" on page 153.
dosteps.c	Use of SNStep/SNStepOver.
editinterface.c	Interfacing to an external editor.
eval.c	Evaluation of expressions using debugger symbol table. See "Evaluating expressions" on page 154.
gdi.c	Simple graphics output. See "GDI functions" on page 152.
menu1.c	Using script to add functions to context menus. See "Adding a shortcut menu item" on page 157.
menu2.c	Using script to add functions to context menus. GDI output.
myint.c	Custom display format for your own types. See "Set custom display templates for types" on page 155.
type128.c	Custom display format for your own types. See "Set custom display templates for types" on page 155.
types.c	Custom display format for your own types. See "Set custom display templates for types" on page 155.
timer.c	Using timer events. See "Setting up a timer callback" on page 153.
update.c	Forcing debugger windows to update.

GDI functions

By default the Debugger's script pane mostly behaves like a normal Debugger TTY pane. It is however possible to overlay a bitmap layer with graphical output on top of the text-based character-map display. In this mode the script pane is effectively two separate layers, the text layer and the graphics layer. The layers are handled quite separately but are overlaid for display. There are a variety of control options to set how the two layers are combined.

Here is a typical sequence for setting a script pane into graphical output mode:

```
//
// Example graphics output using debugger scripting
//
void MyGraphics()
{
    int x,y;
    if(SNGetWindowSize(&x,&y))   // Get size of this script
pane
    {
        SNCreateBmp(x,y);        // create a bitmap of the same
size
         // Note, default display mode will be TEXT+GRAPHICS,
SRCAND
         // - see SNSetGraphMode() for more info
        SNSetColor(255,255,0);   // Set drawing color (Yellow)
        SNEllipse(0,0,x,y);      // Draw an Ellipse to fill the
                                 // whole pane
        SNSetColor(0,0,255);     // (Blue)
        SNRectangle(x/3,y/3,x/3*2,y/3*2);
                                 // Smaller Rectangle
        SNSetColor(255,0,0);     // (Red)
        SNSetFont(24,0,"Arial"); // select font for graphical
text
        SNTextOut(250,50,"This is a TEST STRING!");
                                 // write some text to the
bitmap
        SNUpdateWindows(int mask, int unitnum);
                                 // force a repaint of the pane
    }
}
```

Note that since the graphical output is to a different ‘layer’ to the normal character map TTY output, the plain text output will scroll independently of the graphical overlay. All the usual TTY features such as scrollback buffering and text selection work as normal.

Setting up a timer callback

A timer can be set up to call back to the pane’s callback function with a SM_TIMER message at a specified interval. This is useful if you want a script pane to periodically update with information retrieved from the target console.

The usual sequence to setup a TIMER callback is shown in this simple example:

```
#include <stdio.h>
#include "SNscript.h"

int MyCallback(SNPARAM message, SNPARAM param0, SNPARAM param1,
SNPARAM param2)
{
    static int count = 0;

    switch(message)
    {
    case SM_TIMER:
        count++;
        printf("Timer callback %08X\n", count);
        break;
    }
    return 0;
}

int main(int argc, char** argv)
{
    SNSetCallback("MyCallback");  // set our callback handler
    SNHookMessage(SM_TIMER);      // hook the "timer" message
    SNSetTimer(100);              // 1 per second

    return 0;
}
```

Evaluating expressions

The SNEvaluate function uses the debugger expression evaluator to evaluate an expression in the context of the current PC.

Examples:

```
snval_t    result;
SNEvaluate(&result, 0, "a");
```

will evaluate the target expression "a" and put the result into the script variable "result".

The next example allows you to evaluate the value of a named register:

```
void dumpreg(char* regname)
{
    snval_t   result;
    unsigned int* addr;
    char* pexperr = NULL;
    unsigned int * pquad;
    char   namebuff[16] = {'$', 0};

    strcpy(namebuff+1, regname);
    pexperr = SNEvaluate(&result, 0, namebuff);
    if(pexperr)   // if error
       printf("%s: %s\n", regname, pexperr);
    else
    {
       pquad = result.val.u128.word;
       printf("%s = %08X_%08X_%08X_%08X\n",
           regname, pquad[3], pquad[2], pquad[1], pquad[0]);
    }
}
```

Then in immediate mode enter the name of the register as an argument to dumpreg(), for example:

```
dumpreg("s0");
```

Assigning new values to target variables

It is also possible for a script to assign a value to a specified target-side variable. To do this just evaluate an assignment expression. Remember that both LHS and RHS of the expression are evaluated in the current target console context.

If the PlayStation 2 application is running this is obviously more useful if the expression is a global variable because there will be no current context scope for local variables. The return value of an assignment expression will be the value that is assigned.

Note also that although the script engine itself does not support bitfields (bitfield structures are compiler-implementation and target-console specific), the debugger expression evaluator *does* understand target bitfield structures so it is legal to use bitfield expressions on either side of an assignment.

```
snval_t    result;
SNEvaluate(&result, 0, "gMyCheat.CurrentLevel=1");
```

will assign the integer value 1 to global variable gMyCheat.CurrentLevel. It will also return the value 1 to script variable "result".

Set custom display templates for types

The SNSetTypeDisplay function can be used to set custom display templates to be used for the specified types in watch or local variable panes. You would use this function if you wish to override the way results of a particular type are displayed in the watch pane or local variable pane. To find out what to pass as the ptypestring string just look at what is displayed in the watch or locals pane in the "type" field.

When a watch or local is evaluated for display the result type string is compared with the list of types you have registered (whitespace in either string is ignored). If a match is found then the corresponding template string is used to display the result.

You can set more than one template for any type. If there is more than one template for a type they will all appear on the context menu (mouse right-click) for that entry.

Note: There are some special cases that apply to the template string

[	If the first character of the template is a '[' then everything from there up to the next matching ']' is ignored but is taken to be a title or name for display on the context menu. If no title is specified then the first non-ws portion of the template is used as a name instead.
:	If the first character (after an optional name field) is a colon then the rest of the template string is taken to be the name of a script function. The debugger will call that script function which should return a pointer to the result string to be displayed in the watch

or local-variables pane. The script function will be passed two parameters.

int unit	index that identifies target CPU (0 is main CPU)
char* pexpr	pointer to nul terminated expression string

The ':' special case is a little more involved to write but allows for great flexibility in display of particular types. The script function that is called can perform very complex calculations that would not be possible to specify with a simple format template.

Template string specifications

This string has a similar format to normal printf format specification but the parameter values are embedded in the string rather than as separate parameters.

The usual % style is used just as with printf. You can use %d, %f, %x, %g, %s etc just as with printf including precision and field width specifiers. However, the value to be printed must be embedded in the string itself between {} curly braces following on from the format specifier.

The string portion enclosed in {} will be passed to the debugger expression evaluator and evaluated in the context of the current PC location on that CPU using the currently active target symbol table. Any '@' characters in that string will be substituted with the watch expression (the value which has been evaluated for display).

Examples:

```
SNSetTypeDisplay("int", "[ABC]a = %d{a}, b = %d{b}, c= %d{c}");
```

This is a pretty pointless template for example only. It is used for all displays of type 'int'. It ignores the watch expression which has been evaluated an always displays the value of three variables. It has an optional name 'ABC' which will appear on the context menu if there is more than one template for this type

```
SNSetTypeDisplay("int", "[NIBREV]:MyIntHandler");
```

This is an example of the script function interface. This can be used to handle things that are too complex for the text template. In this case the handler for type int calls a script function char* MyIntHandler(int unit, char* pexpr); which in this example returns a string based on nibble-reversing the original integer value. This template has a name "NIBREV".

```
SNSetTypeDisplay("class test",
"[.members]a = %d{@.m_a}, b = %d{@.m_b}, c= %d{@.m_c}");
```

This example provides a simple one-line display of three members of the class "test". The class can still be opened in the watch pane to display member values but with this template in operation this is no longer necessary as the default display will show some or all of the values. It has a name ".members".

```
SNSetTypeDisplay("class test *",
"[->members]a = %d{@->m_a}, b = %d{@->m_b}, c= %d{@->m_c}");
```

This example does the same but for pointers to the same class. It has a name "->members".

```
SNSetTypeDisplay("class test", "[string]pstr = %s{@.m_pstr}");
```

The final example provides an alternative display format for the class "test". With two templates available for this class you can choose between them using the right-click context menu. It has a name "string". Note the use of the %s format specifier that will try to display the target memory as an ASCII string.

Adding a shortcut menu item

The SNAddMenu function adds a custom menu entry to the shortcut menu for the specified type of debugger pane.

The return value is the unique command ID assigned to that menu entry. When the menu item is invoked this value will be passed to the callback handler to identify the cause. This ID is only unique to that pane, i.e. menu items on other pane types may share the same ID so the handler must also determine the pane type to uniquely identify the cause.

There are two ways to use this function:

1. If the script parameter is not NULL then it is assumed to be a pointer to a string that is the name of the script function to execute. This method causes a direct call to the specified function and has no owning script pane for context. It is run in the global script context as used for debugger startup scripts. Therefore it cannot output text to a script pane or do graphical output. It can however perform stdout to the DBG TTY channel and it can perform normal file I/O etc.

2. If the script parameter is NULL then when the menu item is selected a SM_MENU callback will be issued. To hook this callback see "SNSetCallback" on page 161. The parameters for this type of callback message are

   msg = SM_MENU
   param0 = handle of pane that own the context menu
   param1 = command id for that menu item
   param2 = WT_ pane type (as specified in SNScript.h)

   The advantages of using a callback rather than a direct function mapping is that callbacks can maintain a permanent context, i.e. they can belong to a particular script pane and they change the state of that pane to use graphics or different types of text output. If you want a menu to cause output (text or graphics) to a dedicated script pane then this is the method to use.

Scripting limitations

Scripting support in ProDG Debugger is still at a comparatively early stage of development and as such its specification is subject to change. SN Systems will try to keep the Debugger scripting API constant where possible but you should be aware that any changes that are made may require you to make minor changes to Debugger scripts for use with future Debugger updates.

The following known limitations apply to the current version:

- Scripts are pointer-safe, however there may be some additional built-in functions for interfacing with the Debugger internals which are not currently safe i.e. if you pass them crazy values you may cause the Debugger to crash or exit.

- There is no bitfield support, i.e. you cannot use structures that have bitfields. Since bitfield implementation in C is compiler dependent anyway it would be a bit much to expect the script structure bitfields to precisely match those of the target system. Note however that the Debugger expression evaluator *does* support bitfields so if you want to evaluate target expressions that use bitfields to integers you can still do so – just submit your expression including bitfields to the expression evaluator.

- The script interpreter runs in the same thread context as the rest of the Debugger so there are no problems with synchronization of target communications. However the scripts can make the Debugger unresponsive if they consume a lot of CPU time so you should avoid writing scripts which block for long periods of time (unless you do not mind that the Debugger will be busy during that time). For the same reason indefinitely blocking functions such as scanf() and getchar() are not currently supported. If you accidentally execute a script pane-based script function that does not exit then you can break-in by typing <Ctrl+C> when the window has keyboard focus.

- Strictly speaking there is only *one* Debugger script. You can load and hook-in more script functions from other files at any time but they all become part of a single script program. If you load a file containing a function with the same name as an existing function, the new function will replace the previous one of that name. From that point on, any function that would previously have called the first instance of that function will now call the new one. You should therefore ensure that all function names (and global variables) in different script files are unique.

  Loading a script file is equivalent to executing these three lines in immediate mode:

```
int main(int argc, char** argv){return 0;}
#include "scriptname.c"
main(argc argv);
```

Any previously existing functions called main() will be deleted. The code in the file scriptname.c will then be loaded and compiled and added to the script functions already loaded. If there is a function main() in the new file, then that function will be called. If there is no function main() then the script will be loaded and compiled but nothing will be executed.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Appendix: Debugger Scripting API

SNSetCallback

Prototype

int **SNSetCallback**(
 char*   pFuncName);

Description

Each script window can have a callback function that can be called in response to certain debugger events. The function name passed to this function will be set as the callback function for any messages that the pane chooses to hook. Passing NULL to this function will delete any callback settings for this script/pane so the pane will not receive any further callbacks.

If the executing script is executing in a script pane and terminates with callbacks hooked then the pane will not be in immediate mode, it will be waiting for callbacks. If the script later cancels callbacks by passing NULL to this function then when that script exits the pane will return to immediate mode. You can manually force it back to immediate mode (thus terminating any callback hooks) at any time by using the context menu **Immediate mode** command.

Details of the callback function implementation:

```
int MyCallback(SNPARAM message, SNPARAM param0, SNPARAM
param1, SNPARAM param2 )
```

Details of parameters will depend upon the message. Return value is currently ignored. For future compatibility you should probably return non-zero.

Parameters

The parameters for this function are as follows:

pFuncName

Input: Pointer to name of the callback function. The callback function is call-time compiled and is therefore a null-terminated string (as a char*) and NOT a function pointer.

Returns

If successful: A value of non-zero is returned. *On failure:* A value of 0. This function will fail if called from a script that does not have an associated debugger pane for context (e.g. a script started from the debugger command line).

See Also SNHookMessage, SNUnhookMessage, SNSetTimer, SNAddMenu

SNHookMessage

Prototype

```
int SNHookMessage(
 int                          message);
```

Description This function specifies messages to be passed to the script callback function (set previously with SNSetCallback()). If a message is not "hooked" then that message will not be passed to the callback function.

Note that a single callback function can receive multiple types of callback message. Just issue a SNHookMessage() call for each message type you wish to hook. You would then need to add conditional code to your callback function (typically a *switch(msg)* statement) to respond to the different types of message.

Parameters The parameters for this function are as follows:

message	*Input:* Message number to intercept.

Message numbers are defined in the SNSCRIPT.H header file:

SM_TARGETATN	Target console sends a status-change notification
SM_CHAR	Character key pressed in script pane
SM_KEYDOWN	Key (including system keys) pressed in script pane
SM_TIMER	Programmable timer interval
SM_MENU	Message from the context menu of a debugger pane

The following messages are reserved:

SM_MOUSECLICK	Mouse clicked in script pane
SM_DBLCLICK	Mouse double-clicked in script pane
SM_MOUSEACTIVATE	Script pane activated

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNUnhookMessage

SNUnhookMessage

Prototype

```
int SNUnhookMessage(
 int                          message);
```

Description

This function removes the specified message from the list of notifications required by the current script callback function, i.e. it undoes the effect of a SNHookMessage() function. Note that removing all the hooked messages does not remove the callback so you can unhook all messages and then hook some more to the same callback function. If you wish to delete the callback (and unhook all messages) you can use SNSetCallback(NULL).

Parameters

The parameters for this function are as follows:

message

Input: Message number to no longer intercept.

Message numbers are defined in the SNSCRIPT.H header file:

SM_TARGETATN	Target console sends a status-change notification
SM_CHAR	Character key pressed in script pane
SM_KEYDOWN	Key (including system keys) pressed in script pane
SM_TIMER	Programmable timer interval
SM_MENU	Message from the context menu of a debugger pane

The following messages are reserved:

SM_MOUSECLICK	Mouse clicked in script pane
SM_DBLCLICK	Mouse double-clicked in script pane
SM_MOUSEACTIVATE	Script pane activated

Returns

If successful: A value of non-zero is returned. *On failure:* A value of zero.

See Also

SNHookMessage

SNSetTimer

Prototype

```
int SNSetTimer(
 int                          interval);
```

Description

This function sets up a timer that will callback to the pane's callback function with a SM_TIMER message at the specified interval. This is useful if you want a script pane to periodically update with information retrieved from the target console.

Note: If you specify an interval of zero the timer will be deleted. The callback will not be deleted though so a subsequent SNSetTimer will resume.

Parameters The parameters for this function are as follows:

interval	*Input:* Time interval in milliseconds.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNSetCallback, SNHookMessage, SNUnhookMessage

SNBindKey

Prototype

```
int SNBindKey(
 int                          keycode,
 int                          keydata,
 char*                        pstatement);
```

Description When the specified key is pressed the statement you provide will be executed. Keybindings do not require a script-pane context, i.e. they can be setup in debugger startup or command-line scripts as well as in script panes during debugger execution.

Example to bind <Shift+Ctrl+F2> to call 'myfunc()':

```
#include <snscript.h>
#include <vk.h>
SNBindKey(VK_F2, KS_SHIFT|KS_CONTROL, "myfunc();");
```

Note: the three keys 'Esc','<' (comma) and '>' (dot) are reserved. It is not possible to bind these keys to custom functions.

Parameters The parameters for this function are as follows:

keycode	*Input:* Key code to intercept See Microsoft Windows definitions in WinUSer.h. For your convenience a useful subset of these scancode values are defined in the header file vk.h that is included with the

ProDG Debugger. See the contents of vk.h for more details.

keydata — *Input:* Bit flags to indicate whether SHIFT, CTRL or ALT keys are pressed. These flags can be OR'd together to indicate combinations of shift keys. See definitions in snscript.h.

pstatement — *Input:* Statement to execute when key pressed. The statement you provide will be call-time compiled so it is a simple null-terminated character string, and not a function pointer. C statements should include the trailing ';' character if appropriate. If set to NULL, indicates that current keybinding for that key should be deleted.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNSetstdout

Prototype

```
int SNSetstdout(
 int                          ttychannel);
```

Description By default a script started in a script pane will use that pane for its stdout display and a script started on debugger command line will use the DBG TTY stream for its stdout output. A script can change this behavior by specifying a particular debugger TTY stream to use for output.

Parameters The parameters for this function are as follows:

ttychannel — *Input:* TTY stream to use for output.

Example TTY channel indexes:

```
ALLTTY        0 // buffer 0 gets all console output
DBGTTY        1 // buffer 1 gets DBG internal diagnostic text
TMTTY         2 // buffer 2 gets Target Manager output
MAINTTY       3 // buffer 3 gets main CPU
IOPTTY        4 // buffer 4 gets IOP
LOGTTY        5 // buffer 5 gets LOG stream
```

Returns Previous TTY channel.

See Also SNTxtSetConsole, SNTxtSetTTY

SNTxtSetPos

Prototype

```
int SNTxtSetPos(
 int                  xpos,
 int                  ypos);
```

Description Set cursor position for further text output. This function is only useful in 2D textmap (console) mode. It has no effect if stdout is directed to a TTY buffer or the script pane is in TTY buffer mode.

Parameters The parameters for this function are as follows:

xpos	*Input:* Cursor horizontal position (0 to window width)
ypos	*Input:* Cursor vertical position (0 to window width)

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script is not associated with a debugger pane e.g. if the script is executed from the debugger command line.

See Also SNTxtSetConsole, SNTxtSetTTY, SNTxtHome, SNTxtClrEol, SNTxtClrEop

SNTxtSetConsole

Prototype

```
int SNTxtSetConsole(
 int                  width,
 int                  height);
```

Description Puts the script pane into 2D console output mode. A textmap of the appropriate size is allocated and used for all further text output to that window. If the current script window size is smaller than the new textmap the debugger window will show scrollbars to allow the user to scroll over the textmap. This mode is sometimes more useful for formatted text displays but does not support a scrollback buffer. Note that if a pane is switched from TTY mode to Console mode the TTY buffer is not deleted; therefore a future SNTxtSetTTY(0) can restore the previous TTY-stream based display.

Parameters	The parameters for this function are as follows:
width	*Input:* Width of window buffer in characters.
height	*Input:* Height of window buffer in characters.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script is not associated with a debugger pane e.g. if the script is executed from the debugger command line.
See Also	SNTxtSetTTY

SNTxtSetTTY

Prototype

```
int SNTxtSetTTY(
 int                           buffersize);
```

Description

Set the current script pane into TTY output mode. Any 2D textmap currently attached to the pane will be discarded and the stdout for this pane will instead be sent to a private TTY stream. This mode can have a large scrollback buffer but may be less convenient for 2D text output if displaying formatted text.

If the script is currently set to output to a standard debugger TTY stream then this command will redirect it to the private TTY channel specific to this pane.

Parameters	The parameters for this function are as follows:
buffersize	*Input:* Size of TTY buffer. Minimum is 1024 (1KB), maximum is 1048575 (1MB). You can also specify buffer size as zero; this will leave the buffer size unmodified but will switch script pane output from 2D console mode to TTY buffer mode.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script is not associated with a debugger pane e.g. if the script is executed from the debugger command line.

See Also	SNTxtSetConsole

SNTxtHome

Prototype	int **SNTxtHome**(void);
Description	Home the cursor. In 2D console display mode this function will set the cursor position to top left of the text map. In TTY stream mode this function will clear the TTY buffer, therefore discarding any previously stored text history. Further text output will begin again from the start of the buffer. The size of the buffer will not be changed.
Parameters	None.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script is not associated with a debugger pane e.g. if the script is executed from the debugger command line.

SNTxtClrEol

Prototype	int **SNTxtClrEol**(void);
Description	Clear text to end of line (EOL). In 2D console display mode this function will clear the textmap from the current cursor position to the end of the line, i.e. to the right edge of the textmap. In TTY stream output mode this function has no effect and will return zero. This function is particularly useful when overwriting previous text display.
Parameters	None.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNTxtClrEop

Prototype	int **SNTxtClrEop**(void);
Description	Clear text to end of page (EOP). In 2D console display mode this function will clear the textmap from the current cursor position to the end of the page, i.e. to the end of the textmap. In TTY stream output mode this function has no effect and will return zero. This function is particularly useful when overwriting previous text display.
Parameters	None.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNEvaluate

Prototype

```
char* SNEvaluate(
 snval_t*                result,
 int                     unitnum,
 char*                   pexpstr);
```

Description	This function uses the debugger expression evaluator to evaluate an expression in the context of the current PC (i.e. scope is set according to current program counter value) on the specified CPU. Note: some targets only have one possible CPU context, which should be specified as 0.
Parameters	The parameters for this function are as follows:
result	*Input:* Pointer to buffer to hold the result. *Output:* Result.
unitnum	*Input:* Index that identifies target CPU (0 is main CPU).
pexpstr	*Input:* Pointer to null-terminated text expression to evaluate.
Returns	*If successful:* NULL pointer. *On failure:* Pointer to a plain text error message.

SNGetMemory

Prototype

```
int SNGetMemory(
 void*          buffaddr,
 int            unitnum,
 UINT           addr,
 UINT           count);
```

Description Fetches memory from the target to a buffer on the host PC.

Parameters The parameters for this function are as follows:

buffaddr	*Input:* Pointer to destination buffer on host.
unitnum	*Input:* Index that identifies target CPU (0 is main CPU).
addr	*Input:* Memory address on target CPU.
count	*Input:* Number of bytes to fetch.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNSetMemory

SNSetMemory

Prototype

```
int SNSetMemory(
 void*          buffaddr,
 int            unitnum,
 UINT           addr,
 UINT           count);
```

Description Sends memory from a buffer on the host PC to memory on the target CPU.

Parameters The parameters for this function are as follows:

buffaddr	*Input:* Pointer to source buffer on host.
unitnum	*Input:* Index that identifies target CPU (0 is main CPU).
addr	*Input:* Memory address on target CPU.
count	*Input:* Number of bytes to send.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNGetMemory

SNLoadElf

Prototype

```
int SNLoadElf(
 int               unit,
 char*             pfilename,
 char*             pcmdline,
 int               flags);
```

Description This function loads an ELF executable file into the debugger and/or target CPU. You can also load an IRX module to the IOP by specifying the value 3 for the unit parameter.

Parameters The parameters for this function are as follows:

unit	*Input:* Index identifying target CPU (0 is main CPU, 3 is IOP).
pfilename	*Input:* Name of the file to load.
pcmdline	*Input:* Command line to be passed to that executable.
flags	*Input:* Flags to set how it is to be loaded. The flags value can indicate whether to load symbols into the debugger and/or code into the target console, and/or reset the target CPU first etc. See below:

LFLOAD	// load code
LFSYM	// load symbols
LFEXEC	// execute after code loaded
LFSTARTUP	// load & exec but bp at main/start (PS2 EE & IOP)
LFRESET	// reset before loading code
LFRESETFS	// reset FileServer root directory
LFIMPBPS	// import BPs from MS Visual Studio (PS2 EE & IOP)
LFSLOWSYM	// thorough (not shortcut) symbol loading
LFCLRBPS	// clear breakpoints
LFRESETHOME	// reset fileserver home directory

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNLoadBin

Prototype

```
int SNLoadBin(
 int                    unit,
 char*                  pfname,
 UINT                   address);
```

Description Loads a binary image from a file on the host PC to target memory.

Parameters The parameters for this function are as follows:

unit	*Input:* Index identifying target CPU (0 is main CPU).
pfname	*Input:* Pathname of the file to load.
address	*Input:* Memory address on target CPU.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNSaveBin

SNSaveBin

Prototype	int **SNSaveBin**(int unit, char* pfname, UINT address, UINT count);
Description	Saves a binary image from target memory to a file on the host PC.
Parameters	The parameters for this function are as follows:
unit	*Input:* Index identifying target CPU (0 is main CPU).
pfname	*Input:* Pathname of the file to load.
address	*Input:* Memory address on target CPU.
count	*Input:* Number of bytes to save.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNLoadBin

SNStart

Prototype	int **SNStart**(int unit);
Description	Start the specified target CPU (same as pressing RUN in the Debugger).
Parameters	The parameters for this function are as follows:
unit	*Input:* Index identifying target CPU (0 is main CPU).

Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNStop

SNStop

Prototype

```
int SNStop(
 int                          unit);
```

Description	Halt the specified target CPU (same as pressing STOP in the Debugger).
Parameters	The parameters for this function are as follows:
unit	*Input:* Index identifying target CPU (0 is main CPU).
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNStart

SNSetBP

Prototype

```
int SNSetBP(
 int                          unit,
 UINT                         addr);
```

Description	Set breakpoint on target CPU.
Parameters	The parameters for this function are as follows:
unit	*Input:* Index identifying target CPU (0 is main CPU).
addr	*Input:* Memory address on target CPU.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNClrBP

SNClrBP

Prototype

```
int SNClrBP(
 int                          unit,
 UINT                         addr);
```

Description Clear breakpoint on target CPU.

Parameters The parameters for this function are as follows:

unit	*Input:* Index identifying target CPU (0 is main CPU).
addr	*Input:* Memory address on target CPU.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNSetBP

SNIsRunning

Prototype

```
int SNIsRunning(
 int                          unit);
```

Description Use this function to determine whether code on a particular CPU is running or stopped (e.g. at a breakpoint or exception). Note: This function sends a status query command to the target console. If the target console is unresponsive this command can fail in which case it will return –1.

Note: Some targets may not return "running" as the status immediately after being given an SNStart() command. This is due to race conditions in the target OS and is unfortunately out of our control. For this reason issuing an SNStart() immediately followed by calls to SNIsRunning() are not recommended. A better approach is to hook the SM_TARGETATN event using SNSetCallback() and respond to the "target started" and "target stopped" events.

Parameters The parameters for this function are as follows:

unit *Input:* Index identifying target CPU (0 is main CPU).

Returns *If successful:* 1 if target is running, zero if it is halted. *On failure:* A value of -1.

SNUpdateWindows

Prototype

```
void SNUpdateWindows(
 int                  mask,
 int                  unit);
```

Description Updates Debugger window(s). This function is useful if the script has performed some operation that has changed something on the target console and therefore requires the debugger display to be updated to accurately reflect the new state.

Parameters The parameters for this function are as follows:

mask *Input:* Specifies which window types should be updated. The flags below can be combined with logical OR to select a combination of window types that will be forced to update. A mask value of –1 always means all windows (translates to M_ALL). A mask value of 0 just forces an update of the current script pane (useful to force an update of any graphics/bitmap data associated with the pane).

M_ALL	All pane types
M_NONE	None
M_REGISTERS	Register panes
M_MEMORY	Memory panes
M_DISASM	Disassembly panes
M_SOURCE	Source/text file panes
M_WATCH	Watch panes
M_LOCALS	Local variable panes
M_BREAK	Breakpoint panes
M_STACK	Callstack panes
M_PS2TTY	PS2 TTY panes

M_PS2IOP	PS2 IOP modules panes
M_PS2DMA	PS2 DMA panes
M_PS2EEPROFILE	PS2 Profiler panes
M_AGBTTY	AGB TTY panes
M_NGCTTY	NGC TTY panes
M_WORKSPACE	Workspace panes
M_AGBPALETTE	AGB Palette panes
M_SCRIPT	Script panes
M_NGCPROFILE	NGC Profiler panes
M_KERNELPANE	Kernel and Threading info panes

unit *Input:* Index identifying target CPU (0 is main CPU).

Returns None.

See Also SNRefreshAll

SNRefreshAll

Prototype void **SNRefreshAll**(void);

Description This function causes the debugger to discard everything it knows about the target console, then re-fetch and redraw everything. This can be a convenient alternative to SNUpdateWindows but also has the added advantage of causing the debugger to re-fetch the entire console state (on PlayStation 2 this also discards any DMA chain contents forcing a re-parse and also discards the known IRX modules list forcing them to be re-enumerated).

- Flush debugger memory cache for all target units
- Reload source files for any open source panes
- Flush all DMA panes
- Flush all cached IOP module info
- Update all debugger windows

Parameters None.

Returns None.

See Also SNUpdateWindows

SNSetTypeDisplay

Prototype	int **SNSetTypeDisplay**(char* ptypestring, char* ptemplate);
Description	This function can be used to set custom display templates to be used for the specified types in watch or local variable panes. You would use this function if you wish to override the way results of a particular type are displayed in the watch pane or local variable pane. To find out what to pass as the ptypestring string, look at what is displayed in the watch or locals pane in the type field. When a watch or local is evaluated for display the result type string is compared with the list of types you have registered (white space in either string is ignored). If a match is found then the corresponding template string is used to display the result. You can set more than one template for any type and they will all appear on the context menu (mouse right-click) for that entry. For further details and examples, see "Set custom display templates for types" on page 155.
Parameters	The parameters for this function are as follows:
ptypestring	*Input:* Pointer to string type-name.
ptemplate	*Input:* Pointer to format specification or function name.
Returns	Updated number of templates assigned to this type.
See Also	SNGetTypeDisplay, SNDelTypeHandlers

SNGetTypeDisplay

Prototype	int **SNGetTypeDisplay**(char* pdest, char* ptype, int index);

Description	If the Nth template (N specified by the index value) for type at ptype can be found it will be copied to the destination buffer.
Parameters	The parameters for this function are as follows:
pdest	*Input:* Pointer to destination buffer to receive the template
ptype	*Input:* Pointer to type string, as for SNSetTypeDisplay
index	*Input:* Index number of template to return
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNSetTypeDisplay, SNDelTypeHandlers

SNDelTypeHandlers

Prototype

int **SNDelTypeHandlers**(
 char*   ptype);

Description	This function will delete *all* the templates for the type specified at ptype.
Parameters	The parameters for this function are as follows:
ptype	*Input:* Pointer to type string, as for SNSetTypeDisplay.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNSetTypeDisplay, SNGetTypeDisplay

getcwd

Prototype	char* **getcwd**(char* buffer, int maxlen);

```
char* getcwd(
 char*                  buffer,
 int                    maxlen);
```

Description This function returns the current working directory. If unchanged this will be the current directory assigned to the debugger process.

Parameters The parameters for this function are as follows:

buffer	*Input:* Destination char buffer
maxlen	*Input:* Maximum length to be copied into that buffer

Returns Pointer to the destination buffer that was passed in.

See Also chdir

chdir

Prototype

```
int chdir(
 char*                  dirname);
```

Description This function sets the current working directory. This changes the current working directory for the Debugger process so if you change the directory but do not change it back afterwards you may affect the location where configuration files will be saved when the Debugger exits.

Parameters The parameters for this function are as follows:

dirname	*Input:* Pointer to directory name to be set as current working directory.

Returns *If successful:* Returns a value of 0. *On failure:* A return value of –1 indicates that the specified path could not be found

See Also getcwd

ShellExecute

Prototype

```
int ShellExecute(
 char*                  pverb,
 char*                  pfname,
 char*                  pparms,
 int                    showcmd)
```

Description This function provides an interface to the MS Windows ShellExecute() API function. This allows the script to launch other applications and also to perform shell actions such as sendmail: open: etc. For details please refer to those API docs (and check values in SNScript.h).

Parameters The parameters for this function are as follows:

pverb	Refer to Microsoft documentation.
pfname	Refer to Microsoft documentation.
pparms	Refer to Microsoft documentation.
showcmd	Refer to Microsoft documentation.

Returns Refer to Microsoft documentation.

Examples You can type the following in the script pane in immediate mode:

```
// use the header file that declares the SN built-in
// functions
#include SNscript.h

// open a text file in the default associated
// application
ShellExecute("open","c:\\readme.txt",NULL,SW_SHOW)

// launch a Windows executable program file
ShellExecute("open","c:\\myprogram.exe",NULL,SW_SHOW)

// edit a text file in the default editor (usually
// Notepad)
ShellExecute("edit","c:\\autoexec.bat",NULL,SW_SHOW)
```

SNAddMenu

Prototype

int **SNAddMenu**(

int	panetype,
char*	menutext,
char*	submenutext,
char*	script);

Description

This function adds a custom menu entry to the context menu for the specified type of debugger pane. If your combination of menutext and submenutext identifies an item that already exists then it will be replaced.

The return value is the unique command ID assigned to that menu entry. When the menu item is invoked this value will be passed to the callback handler to identify the cause. This ID is only unique to that pane, i.e. menu items on other pane types may share the same ID so the handler must also determine the pane type to uniquely identify the cause.

For further details and examples, see "Adding a shortcut menu item" on page 157.

Parameters

The parameters for this function are as follows:

panetype	*Input:* WT_ pane type. See SNScript.H for details.
menutext	*Input:* Text to be displayed on context menu. Note that the menutext is used to uniquely identify each menu entry so you cannot add multiple menu entries with the same text.
submenutext	*Input:* Text to be displayed on submenu. To add multiple entries to a single submenu just specify the same menutext but different submenutext for each. If NULL there is no submenu, just a new entry to the top level of the context menu.
script	*Input:* Name of function to be executed. Note that since this call is call-time compiled it is a char* that is the name of the function and is NOT a function pointer. If NULL a SM_MENU callback will be issued.

Returns	*If successful:* command code assigned to this menu item. *On failure:* -1. Note that this function will fail if the maximum number of menu items (currently 16 main menu x 16 submenu items) is exceeded.
See Also	SNGetMenuInfo, SNGetPaneInfo, SNGetPaneTitle

SNGetMenuInfo

Prototype

```
int SNGetMenuInfo(
 int                  panetype,
 int                  command,
 char*                menutext,
 char*                submenutext);
```

Description	This function translates the unique command ID for a pane into the menu entry that caused it. This is intended as a possible way to allow the callback handler to confirm that a particular menu item is the cause of the callback.
Parameters	The parameters for this function are as follows:
panetype	*Input:* WT_ pane type. See SNScript.H for details.
command	*Input:* Command ID for that menu item.
menutext	*Input:* Pointer to buffer to receive menu text. *Output:* Menu text in buffer.
submenutext	*Input:* Pointer to buffer to receive submenu text. *Output:* Submenu text in buffer.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNAddMenu

SNGetPaneInfo

Prototype	int **SNGetPaneInfo**(UINT hwnd, struct paneinfo* pout);
Description	This function is provided to allow the callback handler to obtain more information about the pane that caused the SM_MENU callback.
Parameters	The parameters for this function are as follows:
hwnd	*Input:* Handle to uniquely identify calling pane.
pout	*Input:* Pointer to structure to receive information. *Output:* Pane information. The contents of the paneinfo structure may change, so see the definition in SNScript.H for more details.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNGetPaneTitle

SNGetPaneTitle

Prototype	int **SNGetPaneTitle**(UINT hwnd, char* pout);
Description	This function is provided to allow the callback handler to obtain the titlebar text of the pane that caused the SM_MENU callback. With some panes this is a useful way to obtain extra pane-specific information about the pane (e.g. source file and line of a source pane, byte/word mode of a memory pane etc.).
Parameters	The parameters for this function are as follows:
hwnd	*Input:* Handle to uniquely identify calling pane.

pout	*Input:* Pointer to buffer to receive title string. *Output:* Pane titlebar.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNGetPaneInfo

SNStep

Prototype

```
int SNStep(
 int                    unit,
 int                    flags);
```

Description This function causes the specified CPU to single-step one instruction or one source line.

Parameters The parameters for this function are as follows:

unit	*Input:* Unit number of CPU to step (0 for EE core).
flags	*Input:* 0 = asm level, 1 = source level.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNStepOver

SNStepOver

Prototype

```
int SNStepOver(
 int                    unit,
 int                    flags);
```

Description This function causes the specified CPU to step over one instruction or one source line.

Parameters	The parameters for this function are as follows:
unit	*Input:* Unit number of CPU to step (0 for EE core).
flags	*Input:* 0 = asm level, 1 = source level.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNStep

SNAddr2Line

Prototype

```
int SNAddr2Line(
 int                    unit,
 UINT                   addr,
 char**                 pnamebuff,
 UINT*                  pline);
```

Description	This function converts an address to a filename/line number.
Parameters	The parameters for this function are as follows:
unit	*Input:* Unit number of CPU for context (0 for EE core).
addr	*Input:* Address to look up.
pnamebuff	*Input:* Pointer to char* which will be set to point to the filename string. *Output:* Filename.
pline	*Input:* Pointer to unsigned int variable that will be set to the line number. *Output:* Line number.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNLine2Addr

Prototype	int **SNLine2Addr**(

```
int         unit,
UINT*       paddr,
char*       pname,
UINT        line);
```

Description This function converts a filename/line number to an address.

Parameters The parameters for this function are as follows:

unit	*Input:* Unit number of CPU for context (0 for EE core).
paddr	*Input:* Pointer to unsigned int that will be set to the address. *Output:* Address.
pname	*Input:* Pointer to NUL-terminated filename string.
line	*Input:* Line number.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNAddr2Label

Prototype	int **SNAddr2Label**(

```
int         unit,
UINT        addr,
char*       pout,
UINT*       plen);
```

Description This function returns the label corresponding to the specified address. The label will typically be an assembly language label or a C/C++ function name. If the return is a C++ function name then it will be the demangled form including ( ) and parameter types.

Note: Either or both of the pointer parameters are optional and can be passed as NULL if not required.

Parameters	The parameters for this function are as follows:
unit	*Input:* Unit number of CPU for context (0 for EE core).
addr	*Input:* Address to look up.
pout	*Input:* Pointer to buffer that will receive the label as ASCIIZ string, or NULL if not required. *Output:* Label.
plen	*Input:* Pointer to unsigned int variable that will receive the variable length, or NULL if not required. *Output:* Length of label.
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNRun2Addr

Prototype

```
int SNRun2Addr(
 int                unit,
 UINT               addr);
```

Description

This function sets a temporary breakpoint at the specified address and starts the target CPU. It will behave similarly to the "Run to Cursor" feature of the disassembly and source panes. Since the breakpoint is temporary it will be automatically removed when the target stops.

Note: This function returns immediately. If you want your script to know when the operation has completed then you can use the SNSetCallback() function to hook the SM_TARGETATN message.

Parameters	The parameters for this function are as follows:
unit	*Input:* Unit number of CPU for context (0 for EE core).
addr	*Input:* Address to run to.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNExit

Prototype

```
int SNExit(
 int                 retval);
```

Description This function unconditionally exits the Debugger. This is provided mainly to allow you to use the Debugger as a standalone script engine, e.g. a startup script can be specified on the Debugger command line. The Debugger will connect to a target console, execute the script, and then exit.

Parameters The parameters for this function are as follows:

retval	*Input:* Application return value (currently ignored).

Returns None.

SNGetWindowSize

Prototype

```
int SNGetWindowSize(
 int*                width,
 int*                height);
```

Description This function obtains the current size of the script pane in pixels. This information is useful to set the size of a bitmap layer that you will create for graphical output.

Parameters The parameters for this function are as follows:

width	*Input:* Pointer to variable to receive width of script pane.
height	*Input:* Pointer to variable to receive height of script pane.

Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script context does not have a pane associated with it (e.g. if executed in the autoexec script or via a script specified from the Debugger command line).
See Also	SNCreateBmp

SNCreateBmp

Prototype

```
int SNCreateBmp(
 int                        width,
 int                        height);
```

Description

This function creates a bitmap to be associated with the script pane it is running from. The bitmap will be of the size specified and the default display mode will be TEXTON+GRAPHON and SRCAND i.e. both the text and graphics layers will be enabled and the combine-mode will be set to display both layers in a fairly sensible way. The bitmap will be cleared to the default background color.

If the pane already has a bitmap then the previous bitmap will be discarded and a new bitmap of the specified size will replace it. If this function receives parameters of width=0 and height=0 then it will delete the bitmap associated with that text pane, i.e. there will no longer be a graphical layer associated with that pane; the pane will effectively become a normal text-only script pane.

Parameters

The parameters for this function are as follows:

width	*Input:* Width of bitmap to create.
height	*Input:* Height of bitmap to create.

Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script context does not have a pane associated with it (e.g. if executed in the autoexec script or via a script specified from the Debugger command line).
See Also	SNGetWindowSize, SNSetGraphMode

SNSetGraphMode

Prototype

```
int SNSetGraphMode(
 int                        flags,
 int                        mode);
```

Description

This function sets the display mode for the text (character map) and graphics layers associated with the script pane. The flags value enables you to set the pane to display just text, just graphics, or both and also whether to stretch the bitmap to the same size as the text character map. The mode value specifies how the two layers are to be combined if both text and graphics are enabled for display.

This function can only succeed if the executing script has an associated script pane with a currently active bitmap. Choosing to display just text will disable the graphics display but does not delete the bitmap or prevent further drawing to it; i.e. the bitmap remains intact behind the scenes and the display can be enabled by a further call to SNSetGraphMode().

SRCAND is the most sensible looking combine mode as it results in a standard combination of the two layers. The default graphmode after a SNCreateBitmap() call, is as if you had executed:

```
SNSetGraphMode(TEXTON+GRAPHON, SRCAND);
```

Parameters

The parameters for this function are as follows:

flags

Input: Specifies text or graphics or both. Values (defined in SNScript.h) are as follows. Options can be combined with logical OR:

TEXTON	Enable display of text (character map) layer
GRAPHON	Enable display of bitmap (graphics) layer
STETCHON	Stretch bitmap to fill script pane

mode

Input: Specifies the combine mode for text + graphics. Combine modes for graph + text in script pane (destination = text pane, source = bitmap):

SRCCOPY	dest = source
SRCPAINT	dest = source OR dest
SRCAND	dest = source AND dest
SRCINVERT	dest = source XOR dest
SRCERASE	dest = source AND (NOT dest)
NOTSRCCOPY	dest = (NOT source)
NOTSRCERASE	dest = (NOT src) AND (NOT dest)
MERGEPAINT	dest = (NOT source) OR dest

DSTINVERT	dest = (NOT dest)
BLACKNESS	dest = BLACK
WHITENESS	dest = WHITE

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNGetWindowSize, SNCreateBmp

SNClearBmp

Prototype int **SNClearBmp**(void);

Description This function erases the bitmap layer of the script pane i.e. it clears the bitmap for the associated script pane to the pane background color.

Parameters None.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero. This function will fail if the script context does not have a pane associated with it (e.g. if executed in the autoexec script or via a script specified from the Debugger command line). It will also fail if the associated script pane does not have a bitmap layer.

SNSetColor

Prototype

```
int SNSetColor(
 int                red,
 int                green,
 int                blue);
```

Description This function sets the current drawing color. This color will be used for all further drawing commands.

Parameters The parameters for this function are as follows:

red	*Input:* Red value (0 to 255)

green	*Input:* Green value (0 to 255)
blue	*Input:* Blue value (0 to 255)

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero. This only succeeds if the executing script has an associated script pane with a currently active bitmap.

SNSetPixel

Prototype

```
int SNSetPixel(
 int                 x,
 int                 y);
```

Description This function sets a single pixel to the current drawing color. As with other drawing commands the output is clipped to the size of the pane's bitmap. Parameters that are out of bounds of the current pane bitmap size just cause no visible output, they are not treated as errors.

Parameters The parameters for this function are as follows:

x	*Input:* x coordinate
y	*Input:* y coordinate

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

See Also SNSetColor, SNSetLine

SNSetLine

Prototype

```
int SNSetLine(
 int                    x0,
 int                    y0,
 int                    x1,
 int                    y1);
```

Description

This function draws a straight line in the current drawing color. As with other drawing commands the output is clipped to the pane's bitmap. Endpoints can be beyond the pane's bitmap area, so only the visible portion of the specified line will be drawn to the window. Negative x and y coordinate values are legal.

Parameters

The parameters for this function are as follows:

x0	*Input:* x coordinate of first endpoint
y0	*Input:* y coordinate of first endpoint
x1	*Input:* x coordinate of second endpoint
y1	*Input:* y coordinate of second endpoint

Returns

If successful: A value of non-zero is returned. *On failure:* A value of zero.

SNRectangle

Prototype

```
int SNRectangle(
 int                    x0,
 int                    y0,
 int                    x1,
 int                    y1);
```

Description This function draws a solid rectangle in the current drawing color. As with other drawing commands the output is clipped to the pane's bitmap. Coordinates can be beyond the pane's bitmap area, so only the visible portion of the specified rectangle will be drawn to the window. Negative x and y coordinate values are legal.

Parameters The parameters for this function are as follows:

x0	*Input:* x coordinate of top left corner
y0	*Input:* y coordinate of top left corner
x1	*Input:* x coordinate of bottom right corner
y1	*Input:* y coordinate of bottom right corner

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNEllipse

Prototype

```
int SNEllipse(
 int                 x0,
 int                 y0,
 int                 x1,
 int                 y1);
```

Description This function draws a solid ellipse in the current drawing color. It draws the largest ellipse that will fit within the rectangle described by the parameters. As with other drawing commands the output is clipped to the pane's bitmap. Coordinates can be beyond the pane's bitmap area, only the visible portion of the specified ellipse will be drawn to the window. Negative x and y coordinate values are legal.

Parameters The parameters for this function are as follows:

x0	*Input:* x coordinate of top left corner

y0	*Input:* y coordinate of top left corner
x1	*Input:* x coordinate of bottom right corner
y1	*Input:* y coordinate of bottom right corner
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

SNPolygon

Prototype

```
int SNPolygon(
 int                    count,
 int*                   points);
```

Description

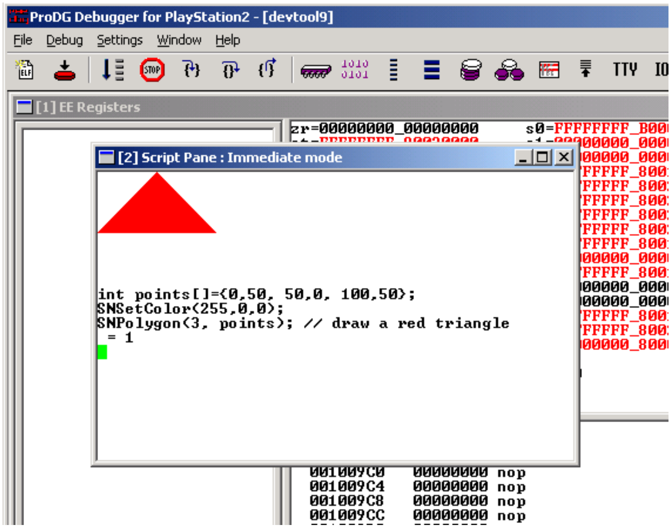
This function draws a solid polygon. The second parameter should point to an array of integer values, two for each point representing pairs of x,y coordinates for each point. See the example below. As with other drawing commands the output is clipped to the pane's bitmap. Coordinates can be beyond the pane's bitmap area, only the visible portion of the specified ellipse will be drawn to the window. Negative x and y coordinate values are legal.

Parameters

The parameters for this function are as follows:

count	*Input:* Number of points in the polygon
points	*Input:* Pointer to array of points
Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.

Example This immediate mode example draws a simple polygon:


ProDG Debugger for PlayStation2 - [devtool9]
File Debug Settings Window Help
[1] EE Registers
[2] Script Pane : Immediate mode

```
int points[]={0,50, 50,0, 100,50};
SNSetColor(255,0,0);
SNPolygon(3, points); // draw a red triangle
 = 1
```

SNTextOut

Prototype

int **SNTextOut**(

int	x,
int	y,
char*	pstr);

Description This function draws text to the specified bitmap coordinates. This text is not part of the pane’s character map, it s rendered graphically to the bitmap layer. As with other drawing commands the output is clipped to the pane’s bitmap. Coordinates can be beyond the pane’s bitmap area, only the visible portion of the specified ellipse will be drawn to the window. Negative x and y coordinate values are legal.

Parameters The parameters for this function are as follows:

x	*Input:* x coordinate of top left corner
y	*Input:* y coordinate of top left corner
pstr	*Input:* Pointer to ACSIIZ text string

Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNSetFont, SNEnumFonts

SNSetFont

Prototype

```
int SNSetFont(
 int                          size,
 int                          flags,
 char*                        pname);
```

Description This function sets the font to be used for further SNTextOut() commands. Some trial and error may be required as not all fonts are available in all sizes. You also need to know the typeface name of the fonts available for use, which may vary from one host system to another. To help you discover the typeface names available we have provided the SNEnumFonts function.

Parameters The parameters for this function are as follows:

size	*Input:* Vertical size of characters in pixels
flags	*Input:* Reserved (should be zero)
pname	*Input:* Pointer to ACSIIZ text string specifying the font name

Returns	*If successful:* A value of non-zero is returned. *On failure:* A value of zero.
See Also	SNTextOut, SNEnumFonts

SNEnumFonts

Prototype int **SNEnumFonts**(void);

Description This function enumerates all the typeface names available and displays those names on stdout (this will usually be the script pane TTY output). This function is provided for diagnostic/test purposes only to help you pick a suitable font for use with SNSetFont.

Returns *If successful:* A value of non-zero is returned. *On failure:* A value of zero.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Index