Getting Started with

ProDG for PlayStation®2

SN Systems Ltd
Version 2.00
March 2001

ProDGforPS2_GS-V2.00 / Getting Started with ProDG for PlayStation 2 v 2.00 / March 2001

Document change control

Ver.	Date	Changes
2.00	28th Mar 2001	Released with ProDG for PlayStation 2 v2.00

Contents

Preface

Overview of ProDG for PlayStation 2

ProDG for PlayStation 2 is SN Systems' suite of fully featured Win32 tools that enable you to build and debug your games for Sony's PlayStation 2.

The Sony PlayStation 2 GNU tools are currently available only for use under the Linux operating system. This means that, to develop your PlayStation 2 applications, you are obliged either to install Linux on your own PC or to connect to a shared Linux PC, to build and debug your applications.

Using the ProDG tools you can now build your application on a Win32 PC and debug it running directly on the Sony PlayStation 2 DTL-T10000 Development Tool.

Currently ProDG for PlayStation 2 consists of:

- Optional **Visual Studio integration**, so you can choose to integrate the ProDG tools into Microsoft Visual Studio, and thus use a familiar integrated development environment while you are using SN Systems development tools.
- **ProDG Build Tools for PlayStation 2** which include the SN Systems dedicated compiler driver, assemblers and linkers together with a Win32 port of the GNU PlayStation 2 tools.
- **ProDG Command-line Utility** for PlayStation 2.
- **ProDG Target Manager** which enables you to control connection to the PlayStation 2 targets in your network.
- **ProDG Debugger for PlayStation 2**, a fully featured Win32 debugger for debugging your PlayStation 2 applications.

Development of ProDG does not stop there, as the tools will be continually improved and upgraded.

Updates of the tools are regularly made available on the SN Systems web site to be downloaded. These are in the form of zip archives for each new tool version. For more information on how to upgrade ProDG tools over the web, see "Updating ProDG over the web on page 14.

What's new in v2.00

- Visual Studio integration configuration dialog
- Much more extensive assember documentation, including SN assembler directive extensions
- New `ps2cc` options: `-fdefault-single-float` and `-Wpromote-double`
- New `ps2ld` linker
- New ProDG Target Manager dialogs for Add Target and Target properties
- Fully configurable ProDG Debugger panes and accelerator keys
- Syntax coloring and horizontal scroll bar in source pane
- Dialog histories so you don't have to keep rekeying
- New ps2dbg switch: -c preserves case of filenames (for Samba users with case sensitivity turned on)
- New Select PS2 Target dialog box
- Text search features in the source pane
- Floating ToolTip style expression evaluation in a source pane
- Basic EE profiling and new profile pane
- DMA error detection and DMA pane improved documentation
- Improved watch pane

About this manual

This *Getting Started* manual is designed to help you install and set up ProDG for PlayStation 2 together with the Sony libraries and toolchain on your Win32 PC. It goes through the complete installation and set-up process and the rest of the manual describes each ProDG tool individually.

Chapter 1: Installation and getting started	How to install and update ProDG for PlayStation 2 and the Sony toolchain and libraries on your Win32 PC, and update the Sony DTL-T10000 Development Tool if necessary.
Chapter 2: Integrating with Microsoft Visual Studio	How to integrate the ProDG Debugger for PlayStation 2 into Microsoft's integrated development environment, Visual Studio.
***Chapter 3:** ProDG Compilers for PlayStation 2*	How to use the compilers and the `ps2cc` ProDG compiler driver for PlayStation 2.
***Chapter 4:** ProDG Assemblers for PlayStation 2*	How to use the ProDG EE, IOP and VU assemblers.
***Chapter 5:** ProDG Linkers for PlayStation 2*	How to replace the GNU PlayStation 2 linker (`ld`) with the SN Systems' linkers (`ps2link` and `ps2ld`) and how to write `ps2link` linker scripts.

Chapter 6: *ProDG Target Manager for PlayStation 2*	How to use the Target Manager to configure the properties of the PlayStation 2 Development Tools in your network, and manage sessions on them.
Chapter 7: *The ProDG command-line utility*	How to use the command-line utility `ps2run`.
Chapter 8: *ProDG Debugger user interface*	How to use and configure the ProDG Debugger for PlayStation 2.
Chapter 9: *Debugging your program*	How to use the ProDG Debugger to load, run and debug your application on the PlayStation 2 Development Tool.
Chapter 10: *Basic EE profiling*	How to profile code usage in your EE application.
Chapter 11: IOP debugging	How to debug code running on the IOP unit.
Chapter 12: *VU and DMA debugging*	How to debug code running on the VU unit, including DMA transfers.
Appendix: *ProDG Debugger reference*	Describes in detail each of the ProDG Debugger panes, shortcut menus and keyboard shortcuts.

Updates and technical support

There will be regular updates to ProDG for PlayStation 2. These will be available to be downloaded from the technical support area of the SN Systems web site, so remember to check out:

http://www.snsys.com/ps2

We recommend that you make regular use of this service and quickly take advantage of any new features added to the software, report or download bug reports, gain answers to questions that may be causing you difficulty and keep up-to-date on news concerning the development industry.

This product is backed by SN Systems' commitment to continual enhancement, development and technical support.

If you experience any difficulties, please do not hesitate to contact our technical support at SN Systems:

Mail: SN Systems Software Ltd
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: http://www.snsys.com/ps2

E-mail (support): support@snsys.com

E-mail (sales): sales@snsys.com

Release history

Version	Description
2.00	***New features:*** Visual Studio integration configuration dialog Much more extensive assember documentation, including SN assembler directive extensions New ps2cc options: -fdefault-single-float and -Wpromote-double New ps2ld linker New ProDG Target Manager dialogs for Add Target and Target properties Fully configurable ProDG Debugger panes and accelerator keys Syntac coloring and horizontal scroll bar in source pane Dialog histories so you don't have to keep rekeying New ps2dbg switch: -c preserves case of filenames (for Samba users with case sensitivity turned on) New Select PS2 Target dialog box Text search features in the source pane Floating ToolTip style expression evaluation in a source pane Basic EE profiling and new profile pane DMA error detection and DMA pane improved documentation Improved watch pane

Chapter 1: **Installation and getting started**

Overview of the installation

This section describes the procedure that you will need to complete to successfully install (or update) ProDG for PlayStation 2.

1. If you are installing over an old version of ProDG for PlayStation 2, first read "Previous installations" on page 6.
2. Install the Sony PlayStation 2 libraries and toolchain (see “Installing the Sony libraries and toolchain“ on page 6 for more details).
3. Install ProDG for PlayStation 2 on your Win32 machine and set up your license file (see “Installing ProDG for PlayStation 2“ on page 9). This will install the latest version of the Sony EE and IOP toolchains.

These steps are described in more detail in the rest of this chapter. The installation of the Sony PlayStation 2 toolchain and libraries is outlined but you may need to refer to the SCEI installation instructions which are available with the Sony CD-ROM, for more information.

When ProDG for PlayStation 2 has been installed:

1. Build your PlayStation 2 application.
2. Start ProDG Target Manager and configure the PlayStation 2 targets in your network (see "***Chapter 6:*** *ProDG Target Manager for PlayStation 2*" on page 73).
3. Start ProDG Debugger and configure the user interface (see "***Chapter 8:*** *ProDG Debugger user interface*" on page 97).

You will then be in a position to debug your PlayStation 2 application.

System requirements

You must have a Win32-based computer connected to a Sony PlayStation 2 DTL-T10000 Development Tool.

The Win32 machine must have at least the following:

- 486 or higher processor running Microsoft Windows 95

- 16MB of RAM
- 300MB of hard disk space
- CD-ROM drive

However for optimum performance we recommend the following:

- Pentium II (or equivalent) or higher processor
- Windows NT, Windows 95, Windows 98, Windows 2000
- 64MB (128MB for Windows 2000 and NT) of RAM
- 8MB of video memory
- At least 300MB of hard disk space
- CD-ROM drive

Note: *Your monitor must be set to at least High Color (16-bit) resolution in order to view different colored panes in the ProDG Debugger; see "Pane colors and fonts" on page 107.*

If you are planning to use the `ps2link` linker, see also the section "ps2link system requirements" on page 49.

Previous installations

If you have previously installed the Sony tools and libraries we recommend that you rename your `\usr\local` directory tree, and reinstall the latest Sony release to a clean `\usr\local` directory. However, it is generally unnecessary to backup the ProDG tools which should be installed over any existing tools.

Note: *If Sony release a new version of their software you will have to repeat all of the installation steps from the beginning.*

Installing the Sony libraries and toolchain

To enable ProDG for PlayStation 2 to work on the Windows platform you will need to duplicate the directory structure of the Sony PlayStation 2 libraries and toolchain on your Win32 platform.

Installing the Sony libraries

- Create an empty `\usr\local` directory on your hard drive. We recommend that you rename any existing `\usr\local` directory tree out of the way.

The Sony PlayStation 2 libraries are shipped as an archive file on the Sony Libraries CD-ROM. This archive file is called `tlib_nnn.tgz` where `nnn` is a Sony version number. This file is located in the following directory:

```
D:\Run Time\Tools-Programming\Standard_Libraries
```

Simply extract the contents of this `.zip` file to the `\usr\local` directory.

When new library releases become available from Sony you can follow the same procedure to update your \usr\local directory.

Whenever new Sony libraries are released it may be necessary to update the PlayStation 2 Development Tool kernel in ROM. The ProDG Target Manager contains a menu option that enables you to do this from your Win32 machine. For more information see “Flashing the kernel“ on page 89.

Installing the Sony toolchain

The Sony EE and IOP toolchains are now installed by the ProDG for PlayStation 2 installer. See “Installing ProDG for PlayStation 2” on page 9 for information on how to do this.

Move IOP libraries and headers

Next you must move the IOP libraries and headers to the correct location [this is a GNU licensing requirement].

Once you have unzipped the libraries and tools then you will need to copy (or move) the contents of

```
<Drive>:\usr\local\sce\iop\install\include
```

and

```
<Drive>:\usr\local\sce\iop\install\lib
```

directories to:

```
<Drive>:\usr\local\sce\iop\gcc\mipsel-scei-elfl\include
```

and

```
<Drive>:\usr\local\sce\iop\gcc\mipsel-scei-elfl\lib
```

respectively, where <Drive> is the drive that you created the \usr\local directory on.

Updating the Sony libraries and toolchain

Approved PlayStation 2 developers can get library and toolchain updates from the official Sony PlayStation 2 web site for your region.

You do NOT normally need to reinstall ProDG for PlayStation 2 when Sony issue patch and minor upgrades to the libraries. However, we recommend that you DO reinstall ProDG for PlayStation 2 when Sony issue a major new release of the libraries, e.g. from v1.6 to v 2.0. This is because Sony put certain libraries in OLD subdirectories. To avoid "contamination" between full releases it is advisable to do a fresh install so you do not have duplicate libraries in the ..\libs directory and the ..\libs\OLD subdirectory.

When Sony releases new toolchain versions, DO NOT install them until you obtain the equivalent toolchain build from SN Systems. Information about the currently supported version of the Sony toolchain is available from the SN Systems web site.

Working with the ProDG EE and IOP build tools

Once you have completed the installation then you will have a directory structure under `<Drive>:\usr\local\sce` that contains the PlayStation 2 libraries and ProDG build tools.

You can invoke the following tools directly on your game files on your Win32 platform.

`ee-gcc`	The GNU PlayStation 2 EE compiler driver.
`iop-elf-gcc`	The GNU PlayStation 2 IOP compiler driver.
`ps2cc`	The SN Systems PlayStation 2 EE and IOP compiler driver.
`ps2dvpas`, `ee-dvp-as`	The SN Systems and GNU PlayStation 2 dvp assemblers.
`ps2eeas`, `as`	The SN Systems and GNU PlayStation 2 EE assemblers.
`ps2iopas`, `as`	The SN Systems and GNU PlayStation 2 IOP assemblers.
`ps2link`, `ps2ld`, `ld`	The SN Systems and GNU PlayStation 2 linkers.

You can also invoke any of the binary utilities:

`ee-addr2line`	Converts `addr` to line `+ f-name`.
`ee-ar`	Archives for the PlayStation 2 EE processor.
`iop-ar`	Archives for the PlayStation 2 IOP processor.
`ee-c__filt`	Equivalent to `-c++filt`, the C++ demangler.
`ee-nm`	Dumps symbol table of object files from the PlayStation 2 EE processor.
`iop-nm`	Dumps symbol table of object files from the PlayStation 2 IOP processor.
`ee-objcopy`	Copies object files from input to output with processing if required.
`ee-objdump`	Dumps object file info from the PlayStation 2 EE processor.
`iop-objdump`	Dumps object file info from the PlayStation 2 IOP processor.
`ioplibdump`	Determines the external functions that are called from a particular `irx` module.
`ioplibgen`	Produces `.ilb` files and library stubs from Sony IOP `.tbl` table files.
`ee-ranlib`	Generates index to archive, therefore speeding up access.

`ee-readelf`	Displays contents of `.elf` files.
`ee-size`	Displays size of sections in `.elf` files.
`ee-strings`	Dumps strings of printable characters from a file.
`ee-strip`	Removes symbol information from `.elf` files.

ProDG IOP build tools mirror the functionality of the EE build tools function, producing code that runs on the IOP processor.

Building PlayStation 2 samples on your Win32 PC

1. Ensure that you have successfully completed installation and set-up.
2. Navigate to `\usr\local\sce\ee\sample` on your Win32 PC, or `\usr\local\sce\iop\sample`, depending on which samples you wish to build. These directories contain all the sample directories for each PlayStation 2 unit.
3. To build a particular sample program, navigate to the required subdirectory under the `\sample` directory.
4. Type `make` and the sample will be built using the Win32 versions of the PlayStation 2 tools.

Note: *Edit the demo `makefile`s and change the line that reads*
`iop-path-setup > PathDefs || (rm -f PathDefs ; exit 1)`
to
`iop-path-setup`
This is because Windows command line cannot execute the pipe command. Note also that the `iop-path-setup` program is currently just a batch tool, which creates the correct path settings in the current directory. In a future release this tool will be replaced. However, if you do not change the positions of the build tools it will work perfectly.

You can also build the PlayStation 2 samples by invoking the SN compiler driver `ps2cc` from the Windows command line; see "Using the SN Systems compiler ps2cc" on page 27 for help with the `ps2cc` syntax.

Installing ProDG for PlayStation 2

You may have downloaded the ProDG for PlayStation 2 installation program from the SN Systems web site, or you may be running it from CD-ROM.

Note: *If you are currently using an evaluation version of ProDG for PlayStation 2 you are advised to request a license key before carrying out the following installation. This will ensure that you have all the information to hand when you install, and your use of the tools will be continuous. For more information see the SN Systems web site.*

This section assumes that you are installing ProDG for PlayStation 2 from the CD-ROM provided, though the process is the same. Once you have installed the ProDG software you can obtain regular updates to the individual tools from the SN Systems web site. These updates are in the form of zip archives. For more information see “Updating ProDG over the web“ on page 14.

Note: *On Windows NT and Windows 2000, you need to be logged in with Administrator rights to be able to make the necessary changes to the registry and environment variables. Contact your system administrator if you require help setting this up.*

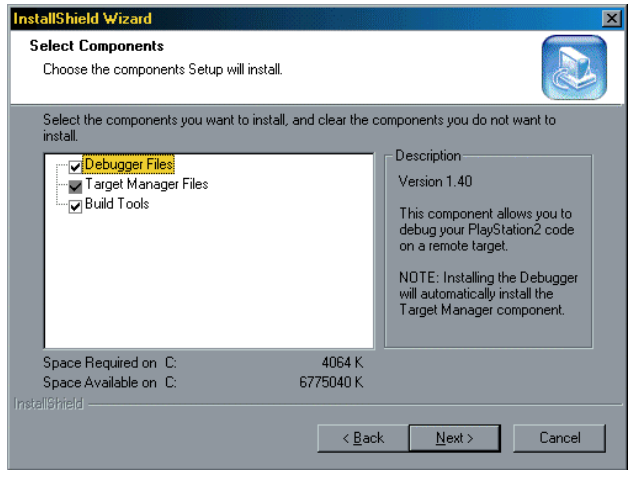
The installer will attempt to write to the system file `\autoexec.bat`. If this is not present on the installation disk, e.g. under Windows ME, please create an empty `\autoexec.bat` and ensure that it is not read-only.

When you put the CD into the drive it should automatically run, and start the installation process. If it doesn't then enter the following into the **Run** dialog accessed from the **Start** menu:

```
E:\setup
```

Where `E` should be replaced by the name of your CD-ROM drive.

1. Click through the instructions until you see the following screen:



2. Clicking on each component will give you a description and version number. of the software it contains. If you select the **Debugger Files** component, the **Target Manager Files** component will also be selected by default, as it is compulsory. Select the required tools. If you don't want to install all the tools now you can install any part of ProDG at a later date by running `SETUP.EXE` and choosing **Modify**.

3. If you selected **Debugger Files**, you will be asked to indicate the directory that you would like it to be installed in. If you have previously installed ProDG Debugger, the install will update it in the same directory. If not, then the default installation directory `C:\Program Files\ProDG for PlayStation2` is proposed.

4. If you selected the **Build Tools** component, the installation will continue and ask you to specify the drive on which you wish these tools to be installed. The drive must be the same drive on which you have already copied the necessary Sony libraries and toolchain (in `\usr\local`, see "Installing the Sony libraries and toolchain" on page 6) and must be in the "C:\" format, i.e. contain a colon-backslash after the drive letter. If you haven't installed the Sony

PlayStation 2 libraries, the ProDG tools cannot be installed. You will need to exit the install and go back and install the PlayStation 2 toolchain and libraries from the Sony CD-ROM.

5. The installer will now detect if you have a previous installation, and try to locate the `sn.ini` file. If it finds one it will offer three choices:
   - **Overwrite the existing file**: The existing `sn.ini` will be replaced by one correct for the current ProDG release.
   - **Save changes to `sn.bak`**: The necessary changes will be saved alongside your existing `sn.ini`, in a file call `sn.bak`, so that you may merge the changes yourself.
   - **Do not modify existing file**: No changes will be made to your existing `sn.ini` file; you will need to manually edit this file to match your installation locations.
6. When the installation is complete you will need to restart your PC unless you are installing to Windows NT or Windows 2000. You can choose to do this at the end of the install process or manually later.
7. Install the **ProDG Visual Studio Integration** if this is required. See "Installation notes" on page 17 for information on how to integrate with Microsoft Visual Studio.
8. Once you have completed the installation and restarted your PC you will need to request a license from SN Systems if you have not yet received one. When you ordered ProDG for PlayStation 2 you would have been provided with a User ID. You will now need to start the ProDG Registration Program and enter your User ID. This can be started via the shortcut that has been automatically put into the **ProDG for PlayStation2** part of the **Start** menu.
9. When you start the program to request a key, it checks your Network card for its unique hardware ID. This is the ONLY information that the program checks for. Once we receive these details from you a key file will be sent back to you. Place this file in the same directory as your software and your product will become officially licensed.
10. The license allows the software to be used only on the machine that you run the registration program on, so if you wish to switch machines at a later date, you will have to contact SN Systems. For more information on licensing see the SN Systems web site.

The sn.ini file

ProDG for PlayStation 2 installs a configuration file called `sn.ini`, which is placed in a directory pointed to by the `SN_PATH` environment variable. The `sn.ini` file is used to support various components of ProDG for PlayStation 2, particularly the Visual Studio integration and the `ps2cc` compiler driver.

These are the default contents of `sn.ini`:

```
[ps2cc]

compiler_path=c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\2.9-ee-
991111
```

```
#compiler_path=c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\2.95.2

c_include_path=c:\usr\local\sce\ee\include;c:\usr\local\sce\
ee\gcc\ee\include;c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\
2.9-ee-991111\include

#c_include_path=c:\usr\local\sce\ee\include;c:\usr\local\sce
\ee\gcc\ee\include;c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\
2.95.2\include

cplus_include_path=c:\usr\local\sce\ee\include;c:\usr\local\
sce\ee\gcc\ee\include;c:\usr\local\sce\ee\gcc\lib\gcc-lib\
ee\2.9-ee-91111\include;c:\usr\local\sce\ee\gcc\include\g++-
2

#cplus_include_path=c:\usr\local\sce\ee\include;c:\usr\local
\sce\ee\gcc\ee\include;c:\usr\local\sce\ee\gcc\lib\gccib\ee\
2.95.2\include;c:\usr\local\sce\ee\gcc\include\g++-2

library_path=c:\usr\local\sce\ee\lib;c:\usr\local\sce\ee\gcc
\ee\lib;c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\2.9-ee-991111

assembler_path=c:\usr\local\sce\ee\gcc\ee\bin

assembler_name=as
#assembler_name=ps2eeas

opt_assembler_name=as
#opt_assembler_name=ps2eeas

dvp_asm_name=ee-dvp-as
#dvp_asm_name=ps2dvpas

dvp_assembler_path=c:\usr\local\sce\ee\gcc\bin

iop_asm_name=as
#iop_asm_name=ps2iopas

linker_name=ld
#linker_name=ps2link

linker_path=c:\usr\local\sce\ee\gcc\bin

linker_script=c:\usr\local\sce\ee\lib\app.cmd
#linker_script=c:\usr\local\sce\ee\lib\ps2.lk

iop_linker_name=ld

startup_module=c:\usr\local\sce\ee\lib\crt0.s

dvp_include_path=c:\usr\local\sce\ee\include

iop_bin_path=c:\usr\local\sce\iop\gcc\mipsel-scei-elfl\bin

iop_lib_path=c:\usr\local\sce\iop\gcc\lib\gcc-lib\mipsel-
scei-elfl\2.8.1

iop_linker_path=c:\usr\local\sce\iop\gcc\mipsel-scei-elfl\
bin

iop_compiler_path=c:\usr\local\sce\iop\gcc\lib\gcc-lib\
mipsel-scei-elfl\2.8.1
```

```
iop_c_include_path=c:\usr\local\sce\iop\gcc\lib\gcc-lib\
mipsel-scei-elfl\2.8.1\include;c:\usr\local\sce\iop\gcc\
mipsel-scei-elfl\include;c:\usr\local\sce\common\include

iop_cplus_include_path=c:\usr\local\sce\iop\gcc\lib\gcc-
lib\mipsel-scei-elfl\2.8.1\include;c:\usr\local\sce\iop\
gcc\mipsel-scei-elfl\include

iop_stdilb=iop.ilb libsd.ilb cdvdman.ilb modhsyn.ilb
modmidi.ilb

stdlib=libgraph.a libdma.a libdev.a libpkt.a libvu0.a
libpad.a

ilb_lib_path=c:\usr\local\sce\iop\gcc\mipsel-scei-elfl\lib

[ps2link]

library_path=c:\usr\local\sce\ee\lib;c:\usr\local\sce\ee\gcc
\ee\lib;c:\usr\local\sce\ee\gcc\lib\gcc-lib\ee\2.9-ee-991111
```

Note: *Lines starting with a '#' are ignored, so you can easily swap between different versions of the build tools, assemblers and linker.*

Troubleshooting installation

Error Number	Description
ERR: 205	Please log on with Administrator rights and run the setup program again.

On Windows NT and 2000, you need to be logged in with Administrator rights to be able to make the necessary changes to the registry and environment variables. Contact your system administrator if you require help setting this up.

ERR: 206 Failed to set `SN_PATH=` environment variable.

You will need to create an `SN_PATH=` environment variable up which contains the path where your `sn.ini` file is located. For example: "`SET SN_PATH=C:\Progra~1\ProDGf~1`"

ERR: 207 Unable to load `\autoexec.bat`.

This can occur under Windows ME. If so, create an `\autoexec.bat` file in the root of your boot drive and re-run the setup program.

ERR: 208 Unable to save `\autoexec.bat`.

Ensure that your `\autoexec.bat` file is not set as read-only.

ERR: 209 The drive you entered does not have a `\usr\local\sce` directory, make sure you enter the correct drive letter. (e.g. `C:\`)

Your Sony software needs to be installed into a `\usr\local\` directory for the ProDG installer to complete successfully. You will need to make sure that this is the case on the drive you specified.

What next?

Once the installation is complete you may need to make some manual changes to your environment, depending on what you specified during the install. You will then need to open ProDG Target Manager and configure at least one target PlayStation 2.

1. The quick way to open the Target Manager is to use the **Start** menu shortcut: **ProDG for PlayStation2 > ProDG Target Manager**.
2. For more information on adding a new target to the Target Manager see "To add a new target" on page 77.
3. Now you can use the ProDG Build Tools to build your application and the ProDG Debugger, ProDG Target Manager and ProDG Command-line utility to load and debug it on the PlayStation 2.
4. If you installed the Visual Studio integration, you can now try accessing the ProDG Debugger from within Visual Studio (see "*Chapter* 2: Integrating with Microsoft *Visual Studio*" on page 17).

Updating ProDG over the web

This section assumes that you have previously successfully installed ProDG for PlayStation 2 v 1.1.

You will find regular updates of all the tools in ProDG for PlayStation 2 on the SN Systems web site (see "Updates and technical support" on page 3). The updates are in the form of zip archives, and this section describes how you can use these to update your ProDG installation.

Updating the ProDG build tools for PlayStation 2

The build tools are split into two zip archives: one that contains the ProDG EE build tools and one that contains the ProDG IOP build tools. Once you have downloaded the two zip archive files from the SN Systems web site you will need to navigate to the zip file location and extract them.

When you extract, make sure that you select the drive that contains the `\usr\local\sce` directory that 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** options 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: **Integrating with Microsoft Visual Studio**

Introduction

This chapter describes how to integrate ProDG for PlayStation 2 with Microsoft Visual Studio. Once you have successfully installed ProDG for PlayStation 2 Visual Studio integration you will be able to create PlayStation 2 projects in Visual Studio, then build and debug them on the DTL-T10000 using ProDG Debugger for PlayStation 2.

The Visual Studio integration has the following main features:

- Builds your project using `ps2cc`, which invokes `ee-gcc`, `iop-elf-gcc`, `dvpasm` and `ld/ps2link`
- Uses Visual Studio source browsing in your project
- Outputs compiler and linker errors/warnings in Visual Studio format so that double-clicking on a build error in the output window opens the source file on the appropriate line in the Visual Studio editor
- Enables ProDG Debugger for PlayStation 2 to be called directly from Visual Studio to debug the current project
- Imports and exports Visual Studio breakpoints at the start/end of a debug session in the ProDG Debugger
- Ability to open source, shown in the ProDG Debugger source pane, in the Visual Studio Editor to enable source file editing

Installation notes

You must have already installed the following software:

- Sony toolchain v 1.6.0 or later
- Sony libraries v 1.6.0 (for IOP support) or later

- Microsoft Visual Studio 6.0
- ProDG for PlayStation 2 v 1.20 or later

Installing the Visual Studio integration

You may have downloaded the ProDG for PlayStation 2 installation program from the SN Systems web site, or you may be running it from CD-ROM. This section assumes that you are installing ProDG for PlayStation 2 from the CD-ROM provided, though the process is the same.

1. Put the CD-ROM in the drive and using Windows Explorer navigate to the Visual Studo integration (VSI) subdirectory:

   `E:\VSI`

   where `E` should be replaced by the name of your CD-ROM drive.

2. Run the `ProDGforVSInn.exe` program in the VSI directory to carry out the installation, where `nn` is variable version information.

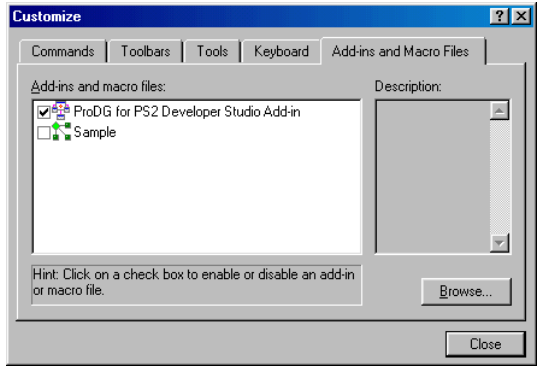
3. During installation you will be asked whether you want the installer to modify your `\autoexec.bat` file, or whether this will be carried out manually.

   Note: *In Windows 2000 installs you will not be given this option.*

4. Your system must then be rebooted in order to set up the environment correctly.

Installing the Visual Studio integration will automatically remove previous installation files and registry settings.

When you start Visual Studio, immediately after doing the install, you will need to click **Customize** in the **Tools** menu and in the dialog that appears select the **ProDG for PS2 Developer Studio Add-in** to enable it:



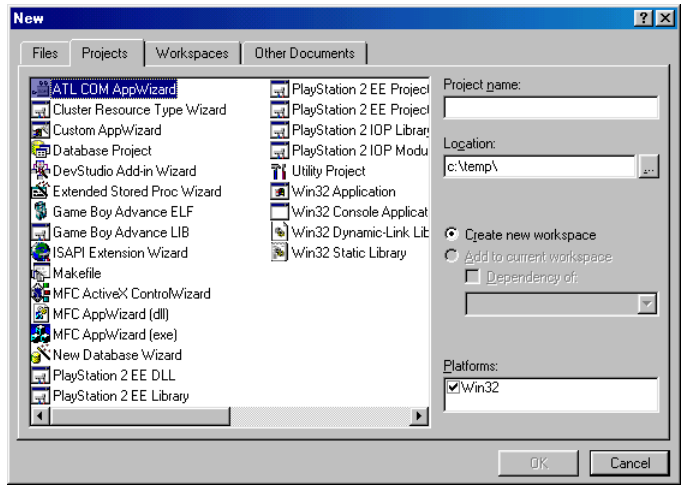
Deinstalling the Visual Studio integration

To deinstall the Visual Studio integration , use the Windows control panel **Add/remove programs** option and select **SN Systems Visual Studio integration**.

Creating a PlayStation 2 project in Visual Studio

Once you have completed the set up steps you can create a new Playstation 2 project in Visual Studio by doing the following:

1. Click **New** in the **File** menu and click the **Projects** tab, and the following dialog is displayed:



2. Select one from the following list of PlayStation 2 AppWizards: **PlayStation 2 EE DLL**, **PlayStation 2 EE Library**, **PlayStation 2 EE Project (ELF)**, **PlayStation 2 EE Project using DLLs**, **PlayStation 2 IOP Library** or **PlayStation 2 IOP Module (IRX)**.

> **Note:** ***PlayStation 2 EE DLL*** *and* ***PlayStation 2 EE Project using DLLs*** *are used to build a DLL and a relocatable application, respectively. See "Building relocatable DLLs" on page 63 for further details.*

3. Enter the name of your new project in the **Project name** field.

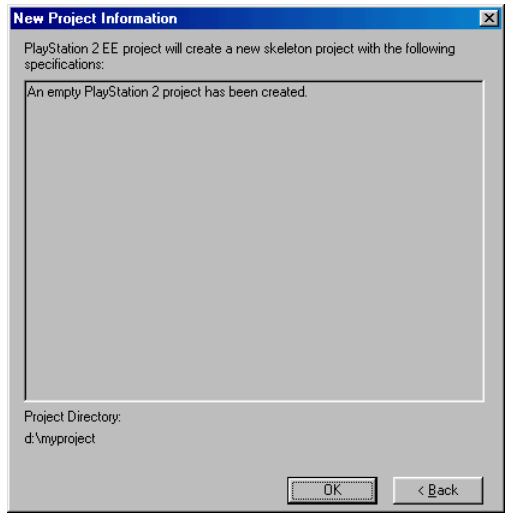
4. In the **Location** field browse to the desired location for the new project file.

   If the project source files are in a different location to the Visual Studio project file, then Visual Studio will not be able to locate any included files *even if they are in the same directory as the source file*. The solution to this is either:

   - (globally for all projects) to add the directory containing the project header files to the included files path in the **Directories** tab of the Options dialog (**Tools** menu); or
   - (for this project) click **Settings** in the **Project** menu, select the **C/C++** tab and with **Category: Preprocessor** selected set an additional include directory in the **Additional include directories** textbox.

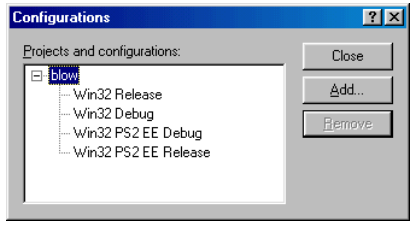
5. Currently the Win32 checkbox must be checked in the **Platforms** dialog, for the PlayStation EE/IOP project wizards to be displayed. Note that this checkbox does not relate to the platform that you are developing for.

6. Click **OK** on the dialog that appears confirming that the new project has been successfully created and the new project is opened in Visual Studio:



The active project configuration

Each PlayStation 2 project generates debug and non-debug build configurations. Select **Configurations** from the **Build** menu option to display the projects and their configurations. For example, a **PlayStation 2 EE Project (ELF)** has four possible build configurations:

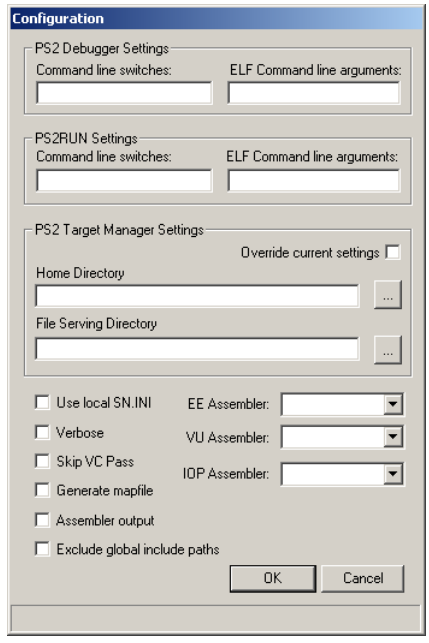


The active project configuration can be set by selecting **Set Active Configuration** from the **Build** menu.

Configuring the Visual Studio integration

You can configure the way the Visual Studio integration behaves on a project by project basis, by creating a new project and then click the **VSI Control Panel** toolbar button:

The VSI Configuration dialog is displayed, similar to the following:



PS2 Debugger Settings

Enables you to set the command-line switches and ELF command-line arguments passed to the ProDG Debugger program, `ps2dbg`. See "ps2dbg command-line syntax" on page 98 for further details.

> **Note:** *You should exercise great care in modifying the default values, as invalid switches may cause unexpected behavior in the ProDG Debugger.*

PS2RUN Settings

Enables you to set the command-line switches and ELF command-line arguments passed to the ProDG command-line utility, `ps2run`. See "***Chapter 7:*** *The ProDG command-line utility*" on page 93 for further details.

PS2 Target Manager Settings

Enables you to set the home and fileserving directories as used by ProDG Target Manager, `ps2tm`. See "***Chapter 6:*** *ProDG Target Manager for PlayStation 2*" on page 73 for further details. To use different settings, click the **Override defaults** checkbox to access the **Home Directory** and **File Serving Directory** fields. Browser buttons are provided if you need to search for different paths.

Use local SN.INI

By default, the Visual Studio integration takes its environment from the `sn.ini` file which is located in the directory pointed to by the `SN_PATH=` environment variable (see "The sn.ini file" on page 11). However, you can get it to use a local `sn.ini` file by checking this checkbox. If a local `sn.ini` file does not exist, you will be given the chance to create one.

Verbose

Enables verbose output during compiling.

Skip VC Pass	By default, C and C++ files are compiled twice in the Visual Studio integration: first using the Visual C compiler in order to do dependency checking, and then by the SN Systems compiler driver, `ps2cc`. See "***Chapter 3:*** *ProDG Compilers for PlayStation 2*" on page 27 for more information). Check this option to skip the Visual C compiler pass and so speed up the compile.
Generate mapfile	Check this option to cause the linker to generate a mapfile.
Assembler output	Check this option to cause the compiler to save an assembler listing.
Exclude global include paths	Check this option to avoid searching for an include file in the global include paths (as listed in **Tools** > **Options** > **Directories** > **Include files** view) but instead to search in the project directory only. In this way you can completely insulate your project from machine-dependent code in the global header files.
EE Assembler	A drop-down listbox enables you to choose between the SN Systems EE assembler, `ps2eeas`, and the GNU EE assembler, `as`.
VU Assembler	A drop-down listbox enables you to choose between the SN Systems VU assembler, `ps2dvpas`, and the GNU VU assembler, ee-dvp-`as`.
IOP Assembler	A drop-down listbox enables you to choose between the SN Systems IOP assembler, `ps2iopas`, and the GNU IOP assembler, `as`.

Press **OK** to save your project settings in the `sn.ini` file, or **Cancel** to quit.

Building your PlayStation 2 project

With the Visual Studio integration you can build your PlayStation 2 project in the usual way in Visual Studio. This section tells you how to do it.

Adding your project files

New projects (except for the IOP) are initialized with certain files as follows:

PlayStation 2 EE DLL	`crt0.s`, `rel.cmd`, `rel.lk`
PlayStation 2 EE Library	`PS2_in_VC.h`
PlayStation 2 EE Project (ELF)	`app.cmd`, `crt0.s`, `ps2.lk`, `PS2_in_VC.h`
PlayStation 2 EE Project using DLLs	`crt0.s`, `relapp.cmd`, `relapp.lk`

These files will certainly need to be edited before building your project. For example, see "Building relocatable DLLs" on page 63 for information on the changes required to `rel.cmd` when building a DLL.

You now need to add your project source files to the new project using the **Add to Project > Files** command on the **Project** menu. The project source files needed by the project `makefile` (`.c`, `.s` and `.dsm`, etc.) must all be added to the project before carrying out the build.

Building your project

Building your project is carried out as usual, by invoking the **Clean**, **Compile** and **Build** (etc.) commands from the Visual Studio **Build** menu.

The following table lists the Visual C compiler switches and how they are translated into their GNU equivalents:

VC	GNU	meaning
`/D`	`-D`	Define proprocessor constant
`/debug`	`-g`	Debug info
`/Fi`	`-include`	Include filename
`/Fo`	`-o`	Output filename
`/GR`	`-frtti`	Enable C++ RTTI
`/I`	`-I`	Include path
`/Od`	`-O0`	No optimization
`/O1`	`-O1`	Optimize for size
`/Os`	`-O1`	Optimize for size
`/Ot`	`-O2`	Optimize for speed
`/O2`	`-O3`	Optimize for speed
`/TP`	`-xc++`	Treat files as C++
`/u`	`-undef`	Undefine all predefined macros
`/W0 or /w`	`-w`	Disable warnings
`/W1, 2, 3`		Default warnings
`/W4`	`-Wall`	Maximum warnings
`/WX`	`-Werror`	Warnings as errors
`/X`	`-nostdinc`	Ignore standard includes

Any other GNU switches can be passed on by adding them to the **Project > Options > (C/C++)** box preceded with a `'-'`, e.g. to enable assembler output listing add `"-Wa,al"` to the options.

The GNU option `-fno-exceptions` (disable exception handling) is on by default for C++ files. If you want to enable exception handling for C++ files, either check the box in **Project Settings > C/C++ > C++ Language** or add `/GX` to the project settings box.

`ps2link` options can be added the same way by including them in **Project > Options > (Link)**.

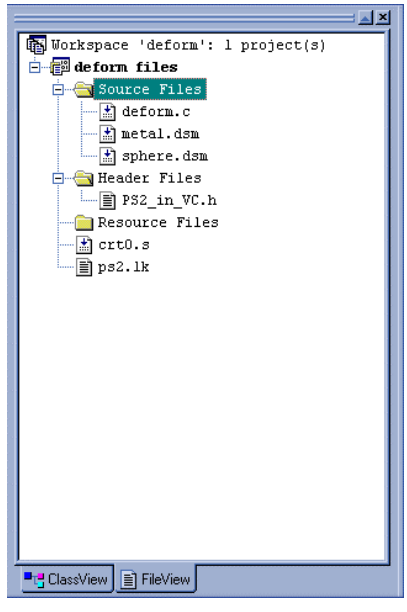
Note: *Visual Studio will link with whichever linker is pointed to by the `linker_name` environment variable in the `sn.ini` file (see "The sn.ini file" on page 11).*

If you are planning to do VU debugging then the GNU linker `ld` MUST be used. The `app.cmd` supplied is the default linker script for the GNU linker `ld`.

Setting up the Sony deform sample

This section describes how to create and configure a PlayStation 2 EE project in Visual Studio to build the Sony sample file `deform.elf` using the ProDG tools.

1. Start Microsoft Visual Studio and ensure that you have enabled the **ProDG for PS2 Developer Studio Add-in** (see "Installing the Visual Studio integration" on page 18).
2. Create a new project using the **PlayStation 2 EE Project (ELF)** project template (**File > New**).
3. Enter `deform` in the **Project name** box.
4. Enter or use the browse button to specify the location as: `drive:\usr\local\sce\ee\sample\vu1` (where `drive` is the drive on which the Sony toolchain and libraries are installed in `usr\local`). Note that the directory `deform` should automatically be appended to the end of the directory path.
5. Once the project has been created, in the file view add the files `deform.c` and `metal.dsm` and `sphere.dsm` to the `Source Files` folder. The files pane should now resemble the following:



6. Click **Build deform.elf** in the **Build** menu, and your project `.elf` file should be successfully built.

Loading and running ELF files

You can load and run your `.elf` file by clicking the **Load ELF** toolbar button:

The `ps2run` command-line switch defaults to `-r`. See "ps2run command-line syntax" on page 93 for more information on `ps2dbg` switches. These can be overriden by creating a `[ps2run]` section in your `sn.ini` file (see "The sn.ini file" on page 11), as in the following example:

```
[ps2run]

switches=-r -t devtool4
args=<args>
```

Any extra switches should be appended to the default value (`-r`).

Integration with the ProDG Debugger

The ProDG Debugger can be invoked once you have successfully built your project in Visual Studio, by clicking the **Run ProDG Debugger for PS2** toolbar button:

Note that the active project configuration will affect your ability to start ProDG Debugger. When you click on the **Run ProDG Debugger for PS2** button, ProDG Debugger for PlayStation 2 will be launched if the project was built with one of the PS2 EE/IOP build configurations as the active project configuration AND the `.elf` file has been successfully built.

If you are currently working with an IOP project, you can start the ProDG Debugger, but this is not how you would normally debug your built IOP module. You would usually debug an IOP project by setting up an EE project that loads and references your built IOP module.

ProDG Debugger and Target Manager are automatically launched with the following command line switches:

```
ps2dbg –vs –r –e
```

The `–vs` switch enables Visual Studio compatibility features. The `–r` and `–e` switches reset the target and load the executable contained in the built project `.elf` file. See "ps2dbg command-line syntax" on page 98 for more information on `ps2dbg` switches.

The `ps2dbg` command-line switches can be overriden by creating a `[ps2dbg]` section in your `sn.ini` file (see "The sn.ini file" on page 11), as in the following

```
[ps2dbg]
switches=-vs -r -e -t devtool4
```

Any extra switches should be appended to the default values (`-vs -r -e`).

Setting breakpoints in your project

When ProDG Debugger is first started it automatically imports any breakpoints that have already been set in your source in Visual Studio.

Note: *Any breakpoints that are set as disabled in Visual Studio will in fact be enabled when ProDG Debugger is started.*

While ProDG Debugger is running, setting or modifying breakpoints in Visual Studio has no effect on ProDG Debugger breakpoints, even if the `.elf` file is rebuilt in Visual Studio and re-downloaded. However, any breakpoints set in ProDG Debugger will be immediately reflected in Visual Studio. We therefore recommend that you set or modify breakpoints only in the ProDG Debugger source or disassembly panes. These breakpoints will be automatically updated in your source files as viewed by Visual Studio.

When you exit ProDG Debugger, all of the breakpoints still set will be exported back to Visual Studio.

Editing your source in Visual Studio

Any time you are debugging your project in ProDG Debugger you can switch from the source pane to the Visual Studio editor. This can be done by moving to the part of the file that you wish to edit, and clicking **Edit in Visual Studio** from the source pane shortcut menu.

Any changes you make to the source will need to be rebuilt into a new `.elf` file. You can then either restart ProDG Debugger from Visual Studio by clicking the **Run ProDG Debugger for PS2** toolbar button or download the newly built `.elf` file using the ProDG Debugger **Load ELF file** option (**File** menu).

Note that setting or unsetting breakpoints in Visual Studio, *while ProDG Debugger is running*, will have no effect (see "Setting breakpoints in your project" on page 26).

Chapter 3: **ProDG Compilers for PlayStation 2**

Using the SN Systems compiler ps2cc

The GNU compiler drivers for the EE and IOP units can be replaced by the SN Systems `ps2cc` compiler driver. This has the advantage of enabling a PlayStation 2 game to be built without having to create a `makefile`.

The `ps2cc.exe` executable can be put anywhere on your search path, but the program must be able to locate your `sn.ini` (SN Systems configuration) file by interrogating an environment variable `SN_PATH=`.

In order to build successfully, the following environment variables (shown here with typical settings) must be set up in the `[ps2cc]` section of your `sn.ini` configuration file:

```
[ps2cc]
....

#assembler_name=as
assembler_name=ps2eeas

opt_assembler_name=as
#opt_assembler_name=ps2eeas

#dvp_asm_name=ee-dvp-as
dvp_asm_name=ps2dvpas

dvp_assembler_path=c:\usr\local\sce\ee\gcc\bin

iop_asm_name=as
#iop_asm_name=ps2iopas

#linker_name=ld
linker_name=ps2link

linker_path=c:\usr\local\sce\ee\gcc\bin

#linker_script=c:\usr\local\sce\ee\lib\app.cmd
linker_script=c:\usr\local\sce\ee\lib\ps2.lk
```

```
iop_linker_name=ld
startup_module=c:\usr\local\sce\ee\lib\crt0.s
dvp_include_path=c:\usr\local\sce\ee\include
....
```

Note: *ps2cc ignores lines starting with a '#', so you can easily swap between GNU's ld and the SN Systems linker ps2link, as shown in the example above.*

You are now all set to use ps2cc instead of ee-gcc. Replace calls to ee-gcc ($(prefix)-gcc) in your makefile with calls to ps2cc. If you have already changed your makefile to use ps2link the make will fail, otherwise it should link fine, whichever linker you choose in your sn.ini (since ps2cc changes the calling syntax automatically).

You can also run ps2cc from the DOS command line, as follows:

```
ps2cc [options] filename [filename…]
```

How ps2cc interprets input files

ps2cc can accept any of the following types of file as input and applies the following actions according to the file extensions:

File Type	Extensions	Actions
C source	.C	Pre-process, Compile, Assemble, Link
C++ source	.CC, .CPP	Pre-process, Compile, Assemble, Link
Preprocessed C source	.I	Compile, Assemble, Link
Preprocessed C++ source	.II, .IPP	Compile, Assemble, Link
Compiler-sourced assembler	.S	Assemble, Link
User-sourced assembler	.ASM	Pre-process, Assemble
VU assembly code	.DkSM	Assemble with DVP Assembler
C header	.H, .HPP	None
Object files	All others	Link only

Files with extensions that are not recognised as indicating any specific file type are treated as object files and passed only to the linker. This includes .o files, the standard object file extension.

There is no restriction on how many different extensions can be used; ps2cc can compile many C and C++ files in a single invocation and will apply the correct compiler to each.

The actions taken are also subject to control options such as -c which will omit automatic linking.

It is also possible to use the -x… option to specify that all subsequent files on the command line are of a given type, overriding the type as normally indicated by the file extension. You can also specify several -x… options and each will affect all subsequent files until the next -x… option appears.

-x argument	Files are assumed to be..
C	C source
Cpp-output	Pre-processed C source
c++	C++ source
c++-cpp-output	Pre-processed C++ source
Assembler	Assembler
Assembler-with-cpp	Assembler
c-header	C header
None	Object

Options

Once ps2cc has interpreted the relevant *file type*, any compiler *options* are specified in the command line—each one preceded by a hyphen ('-').

The following tables group the various compiler options according to type (options marked with "*****" are new in ps2cc v2.70):

Process control and output

Options	Actions
-c	Compile to an object file. If an output file is specified (via the -o option), all output is sent to this file. Otherwise the output file takes the input filename, with a new extension of .o.
-iop	Compiles for the IOP (default=EE)
-S	Compile to assembler source. If no output file is specified, the output file is the original filename with a new extension of .S.
-E	Pre-process only. If no output file is specified, output is sent to the screen.
-o FILE{,MAPFILE}	Specify the output filename FILE rather than using the default. To create a map file, add the name of the output MAPFILE immediately following a comma (no spaces)
-v	Verbose mode – print all commands before execution.

`-save-temps`	Preserve intermediate temporary files such as pre-processor output and compiler-generated assembler source.
`-x type`	Treat subsequent input files as being of type `type`.

C/C++ language options

Options	Actions
`-f…`	Specify a compiler option / optimization (full list in GNU compiler documentation).
`-fdefault-single-float*`	Forces singles to be used instead of doubles as the default for constants, so that you don't have to specify the "f".
`-ansi`	Check code for ANSI compliance.

Warning options

Options	Actions
`-Wall`	Enable all warnings.
`-Wpromote-double*`	Warns you if the compiler decides to make a constant a double rather than a single.
`-w`	Disable all warnings.
`-W…`	Suppress individual warnings.

Debugging options

Options	Actions
`-g`	Generate debug information for source-level debugging (required to use ProDG Debugger).

Optimization options

Options	Actions
`-G SIZE`	Set variable size for gp register optimization: 0=none
`-mgpopt`	Improve gp register optimization
`-O0`	No optimization (default).
`-O or -O1`	Standard level of optimization.
`-O2`	Full optimization.
`-O3`	Full optimization and function inlining.
`-Os`	Optimize to make the code as small as possible. Note that this option is only available in version 2.95.2 of the compiler.

Preprocessor options

Options	Actions
`-I DIR`	Add this path to the list of directories searched for include files.
`-D NAME`	Define pre-processor symbol `NAME`.
`-D NAME=DEF`	Define pre-processor symbol `NAME` with value `DEF`.
`-U NAME`	Undefine the symbol `NAME` before pre-processing.
`-Wp,…`	Specify an option for the pre-processor (full list in GNU documentation).

Assembler option

Options	Actions
`-Wa,…`	Specify an option for the assembler.
`-Wd,…`	Specify an option for the DVP assembler.

Linker options

Options	Actions
`-l LIBRARY`	Include specified library `LIBRARY` when linking.
`-L DIR`	Add this path to the list of directories searched for libraries.
`-X… or -Wl,…`	Specify an option to be passed to the linker.
`-nostdlib`	Do not include the standard libraries listed in `sn.ini`.
`-linkscript file`	Use the specified file as the linker script.

Machine-dependent options

Options	Actions
`-m…`	Specify a machine-specific compiler option (full list in GNU compiler documentation).

Using a response file

To save repeatedly typing long command lines you can use a *response file*. To create a response file, enter the options into an ASCII text file, separated by spaces, tabs or newlines. When you invoke `ps2cc` on the command line, you can specify that a response file is to be used by giving the name of the file preceded by a 'commercial at' ('`@`') character, e.g.

```
ps2cc @myresponsefile
```

`ps2cc` uses the contents of the '`myresponsefile`' to obtain its arguments.

Examples

```
ps2cc -c -O2 main.c objects.c pluscode.cpp
```

This pre-processes, compiles and assembles `main.c`, `objects.c` and `pluscode.cpp` to produce three object files, compiled with optimizations and containing no debug information. The files `main.c` and `objects.c` are compiled with the C compiler; whereas `pluscode.cpp` is compiled with the C++ compiler.

```
ps2cc -c *.c -I. -Ic:\include
```

This pre-processes, compiles and assembles every `.c` file in the current directory to make a `.o` file. Files included with `#include <…>` are searched for in the current directory and then in `c:\include`.

```
ps2cc -g -O2 *.c *.dsm -o main.elf
```

This pre-processes, compiles all `.c` files, assembles all `.dsm` files, producing a set of object files `*.o` which are then linked to build the executable `main.elf`. Compiler optimizations are enabled and debug information is included in the output.

Compiling C++ files for the IOP

The SN IOP C++ compiler is built using the latest 2.95.2 source code. It slots into and works directly with the ProDG Debugger, and with the Visual Studio integration, if used. It allows you to pass a `.CPP` (C++ source) file to the `ps2cc` compiler and specify the `-iop` option (compile for the IOP), provided that you make some minor changes to your source code. See "Wrap SCE headers with 'extern "C" {…}'" on page 33 and "C++ global constructors and destructors" on page 34.

Installing SN IOP C++ compiler

If you are installing these tools from a zip, follow this procedure:

1. Ensure the relevant library release from Sony have been installed.
2. Unzip this file onto the root.
3. Add the absolute path of `\usr\local\sce\iop\gcc\bin` to your `\autoexec` PATH setting, e.g.

   ```
   SET PATH=%PATH%;c:\usr\local\sce\iop\gcc\bin
   ```

4. Now you must set the `PS2_DRIVE` environment, unless it is already set up (by the EE tools) , e.g.

   ```
   set PS2_DRIVE=c
   ```

   Note: *There is no colon-slash after the drive letter*

   and add the following two lines:

   ```
   SET IOP_CPATH=%PS2_DRIVE%:\usr/local/sce/iop/gcc/lib/gcc-lib/mipsel-scei-elfl/2.8.1/include
   ```

```
SET
IOP_CPATH=%IOP_CPATH%;%PS2_DRIVE%:\usr/local/sce/iop/gcc/m
ipsel-scei-elfl/include
```

5. Now unzip the tools from the zipfile. Extract these files into the drive where you have the GNU software. The relative paths will then place the files in the correct directories.
6. Note that if you have installed the IOP tools through InstallShield, you do not need to do this since InstallShield will have done it for you.

Setting up install directories

It is important to note that as with Linux, after installing the `1.6.x` or `2.0.x` libs, you must copy the contents of the '`iop\install`' to '`iop\gcc\mipsel-scei-elfl`'.

Building IOP demos

To build the IOP demos, you must change the `makefile` line that reads:

```
iop-path-setup > PathDefs || (rm -f PathDefs ; exit 1)
```

to

```
iop-path-setup
```

This is because DOS cannot execute the command as written.

Note: *The `iop-path-setup` program is currently just a batch tool which creates the correct `path defs` into the current directory. In the release tools this will be replaced but if you don't change the positions of the build tools it will work perfectly.*

It is assumed that you have already installed the Win32 EE Tools, purely for the `make.exe` program. If you have not, then obtain `make.exe` from the internet or from the Win32 EE tools and put it in the `\usr\local\sce\iop\gcc\bin` directory.

Wrap SCE headers with 'extern "C" {...}'

In order to use C++ IOP compiler you will need to wrap the Sony includes with extern "C"; the most convenient way of doing this is in your source code, as in the following example:

```
...
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
extern "C" {
#endif
#include <kernel.h>
#include <sys\file.h>
#include <sif.h>
#include <stdlib.h>
#include <stdio.h>
#include <sifrpc.h>
```

```
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
}
#endif
...
```

C++ global constructors and destructors

The C++ port has been written so that you have to manually call constructors and destructors. The functions you use are `SN_CALL_CTORS`; and `SN_CALL_DTORS`, e.g.:

```
...
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
extern "C" {
#endif
#include <stdio.h>
#include <kernel.h>
#include <sysmem.h>
#include <sif.h>
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
}
#endif
#include <libsniop.h>  //Contains definitions for CTOR and
DTOR calls.
#define BASE_priority  32
class foo
{
public:
 int hoo;
 int bar;
 foo () {hoo = 40; printf("\nfoo Constructor");}
 ~foo () {hoo = 0; bar = 0; printf("\nfoo Destructor.");}
};
foo globfunc;  //Global object of class foo
ModuleInfo Module = { "cxx_fiddling", 0x0101 };
int thread1(void)
{
 printf("\nThread1 startup.");
 printf("\nCall Global Ctors.");
 SN_CALL_CTORS;  //Call all global CTORs
 globfunc.hoo = 32;
 globfunc.bar = 256;
 printf("\nCall global Dtors.");
 SN_CALL_DTORS;  //Call all global DTORs
 printf("\nStuff happening after global destructor call.");
 return 0;
}
int start (void)
```

```
{
    struct ThreadParam param;
    int thread;
    printf("\nStartup thread init.");
    CpuEnableIntr();
    if (!sceSifCheckInit())
    {
 sceSifInit();
    }
    param.attr         = TH_C;
    param.entry        = thread1;
    param.initPriority = BASE_priority - 2;
    param.stackSize    = 64*1024;
    param.option       = 0;
    thread = CreateThread (&param);
    if (thread > 0)
    {
 StartThread (thread, 0);
 printf ("\nThread 1 started.");
 printf ("\nStartup thread terminated.");
 return 0;
    }
    else
    {
 printf ("\nStartup thread terminated.  Errors
encountered");
 return 1;
    }
}
...
```

The logic behind this is that most IOP programs terminate the `start()` thread before executing any program code. You can call global constructors and destructors from non-startup threads, and access them as usual until they are destroyed. If the globals were tied to `start()` then they would be destroyed once `start` terminates, as with `main()` in C++ programs on the EE. A single call to either of these `SN_CALL_...` macros will initialise or destroy all global objects from every source file which is linked to your program.

You will need to `#include <ioplibsn.h>` for the definitions for C++ memory functions and also for the global constructor and destructor calling code.

IOPFIXUP error: unresolved symbols

If you get unresolved symbols for `malloc` and `free`, which are used for `new` and `delete`, or `exit()` and `_exit()`, linking with `libsniop.a` will resolve them. This maps `malloc` and `free` to `AllocSysMemory` and `FreeSysMemory`, and patches in termination code for `exit` and `_exit`.

Doubles and float software emulation

You now have access to float and double types and all computations involving them, although conversion from float to double is currently not possible. Use doubles where possible.

Note that the IOP `printf` does not support floating point output. To obtain this you must link with `libsniop.a`, by adding '`-lsniop`' to your `makefile` or Visual Studio integration link stage.

Note: *This is software emulation and is not really suitable for use in release code which needs to run quickly.*

Using the ioplibdump utility

The `ioplibdump` utility is used to determine the external functions that are called from a particular `irx` module and the module they are from. It provides a list of:

- the modules that must be loaded for the `.irx` to run
- the `.ilb` function numbers for external library calls

The syntax is as follows:

```
ioplibdump <objectfiles> : <stub_ilb_data>
```

For example:

```
ioplibdump cxtmdm.irx
```

will list all the external function information from `cxtmdm.irx`. However,

```
ioplibdump cxtmdm.irx:iop.ilb
```

will list all the external information from `cxtmdm.irx`, but it will also examine the `iop.ilb` file to see if it can find any of the function names and dump these too. Note that you can specify any number of `.irx` and `.ilb` files, e.g.

```
ioplibdump cxtmdm.irx client.irx mylib.irx:iop.ilb
ilink.ilb
```

In the above example, each irx file will be examined and compared with each `.ilb` file and all the external function names will be listed.

Using the ioplibgen utility

The `ioplibgen` utility produces `.ilb` files and library stubs from Sony IOP `.tbl` table files. To create the entry table source for the `.tbl` file the syntax is as follows:

```
ioplibgen [input filename(.tbl)] -e entry_table_source(.s)
```

To create the `ilb` calling stub for the library defined in the `.tbl` file the syntax is as follows:

```
ioplibgen [input filename(.tbl)] -d stub_ilb_data(.ilb)
```

Chapter 4: ProDG Assemblers for PlayStation 2

Using the SN Systems assemblers

ProDG Build Tools for PlayStation 2 include the three SN Systems assemblers: `ps2eeas`, `ps2iopas` and `ps2dvpas`.

The EE and IOP assemblers can be invoked directly from the EE or IOP (`ee-gcc`, or `iop-elf-gcc`) compilers using the `-snas` command line option. However, the SN Systems VU assembler `ps2dvpas` is used on the command line to assemble `.dsm` files that have been written in VU microcode (see "The ProDG VU assembler ps2dvpas" on page 45).

Specifying options

Using the same mechanism as when the GNU assembler is invoked via the C compiler you can use the `-Wa` option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas. Usually you do not need to use this `-Wa` mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the `-v` option to see precisely what options it passes to each compilation pass, including the assembler.)

The command-line options and directives for the SN Systems' assemblers are compatible with the GNU assemblers except for some GNU command-line switches which are ignored or produce errors. See "Unsupported switches and directives" on page 41.

In addition, if you specify the `-sn` option, you will have access to the SN Systems directives. See "SN Systems directives" on page 43.

Command-line syntax

After the program name the command line may contain options and filenames. Options may appear in any order, and may be before, after, or between filenames. The order of filenames is significant.

'--' (two hyphens) by itself names the standard input file explicitly, as one of the files to be assembled.

Except for '--' any command line argument that begins with a hyphen ('-') is an option. Each option changes the behavior of the assembler. No option changes the way another option works. An option is a '-' followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one filename to follow them. The filename may either immediately follow the option's letter or it may be the next command argument. These two command lines are equivalent:

```
as -o my-object-file.o mumble.s

as -omy-object-file.o mumble.s
```

Input files

The phrase *source program* or *source*, describes the program input to one run of the assembler. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run the assembler it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give the assembler a command line that has zero or more input filenames. The input files are read (from left filename to right). A command line argument (in any position) that has no special meaning is taken to be an input filename.

If you give the assembler no filenames it attempts to read one input file from the assembler's standard input, which is normally your terminal. You may have to type <Ctrl+D> to tell the assembler there is no more program to assemble.

Use '--' if you need to explicitly name the standard input file in your command line.

If the source is empty, the assembler produces a small, empty object file.

Filenames and line numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See "Error and warning messages" on page 39.

Physical files are those files named in the command line given to the assembler.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical filenames help error messages reflect the original source file, when the assembler source is itself synthesized from other files. The directive `.app-file string` is used.

Output (object) file

Every time you run the assembler it produces an output object file. Its default name is `a.out`. You can give it another name by using the `-o` option. Conventionally, object filenames end with `.o`.

The object file is meant for input to the linker. It contains assembled program code, information to help the linker integrate the assembled program into a runnable file, and (optionally) symbol information for the debugger.

Error and warning messages

The assembler may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs the assembler automatically. Warnings report an assumption made so that assembly could continue for a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where `NNN` is a line number). If a logical filename has been given (using the directive `.app-file string`) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (using the directive `.line line-number`) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed.

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The filename and line number are derived as for warning messages.

Symbol names

Symbol names begin with a letter or with one of `'._'`. On most machines, you can also use $ in symbol names. That character may be followed by any string of digits, letters, dollar signs and underscores. For the AMD 29K family, `'?'` is also allowed in the body of a symbol name, though not at its beginning. Case sensitivity is also significant: `foo` is not the same as `Foo`.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local symbol names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names `0,1...9`. To define a local symbol, write a label of the form `N` (where `N` represents any digit). To refer to the most recent previous definition of that symbol write `Nb`, using the same digit as when you defined the label. To refer to the next definition of a local label, write `Nf`---where `N` gives you a choice of 10 forward references. The `b` stands for backwards and the `f` stands for forwards.

Local symbols are not emitted by the current GNU C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L`

All local labels begin with `L`. Normally both the assembler and linker forget symbols that start with `L`. These labels are used for symbols you are never intended to see. If you use the `-L` option then the assembler retains these symbols in the object file. If you also instruct the linker to retain these symbols, you may use them in debugging.

digit

If the label is written 0: then the digit is 0. If the label is written 1: then the digit is 1. And so on up through 9:.

C-A

This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value `\001`.

ordinal number

This is a serial number to keep the labels distinct. The first 0: gets the number 1; The 15th 0: gets the number 15; *etc.* Likewise for the other labels 1: through 9:. For instance, the first 1: is named L1C-A1, the 44th 3: is named L3C-A44.

The special dot symbol

The special symbol `'.'` refers to the current address that the assembler is assembling into. Thus, the expression `melvin: .long` defines `melvin` to contain its own address. Assigning a value to `'.'` is treated the same as a `.org` directive. Thus, the expression `.=.+4` is the same as saying `.space 4`.

Symbol attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, the assembler assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the `.text`, `.data`, `.bss` or `.absolute` sections the value is the number of addresses from the start of that section to the label. Naturally for `.text`, `.data` and `.bss` sections the value of a symbol changes as the linker changes section base addresses during linking; `.absolute` symbols' values do not change during linking—that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is `0` then the symbol is not defined in this assembler source file, and the linker tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally) other information for linkers and debuggers. The exact format depends on the object-code output format in use.

Unsupported switches and directives

The following assembler command line switches are either ignored or produce errors if you use them:

Switch	How it is handled
`-a`	Ignored
`-D`	Ignored
`-f`	Ignored
`--itbl`	Generate an error
`-J`	Ignored
`-K`	Ignored
`--listing-lhs-width`	Ignored
`--listing-lhs-width2`	Ignored
`--listing-rhs-width`	Ignored

`--listing-cont-lines`	Ignored
`-membedded-pic`	Generate an error
`-nocpp`	Ignored
`-R`	Generate an error
`--traditional-format`	Ignored
`-w`	Ignored
`-X`	Ignored
`-Z`	Ignored

The following table shows the directives that are not currently supported by the SN Systems assemblers and how they are handled:

Directive	How it is handled
`.abicalls`	Generate an error
`.cpadd`	Generate an error
`.cpload`	Generate an error
`.cprestore`	Generate an error
`.eject`	Ignored
`.endfunc`	Ignored
`.format`	Ignored
`.func`	Ignored
`.gpword`	Generate an error
`.ident`	Ignored
`.insn`	Ignored
`.lflags`	Ignored
`.linkonce`	Generate an error
`.list`	Ignored
`.livereg`	Ignored
`.llen`	Ignored
`.lsym`	Ignored
`.name`	Ignored
`.noformat`	Ignored
`.nolist`	Ignored
`.nopage`	Ignored
`.org`	Ignored
`.plen`	Ignored
`.rva`	Ignored

.sbttl	Ignored
.spc	Ignored
.struct	Ignored
.title	Ignored

SN Systems directives

The following table list the SN Systems directives that are activated by means of specifying the -sn switch on the command line.

Note: *You can access the value of the rs counter directly through the variable __rs.*

Directive	Description
.endscope	See .scope (below)
.equr newreg, reg	Specify an alternative name for a register See "Naming registers and register fields" on page 44 for sample code
.rpalloc poolname, newreg, newreg, ...	Performs a register equate for each of the newregs by getting a register value from the specified register pool poolname
.rpfree poolname, reg, reg, ...	Releases each specified reg back into the pool poolname and marks the register name as undefined so you can't use it again by accident
.rpinit poolname, reg, reg, ...	Creates a register pool of the specified poolname and makes the list of registers available for allocation from that pool
.rsb name,count	Aligns the rs counter to the byte boundary, assign the rs counter value to the name and advance the rs counter by count * size
.rsd name,count	Aligns the rs counter to the doubleword boundary, assign the rs counter value to the name and advance the rs counter by count * size
.rsh name,count	Aligns the rs counter to the halfword boundary, assign the rs counter value to the name and advance the rs counter by count * size
.rsq name,count	Aligns the rs counter to the quadword boundary, assign the rs counter value to the name and advance the rs counter by count * size
.rsreset {expression}	Sets the rs counter to 0 or the value of expression if it is specified
.rsw name,count	Aligns the rs counter to the word boundary, assign the rs counter value to the name and advance the rs counter by count * size

`.scope` `.endscope`	Delimit a scope for local labels; any label beginning with the '@' character will be local to that scope See "Scope delimiting" on page 44 for sample code

Naming registers and register fields

The `.equr` SN Systems directive is available for naming registers and register fields. For example:

```
.equr MyRegister, VF03
.equr MyXField  , VF03x
```

If a floating-point register is assigned a name, then fields of that register can be accessed using a tail. All tails must be in lower case. For example:

```
.equr MyReg , VF04
ADDw.x MyReg.x, VF00x, VF08w      NOP
```

It is intended that tails can be used on register names assigned using `.rpalloc`, although this has not been tested:

```
.rpinit MyPool , VF01 ,VF02
.rpalloc MyPool , MyReg
ADDw.x MyReg.x, VF00x, VF08w      NOP
```

Scope delimiting

The following sample code segment show you how to use the scope delimiter directives `.scope` / `.endscope`:

```
.scope
@L1:
   bnez  $4,@L1
   add   $4,-1
.endscope

.scope
@L1:
   bnez  $5,@L1
   add   $5,-1
.endscope

Scopes can be nested.

e.g.

.scope
@L1:
   nop
.scope
@L1:
   bnez  $5,@L1  // goes to second $L1
   add   $5,-1
.endscope
   bnez  $4,@L1 // goes to first $L1
   add   $4,-1
.endscope
```

It is not possible to access local labels in one scope from a nested scope. All local labels not within a `.scope/.endscope` pair are in the global scope, i.e. there is no scoping between non-local label names.

The ProDG VU assembler ps2dvpas

The SN Systems VU assembler `ps2dvpas` directly replaces the GNU assembler `ee-dvp-as` and is completely compatible with it. To use it to build any VU microcode in your application you will need to substitute `ee-dvp-as` with `ps2dvpas` (`DVPASM` variable) in your `makefile`, or invoke it directly on the command line:

```
ps2dvpas <options> <input file>
```

The following table lists all of the available options:

Options	Actions
`-a[list-options]`	Produce listing (not implemented)
`--defsym sym=value`	Define integer symbol
`-G<size>`	Set size of data items placed in `.sdata/.sbss`
`--gstabs`	Produce STABS debug info
`--help`	Print help
`-I <directory>`	Specify directory to search for include files
`-L`	Output local symbol information
`--keep-locals`	Same as –L
`-o <output file>`	Specify output filename
`-sn`	Activate SN Systems extensions See "SN Systems directives" on page 43 for further information
`--version`	Print version information
`-W`	Disable warnings

The following list briefly describes all of the default directives that are understood by `ps2dvpas`. This does not include the standard GNU directives or the VU opcodes which can be found in the Sony documentation.

Directive	Description
`.data`	Switches to the `.vudata` section
`.dmadata`	Labels a block of DMA data
`.dmapackvif`	Flags whether the first part of DMA data should be packed in with the `dma` tag
`.enddirect`	Ends a VIF `direct` or `directhl` block

.enddmadata	Ends a DMA controller operation
.endgif	Ends a GIF operation
.endmpg	Ends a VIF mpg operation
.endunpack	Ends a VIF unpack operation
.quad	Specifies 128 bit data words
.text	Switches to the .vutext section
.vu	Switches to VU opcode mode
.word	Specifies 32 bit data word

DMA/VIF/GIF operations

The following sections list the DMA, VIF and GIF operations that are supported by ps2dvpas. More information on these can be found in the Sony documentation.

DMA controller operations

Dmacall	Dmaref
Dmacnt	Dmarefe
Dmaend	Dmarefs
Dmanext	Dmaret

VIF operations

Base	Mscnt
Direct	mskpath3
directhl	Offset
Flush	Stcol
Flusha	Stcycl
Flushe	Stcycle
Itop	Stmask
Mark	Stmod
Mpg	Strow
Mscal	Unpack
Mscalf	Vifnop

GIF operations

Gifimage	Gifreglist
Gifpacked	

Chapter 5: **ProDG Linkers for PlayStation 2**

Introduction

A linker is a fast and flexible tool that enables you to create your game from its component object files. It allows you to lay out your code and data in a model of the target machine's memory and control the order in which they appear in the game image.

This chapter explains the differences between the ProDG Linker for PlayStation 2, `ps2link`, and the GNU PlayStation 2 linker, `ld`, and then goes on to describe how to use `ps2link` and write control scripts for it. It also describes the new linker `ps2ld`, which unifies some of the functionality of `ee-ld` and `ps2link` but in a much faster linker. This release only supports `ee-ld` command line options and script files but future releases will support `ps2link` syntax also.

Linkers included in ProDG for PlayStation 2

ProDG for PlayStation 2 includes the GNU PlayStation 2 linker, `ld`. This is considered the 'reference' linker, i.e. your game should link using this.

- We don't supply documentation for `ld`. There is reasonably up-to-date documentation at various URLs, including http://www.redhat.com/support/manuals/gnupro99r1/5_ut/b_Usingld/ld.html.
- `ld` is used by default in the Sony samples.
- The Sony libraries come with a standard `ld`-format linker script which works for all the samples. This script is installed as `ee/lib/app.cmd`.
- `ld` can be called automatically from `ee-gcc`.

We also ship our own linker, `ps2link`. This is more advanced but also more 'beta' than `ld`, so your game may not link with it.

- `ee-gcc` cannot automatically call `ps2link`. However, our replacement compiler driver `ps2cc` can automatically call `ps2link`. See "Using the SN Systems compiler ps2cc" on page 27.

- You need to change your `makefile` to use this linker. See "Calling ps2link instead of ld" on page 50.
- We include a standard `ps2link`-format linker script which works for all the samples. This script is installed as `ee/lib/ps2.lk`. See "Example linker script" on page 61.

ProDG for PlayStation 2 also includes `ps2ld`, which will eventually unify the functionality of `ee-ld` and `ps2link`. This release only supports `ee-ld` command line options and script files but future releases will support `ps2link` syntax also. The major advantage in using `ps2ld` however, is that it is considerably faster than either `ee-ld` or `ps2link`. See "Using ps2ld" on page 70.

Benefits of ps2link

There are some things that `ps2link` does better than `ld`, which is why you should consider switching to the SN Systems linker:

- `ps2link` is up to 40% faster than `ld`
- `ps2link` requires much less virtual memory than `ld`, especially for debug builds
- `ld` will fail to find symbols required by a library if the symbols are defined in libraries specified earlier on the command line. `ps2link` has no such problem.
- `ld` will only search library search paths if you refer to a library using the `-l<name>` option, in which case the library *must* be called `lib<name>.a`, e.g. `-lgraph` searches for `libgraph.a`. But if you refer to the library by its full name, the library search paths are not used. `ps2link` always searches library search paths.

Features

These are the main features of `ps2link` in this release of ProDG for PlayStation 2:

- Produces ELF image compatible with ProDG Debugger, ProDG Target Manager, and `dsedb`.
- Produces debug info compatible with ProDG Debugger and `dsedb`.
- Supports PlayStation 2 object files and libraries built with GNU tools.
- Supports C++, including templates, global object construction/destruction, and exceptions.
- Supports a subset of the SN linker script language as used on PlayStation and Nintendo 64.
- Emulates GNU `ld` error/warning message format.
- Can emulate Visual Studio error/warning message format.
- Demangles symbol names in messages and MAP file.

- Removes unused and duplicate functions and class data from the image.
- Requires a ProDG tools license (the same license as the ProDG Debugger).

ps2link system requirements

Notwithstanding the ProDG for PlayStation 2 system requirements we strongly recommend that your system has at least 256 MB RAM to avoid linker swapping.

A rule of thumb for `ps2link` memory requirements is that you need at least half as much memory as the total size of the object and library files you're linking, if you want it to link in a realistic timescale.

Inadequate memory under Windows 98 just slows the linker; whereas under Windows 2000 it cripples it. Here are some sample timings for a huge link job (315Mb of object files), which illustrate this effect:

Under Windows 98:

With 512 MB: 1m 44s
With 256 MB: 3m 07s
With 128 MB: 5m 52s

Under Windows 2000:

With 512 MB: 1m 58s
With 256 MB: 4m 05s
With 128 MB: over three hours!

Sections and groups

In order to understand the power of `ps2link` and to write a linker script which makes the most of your target and your game, you will first need to have a basic understanding of *sections* and *groups*.

A *section* is a block of bytes which the compiler (or you, if you are coding in assembler), knows contains information of a similar type. For example, all program code goes into the section called `.text`, initialized variables go into the `.data` section and uninitialized variables go into the `.bss` section.

Object files contain a list of section names and the data to be placed in each one. At its simplest, linking is the process of combining all the `.text` sections into one large `.text` section and then locating it somewhere in the target memory, and repeating for each named section.

With `ps2link` you have considerably more control over the target image through the concept of *groups*. A group is a container which can hold any number of sections. Groups will be defined in your linker script; they have properties including their `ORG` address (where they are loaded into PlayStation 2 memory) and their `OBJ` address (the address to which they must be relocated before they will work in PlayStation 2 RAM).

You will then assign sections to groups based on the section names. You can add a unique prefix to the section names from each object file or from any library, so that you can locate the sections from one object file in a completely different area of memory from those taken from another.

Replacing the GNU linker ld with ps2link

This section covers how to get up and running with `ps2link` instead of the GNU PlayStation 2 linker (`ld`).

- You must write a new linker script, which may be done by modifying the example linker script `ps2.lk` (see "Example linker script" on page 61). This is installed in `\usr\local\sce\ee\lib` if you installed the ProDG EE build tools.
- You must ensure that every section is placed in a group in your linker script. For more information about the linker scripts (see "Linker control script language" on page 54).
- Write or modify your existing `makefile` so that `ps2link` is called instead of the GNU PlayStation 2 `ld` linker. This `makefile` automatically finds the `ps2.lk` file in its default location (see below).

Calling ps2link instead of ld

The `ee-gcc` compiler driver cannot be set up to directly call `ps2link` instead of `ld`. This means that you will need to amend your `makefile` (or write one) to make the compiler driver create object files (add the `-c` option to the `ee-gcc` command line) and then explicitly call `ps2link` to make the final output files.

Here is an example of the type of `makefile` that you will need to write to call `ps2link`. This `makefile` builds the Sony demo `blow.elf`.

```
SHELL       = /bin/sh

TOP         = ../../..
LIBDIR      = $(TOP)/lib
INCDIR      = $(TOP)/include

TARGET      = blow
OBJS        = $(TARGET).o physics.o data.o fireref.o firebit.o \
              src.o wood.o grid.o

LCFILE      = $(LIBDIR)/ps2.lk
LIBS        = $(LIBDIR)/libgraph.a \
              $(LIBDIR)/libdma.a \
              $(LIBDIR)/libdev.a \
              $(LIBDIR)/libpkt.a \
              $(LIBDIR)/libpad.a \
              $(LIBDIR)/libvu0.a

PREFIX      = ee
AS          = $(PREFIX)-gcc
CC          = $(PREFIX)-gcc
LD          = ps2link
DVPASM      = $(PREFIX)-dvp-as
OBJDUMP     = $(PREFIX)-objdump
RUN         = dsedb -r run
RM          = /bin/rm -f

CFLAGS      = -g -Wall -Werror -Wa,-al -fno-common
CXXFLAGS    = -g -Wall -Werror -Wa,-al -fno-exceptions -fno-common
ASFLAGS     = -c -xassembler-with-cpp -Wa,-al
DVPASMFLAGS = -g
```

```
LDFLAGS     = -l $(LIBDIR) -l $(TOP)/gcc/ee/lib -l \
              $(TOP)/gcc/lib/gcc-lib/ee/2.9-ee-990721
TMPFLAGS    =

.SUFFIXES: .c .s .cc .dsm

all: $(TARGET).elf

$(TARGET).elf: $(OBJS) $(LIBS)
   $(LD) $(LDFLAGS) $(OBJS) $(LIBS) \
   @$(LCFILE),$(TARGET).elf,$(TARGET).map

crt0.o: $(LIBDIR)/crt0.s
   $(AS) $(ASFLAGS) $(TMPFLAGS) -o $@ $< > $*.lst

.s.o:
   $(AS) $(ASFLAGS) $(TMPFLAGS) -I$(INCDIR) -o $@ $< > $*.lst

.dsm.o:
   $(DVPASM) $(DVPASMFLAGS) -I$(INCDIR) -o $@ $< > $*.lst

.c.o:
   $(CC) $(CFLAGS) $(TMPFLAGS) -I$(INCDIR) -c $< -o $*.o > $*.lst

.cc.o:
   $(CC) $(CXXFLAGS) $(TMPFLAGS) -I$(INCDIR) -c $< -o $*.o > $*.lst

run: $(TARGET).elf
   $(RUN) $(TARGET).elf

clean:
   $(RM) *.o *.map *.lst core *.dis *.elf
```

Invoking the linker

The linker can be invoked on the command line in the following way:

```
ps2link [switches] [files] @scriptfile,output,mapfile
```

`switches`	Any of the command-line switches described in the following section. These must be preceded with a hyphen "-".
`files`	Can be object files or libraries. The linker will use these files in addition to any files specified in the linker script.
`scriptfile`	The linker treats this file as its control script. See below for a description of the format. The SN Systems convention is to use `.lk` as the extension for linker scripts.
`output`	The destination file for the linked code. This file should have a `.elf` extension. The symbol table information used by the debugger is also written to the output file.
`mapfile`	The destination file for the section/symbol map. This filenormally contains just a dump of all groups and sections written out by the linker, showing their `OBJ` and `ORG` start and end addresses and sizes. There is also a list

of the symbols in your program. This list appears twice, once sorted by name and once by address.

Note: *Quoted filenames, e.g. "My Mapfile", are allowed.*

ps2link command-line switches

The command-line switches are preceded by a dash (-), and can be any of the following:

Switches	Actions
`-ci`	Link case-insensitively.
`-e sym=val{;sym=val..}`	Define one or more symbols with the given value (like EQU directive).
`-entry symbol`	Specify symbol as program entry point.
`-exceptions`	Enable exceptions.
`-G n`	Specify small data threshold.
`-l path`	Add search path for libraries.
`-li path`	Add search path for object files.
`-o address`	Set the initial ORG address to `address`.
`-st`	Produce a static coverage report.
`-strip`	Strip unused elements from the program.
`-we`	Treat warnings as errors.
`-wm`	Warn of multiple declarations of XBSS symbols in C.
`@@file`	Read response file for additional options.

The rest of this section describes the switches in more detail.

Link case-insensitively (`-ci`)

Use this switch to force `ps2link` to ignore the case of all symbols. Normally `ps2link` preserves the case of all symbols. You almost certainly won't need this switch.

Define symbols with their values (`-e sym=val`)

Use this switch to define one or more symbols with the given value on the command line. This is equivalent to the `EQU` directive in the linker script. Spaces are not permitted in the `sym=val` expression, though a space between the `-e` switch and the symbol assignment is optional.

The ability to define symbol values on the command line enables you to maintain a linker script that can be used for more than one project. In this way any specific project symbols can be defined on the command line.

For example:

```
-e _gp=__lit8_obj+0x7ff0
```

which defines a symbol `_gp` which is calculated as the obj address of the `.lit8` section plus the hex constant `0x7ff0`.

Specify symbol as program entrypoint `-entry symbol`

By default the linker looks for the symbol `ENTRYPOINT` and uses that as the run address for your image and the entry point for the static coverage scan. Use this switch to tell the linker that your program entry point is a different symbol.

Enable exceptions `-exceptions`

The GNU C compiler `gcc` creates certain routines to initialize and close down exception handling if it finds you using any exception constructs (`try` / `catch`) in your code. These routines must be called before program start and after program end. The way `ps2link` does this is to require you to specify `-exceptions` on the command line. This will make it look for a couple of routines in your program which are expected to perform this initialization and shutdown work. These routines are called `sn_reg_frame` and `sn_dereg_frame` and are implemented in `sn_exceptions.o`, which is shipped with the linker. If you don't link with this object file, the linker will explicitly tell you so. But if you do, exceptions will work fine.

Specify small data threshold `-G n`

When generating common variables (variables defined but not initialized in your source) the linker needs to know whether the compiler is treating them as "small data" (accessed relative to the GP register) or as regular "large" data. The value for the compiler's –G option specifies this threshold. You should give the same threshold value to the linker to maintain consistency. The default value is 8, the same as the compiler's default –G value.

Add library search path `-l path`

The path that you specify using the `-l` switch is added to the set of paths that `ps2link` uses to search for libraries (see "INCLIB statement" on page 59). `ps2link` always tries to locate files relative to the current directory first.

Add include search path `-li path`

The path that you specify using the `-li` switch is added to the set of paths that `ps2link` uses to search for object files (see "INCLUDE statement" on page 59). `ps2link` always tries to locate files relative to the current directory first.

Produce static coverage report `-st`

This switch tells the linker to produce a report on unused functions and variables in your image. See "Dead-stripping" on page 69 for more details.

Strip unused elements from the program `-strip`

This switch tells the linker to remove unused functions and variables from your image. See "Dead-stripping" on page 69 for more details.

Treat warnings as errors `-we`

Use this switch to specify that `ps2link` treats warnings as errors.

Detect duplicate C variable definitions `-wm`

Use this switch to indicate that `ps2link` checks for duplicate uninitialized variable definitions. This means situations where you have defined a globally-visible variable with the same name more than once. (In ANSI C you should define a variable once, and export it in a header using `extern`, but the alternative semantics are still supported by `gcc`.)

The compiler will complain if there is a compile-time problem in resolving these multiple definitions. But a linker cannot do anything about it if the compiler accepts the code and must allocate a single common variable of the largest requested size. If you accidentally declare the same name as two different but compatible types (e.g. `short` and `long`) then references to the smaller types may break at runtime (particularly on big-endian targets). `ps2link` will therefore report this as a warning and refer you to the object files where the clashing declarations were found. If the declarations indicate different sizes you'll get an additional warning.

Linker control script language

`ps2link` requires a linker script, which is a map of how your game looks both in the console memory and in the files that `ps2link` will create. It sounds complicated and can be tricky at first but is reasonably logical once you get the hang of it.

The `ps2link` linker script is written in a simple language which defines the following:

- groups and their properties
- the allocation of sections to groups
- object files to be read to obtain sections, symbols, and code/data
- library files to be used to resolve references to symbols not found in the specified object files

The script is line-based. Comments can be included if you prefix them with a semicolon, i.e. the part of a line after a semicolon is ignored by the linker. Blank lines are also ignored, therefore you can use them to make the script easier to read.

Sections and groups

`ps2link` works with the concept of *groups*. A group is an area of memory with certain properties such as:

- where it is in the console memory, referred to as OBJ-space
- where it is in the code image, referred to as ORG-space
- whether it has a maximum size

Groups contain *sections*. Sections are created by the compiler to distinguish types of program data:

- `.text` contains your game code
- `.data` and `.rodata` contain initialized variables, and `.sdata` contains initialized variables below a certain size threshold
- `.bss` contains uninitialized variables, and `.sbss` contains uninitialized variables below a certain size threshold
- `.ctors` and `.dtors` contain addresses of functions required to create global objects in C++

The purpose of a linker is to clump together all the "like-named" sections from all the input objects and libraries, forming single large sections. It then allocates these sections to groups. The content of the groups, and the symbolic debug information describing them, are written out to code and symbol files, and a map file lists the addresses of all your sections, groups, and symbols in ASCII readable form.

This is what `ps2link` does, and the linker script is the means of controlling it.

Groups are written into the code outputs in the order they occur in the script. Their ORG addresses are the addresses to which they are loaded. The OBJ address of a group (which can be different) is the address where it must be relocated to, for it to work.

Script syntax

This is the syntax for a `ps2link` linker script, in approximately the order the directives should appear.

In all examples, white space is optional except that any directive which starts with a name rather than the directive keyword itself, must be in the leftmost column of the script, whereas all other directives must be indented at least one space.

Numbers are assumed to be decimal unless preceded by `0x` or `$`, and names (specifically filenames) can't contain a semicolon, space, or comma unless they're enclosed in string quotes.

These are the commands recognised in the script language:

`INCLUDE filename[,prefix]`

Include sections from `filename`, optionally prefixing their names with `prefix`.

`INCLIB filename[,prefix]`

Use `filename` as a library to resolve unresolved references, optionally prefixing all section names with `prefix`.

`ORG address`

Set the ORG address to `address`.

`Name EQU value`

Define a symbol with the given `value`.

`Name GROUP attributes`

Define name as a group with the specified attributes.

```
SECTION[.align] name[,group]
```

Declare name as a section, optionally giving it alignment align and assigning it to group group.

```
SECTALIGN number
```

Set the default section alignment for all subsequent sections.

Each line should begin with a space or tab unless it defines a name, in which case the name must start in the first column. You can use spaces or tabs to separate elements on each line.

Command reference

The rest of this section describes in more detail the commands that can be put into a linker script.

ORG statement

This directive specifies the start address of the image in ORG space.

```
ORG <address>
```

You should always have one ORG directive, near the top of your script. You can re-ORG subsequent parts of the image later in the script. You can also specify an initial ORG address with the -o option. If you do not have an ORG address, 0 is assumed.

GROUP statement

This statement defines a group with the given attributes. The linker creates five symbols for each group describing its position in OBJ space and ORG space. These symbols are as follows:

Symbols	Definition
_groupname_obj	The start of the group in OBJ space.
_groupname_objend	One more than the end of the group in OBJ space. This value will have the same alignment as the group OBJ address.
_groupname_org	The start of the group in ORG space.
_groupname_orgend	One more than the end address of the group in ORG-space. This value will have the same alignment as the group ORG address.
_groupname_size	The actual size of the data in the group (un-aligned).

Period characters in group names are converted to underscores in these symbols.

You can refer to these names in your C source by declaring them to be external arrays of chars of unspecified size, e.g.

```
extern char _text_obj[];
```

group

This directive creates a group.

```
<name>    group    <attributes>
```

Groups represent "units of output" as well as "blocks of memory". Each group will generate some code, some symbols, and some map information. You can decide where these will be written to, or let them go to the default outputs.

group attributes

The group attributes determine how `ps2link` handles the group. Currently you can use the BSS attribute after the name of your group. Additional attributes will be added in future versions of the linker.

bss

This means the group can only contain empty sections, and will not normally be written out into the output image, since the standard startup code will zeroize this area of memory. If any initialized data ends up in a group marked `bss`, `ps2link` will report an error.

SECTALIGN statement

This directive specifies the default alignment for subsequent section directives.

```
sectalign <alignment>
```

`alignment` can be any power of 2 from 0 to 16, i.e. any appropriate integer from 1 to 65536.

Sections are normally aligned using the following rules:

- the default alignment is 1
- `sectalign` directives can change this
- specifying an alignment in a section directive changes it for that section
- any part of a section specified with higher alignment in an object file uses that alignment instead

The default alignment created by the PS2 compiler depends on the section in question and your compiler options, but is typically at least 8 bytes.

SECTION statement

The `SECTION` statement declares sections and allocates them to groups.

```
section<.alignment>    sectionname,groupname
```

`alignment`	can be any power of 2 from 0 to 16, i.e. any appropriate integer from 1 to 65536. Section chunks will be aligned to at least this boundary—possibly more if an individual chunk from an object file needs a greater alignment.
`sectionname`	is the name of the section to recognise

groupname is the name of the group which will hold the section.

- You can use a wildcard syntax to match section names. The * (asterisk) character will match zero or more characters, while ? (question mark) matches exactly one character. When ps2link tries to find a section directive to match a section declaration in an object file, all section directives without wildcard characters are checked, and then all specifications with wildcards are checked in order of appearance.

Sections appear in a group in the order they're defined in section directives. Sections which match wildcarded specifications appear within the group in the position which would have been occupied by the wildcarded directive, allowing you to collect unused sections anywhere you like. If several sections match a wildcarded directive, they appear in the order they were encountered.

- You cannot provide two non-wildcarded mappings for the same section.

All sections from included files (and from object files that have been included via the INCLIB statement) and which have exactly the specified name, are combined into a single block and placed in the designated group. You should apply a prefix in the INCLUDE or INCLIB statement if you want the sections of a file to be allocated to another group.

ps2link creates descriptor symbols for each section in the same way as for groups:

Symbol	Definition
_sectionname_obj	The start of the group in OBJ space.
_sectionname_objend	One more than the end of the group in OBJ space. This value will have the same alignment as the group OBJ address.
_sectionname_org	The start of the group in ORG space.
_sectionname_orgend	One more than the end address of the group in ORG space. This value will have the same alignment as the group ORG address.
_sectionname_size	The actual size of the data in the group (unaligned).

Period characters in the section name are converted to underscores in these symbols. The symbols for a section called .text are thus prefixed with two underscores. You can refer to them in your C code by making a similar declaration to that for the special GROUP names given above.

INCLUDE statement

This directive includes an object file in your game.

```
include object <,sectionprefix>
```

Including an object file means that its sections will appear in the output image.

ps2link searches for the object file relative to the current directory and then using a set of search paths (see the -li option and the sn.ini include_path= entry).

sectionprefix is applied to the name of all sections found in the object file.

ps2link can include object files direct from library files. The syntax for this is

```
include lib(object) <,sectionprefix>
```

ps2link must understand the format of the library file lib. Currently ps2link only understands the formats used by SN and Sony librarian tools. If ps2link can understand the library and finds an object named object in it, it includes that object, as though you had extracted it from the library and included it directly.

- You must write an INCLUDE statement for each object file in your game.
- You can prefix the names of the sections in an object file with a name, typically the name of the group to which you intend to allocate them. However, there does not have to be any correlation between the prefix you choose and the group to which a section is allocated.

INCLIB statement

This directive refers ps2link to a library for resolving symbols. References to functions and variables not defined in your game must be resolved from libraries for the game to link properly. You must write an INCLIB statement for each library which you want to use.

```
inclib library <,sectionprefix>
```

ps2link searches for the library file relative to the current directory and then using a set of search paths (see the -l option and the sn.ini library_path entry). ps2link will search a set of directories for each library file you include using INCLIB, and will use the first version it finds, should there be a choice.

After resolving as many references as possible between all the object files named in your script, and checking imported symbol files, ps2link turns to the list of libraries, and uses their symbol indexes to pull in additional object files from those libraries to resolve the remaining references. Including these objects may then lead to further unresolved symbols and further library resolution.

ps2link always tries to find unresolved symbols by searching the libraries in the order specified in the script. Sometimes the order in which libraries appear can be crucial, especially if different libraries have different versions of functions under the same name. You can use the -wl option to make ps2link report duplicate symbols in different libraries, to alert you to where this happens.

ps2link does not include the whole of a library just to resolve a single symbol. It adds only the single object from the library which contains the symbol resolution.

- You can add a further name to the section names in library object files as a prefix, typically the name of a group to which you intend to allocate them. However, there does not need to be any correlation between the prefix and the group to which a section is allocated.

Note: *All object files taken from a library will get the same prefix. If you want to use different prefixes for different object files, you will need to take more advanced steps, such as breaking the library into two or more smaller libraries or extracting the object files and using* `INCLUDE`.

EQU statement

This statement will define a global symbol by giving it a specified value. This directive lets you equate a name to an expression, which at present can be a symbol name or a decimal or hexadecimal constant.

```
newname     equ     expr
```

`newname` becomes another globally visible name for the result of evaluating `expr`.

Section and group descriptors

ps2link creates five symbols for each group in the script. These are:

`_groupname_obj`	the start address of the group in OBJ-space
`_groupname_objend`	the end address of the group in OBJ-space
`_groupname_org`	the start address of the group in ORG-space
`_groupname_orgend`	the end address of the group in ORG-space
`_groupname_size`	the size of the group's contents

ps2link also generates a symbol called `_sectionname` for every section which it recognises or creates. The value of the symbol is the start address of the section in OBJ-space.

Periods in group or section names are converted to underscores in these symbols' names, so the size of the `.text` section is represented by the symbol `__text_size`.

These symbols can be referenced from your programs as variables of type `extern char[]`.

Example linker script

The following is an example linker script, `ps2.lk`, which will work for the standard PlayStation 2 demos. It is installed in `\usr\local\sce\ee\lib` if you installed the ProDG EE build tools.

```
;       SN Systems default linker script for PS2.
;       Default libraries. (Supply others on the command line.)

        inclib  libc.a
        inclib  libkernl.a
        inclib  libgcc.a
        inclib  libm.a

;      The heap size and stack details are defined here.

_heap_size      equ     0xffffffff
_stack  equ     0xffffffff
_stack_size     equ     0x00100000

;       Groups represent entries in the output ELF's program headers
;       table. Each contains one or more sections.
;       A group only appears in the PHDRS table if it is named and
;       has nonzero size.

        org 0x00100000

indata  group

        section *.indata,indata

;       This group is for the program's code and initialized data.

        org 0x00200000

text    group

        sectalign 8
        section .text,text
        section .vutext,text
        section .reginfo,text

        sectalign 16
        section .data,text
        section .vudata,text
        section .rodata,text
        section .rdata,text
        section .gcc_except_table,text

;       Collect everything else which is part of the image here.
;       (Subsequent section directives get a chance to collect
;        contents first.)

        section *,text

;       Set the GP register's value.
;       The total size of these sections (from .lit8 to .scommon)
;       cannot exceed 64K.

_gp     equ     __lit8_obj+0x7FF0
```

```
        section .lit8,text
        section .lit4,text
        section .sdata,text

;       This group is for uninitialized data

bss     group   bss

;       This is the start marker for the startup code's zeroing
;       routine.

_fbss   equ     _bss_obj

;       These sections are to be zeroized by crt0.o.

        section .sbss,bss
        section .scommon,bss
        section .bss,bss
        section .vubss,bss

;       This is crt0.o's marker for the start of the heap.

_end    equ     _bss_objend

;       This group is for the scratchpad.

        org 0x70000000

spad    group

        sectalign 4

        section .spad,spad
```

The linker and libraries

Before `ps2link` looks at the specified library files, it attempts to resolve all references using symbols from the object files and symbols generated by the linker.

If an unresolved symbol reference is then found in a library, the object module containing that symbol is read from the library and `ps2link` acts as though it had been added to the bottom of the linker script; its section contents are appended to the program image `ps2link` is building, and its unresolved references are added to the list, possibly requiring more library objects to be loaded.

By loading only the object module containing the referenced symbol, and not the entire library, `ps2link` minimizes the overhead of using libraries as far as is possible. If the libraries are efficiently organized, ideally with only one function per object module, this will ensure that your program is not made bigger than necessary by irrelevant library code.

If you want to ensure that a particular library object is always linked in, extract it from the library using `PSYLIB2` and add an appropriate `INCLUDE` statement to your linker script.

ps2link searches libraries for missing symbols in the order they are specified in the linker script and will load the required module from the first library seen to contain the symbol in question. This means that you can override a library's implementation of a function by writing a reference to a library file which contains an alternative implementation to it, and placing this line before the INCLIB statement of the first library. Alternatively you can write a function with the same name, compile it into an object file and include this explicitly in your script.

Building relocatable DLLs

You can now build relocatable DLLs for the PlayStation 2 EE processor. This enables you to build programs which can dynamically load and unload code modules to any address (subject to correct code alignment) which the system will then relocate and link into other code which has already been loaded. The linking is performed by name so that you can simply write a call to a function in a different module in the usual way even though that module is not built into the same .elf file as the caller. Modules can be loaded in any order and any module can refer to symbols in any other module, irrespective of the order of loading. The debugger is able to automatically identify which modules are loaded and find and load the debug information associated with them.

Building the relocatable DLL modules and the main program module is achieved by the use of the program ps2dlllk which accepts a script file and a normal linker command line, reads the script, calls the linker and then processes the output produced by the linker to create the relocatable DLL (.rel extension plus debug information in a .elf file) or the main program file (.elf extension). The only variation required to the standard linker command line is the use of a modified linker control file (rel.cmd to build a DLL or relapp.cmd to build the main program) in place of the standard app.cmd.

Invoking the DLL linker

The PlayStation 2 DLL linker program ps2dlllk can be invoked on the command line in the following way:

```
ps2dlllk <script-file> <linker> <linker command line args>
```

e.g.

```
ps2dlllk physics.lk ee-gcc -T rel.cmd -o physics.elf
physics.o -nostartfiles
```

This will create physics.rel containing the code and relocation information and physics.elf containing the debug information.

Note: *Currently only the GNU linker is supported, either directly as ld.exe or via ee-gcc.exe.*

Some restrictions that currently apply are :

- The whole program must be built without GP optimization by specifying -G0.
- There is no support for unmangling C++ names in the ps2dlllk script files

- If a call is made to a function that hasn't been loaded yet then address 0 will be called and the program will crash. There is no automatic loading of required DLLs. It is up to the programmer to make sure they are in memory before using them.
- The file `relapp.cmd` contains a definition for the `.sndata` sections specifying its size explicitly as 16384 bytes:

  ```
  .sndata ALIGN(128): {sn_dll_header_root = .; . += 16384;}
  ```

  If your program is large then this default may not be enough. In this case `ps2dlllk` will print an error message which will specify how big this section needs to be and you will then have to edit this file to increase the size.

Files required

`ps2dlllk.exe`	The DLL linker
`libsn.a`	Contains functions to link into the main program
`libsn.h`	Header file for `libsn.a`
`rel.cmd`	GNU linker script for building DLLs
`relapp.cmd`	GNU linker script for building the main program

`ps2dlllk.exe` will be placed with your other ProDG executables. `libsn.a`, `rel.cmd` and `relapp.cmd` should be in `/usr/local/sce/ee/lib`. `libsn.h` should be in `/usr/local/sce/ee/include`.

Script file for ps2dlllk

The script file is a sequence of commands. White space (spaces and tabs) are ignored.

The script file syntax for `ps2dlllk` is :

`; comment`

A semi-colon marks the start of a comment which continues to the end of the line

`.main`

This directive tells `ps2dlllk` that the module being built is the main program rather than a relocatable DLL.

```
.export wildcard-name
.noexport wildcard-name
```

These directives control which symbols are exported from the module (DLL or main program) for other modules to use. By default, all global symbol names are exported.

The patterns specified by `.export` and `.noexport` directives are applied in the order they are listed in the script file. The pattern can be an explicit name, e.g.

```
.export start   ; exports the symbol start
```

or can end with a `*` to match any sequence of characters, e.g.

```
.noexport *     ; don't export any symbols
.export   X*    ; export all symbols beginning with X
```

`.reference symbol-name`

This tells the linker to act as if the specified symbol name had been referenced in the code being linked and so to include the module defining the symbol from one of the specified libraries in the output file even if there is no other use of the symbol in the program. This allows you to force particular library routines into either the main program or particular DLLs where they can then be shared by all the other DLLs.

`.resolve dll-file-name`

This directive specifies that `ps2dlllk` should search the specified relocatable DLL file (`.rel` extension) for any symbol names that it makes use of but does not define and then to ensure that these symbols are defined in the main program or relocatable DLL being created. This would typically be used in the main program's script file to make sure that all required libraries are available but could be used in a relocatable DLL too.

`.nodebug`

This directive specifies that debug information should not be generated for this program / DLL.

`.index symbol-name index-number`

This directive is used to build a table of pointers to functions in the module being created which will allow access to these functions by indexing into the table rather than by name. This can be used to avoid the situation where more than one module defines the same name but only one of them can be accessed.

The index number can be any integer >= 0. The table will be of a size as defined by the highest index number used so using unnecessarily large numbers will lead to a large table being created.

A pointer to the index table for a relocatable DLL is returned from the call to the function `snDllLoaded()`. See below.

Associated library functions

These are the functions required by the main program to use the DLL system. They are supplied in `libsn.a` which should be linked into the main program.

```
int snInitDllSystem (void** index_pointer);
```

You should call this function at the start of your program to initialize the DLL system. If a non-null parameter is specified then the address of the index table for the main program will be returned there.

```
int snDllLoaded (void* buffer, void** index_pointer);
```

Call this function after a relocatable DLL has been loaded into memory. The first parameter points to the memory where the DLL has been loaded. This must be aligned to the boundary required by the DLL. If it isn't then the error code `SN_DLL_BAD_ALIGN` is returned. Typically, an alignment of 128 byte will suffice.

If the second parameter is not null then the address of the index table for the DLL is returned there.

```
int snDllUnload (void* buffer);
```

This routine should be called before the memory containing a DLL is freed. The first parameter is the base address of the memory containing the DLL.

```
int snDllMove (void* destination, void* source, void**
index_pointer);
```

This routine moves a DLL from one location to another making all required adjustments. This allows you to defragment your allocated memory but:

1. Any pointers to code or data in the DLL will not be fixed up.
2. A DLL cannot move itself nor call a routine that moves it.

The return codes from these functions are defined in `libsn.h` and are as follows:

Error	Definition
SN_DLL_SUCCESS	0 - operation succeeded
SN_DLL_NOT_A_DLL	1 - the buffer doesn't seem to contain a DLL
SN_DLL_BAD_VERSION	2 - the DLL version is not supported by this code
SN_DLL_INVALID	3 - some data in the DLL header were invalid
SN_DLL_BAD_ALIGN	4 - the DLL is not aligned to the required boundary
SN_DLL_NOT_LOADED	5 - The DLL has not been loaded so can't be unloaded
SN_DLL_TOO_MANY_MODULES	6 - Too many modules loaded

Example

This example shows how to modify the Sony `vu1/blow` sample to make the module `physics.c` into a relocatable DLL.

Modifications to the makefile

1. Remove `physics.o` from the list of object files:

```
OBJS = crt0.o \

       $(TARGET).o data.o fireref.o firebit.o src.o wood.o
       grid.o debug.o
```

2. Add `libsn.a` to the list of libraries:

```
LIBS =   $(LIBDIR)/libgraph.a \
         $(LIBDIR)/libdma.a \
         $(LIBDIR)/libdev.a \
         $(LIBDIR)/libpkt.a \
         $(LIBDIR)/libpad.a \
         $(LIBDIR)/libvu0.a \
         $(LIBDIR)/libsn.a
```

3. Make sure that `-G0` is specified on the compiler command line:

```
CFLAGS = -G0 -g -Wall -Werror -fno-common
```

4. In the build rule for the main program:

   - Add `physics.elf` as a file that the main program is dependent on. This will ensure that it is always built before the main program so that the `physics.rel` file can be used in a `.resolve` directive of the main program's `ps2dlllk` script file.
   - Prefix the call to the GNU linker with a call to `ps2dlllk` and its script file.
   - Change the GNU linker script filename to `relapp.cmd`.

```
$(TARGET).elf: $(OBJS) $(LIBS) physics.elf

ps2dlllk blow.lk $(LD) -o $@ -T
/usr/local/sce/ee/lib/relapp.cmd \

$(OBJS) $(LIBS) $(LDFLAGS)
```

   The file `blow.lk` contains the two lines

```
.main                ; this is the main program
.resolve physics.rel ; make sure we supply all routines
                     ; needed by physics.rel
```

5. Add a rule to build `physics.elf` from `physics.o` using `ps2dlllk`:

```
physics.elf: physics.o

ps2dlllk physics.lk $(LD) -o $@ -T
/usr/local/sce/ee/lib/rel.cmd \

physics.o -nostartfiles
```

   For this example, the file `physics.lk` can be empty which will result in all global symbols defined in `physics.o` being exported.

Modifications to blow.c

1. Add a `#include` of `libsn.h`

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

2. In the function `main()` add the following at the start of the function:

```
int f;
int physlen;
char* physbuf;

/* Initialize the SN PS2 relocatable DLL system */
if (snInitDllSystem(0))
{
printf("Failed to initialize DLL system\n");
return 1;
}

/* Read in the physics.rel file to allocated memory on a 128 byte
boundary */
f =
sceOpen("host0:/usr/local/sce/ee/sample/vu1/blow/physics.rel",
SCE_RDONLY);
```

```
if (f < 0)
{
printf("Error opening physics.rel\n");
return 1;
}

physlen = sceLseek(f, 0,SCE_SEEK_END);
sceLseek(f, 0, SCE_SEEK_SET);
physbuf = malloc(physlen + 127);
physbuf = (char*) (((int)physbuf + 127) & ~127); /* Align to 128
byte boundary */
sceRead(f, physbuf, physlen);
sceClose(f);
FlushCache(0);

/* Inform the system that a DLL has been loaded. This will
relocate it and hook up all inter-module references */

if (snDllLoaded(physbuf, 0))
{
printf("Failed to install physics.rel DLL\n");
return 1;
}
```

It is now possible for the main program to call the `SetParticlePosition()` function in the `physics.rel` module and for that module to call library functions in the main program.

The DLL checker

The ProDG PlayStation 2 DLL checker program `ps2dllcheck` is used to check for undefined symbols in a DLL (`.rel` file) or PlayStation 2 EE executable (`.elf` file), which has been linked using the ProDG DLL Linker for PlayStation 2, `ps2dlllk` (see "Building relocatable DLLs" on page 63). Provided the DLL and `.elf` file have been built correctly, all of the undefined symbols in the `.elf` should refer to functions in the DLL, and *vice versa*.

You should run this utility after building a DLL and before debugging it, in order to check that all undefined symbols are mutually resolved, as calls to symbols which cannot be referenced will cause an exception to be thrown and will greatly slow down debugging your code.

Note: *You can use the ProDG PlayStation 2 DLL checker to find function calls which may have been accidentally misspelt in your source, as these will also be listed as undefined symbols.*

ps2dllcheck command-line syntax

The PlayStation 2 DLL checker program `ps2dllcheck` can be invoked on the DOS command line as follows:

```
ps2dllcheck <file> <file> ... <file>
```

where `<file>` is the name of a DLL or `.elf` file built with `ps2dlllk`. If `<file>` has not been built using `ps2dlllk`, then this error message is displayed:

```
File <file> is not a DLL or an ELF created with PS2DllLk
```

Displaying undefined symbols

The program `ps2dllcheck` lists undefined symbols similar to the following:

```
Undefined symbols :
SetParticle Position
```

Symbols which are undefined in a `.elf` file are assumed to be resolved in a DLL, and *vice versa*. You can check that this is the case by listing both filenames as arguments to the DLL checker program, as in the following example:

```
ps2dllcheck blow.elf physics.rel
```

If all of the undefined symbols, found by checking each file separately, are resolved by examining the other file(s) in the list, then you should see the message:

```
No undefined symbols
```

When you have established that all symbols are resolved then you can safely start debugging your application.

Dead-stripping

`ps2link` can detect and remove unused code and data in your game image. Associated debug info is also removed. The result is as though those functions and variables had not been compiled in your original source files.

Dead-stripping affects only the output image generated by `ps2link`; your object files and libraries are not changed in any way by the linker.

We recommend that you turn on dead-stripping for C++ projects to avoid GP segment bloat. Alternatively, try compiling with the `-fno-keep-static-consts` option to stop the compiler from emitting `consts` into `.sdata`.

To enable dead-stripping, add the `–strip` switch to the `ps2link` command line.

Dead-stripping produces a report in a file called `statcov.txt`. This report lists the items which have been stripped from the image. You can choose just to produce the report, without actually doing the stripping, with the `–st` switch.

`ps2link` detects unused items by starting at the program entry point and checking which objects are referenced by it, recursing as it goes. The default entry point is the symbol `ENTRYPOINT`, which is defined in the startup code source, `crt0.s`. If you have a different entry point symbol you should use the `-entry` switch to specify it.

Occasionally `ps2link` will incorrectly detect functions or data items which it believes can be removed from the image but which are actually required by your game to function. In these cases you may find the following options useful to tweak `ps2link`'s dead-stripping capabilities:

`-stripmin n`	Do not strip any item of `n` bytes or smaller. The default threshold is 16.

Note: *We recommend that a `-stripmin` value less than 8 should not be used; below that level there are occasional misalignment artifacts, usually in the libraries.*

`-nostriplib`	Do not perform stripping in any library object file. (Stripping still takes place in regular object files.)
`-nostripobj`	Do not perform stripping in any object file. (Stripping still takes place in library object files.)
`-ns name1 name2 ... –ns`	Preserve the named items.
`-nsf file1 file2 ... -nsf`	Preserve any items in files whose names include any of the specified wildcardable patterns.

Additional notes

The complete list of sections likely to appear in compiler output is:

Sections	Use
`.text`	Program code
`.data`	Initialized variables
`.rodata`	Read-only data such as strings
`.bss`	Uninitialized variables
`.sdata`	Initialized variables (small data)
`.sbss`	Uninitialized variables (small data)

Using ps2ld

`ps2ld` is a replacement for `ee-ld` which will eventually unify all the functionality of both `ee-ld` and `ps2link`. The main advantage to be gained in using `ps2ld` is that it is considerably faster than either `ee-ld` or `ps2link`. Furthermore, you will not have the problems associated with deciding which linker to use for a particular project.

This release however, only supports the `ee-ld` command line options necessary to build PlayStation2 programs and script files but future releases will also fully support the `ps2link` syntax.

In addition, `ps2ld` has been extended to include unused function stripping. This is activated by adding `–strip-unused` to the linker command line or `–W1,-strip-unused` to the `ee-gcc` command line.

`ps2ld` is also compatible with Visual Studio Integration and `ps2dlllk`.

Building with ps2ld

1. Rename the original GNU `ld.exe` file to something different.
2. Rename `ps2ld` to `ld.exe` and place it in `\usr\sce\local\sce\ee\gcc\ee\bin`.

3. Amend the `sn.ini` file so that it will use `collect2` to call the linker you have named as `ld.exe` in the above directory.

Additional command line switches in ps2ld

Switch	Description
`-strip-unused`	Removes unused function code
`-report-unused`	Writes a list of unused functions to the file `statcov.txt` (same as `ps2link`)

Unsupported command line switches in ps2ld

All the `ee-ld` command line switches are recognised by `ps2ld` but `ps2ld` will issue a warning when the following unimplemented switches are used:

Switch	Description
`-aarchive`	Shared library control for HP/UX compatibility
`-ashared`	Shared library control for HP/UX compatibility
`-adefault`	Shared library control for HP/UX compatibility
`-architecture`	Set architecture
`-A`	Set architecture
`-Bdynamic`	Link against shared libraries
`-Bstatic`	Do not link against shared libraries
`-Bsymbolic`	Bind global references locally
`-call_shared`	Link against shared libraries
`-c`	Read MRI format linker script
`-dy`	Link against shared libraries
`-dn`	Do not link against shared libraries
`-dynamic-linker`	Set the dynamic linker to use
`-embedded-relocs`	Generate embedded relocations
`-EB`	Link big-endian objects
`-format`	Specify target for following input files
`-filter`	Filter for shared object symbol table
`-force-exe-suffix`	Force generation of file with .exe suffix
`-f`	Auxiliary filter for shared object symbol table
`-F`	Filter for shared object symbol table
`-h`	Set internal name of shared library
`-mri-script`	Read MRI format linker script
`-m`	Set emulation
`-non_shared`	Do not link against shared libraries

-oformat	Specify target of output file
-relax	Relax branches on certain targets
-rpath dir	Adds a directory to the runtime library search path
-rpath-link DIR	Try to locate required shared library files in the specified directory
-soname	Set internal name of shared library
-split-by-file	Split output sections for each file
-split-by-reloc	Split output sections every COUNT relocs
-shared	Create a shared library
-task-link	Do task level linking
-traditional-format	Use same format as native linker
-version-script	Read version information script
-wrap	Use wrapper functions for SYMBOL

Unsupported script file directives in ps2ld

The following script file directives are not implemented in `ps2ld` and will generated an error if used:

`OUTPUT_ARCH`

`OUTPUT_FORMAT`

`CONSTRUCTORS`

`HLL`

`SYSLIB`

`VERSION`

`TARGET`

The following directives are accepted by `ps2ld` but will be ignored:

`PHDRS`

`NOCROSSREFS`

`MEMORY`

`SORT`

`KEEP`

`CREATE_OBJECT_SYMBOLS`

Chapter 6: **ProDG Target Manager for PlayStation 2**

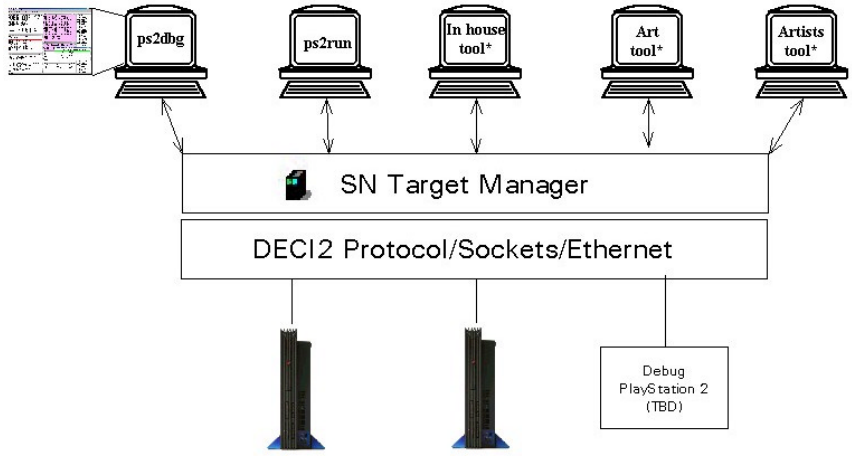
Overview of the Target Manager

A *target*, in this context, is a DTL-T10000 development tool on a LAN.

ProDG Target Manager for PlayStation 2 is used to control access to any PlayStation 2 targets that you have on your network. It provides control, host fileserving and TTY output for all PlayStation 2 development kits on a LAN.

- In order for applications that communicate with the PlayStation 2 target to run, (ProDG Debugger, `ps2run`, etc.) the Target Manager must be running and have at least one target set up.
- This application has been developed separately so that multiple tools can connect to the target via the same interface, and to allow you to connect to targets, maintain sessions and view session information outside of your particular application.

Starting ProDG Target Manager prior to ProDG Debugger enables you to set up a session on the required PlayStation 2 that you can use with the Debugger or with `ps2run`. This allows you to maintain a session with a particular target even when you start and stop the Debugger. This way of working also has the advantage of allowing you to lock out a session on the required target PlayStation 2 for as long as you wish to work with it. This means that when you exit the Debugger you can remain connected to the target for further use with the Debugger or `ps2run`.



Accessing PlayStation 2 targets from your own application

ProDG Target Manager for PlayStation 2 SDK is now available as a separate product for those who wish to access PlayStation 2 targets from their own applications, for example, art preview and internal tools.

For further information, visit the Products area of the SN Systems web site (see "Updates and technical support" on page 3 for contact information).

Launching the Target Manager

The ProDG Target Manager for PlayStation 2 can be launched via the command line or via the **ProDG for PlayStation2 > ProDG Target Manager for PS2** shortcut in the **Start** menu, and is automatically started when you start ProDG Debugger.

The **Start** menu shortcut just accesses the `ps2tm` command line, and can therefore be customized to start the Target Manager in the state that you wish. For example you can specify that the Target Manager always tries to connect to a particular target on start-up.

The Target Manager must be running and have at least one target set up for any of the other tools to be able to communicate with the target PlayStation 2.

ps2tm command-line syntax

The Target Manager command line is the following:

```
ps2tm [<options> [<args>]]
```

where `<options>` can be either commands or options that relate to those commands.

The command switches can be any of the following:

Command switches	Actions
`-?`	Show a list of all the command-line options
`-a <target>`	Enables you to add a new target. This must be followed with the `-i <ip address>` argument, and any other arguments as required
`-t <target>`	Allows you to modify the parameters of an existing target. This can be followed by a list of arguments to do whatever is required to the specified target
`--delete <target>`	Deletes the specified target
`-m`	Starts the Target Manager in a minimized state (system tray icon)

The two commands `-a` and `-t` are used to add or indicate a particular target. If you put either one of these on the `ps2tm` command line, you can follow it with

any number of target options that enable you to specify what you wish to do on the target.

The options that can be used on a target are the following:

Option switches	Actions
`-f <path>`	Set the file serving directory path for the specified target.
`-h <path>`	Set the home directory for the specified target.
`-i <ip address>`	Set the network address for the specified target. Note that this will fail if you are already connected to the target (i.e. `-devtool1 -c -i add1` will fail).
`-b <ee_boot>, <iop_boot>`	Set boot parameters for the specified target. These are the parameters that are to specify behavior when you reboot your PlayStation 2 (see "Resetting the target" on page 85).
`-p <port>`	Set the port number for the specified target (by default this is set to 8510). Note that this will fail if you are already connected to the target.
`-c`	Connect to the specified target.
`-d`	Disconnect from the specified target.
`-r`	Reset the target, which can only be used when you are connected to the target (for example `-c -r` will always work).

For a particular target, you can enter any number of options, in any order, though there are some exceptions:

- You cannot change the port number or IP address after connecting to a target
- You cannot reset the target unless you have previously connected to it

If you try to do something that fails, then remaining changes on the current target will be ignored and the next target command on the command line will be executed.

For example, the command:

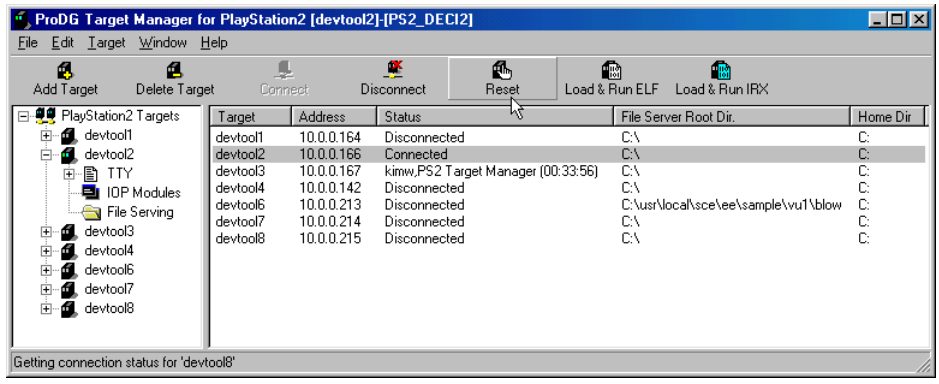
```
ps2tm -a Devtool1 -i <ip_address> -c -r
```

will start the Target Manager, add a new target called "`Devtool1`" with the specified IP address, connect to it and reset it, whereas

```
ps2tm -a Devtool1 -i <ip_address> -c -r -t Devtool2 -c -v
```

will do the same as the previous example, but in addition connect to and reset the target called "`Devtool2`".

The Target Manager user interface



The Target Manager window includes five menus: **File**, **Edit**, **Target**, **Window** and **Help**. These contain the commands that enable you to add and configure targets, connect to them and load files on selected targets.

Below the menu bar is a toolbar containing buttons that enable you to access the most useful commands rapidly.

The rest of the Target Manager window resembles Windows Explorer and shows any PlayStation 2 targets that you might have added on the left and their properties in the right part of the window.

In the left-hand part of the window, targets to which you are already connected have the LEDs hilighted on the target icon.

In the right-hand part of the window, the target properties are arranged in columns labelled **Target**, **Address**, **Status**, **File Server Root Dir.** and **Home Dir**. The column widths may be resized by dragging the column separators either left or right.

To update target connection status

At any time you can refresh display of the current connection status and directory settings for all of the available targets, by pressing the shortcut key `<F5>`. Pressing `<F5>` should always result in the connect time for connected targets being updated, and any targets that have been newly connected or disconnected.

To sort rows in Target Manager main window

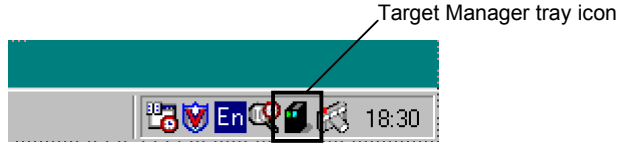
You can sort the rows displayed on the right by clicking on either the **Target** or **Address** column headings. This will sort the listed targets alphabetically by their target name or IP address, respectively.

To expand out a target

You can view the different TTY output streams of any target, by expanding the target folder in the left part of the window, then expanding the TTY folder, and finally selecting the module whose TTY output you wish to monitor. TTY output will then be sent to the right part of the window.

Target manager tray icon

When you minimize the Target Manager it becomes a system tray icon. If you wish to open the main window again at any time, you can either double-click the tray icon or click the **Open** command in the tray icon shortcut menu:



Keyboard shortcuts

Many of the most frequently used commands in the Target Manager can be accessed via keyboard shortcuts. Currently the keyboard shortcuts are not customizable.

For detailed information on the Target Manager keyboard shortcuts, see "Keyboard shortcut reference" on page 191.

Exiting the Target Manager

The Target Manager can either be closed from a menu option in the main window or through its tray icon shortcut menu.

To exit the Target Manager

1. If the Target Manager main window is displayed, click **Close** in the **File** menu. However if the Target Manager is minimized you can click **Exit** in the icon shortcut menu.
2. In both cases a dialog appears asking if you are sure that you wish to exit the Target Manager.
3. Click **OK**.

Note: *If you are connected to a target when you exit the Target Manager, this connection will be terminated.*

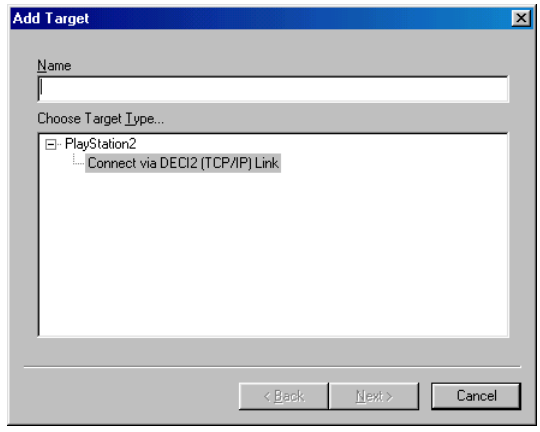
Adding and removing targets

Before you start the Debugger the Target Manager must be used to set up the properties of the PlayStation 2 targets in your network.

This section describes how you set up the targets in the Target Manager to enable you to connect to the different PlayStation 2 targets in your network.

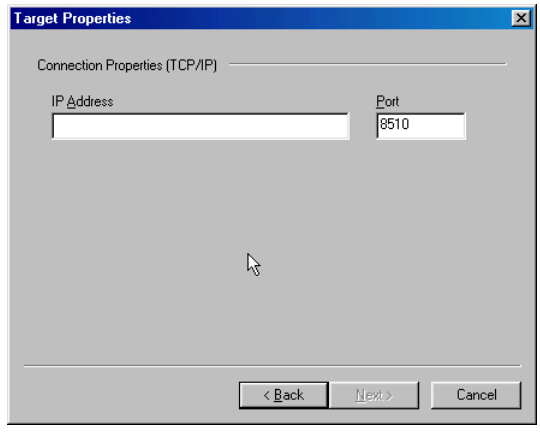
To add a new target

1. Click **Add Target** in the **File** menu (or use the **Add Target** toolbar button, or the `<Insert>` shortcut key), and the following dialog is displayed:



2. Enter a name to identify the target session, in the **Name** field. Then click **Next**.

   The Target Properties dialog is displayed:



3. Enter the IP address of the PlayStation 2 that you would like to connect to in the **IP Address** field.
4. Enter the connection port in the **Port** field (this would normally not be changed from `8510` unless your network administrator has modified this).
5. Click **Next**.

   The Add Target Completed confirmation window is displayed, with details about the target to be added.

6. If you wish to proceed, click **Finish** to add the target. The new target and its properties appear in the main part of the Target Manager window.

Note: *It is possible to change the properties of a target when it is not connected. For more information see "Configuring target" on page 80.*

To remove a target

1. Click on the target that you wish to delete in the main window.

2. Click **Delete Target** in the **File** menu (or use the **Delete Target** toolbar button, or use the <Delete> shortcut key). A dialog is displayed asking you to confirm that you wish to delete the target.
3. Click **OK**.

   The target is deleted. If you were connected to the target when you delete the session then you are automatically disconnected.

File serving using the SIM device

The ProDG Debugger and Target Manager provide full Windows support for the `sim:` device for file serving. This means that your application can file serve using the `sim:` device to refer to any files that might need to be opened.

This directory is a property of each target referred to as the **Current Dir** in the Target Manager, and can be changed at any time.

Another property of a target in the Target Manager is the **Home Dir**. If you are migrating from Linux you may have entered absolute filenames in your source using the "~" character, to refer to your home directory. The Target Manager and Debugger support these path names and will replace the ~ directory, in any file path name, with the **Home Dir** specified.

For example if your file was found in `~/myappfiles` on Linux then the Debugger will look for the file in `C:\myappfiles` on your Win32 machine if the Target Manager home directory is set to `C:`

The file serving directory can also be quickly changed in the Debugger to the directory you load a `.elf` file from when you load manually.

When running your application on the PlayStation 2 you can view the file serving statistics in real-time in the Target Manager (see "Viewing your application file serving" on page 89).

To change the file serving directory

The `sim:` device file serving directory can be changed at any time.

1. Select the target for which you wish to change the file server root, in the left-hand list of PlayStation 2 targets.
2. Click **Set File Server Root** in the **Target** menu, and a folder browsing dialog appears.
3. Browse until you locate the directory that you would like to be used for file serving and click on it.
4. Click **OK**.

   The properties of the selected target should be updated in the right side of the main window to show the newly selected file server root directory under the **Current Dir** title.

It is also possible to change the file serving directory from inside your code using the following function call:

```
sceOpen("host:SETROOT:<path>", SCE_RDONLY);
```

e.g.

```
sceOpen("host:SETROOT:d:\\", SCE_RDONLY);
```

To set the home directory

The home directory is the directory on your Win32 PC that is used to replace any file paths that you might have specified with a "~" (representing your Linux home directory) in your application source.

1. Select the target for which you wish to change the home directory, in the left hand target list in the Target Manager main window.
2. Click **Set Home Directory** from the **Target** menu, and a folder browsing dialog appears.
3. Browse until you locate the directory that you would like to be used to replace the Linux home directory (~) and click on it.
4. Click **OK**.

   The properties of the selected target should be updated in the right side of the main window to show the newly selected home directory.

It is also possible to change the home directory from inside your code using the following function call:

```
sceOpen("host:SETHOME:<path>", SCE_RDONLY);
```

e.g.

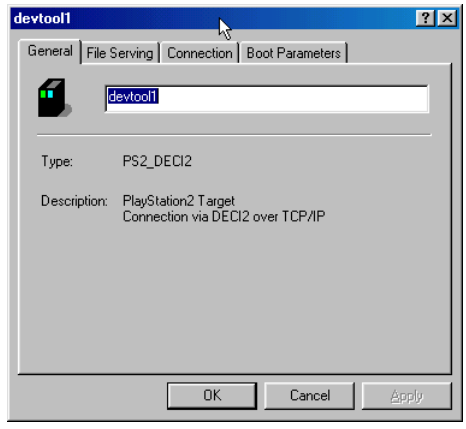
```
sceOpen("host:SETHOME:d:", SCE_RDONLY);
```

Configuring targets

If you wish to change any of the properties of an existing target in a single dialog, you can do this in the target property sheet. However, you will not be able to change any connection properties if you are already connected to the target; they are greyed out in the dialog.

To modify the target properties

1. Select the required target in the left hand list.
2. Click **Target Properties** in the **Target** menu, or right-click to display the shortcut menu and then select **Properties**, or use the `<alt+enter>` shortcut key. The property sheet for the selected target is displayed.



In this dialog you can modify all the properties of the selected target, including its name, IP address, port, home directory and file serving directory and its boot parameters. Note that you cannot modify the name, IP address or port of a target when connected to it. You will need to disconnect first. For more information on boot parameters, see "Resetting the target" on page 85.

3. Modify the required properties.
4. Click **OK**.

Connecting to targets

ProDG Target Manger can either be launched via the command line or Windows shortcut, or it is launched automatically by the Debugger. For any other tools (Debugger or `ps2run`) to be able to interact with a particular target, it must already be configured in the Target Manager.

When the Debugger or `ps2run` exit, by default they will leave the target in the state that it was in when they started. So, for example, if you connect to an existing session when you start the Debugger, the session will remain connected when you quit the Debugger. However you can modify this behavior using options on their command lines to specify, for example, that you are always disconnected.

To connect to a target

1. Select the target that you would like to connect to in the right-hand list of the Target Manager main window.
2. Double-click on the target name, or click **Connect** in the **Target** menu (or click the **Connect** toolbar button, or the <c> shortcut key).

   If your connection was successful then the target properties in the right-hand part of the window will be updated to show `Connected` in the **Status** column. However if another user is currently connected to the target a dialog is displayed saying that the target is in use by another user, and the identity of the user that is currently connected is displayed in the **Status** column. The **Status** field for another connected user takes the form:

   `<computer>@<domain>,<app> (<connect time [hh:mm:ss]>)`

e.g.

```
Mike@snsys.com,PS2 Target Manager (02:00:10)
```

Your connection may also time out which means that the target is unavailable on the network. There may be several reasons for this, including the following:

- The selected target may not be connected to the network correctly or turned on.
- The TCP/IP software may not be correctly configured. For example, the PlayStation 2 may have a duplicate IP address. To test for this you can power off the PlayStation 2 development platform and `ping` the IP address. If you receive a response then the IP address is duplicated on your network.
- When you enter the IP address or DNS name to identify the PlayStation 2 target there is no verification of the name when you input it, therefore when you connect you may find that you have badly specified the IP address or DNS name or that the DNS name you entered is not associated with the correct IP address.
- The PlayStation 2 Development Tool may simply need to be rebooted.

Disconnecting from targets

When the Debugger or the `ps2run` command exit, then you may still be connected to the PlayStation 2 target in the Target Manager. If you wish to disconnect from the target because you have finished working with it, then you will need to do this in the Target Manager.

To disconnect from a target

1. Select the connected target that you would like to disconnect from in the right hand list of the Target Manager main window.
2. Double-click on the target name, or click **Disconnect** in the **Target** menu (or click the **Disconnect** toolbar button, or the `<d>` shortcut key). The target properties will update to show `Disconnected` in the **Status** column.

It is also possible to disconnect from the target using the following function call:

```
sceOpen("host:DISCONNECT:", SCE_RDONLY);
```

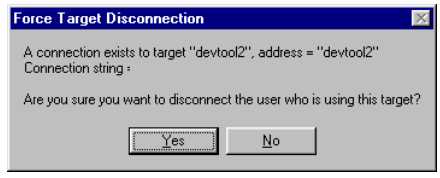
To forcibly disconnect a target

The Target Manager also allows you to forcibly disconnect another user who is connected. Target manager will not allow you to forcibly disconnect from a target to which *you* are already connected.

Forced disconnection is useful if someone has connected via `dsidb/dsedb` but has closed the telnet connection; the session can be disconnected via the Target Manager without having to reset the DTL-T10000.

Note: *The connected user will not receive any warning that they are being disconnected, and may lose valuable work as a result, so this option should be used only when absolutely necessary.*

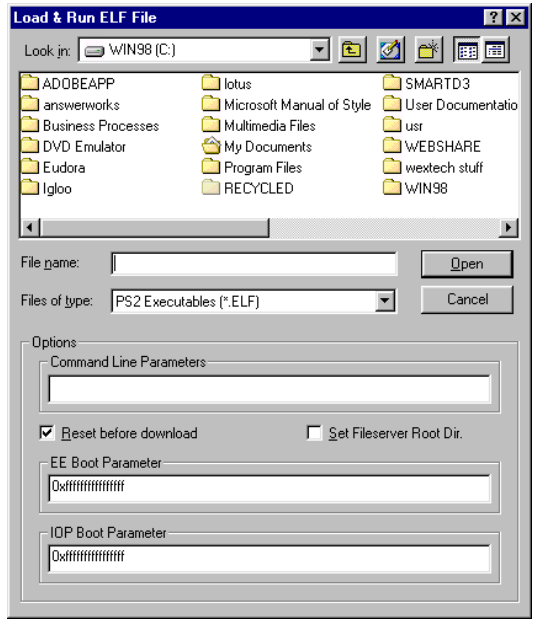
1. Click **Force Disconnect** in the **Target** menu.
2. A connect status query is performed to determine if any users are connected via either the EE or IOP debug protocols. If so, the Force Target Disconnection confirmation dialog is displayed:



3. Click **Yes** provided you are absolutely sure you want to forcibly disconnect the user who is using this target.
4. The target properties will update to show `Disconnected` in the Status column.

Loading and running ELF files

You can load and run `.elf` files on the target directly from the Target Manager. To do this click the **Load and Run ELF** button on the Target Manager toolbar, or click the command in the **Target** menu. A dialog appears in which you can select the `.elf` file to be loaded:



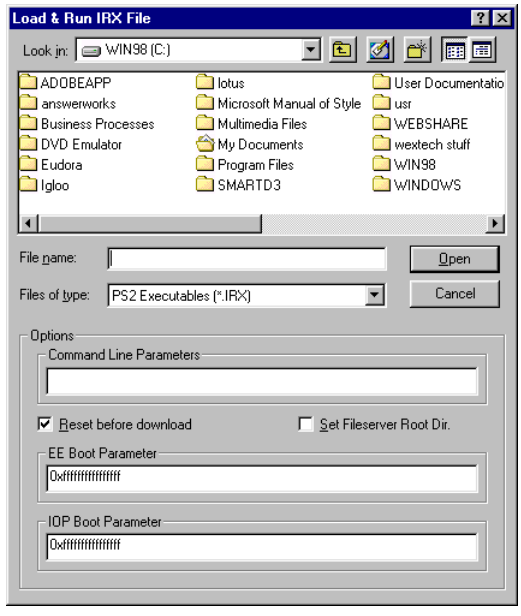
A space is provided to allow you to specify **Command Line Parameters**, i.e. arguments that are passed in with the program name.

If the **Set Fileserver Root Dir.** checkbox is checked, then the directory in which the `.elf` file has been selected from is set as the Fileserver Root directory for that target.

You can specify that the target is reset before the program load, by setting the **Reset before download** checkbox. The dialog also enables you to set the boot parameters for the EE and IOP units (see below), which are used during a reset.

Loading and running IRX files

You can load and run `.irx` files on the target directly from the Target Manager. To do this click the **Load and Run IRX** button on the Target Manager toolbar, or click the command in the **Target** menu. A dialog appears in which you can select the `.irx` file to be loaded.



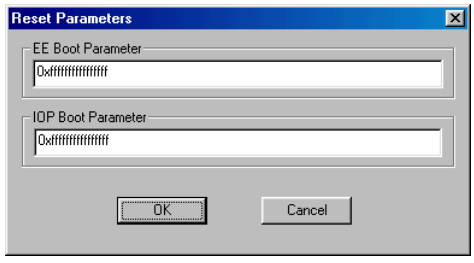
A space is provided to allow you to specify **Command Line Parameters**, i.e. arguments that are passed in with the program name.

If the **Set Fileserver Root Dir.** checkbox is checked, then the directory in which the `.irx` file has been selected from is set as the Fileserver Root directory for that target.

You can specify that the target is reset before the program load, by setting the **Reset before download** checkbox. The dialog also enables you to set the boot parameters for the EE and IOP units (see below), which are used during a reset.

Resetting the target

At any time the target can be reset using the **Reset** button. A dialog appears in which you can set the boot parameters for the EE and IOP units separately. Click **OK** to start the target reset. If the Debugger is currently connected to the target that you are resetting, then you will need to load your application `.elf` file to the target to continue debugging. In addition any breakpoints that you have already set in the application source in the Debugger will be discarded.



This command always resets both PlayStation 2 units but their behavior following a reset is determined by the boot parameters. Within the Target Manager the boot parameters are specific to each target and can be changed in either the Target Properties sheet (see "Configuring target" on page 80), or when you reset the target or load and run a `.elf` or `.irx` file.

For more information on the boot parameters, please refer to the Sony PlayStation 2 Developer Tool documentation.

Viewing output from the PlayStation 2

You can expand each target node listed in the left part of the window to show the content of the different output streams originating from the PlayStation 2 processors.

If you are not currently connected to the selected target, the TTY panes will still show any output resulting from a previous connection.

The first level under the target node is split into three directories: **TTY**, **IOP Modules** and **File Serving**.

Viewing TTY stream output

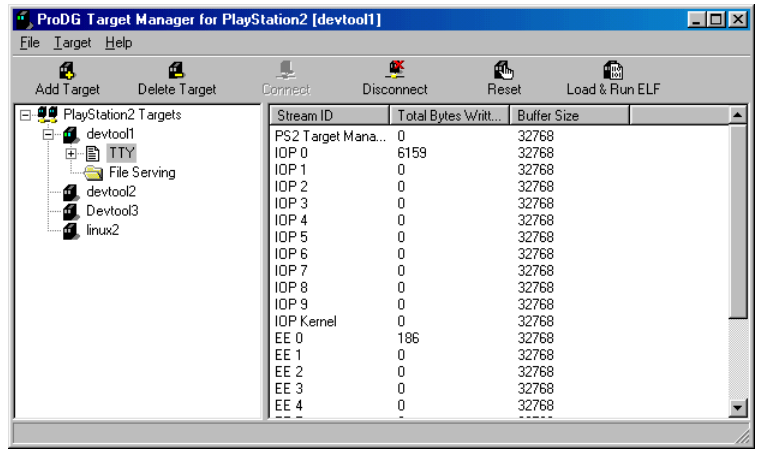
You can view a summary of the TTY streams, or each TTY stream can be displayed individually.

To view a summary of TTY stream output

- Click on the **TTY** directory to select it.

  The right-hand pane switches to show a list of TTY streams available: **PS2 Target Manager**, **IOP streams 0-9** and **IOP Kernel**, **EE streams 0-9** and **EE Kernel**, and **Network**.

For each stream the **Total Bytes Written** column contains the number of output bytes to that stream, while the **Buffer Size** column shows the size of the TTY buffer for the stream:



To view an individual TTY stream

This procedure assumes that you are already viewing the TTY node for a chosen target (see "To view a summary of TTY stream output" on page 85).

There are two ways to view an individual TTY stream:

- In the list of TTY streams in the right-hand pane, double-click on a row to select a particular stream.

or

- Double-click on the **TTY** directory, or click on the + sign next to it.

  The **TTY** directory expands to show a tree view of all TTY output streams available.

  When a TTY pane contains new output it is indicated by a red icon in the TTY stream list.

Select an individual TTY stream from the list presented.

The right-hand pane switches to show any TTY output from that stream.

To clear TTY stream output

TTY streams are cleared automatically when you shut down ProDG Target Manager. However, the output will persist after a connection has been closed, so it may be useful to know how to clear the output from a stream when necessary.

1. Select the stream to be cleared.
2. Right-click on the right-hand pane to display the shortcut menu:

3. Click **Clear** to clear the output.

To copy TTY stream output to another application

TTY stream output can be saved to another application for later analysis using the standard Windows cut, copy and paste operations:

1. Select the stream to be copied.
2. Select the text to be copied by swiping it with the mouse button depressed, or click **Select All** or `<Ctrl+A>` from the TTY pane shortcut menu if you want to select everything in the pane.
3. Press `<Ctrl+C>` to copy the selected text to the clipboard, or click **Copy** from the TTY pane shortcut menu.
4. Press `<Ctrl+V>` to paste the selected text from the clipboard to the application, or select **Paste** in the application's menu.

To change the TTY stream font

The TTY stream font can be altered to suit your preference.

1. Select **Set Font** from the TTY pane shortcut menu.

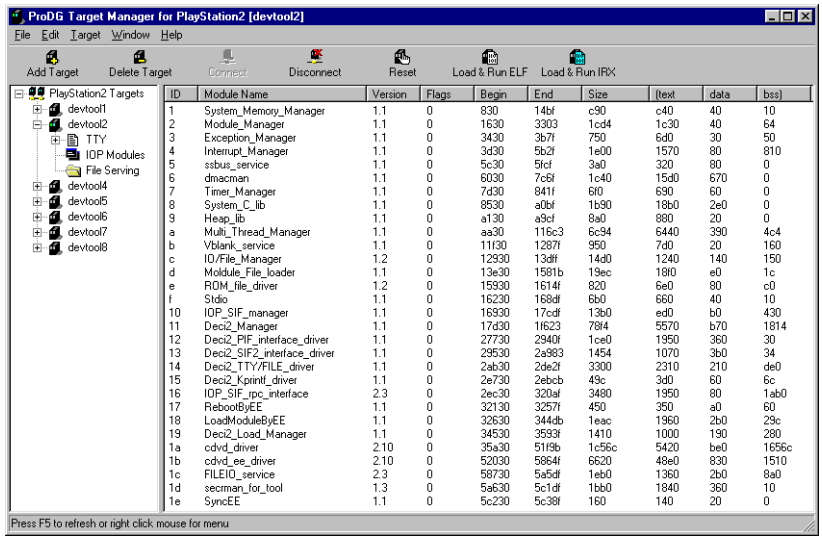
   The Font dialog is displayed for you to set the font face, style and point size.

2. Press **OK** to accept the new settings.

The settings will apply to all TTY stream panes, not just the current one.

Viewing IOP modules loaded

You can view a table showing all the IOP modules currently loaded on the IOP processor, by selecting the **IOP Modules** directory under the target icon in the left-hand side of the Target Manager window.



ID	Module Name	Version	Flags	Begin	End	Size	(text	data	bss)
1	System_Memory_Manager	1.1	0	830	14bf	c90	c40	40	10
2	Module_Manager	1.1	0	1630	3303	1cd4	1c30	40	64
3	Exception_Manager	1.1	0	3430	3b7f	750	6d0	30	50
4	Interrupt_Manager	1.1	0	3d30	5b2f	1e00	1570	80	810
5	ssbus_service	1.1	0	5c30	5fcf	3a0	320	80	0
6	dmacman	1.1	0	6030	7c6f	1c40	15d0	670	0
7	Timer_Manager	1.1	0	7d30	841f	6f0	690	60	0
8	System_C_lib	1.1	0	8530	a0bf	1b90	18b0	2e0	0
9	Heap_lib	1.1	0	a130	a9cf	8a0	880	20	0
a	Multi_Thread_Manager	1.1	0	aa30	116c3	6c94	6440	390	4c4
b	Vblank_service	1.1	0	11f30	1287f	950	7d0	20	160
c	IO/File_Manager	1.2	0	12930	13dff	14d0	1240	140	150
d	Moldule_File_loader	1.1	0	13e30	1581b	19ec	18f0	e0	1c
e	ROM_file_driver	1.2	0	15930	1614f	820	6e0	80	c0
f	Stdio	1.1	0	16230	168df	6b0	660	40	10
10	IOP_SIF_manager	1.1	0	16930	17cdf	13b0	ed0	b0	430
11	Deci2_Manager	1.1	0	17d30	1f623	78f4	5570	b70	1814
12	Deci2_PIF_interface_driver	1.1	0	27730	2940f	1ce0	1950	360	30
13	Deci2_SIF2_interface_driver	1.1	0	29530	2a983	1454	1070	3b0	34
14	Deci2_TTY/FILE_driver	1.1	0	2ab30	2de2f	3300	2310	210	de0
15	Deci2_Kprintf_driver	1.1	0	2e730	2ebcb	49c	3d0	60	6c
16	IOP_SIF_rpc_interface	2.3	0	2ec30	320af	3490	1950	80	1ab0
17	RebootByEE	1.1	0	32130	3257f	450	350	a0	60
18	LoadModuleByEE	1.1	0	32630	344db	1eac	1960	2b0	29c
19	Deci2_Load_Manager	1.1	0	34530	3593f	1410	1000	190	280
1a	cdvd_driver	2.10	0	35a30	51f9b	1c56c	5420	be0	1656c
1b	cdvd_ee_driver	2.10	0	52030	5864f	6620	48e0	830	1510
1c	FILEIO_service	2.3	0	58730	5a5df	1eb0	1360	2b0	8a0
1d	secrman_for_tool	1.3	0	5a630	5c1df	1bb0	1840	360	10
1e	SyncEE	1.1	0	5c230	5c38f	160	140	20	0

This displays a view of the IOP modules with detailed information for each module, including the module **Version**, where the module is loaded in memory (**Begin** and **End**), the **Size** of the module, and the size of the `text`, `data` and `bss` sections.

You can alphabetically sort the IOP module display by clicking on either the **ID** or **Module Name** column headings.

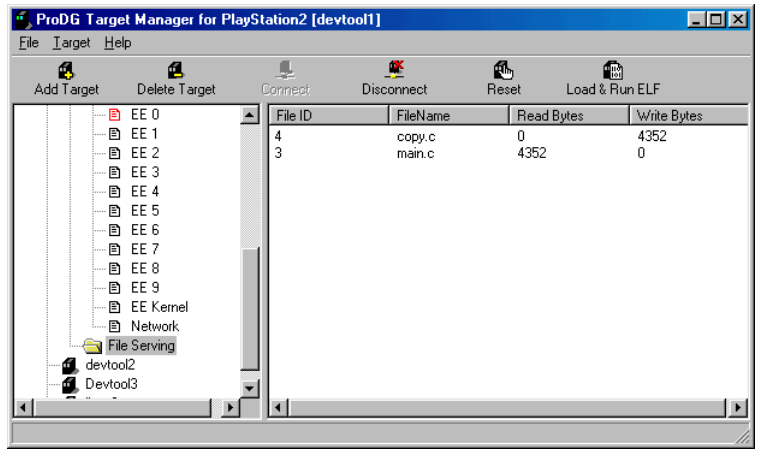
To refresh the IOP Modules table

The IOP Modules table is refreshed automatically only after selecting the **Connect** or **Reset** menu options from Target Manager (not if the connection / reset is done from ProDG Debugger).

At all other times if you need to refresh the IOP modules list, press `<F5>` or right-click on the pane to display the IOP Modules shortcut menu and select the only available option: **Refresh List**.

Refreshing the list will only work if the IOP is running. If the operation cannot be carried out the display will remain blank and no error will be displayed.

Viewing your application file serving



If you open the **File Serving** directory for the target that you are currently debugging, while your application is running on the target, you can see in real time the files that have been opened and written to, or read from, by your application. In addition the total number of bytes that have been written to each file or read from each file, while your application is running, is shown.

Flashing the kernel

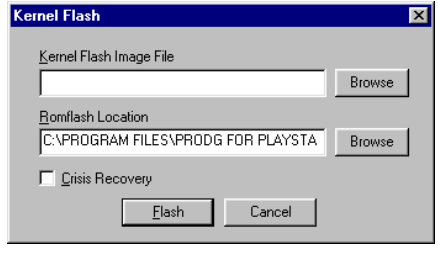
Whenever new Sony libraries are released it may be necessary to update the PlayStation 2 Development Tool kernel held in EEPROM. This can be achieved by using the Flash Kernel option provided in Target Manager.

It is also possible to recover from a situation where a ROM flash has created an error, by re-flashing from the DTL-T10000's shadow ROM.

To flash the target kernel

1. Connect to the target to be flashed.
2. Click **Flash Kernel** from the **Target** menu.

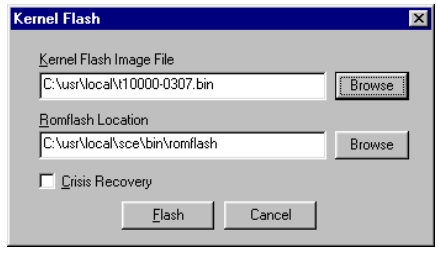
   The Kernel Flash dialog is displayed:



3. Using the **Browse** button, browse for the **Kernel Flash Image File**. This is normally located in your `\usr\local` directory and will have a `.bin` filename extension, e.g. `C:\usr\local\t10000-0307.bin`.

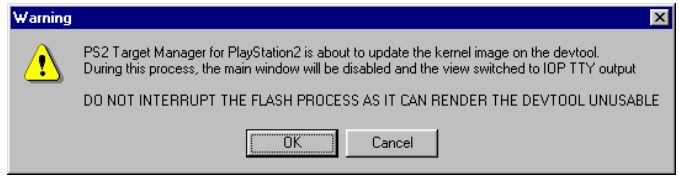
4. The **Romflash Location** field defaults to the directory in which the ProDG Target Manager program `ps2tm.exe` is found, plus the filename `romflash`. Using the **Browse** button, browse for the `romflash` program in its actual location. This is normally in your `\usr\local\sce\bin` directory, e.g. `C:\usr\local\sce\bin\romflash`.

   The Kernel Flash dialog should now look similar to the following:



5. Make sure that the **Crisis Recovery** checkbox is clear. Click **Flash**.

   The following warning message is displayed:



6. Read the warning message carefully! Provided you are sure you want to proceed, click **OK**.

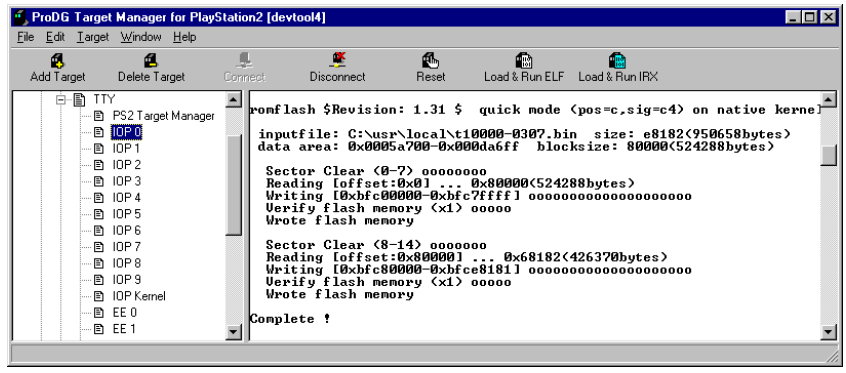
> **Note:** *It is VERY IMPORTANT that the ROM flash process is allowed to continue to completion, otherwise it may be difficult to reconnect to the DTL-T10000.*

7. The existence of both the ROM flash module and kernel flash image files are verified. Then the main Target Manager window is locked to prevent the user from disconnecting halfway through the flash.

   First a reset of the target is issued with the boot parameters `ee=0,iop=7` (see "Resetting the target" on page 85). Then the ROM flash module is downloaded to the IOP and run, passing in the name of the kernel flash image file as an argument.

8. During flashing the TTY window is switched to IOP channel 0 so that the results of the flash can be observed.

   When the flash has completed, TTY / IOP 0 output similar to the following will be seen:



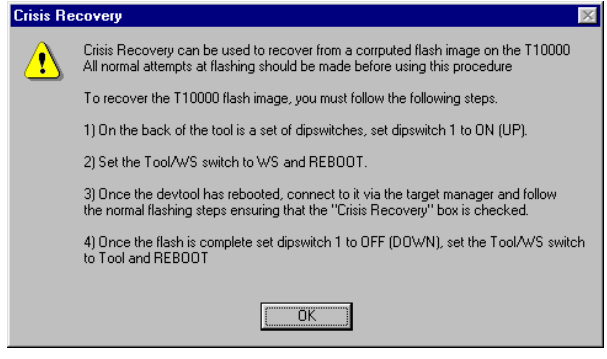
Finally, a reset of the target is issued with the boot parameters `ee=0`, `iop=0` (see "Resetting the target" on page 85).

9. The DTL-T10000 EEPROM has now been updated with the new version of the kernel flash image file.

To recover from a ROM flash crisis

Crisis recovery should only be necessary if a ROM flash has aborted with an error, or if an invalid flash image file has been used, so rendering the kernel unbootable. In this situation you would not be able to connect and/or communicate with the DTL-T10000 from Target Manager.

1. Click **Flash Kernel** from the **Target** menu.
2. Check the **Crisis Recovery** checkbox.
3. Read the Crisis Recovery instructions carefully:



The instructions displayed form the rest of this procedure:

4. On the back of the DTL-T10000 are some dip switches. Set dip switch 1 ON (up).
5. Set the Tool / WS switch to WS and reboot the DTL-T10000.
6. Now connect to the target from Target Manager.

   Follow the normal flashing procedure (see "To flash the target kernel" on page 89), but ensuring that the **Crisis Recovery** checkbox is still checked.

In this situation booting is from the shadow ROM on the workstation. Then the `romflash` module is downloaded to the IOP and run, passing in the name of the kernel flash image file as an argument. Because the `-mpu4shadow` parameter has been specified, the EEPROM on the Development Tool is then updated with the specified kernel image file. During flashing the TTY window is switched to IOP channel 0 so that the results of the flash can be observed.

7. When the flash operation has completed, set the DTL-T10000 dip switch 1 to OFF (down), set the Tool / WS switch to Tool and reboot the DTL-T10000.

Chapter 7: **The ProDG command-line utility**

Overview

This section contains information on the command-line utility `ps2run`, that is available as part of ProDG for PlayStation 2.

The ps2run command line utility

`pS2run` is a command-line tool for communicating with a PlayStation 2 target, which is used in conjunction with the ProDG Target Manager for PlayStation 2 application: `ps2tm`.

The Target Manager (`ps2tm`) must be running for `ps2run` to work properly. `ps2run` has options which can reset the target, set its file server directory, load an ELF file, download a binary file and disconnect from the target.

ps2run command-line syntax

`pS2run` can either be used directly from the command line, or in batch or make files.

The `ps2run` command line is as follows:

```
ps2run <options> [<file> [<args>]]
```

Use `<file>` [and optional `<args>`] to specify the name of the executable `.elf` file (and optional arguments to that program) which you wish to run on the target.

Omitting `<file>` and `<args>` just performs the specified options.

The following options may be entered as arguments to `ps2run`:

`-b <address>`	Enables you to specify that a binary download will take place, to the hexadecimal address that you specify.
`-d`	Disconnects from the target when `ps2run` has finished running. However, if there was an existing session in the Target Manager that you connected to, then `ps2run` will leave the session intact.
`-da`	Specifies that you are always disconnected from the target when `ps2run` finishes running.
`-f <path>`	Enables you to set the path of the file serving directory.
`-h <path>`	Set home directory to `<path>`.
`-nd`	Enables you to specify that you are not disconnected from the target when the `ps2run` finishes running (default).
`-nr`	The target is not reset. This is the default behavior.
`-nx`	Enables you to specify that the code is not executed on the target after loading. This switch is ignored if you are downloading a binary file.
`-p`	Enables you to specify that `stdout` stream is displayed. No `.elf` file is necessary with this switch.
`-q`	Enables you to specify that `ps2run` is in quiet mode providing there are no errors.
`-r <ee_boot>, <iop_boot>`	Resets the PlayStation 2 target before loading and running your application. The boot parameters enable you to specify the behavior of the target after reset (see "Resetting the target" on page 85).
`-t <name>`	Enables you to specify the target to connect to using its name (as shown in the Target Manager, see "To add a new target" on page 77). The name must be enclosed in quotes if it has spaces in it (e.g. "`my devtool`"). If you enter a name that cannot be identified in the list of targets on the Target Manager, or you do not enter a name, then a dialog appears asking you to select from the available targets when you launch the Debugger. Note that if the name of the target contains spaces you must enclose it in double quotes on the command line.
`-x`	Enables you to specify that the code is executed on the target after load. This is the default behavior when you load

	a `.elf` file. This switch is ignored if you are downloading a binary file.
`-xs`	Specifies that code is executed but stopped at entry point.

Specifying target name

If a target name is specified using the `-t <target>` option then the target name is matched (case sensitively) against the names listed in the ProDG Target Manager for PlayStation 2.

If no target name is specified then the environment variable `PS2TARGET` is searched for. This environment variable can be set manually by inserting a line like the following into your `autoexec.bat` file:

```
SET PS2TARGET=<target_name>
```

If `PS2TARGET` has been set, the target name is taken from this, otherwise if there is only one target available then that will be used, otherwise if a single target is currently connected in the Target Manager then that target will be used.

If all of the above fail, then `PS2Run` will stop with an error.

To set the file server directory

After connecting to the target, the file server directory is set if one is specified using the `-f <path>` option. This means that, when the target is reset, loading of `iopconf` will occur from this directory.

If the reset option is specified using the `-r` option, then the target is reset.

To specify a binary download

A binary download puts an exact image of a file onto the target. This would typically be graphics or level data, not code.

If a binary download is specified using the `-b <address>` option, then the file is downloaded to the specified address. The address is given in hexadecimal either with or without a `0x` prefix, for example:

```
ps2run -b 0x200000
```

and

```
ps2run -b 200000
```

are the same.

The target is not started after downloading a binary file.

Specifying an ELF file

If binary download is not specified then the file is assumed to be an `.elf` file. Checks are performed to make sure it is an `.elf` file. If it is, then it is loaded to the target.

The program will then be executed unless the "no execution" option is specified using the –nx option.

Showing TTY output

Debugging with ps2run is most easily achieved using two instances of the program running in separate MS-DOS sessions.

In one MS-DOS box use the command:

```
ps2run -t <name> -p
```

to intercept all TTY output from your target.

From a second MS-DOS box you can then reset the target and run the program you wish to debug by invoking the command:

```
ps2run -t <name> -r <file>
```

where <file> is the name of the .elf file to be executed.

Disconnecting from the target

After all operations are performed the target can either be left connected so that file serving can take place, or it can be disconnected so that other users can connect to it.

- -nd specifies no disconnect (the default)
- -da always disconnects
- -d ensures that the target is left in the same state as before the ps2run command was executed. Therefore if you were already connected to the target in the Target Manager then you use ps2run, when ps2run has finished the connection will remain.

Quiet mode

ps2run prints a progress report unless the -q option is specified, in which case it only prints anything if an error occurs.

Return values

The return code of the program is 0 if everything worked or 1 if there were any errors.

Chapter 8: **ProDG Debugger 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, watchpoints, 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 IO 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 107.

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. See "ps2dbg command-line switches" on page 99 for details.

`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 a `.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.

ps2dbg command-line switches

The valid switches are:

`-b` specifies that breakpoints are not persisted in the configuration file when the Debugger exits.

`-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.

`-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.

`-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.

`-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` don't disconnect from the target when the Debugger exits. This overrides the default Debugger behavior which 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 which is to leave the target in the state it was in when the Debugger started.

`-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.

`-x` code is to be executed on the target after load.

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".

Connecting to targets

Using the command-line options you can start the Debugger with a .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 a .elf file can be loaded via the toolbar and menus.

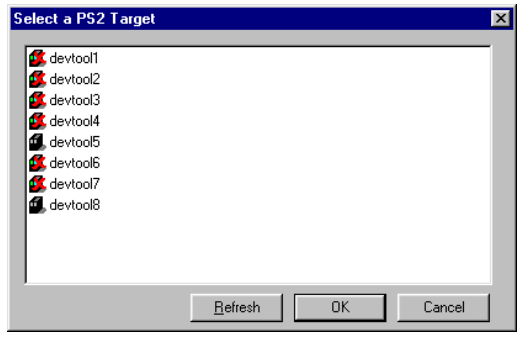
When you start the Debugger the Target Manager is automatically started, and a dialog appears in which any targets that you have set up are listed (depending on your Debugger command line options). You can also change the target that you are working on while using the Debugger using the **Select Target 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 113).

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 which are in use are shown with a red cross, whereas targets which are available are shown in black. Also, targets to which you are already connected have the LEDs hilighted 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.

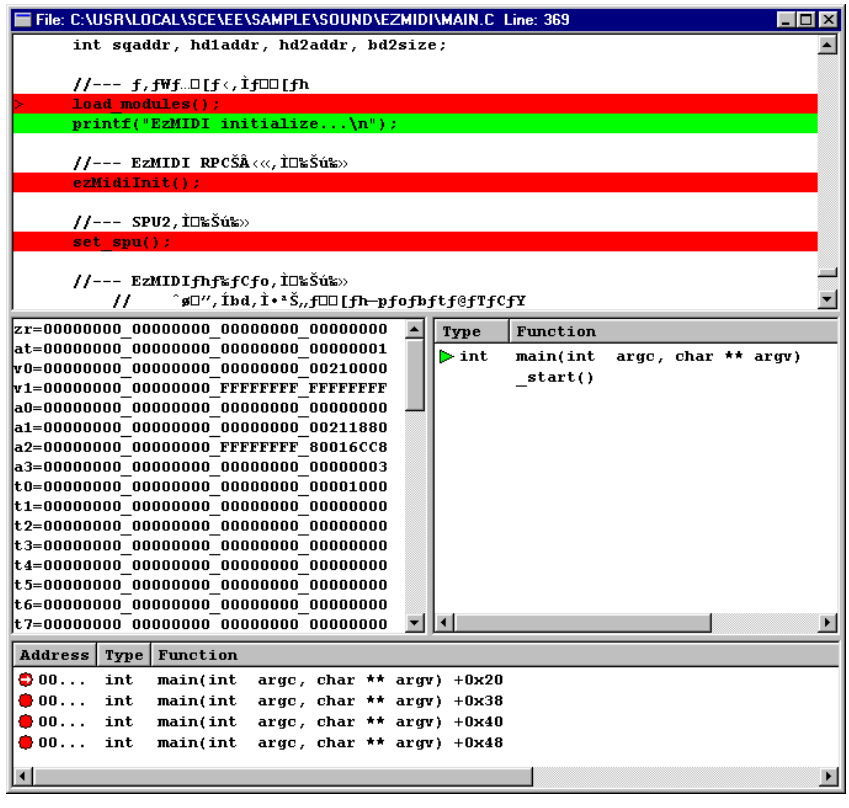
Target and processor status

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:

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

Windows and panes

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. If there is more than one pane in a window it is referred to as a split pane view, as in this example:



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.

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 107).

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 104.

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 106.

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, but for more detailed information see "***Appendix:*** *ProDG Debugger reference*" on page 161.

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.

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.

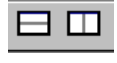
- Press the standard Windows shortcut `<Ctrl+Tab>` and the active window will change to the next ProDG Debugger window in the list.

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.

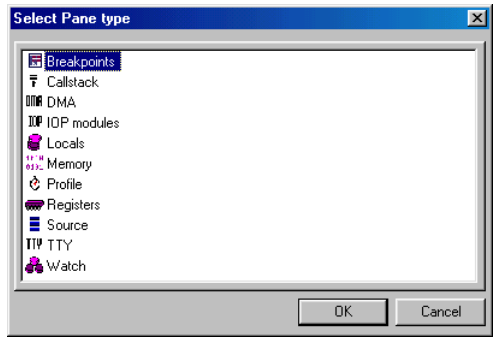
   The pane is split horizontally or vertically, and the new pane will contain an identical pane type to the orginally selected pane.
3. You now need to use the **Change View** option in the shortcut menu to indicate the required pane in the newly created pane site (see "To change the type of a pane" on page 104).

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 listbox 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

Panes are deleted by removing one of the pane borders. For example if you have a split pane view with two panes in it, and the left pane is the current pane, the right pane can be deleted by deleting the right edge of the left pane. To carry out this deletion, do the following:

1. Select the pane that is to expand into the pane to be deleted.
2. Click **Pane > Delete Pane**, and in the submenu that is displayed select the edge to be deleted.
3. Once you have deleted the required edge, the current pane 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 is the active pane 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 listbox 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 listbox 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.

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.

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.

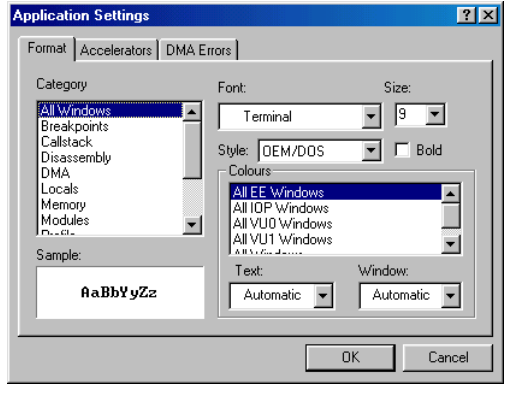
When the Debugger application exits it saves the current configuration in the working directory in a plain text file `dbugps2.ps2`. When the Debugger application starts up it tries to restore the current configuration from this file in the current working directory.

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:



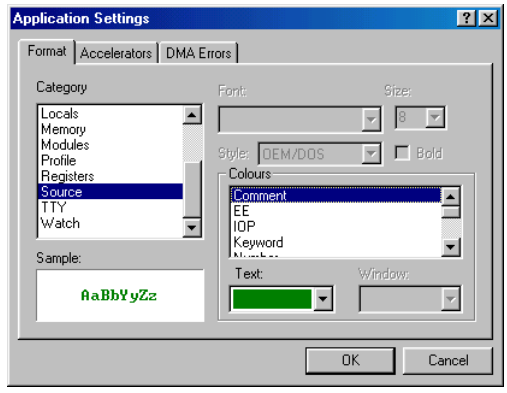
2. The **Category** list contains all of the available pane types. When you select a different pane type the **Colours** list updates to show the CPU types available for that pane. However, for the **Source** category the **Colours** list displays some additional entries that allow you to define syntax coloring (see "To set syntax coloring in source panes" on page 108), and for the **All Windows** category the **Colours** list displays all the available CPUs plus **All Windows**. Choose **All windows** if you wish to define colors and fonts which 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 Colours 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** listbox.
6. Finally, you can choose the color of the text from the **Text** drop-down listbox and of the window background from the **Window** drop-down listbox. 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` which 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 **Colours** 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:



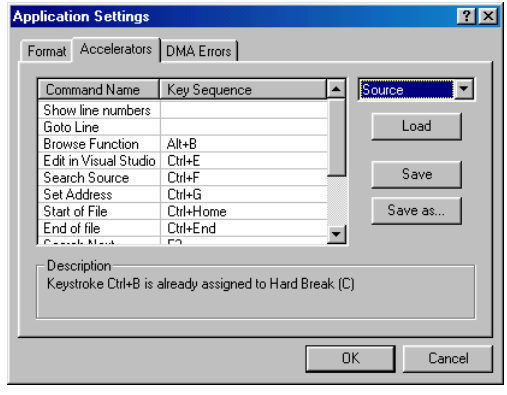
3. In the **Colours** section, you can now set the coloring individually for comments, keywords, numbers, operators, strings and user-defined keywords.

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.

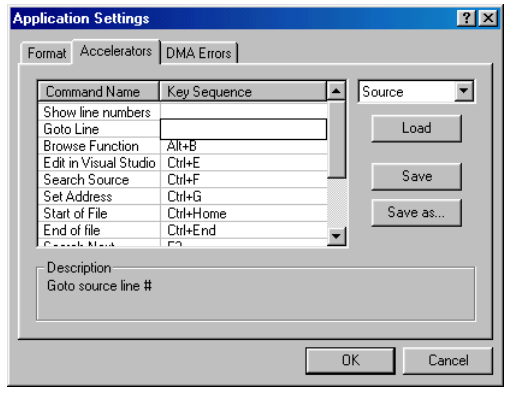
2. In the listbox at the top right, select the pane type for which you wish to set the shortcut menu accelerator keys, for example **Source** pane. There is also an **Application** pane type at the end of the list which allows you to set accelerator keys for ProDG Debugger application menu options. According to the pane type chosen, the list of command names and key sequences will change in the main part of the dialog.



3. The **Description** field provides a slightly more detailed descrition 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 editbox is displayed:



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 editbox.

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.

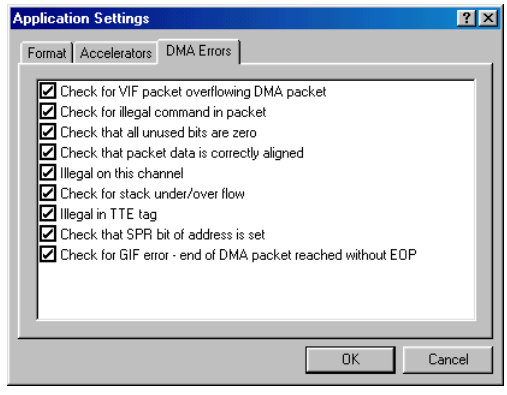
3. Press **Load** to cause a file browser to be displayed. You can then select a named akm file to load accelerator keys from. The default file is called `ps2dbg.akm`.

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 "ProDG Debugger for PlayStation 2 shortcut keys" on page 191.

DMA errors detected

You can configure which DMA errors are reported in the DMA pane (see "DMA pane" on page 185) 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.



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.

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 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 which 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.

Chapter 9: **Debugging your program**

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.

Loading and running ELF files

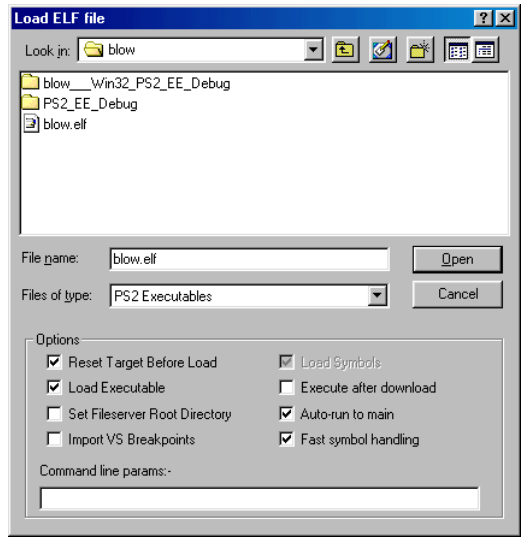
You can load and run `.elf` files on the target directly from the Debugger. You may be doing this after starting the Debugger if you did not specify a `.elf` file on the command line, or after having switched targets or just to load a newly built application.

Note: *You can also load a `.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	to reset the target before your `.elf` file is loaded.
Load Executable	to indicate that you wish the application executable contained in the `.elf` file to be loaded as well as the symbol information.
Set Fileserver Root Directory	to set the application file serving directory to the `.elf` file directory.
Import VS breakpoints	to specify 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.
Execute after Download	to specify 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	to specify that the `.elf` file will be run to the main routine and stopped, once it is loaded on the target.
Command line params	enables you to specify any command-line parameters for your PlayStation 2 application.
Fast symbol handling	assumes all types of the same name in different modules are identical. If you have different types with the same name in different modules then you should not check this option and enable safe symbol loading instead.

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 program 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 "Breakpoints and stepping" on page 116.

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.*

Breakpoints and stepping

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 program on the PlayStation 2" on page 115.
- You can issue a "break 1" assembler instruction if you want to be able to control the placing of breaks at compile time.
- 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.

Setting breakpoints at compile time

You can set a breakpoint by issuing a "break 1" call in assembler as part of your program. This signals to the Debugger that a breakpoint is intended.

Note: *You must click **Run** if you want the Debugger to break on a "break 1" instruction.*

Only the "break 1" instruction does this; other break opcodes do not and it only happens in response to **Run**, not **Step** or similar.

The Debugger first checks that the PC is pointing at the break instruction, and then advances the PC so that you can continue to step through your code.

Setting and viewing breakpoints

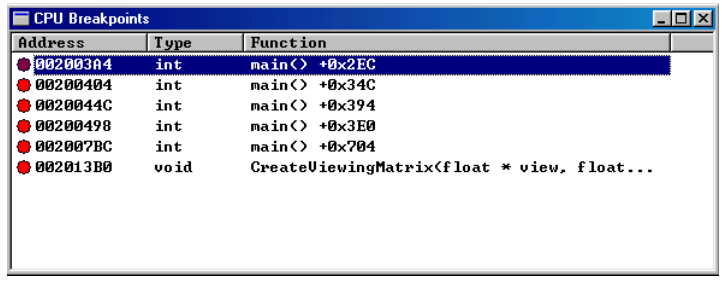
A breakpoint indicates that you want the execution of your program to stop when the program counter reaches the specified line of C, C++ or assembly code.

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 98).

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

To view breakpoint information

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. For more information see "Breakpoints pane" on page 179.



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 break-pointed 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.

Single-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 169, and "Source pane" on page 172. 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 which 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.

Viewing program 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).

For more information on using the source pane, see "Source pane" on page 172.

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.*

To navigate through your program source

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

- 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 listbox 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 first line of the source file, use the Start of file accelerator key:

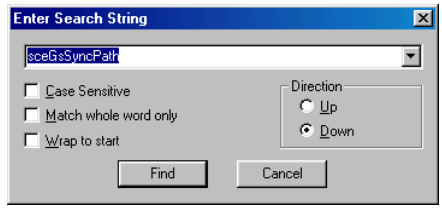
1. From the **Settings** menu option, select **Options** to display the Application Setting dialog. Make sure that the **Accelerators** tab is selected.
2. In the listbox at the top right, select **Source**.
3. Scroll down the **Command Name** list until the accelerator key setting for **End of file** 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.

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.

Case Sensitive	this checkbox will cause a match to fail unless the matched string is case-identical to the search string.
Match whole word only	this checkbox will cause a match to fail unless the matched string is a whole word, i.e. matched substrings will not be detected.
Wrap to start	this checkbox 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 radiobuttons determine the direction of the search, either upwards or downwards from the cursor position.

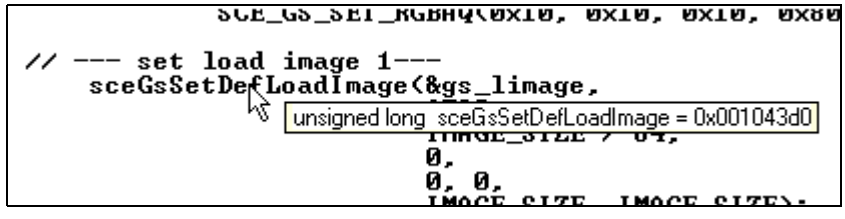
Note: *These options are persistent only for the lifetime of the application, i.e. they are not stored in the configuration file on shutdown.*

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.

To evaluate an expression using a floating ToolTip

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:



For more information on using the source pane, see "Source pane" on page 172.

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 Souce 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.*

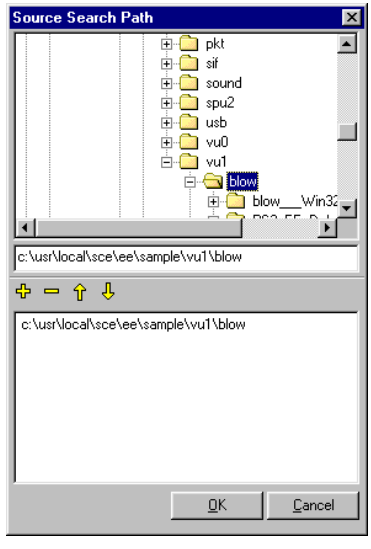
For more information on using the source pane, see "Source pane" on page 172.

To add directories to the source search paths

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.

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.

Viewing program 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.

For more information on using the disassembly pane, see "Disassembly pane" on page 169.

Viewing and modifying registers and memory

You can view all of the DTL-T10000 registers and memory in the registers and memory panes respectively. You can also modify the values of any register or memory address.

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 which shows you all the registers and their current values.

For more information on using the registers pane, see "Registers pane" on page 164.

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.

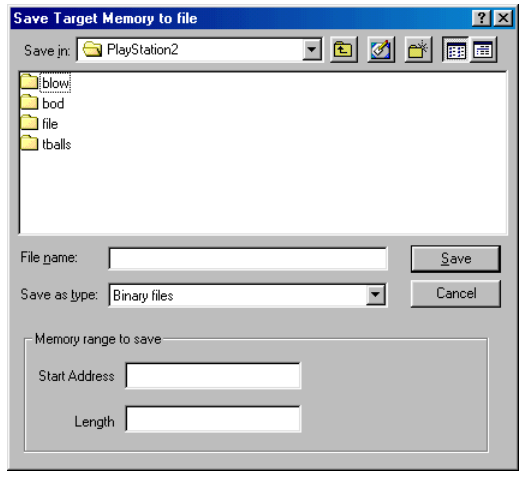
For more information on using the memory pane, see "Memory pane" on page 166.

Saving and downloading target memory

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

To save part of the target memory to a binary file

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

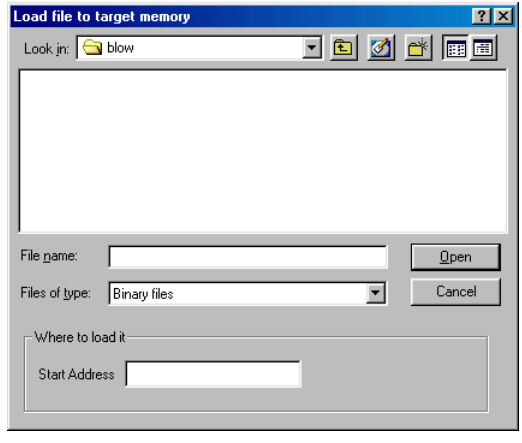


2. Enter the **Start Address** and the **Length** of the memory range in the fields provided for this purpose. You can either use `0x` as a prefix or if you leave this out the fields will assume that you are entering hexadecimal values.

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**.

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 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 local and watched variables

You can view the local variables of the function that the program counter is positioned in, in the local variables pane. 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 view the 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 which shows you all the variables in the current function and their current values.

For more information on using the locals pane, see "Locals pane" on page 176.

To create a new watch pane and add watches

1. 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.

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

3. 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 107).*

For more information on using the watch pane, see "Watch pane" on page 177.

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 listbox at the top right, select **Watch**.

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.

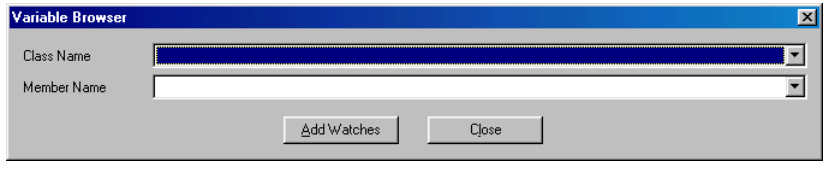
The quick way to expand the array or structure is to double-click on it in the locals or watch view. In addition you can quickly collapse an expanded variable by double-clicking it again.

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.

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 which shows you current state of the program stack.

For more information on using the call stack pane, see "Call Stack pane" on page 181.

Viewing TTY output

You can view TTY output from your program in a TTY pane. This displays debug and other `printf`-type output.

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 which can then be configured to filter certain types of TTY stream, using shortcut menu options.

For more information on using the TTY pane, see "TTY pane" on page 182.

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.

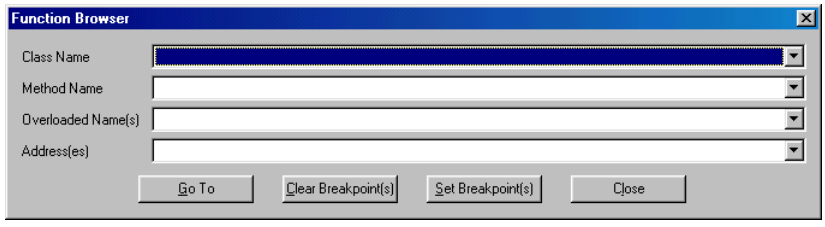
Expressions can be used wherever a Debugger dialog asks you for an address, expression or value. The watch and locals pane refer to it constantly, so it is essential that you understand its operation in order to exploit its full power.

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 or disassembly panes shortcut menus .



Name demangling

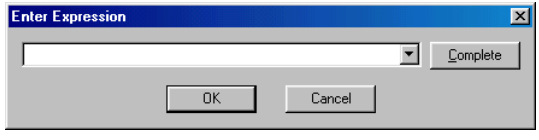
You can choose to display "demangled" C++ function names at various places in the Debugger, for example in a breakpoints or call stack frame.

To demangle C++ function names

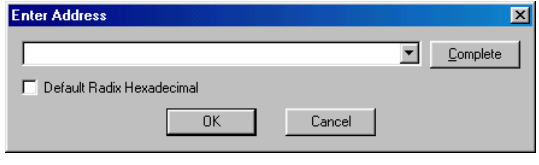
To demangle C++ function names this you simply need to copy the SN Systems file `demangle.dll` to your ProDG for PlayStation 2 program directory. When the Debugger detects this DLL it will automatically convert "mangled" function names into "demangled" ones.

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.

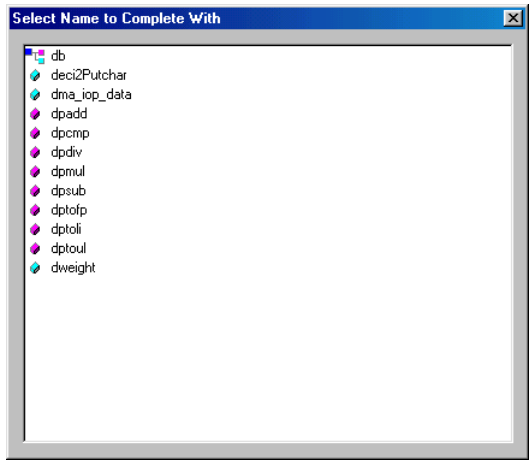
Name completion

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 listbox.

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:



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 128).

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 129).

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

[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 signifigant PlayStation 2 performance bottlenecks.

The profile pane collects and analyses data produced by a small efficient interrupt handler which 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 which 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.

Building for EE profiling

This is what do you need to do to make use of the EE profile pane:

1. Include the header file `libsn.h` in your code.
2. Add one function call to your program to turn on profiling.
3. Link with `libsn.a`.

That's all there is to it – the Debugger will do the rest. Your PlayStation 2 program should run at pretty much full speed. If you are using SN's DMA debugging then you are probably already linking `libsn.a` to replace a few problematic

PlayStation 2 library functions. For that reason it is a good idea to make `libsn.a` the first library in the library list in your `makefile`.

How to use profiling in your application

This example is based upon a simple PlayStation 2 sample program but the theory should be easily applied to any PlayStation 2 EE application.

1. Make sure you have put up-to-date versions of `libsn.a` and `libsn.h` in your EE library and include directories respectively. Put the `SNProfil.irx` file somewhere convenient to load it at runtime (we recommend you put it in the C:/usr/local/sce/iop/modules directory along with your usual IOP modules).

2. Edit your main source file, i.e. the one which contains your `main()` function, to #include `libsn.h`. You can make this the first include file if you wish as it has no dependencies on any SCE headers.

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

3. Edit your `main()` function to provide a fixed-location profile data buffer and pass it in a call to `snProfInit()` to start the profile data collection. Note the call to `snDebugInit()` which is not strictly necessary for profiling but it installs other SN debug extensions which you may find useful.

   ```
   main()
   {
   static u_long128 profdata[2048]; // quadword aligned, can be 2K
                                    // to 64K bytes
   snDebugInit();
   sceSifInitRpc(0);
   // Load the SNProfil module
   if(sceSifLoadModule("host0:/usr/local/sce/iop/modules/SNProfil.ir
   x", 0, NULL) < 0)
   {
   printf("Can't load SNProfil module\n");
   exit(-1);
   }

   if(snProfInit(_4KHZ, profdata, sizeof(profdata)) != 0)
   printf("Profiler init failed\n"); // see SN_PRF… in LIBSN.H
   // rest of user code follows on from here...
   }
   ```

4. Edit your `makefile` to link with `libsn.a` before other libraries:

   ```
   LIBS = $(LIBDIR)/libsn.a \       # add this line
   $(LIBDIR)/libgraph.a \
   $(LIBDIR)/libdma.a \
   $(LIBDIR)/libdev.a \
   $(LIBDIR)/libpkt.a \
   $(LIBDIR)/libpad.a \
   $(LIBDIR)/libvu0.a
   ```

5. It is not really necessary for profiling but whilst you are editing your `makefile` perhaps this is a good time to check that you are using the SN Systems' `ps2dvpas` VU assembler (to obtain better VU debug information).

Also to make debugging generally easier you should turn optimization off by removing –O2 from CFLAGS. Try these three lines in your `makefile`:

```
DVPASM = ps2dvpas
CFLAGS = -g -Wall -Werror -Wa,-al -fno-common
DVPASMFLAGS = -g
```

6. Now build your program.
7. Run the Debugger, load up the ELF and start it running.

   If you open a profile pane in the Debugger you should be able to see real-time profile results updating as the program runs. See "Profile pane" on page 188 for details on how to use the profile pane.

Things to look for

If your profile display shows signifigant 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 which 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 signifigant 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`"].

EE profiler API

There are a few simple functions available in `libsn.a` to allow you to control profiler data collection.

int snProfInit(UINT32 interval, void buffstart, int bufflen);*

```
UINT32 interval;  // Sample interval in CPU clocks (@300MHz)

//(LIBSN.H defines constants you can use _1KHZ, 2KHZ, _4KHZ,
_10KHZ, _20KHZ)

void* buffstart;  // Start address of profile sample buffer

int bufflen;      // length of the above sample buffer (in
bytes)
```

Remarks: This is the function you must call in your application to initialise the profiler functionality. This will hook the timer interrupt, setup the profile buffer, and start the timer running. Note that the buffer will be written to from kernel code from this point on so it must be at a fixed address and always available.

Example:

```
static u_long128 ProfData[8192]// quadword aligned, can be
                               // 2K to 64K bytes

snProfinit( _4KHZ, ProfData, sizeof( ProfData ) );
```

void snProfSetInterval(UINT32 interval);

```
UINT32 interval;  // Sample interval in CPU clocks (@300MHz)

//(LIBSN.H defines constants you can use _1KHZ, 2KHZ, _4KHZ,
_10KHZ, _20KHZ)
```

Remarks: This can be called at any time after snProfInit() to change the sample interval. The header file defines a few typically useful rates but you can set any interval you like. The interval is specified in CPU clocks so:-

```
samples_per_second = 300000000/interval
```

or to calculate the interval required for a particular sample frequency:-

```
interval = 300000000/samples_per_second
```

Example:

```
snProfSetInterval( 300000000 / 40000 );   // set 40KHz
sample rate
```

void snProfSetRange(int profmask, void startpc, void* endpc);*

```
int profmask;     // profile mask value – samples only valid
when
// (flags & profmask) != 0

void* startpc;    // samples below this location are
discarded

void* endpc;      // samples above this location are
discarded
```

Remarks: The default value for both mask and flags is 1. The default PC range is 0 to 0xFFFFFFFF. So unless you change these all samples will be collected and made available to the Debugger. By changing these values you can select which bits of your code will actually be profiled and which will be ignored. Note that if a signifigant proportion of samples are discarded because of this filter then it will take that bit longer for the sample buffer to fill up, therefore your Debugger profile pane will update more slowly. If you wish to compensate for this you can decrease your buffer size or increase your sample rate. If you filter out all of the samples, i.e. no code in the PC range or flag values you specify is being called at all, then the profile window will stop updating altogether.

Example:

```
// set profiling to accept all flag values of samples within
this module.

snProfSetRange(-1, firstfuncinthismodule,
firstfuncinnextmodule);
```

See “Targeting profiling to specific situations” on page 136 for more detail of how to use these functions to selectively profile regions of your program or pre-set conditions.

extern int snProfSetFlagValue(int value);

```
int value; // profile flags will be set to this absolute
value
```

Remarks: The profiler flags can be viewed as 32 different boolean conditions. This function allows you to set the current state of your program to something which can be selectively filtered by the profiler. By making use of this feature you can then use features of the Debugger profiler pane to select just particular parts of your program ie. "show me profile results from just the parts of my program where flag 4 is set".

Note: *Think of these as logical binary flags, not numbers. Because the profiler uses a user specified mask to select different bits of your code a mask value of 3 will accept profile flag values with either bit 0 or bit 1 set and will reject any sample which occurs when bit 0 and 1 are both zero*

i.e. If mask = 3 then flag values of 1,2, or 3, or 5 etc will be accepted but flag values of 4, 8 etc will be rejected.

See "Targeting profiling to specific situations" on page 136 for more detail of how to use these functions to selectively profile regions of your program or pre-set conditions.

```
int snProfSetFlags(int flags);

int flags; // flag bits to set (32 bits for 32
'conditions')
```

Remarks: The profiler flags can be viewed as 32 different boolean conditions. This function allows you to set the current state of your program to something which can be selectively filtered by the profiler. This function can only set additional bits, it will never clear them. To set the entire flags word to a definite value you should use snProfSetFlagValue() instead.

See "Targeting profiling to specific situations" on page 136 for more detail of how to use these functions to selectively profile regions of your program or pre-set conditions.

```
int snProfClrFlags(int flags);

int flags; // flag bits to clear (32 bits for 32
"conditions")
```

Remarks: The profiler flags can be viewed as 32 different boolean conditions. This function allows you to set the current state of your program to something which can be selectively filtered by the profiler. This function can only clear additional bits, it will never set them. Bits which are set to 1 in the flags parameter will be cleared in the flags word of the profiler's control block. To set the entire flags word to a definite value you should use `snProfSetFlagValue()` instead.

See "Targeting profiling to specific situations" on page 136 for more detail of how to use these functions to selectively profile regions of your program or pre-set conditions.

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 (ie 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 from that point on so if the target is halted a breakpoint so no more profile data is being collected then changing the flag value will not affect the display of sample data already collected. A later release of the profiler lib and Debugger support may allow you to retrospectively apply the filtering to already collected sample set even if the target is not currently profiling.*

Known problems

There are currently some problems with PlayStation 2 SIF useage 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 how to debug an IOP module

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. In order to debug an IOP module there must be a calling EE application which loads the module on request.

This section describes the different ways to debug an IOP module. 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 application load the module or maybe you wish to debug an already loaded module. In which case you can debug it by setting a breakpoint at the entry point of the IOP code. When you do this the IOP will stop each time a command is sent to the IOP, and you can continue debugging from the entry point of the IOP module.

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 142).

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.

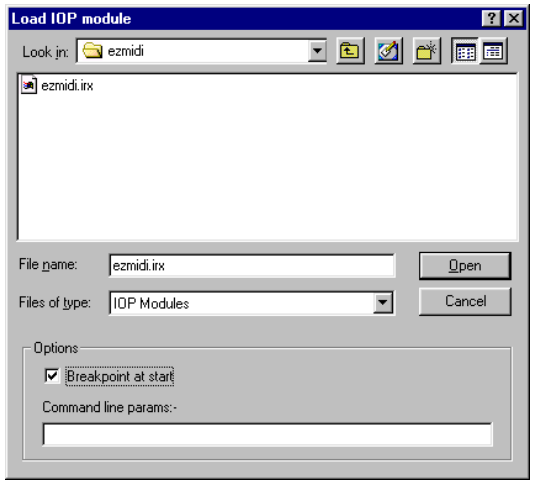
The main reason to load an `.irx` file during debugging is so that you can debug it directly from the point it is loaded onto the IOP processor. If you do this then you will need to select the **Breakpoint at Start** option in the Load IOP Module dialog.

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.



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.

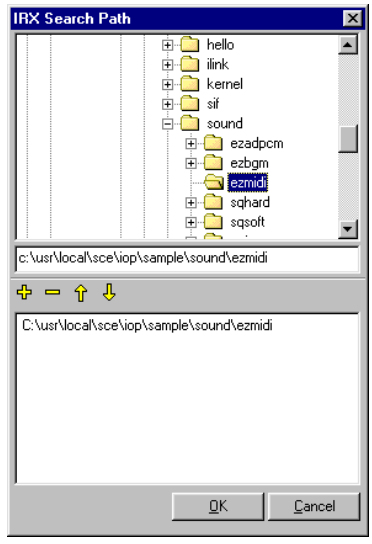
Setting the IRX search path

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 103). 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).

Listing IOP modules currently loaded

A list of currently loaded IOP modules can be viewed by displaying the IOP Modules pane.

To view the IOP modules

If you wish to view the IOP modules:

1. Click **New Window > IOP modules** in the **Window** menu or click the **IOP modules view** button in the toolbar.
2. A new window containing an IOP modules 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.

If you double click 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 module which the PC is currently in is shown with a '>' marker.

See "IOP Modules pane" on page 183 for further information.

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.

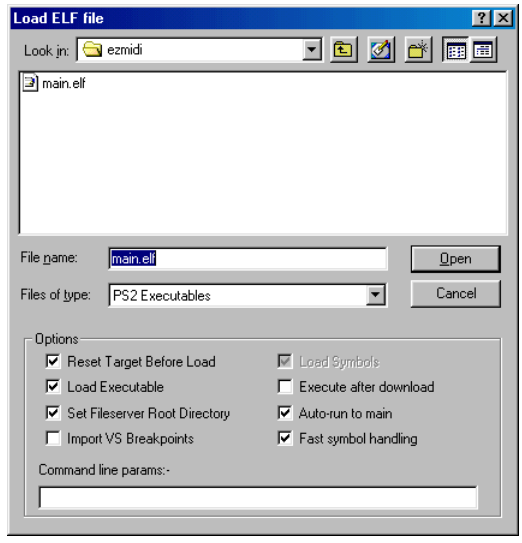
Debugging IOP modules in the Sony ezmidi sample

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. This means adding `-G0` and `-g` to the `CFLAGS` variable in your `makefile`, e.g.,

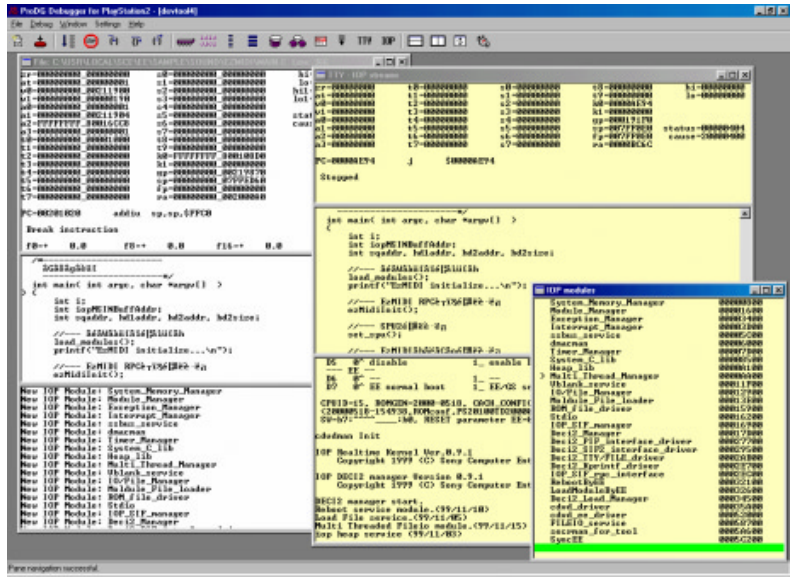
```
CFLAGS = $(INCDIR) -I. -Wall -G0 -g
```

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:



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:



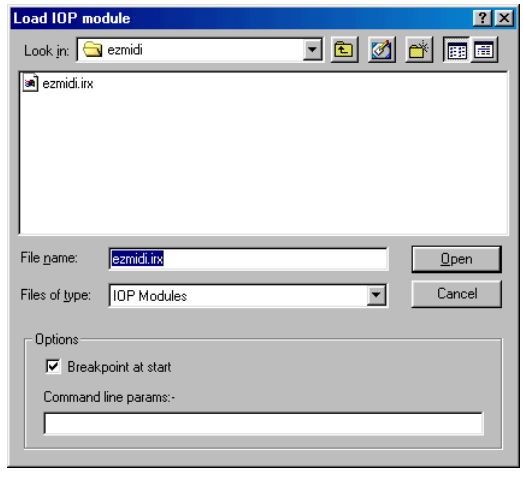
5. Now make the main CPU source pane the active pane and start to step through your code executing on the EE. Step into the `load_modules` function.

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 start-up. If you wish to see how to debug the module from its entry point from the EE application step through the code to load the `ezmidi` module and go to step 12.

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.



   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 and if you loaded the `ezmidi` module manually then you will need to move the program counter past the load line of code 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. 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.
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 which 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.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 12: **VU and DMA debugging**

Overview of 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. At the start of the function `main()` in your program call the `snDebugInit()` function and check the result is not `0`. e.g.

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

This will install debug extensions which will enable the Debugger to access the accumulator registers in the EE, VU0 and VU1 floating point units. It should now be possible to see these register values in the register windows.

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).
6. You must ensure that you use the GNU linker `ld` if you plan to do VU debugging.

Creating new VU Debugger windows

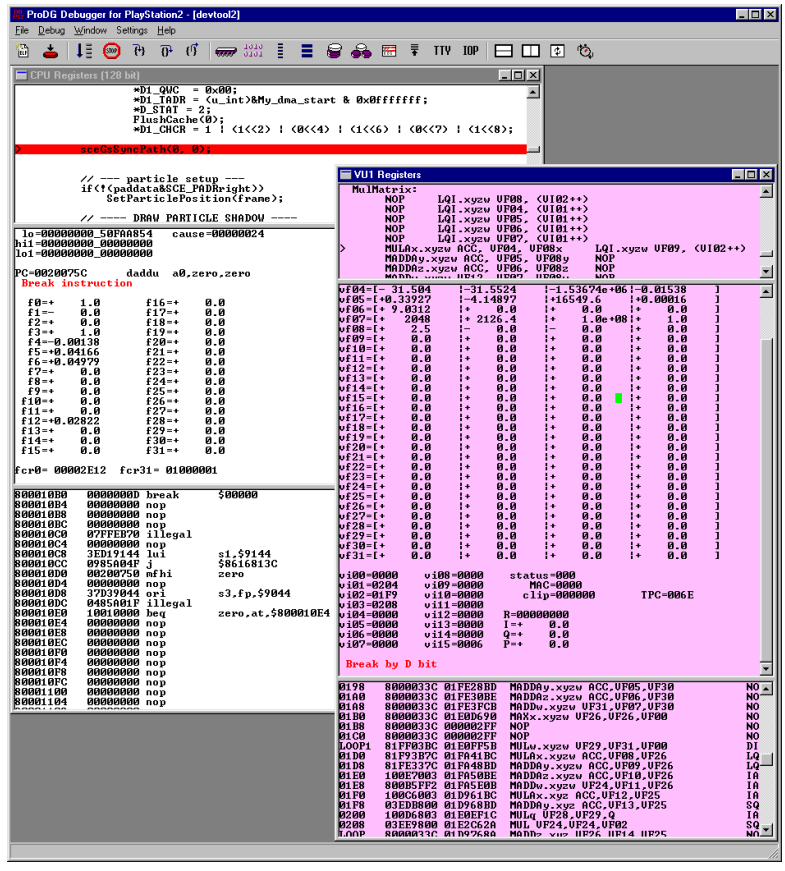
VU debugging takes place by creating a split-pane container with separate Debugger panes, just as you did for EE debugging (see "Creating new Debugger windows" on page 103). Just as with IOP debugging, this would normally be set up in addition to an EE split-pane container.

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:



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

Overview of how to debug a VU module

This section describes how to debug a VU module.

1. Set the D bit on the first VU instruction in your code. 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 154 for details.

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.

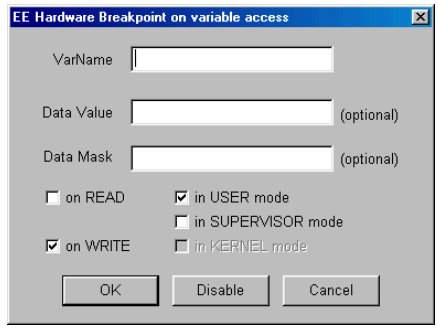
You can further refine when a hardware break is triggered, by specifying a mask or data value that must be present to cause the break to occur (rather than it being triggered each time the variable or address is read from or written to).

Note: *Currently there is an anomaly in the PlayStation 2 kernel which means that if you set a hardware breakpoint and leave it enabled when you close the Debugger, or load another application on the target, the next debug session may behave strangely (the kernel or application start up code will halt or the kernel may crash with a TLB exception). For this reason it is recommended that you disable hardware breakpoints before closing the Debugger or loading the new application* `.elf` *file.*

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:



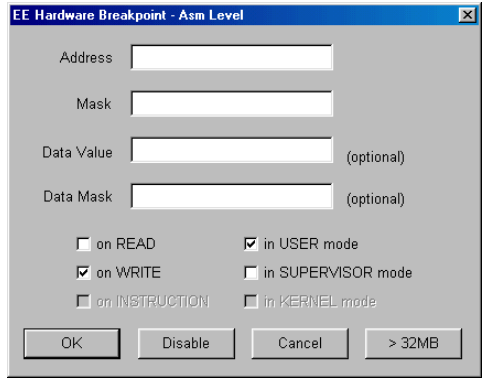
Hardware breakpoint set up dialog for C mode

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.

4. 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.

5. 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.

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

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

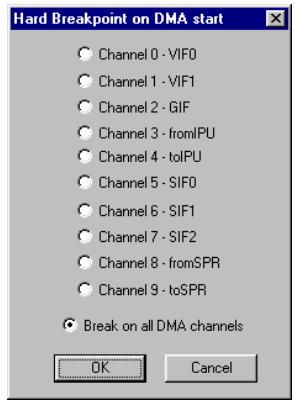
8. Click **OK** to set the hardware breakpoint and enable it.

   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.

> **Note:** *You will also need to build your program for 32MB or it will trip over its own stack and may kill the PlayStation 2 kernel.*

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:



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 150) or assembly level (see "To set an assembly level hardware breakpoint" on page 151) 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.

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 152).

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:

1. Click **New Window > DMA** in the **Window** menu or click the **DMA channel view** toolbar button.
2. 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.

For more information on using the DMA pane, see "DMA pane" on page 185.

Overview of how to debug 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 152).
2. Create a DMA pane; see "To view a DMA channel" on page 154.
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.

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 diassemblies 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.

Detecting DMA errors

The range of DMA errors detected by the Debugger can be configured from the Application Settings dialog. See "DMA errors detected" on page 111 for details.

From the DMA pane, two shortcut menu options are available for detecting DMA chain errors:

Parse DMA list for errors will progress 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.

The following table lists the DMA errors detected by the DMA error parser:

Error	Interpretation
`size`	VIF/MPG/UNPACK data overflows DMA packet
`illegal`	Packet is not a legal command
`bits`	Packet has unused bits that are not zero
`align`	VIF code is not correctly aligned (DIRECT require 128 bit align and MPG requires 64 bit alignment)
`channel`	Operation is not legal on this DMA channel
`stack`	Stack overflow or underflow (VIF stack is limited to 2 levels)
`illegal in tag`	This op cannot fit into the tag with TTE mode enabled.
`(SPR=0)`	SPR bit of address is not set but address is scratchpad
`no GS EOP`	End of packet reached without seeing GS EOP

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 "bits" 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.

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 walkthough 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
   ```

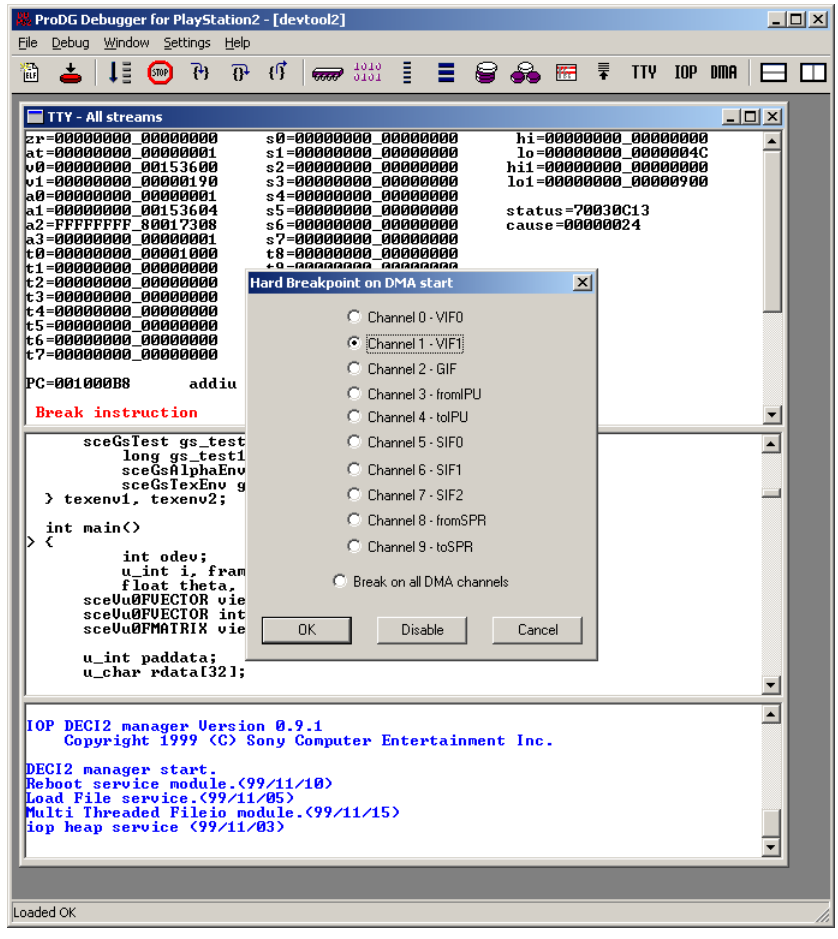
3. Delete all object and `.elf` files in that directory to force a full build.

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 113) to reset the target, load the `blow.elf` code and symbols, run to main. OR specify all that on the command line:

   ```
   ps2dbg -ref blow.elf
   ```

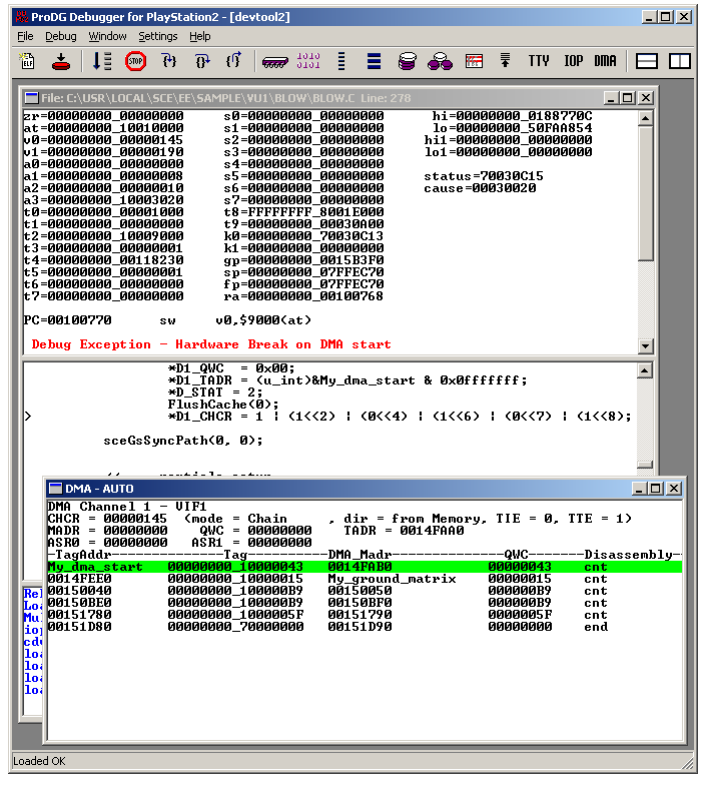
6. From the **Debug** menu, select **EE DMA Hardware Break** to bring up the Hard Breakpoint on DMA start dialog (see "To set a DMA channel hardware breakpoint" on page 152) and select channel 1 (VIF1) like this:



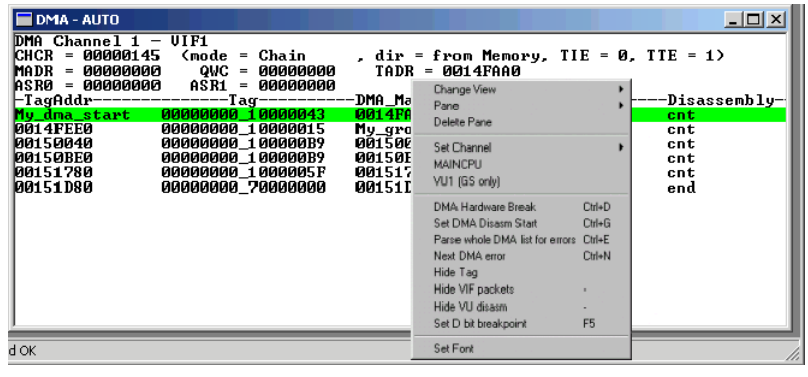
7. Start the target PlayStation 2 running. It will stop at the next attempt to trigger the VIF1 DMA channel. Note that the application has actually halted on the instruction that will trigger the DMA, i.e. the DMA has not happened yet.

   If you now open a new DMA pane - see "DMA pane" on page 185 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:



8. The context menu for the DMA pane provides several useful options.



Set Channel	allows you to set the DMA pane to monitor a particular channel or to AUTO which just monitors the last channel which triggered a DMA Hard Break.
Main CPU	displays DMA chains in EE RAM.
VU1 (GS only)	displays GS packets in VU1 RAM.
DMA Hardware Break	brings up the DMA hard break dialog you have already seen.

Set DMA Disasm Start allows you to point a DMA pane at an arbitrary address so, for example, you can examine an incomplete DMA chain under construction.

Parse whole DMA list for errors will progress 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.

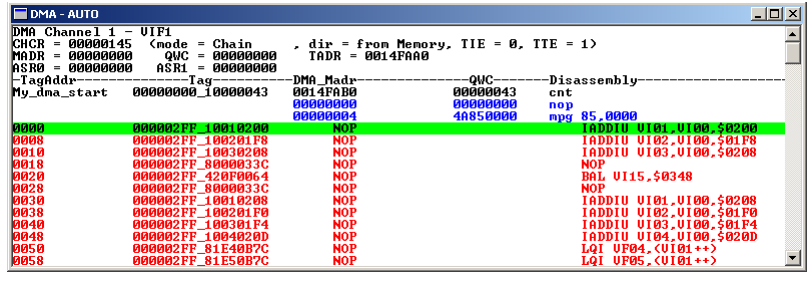
Next DMA error if the above parse found more than one error this will take you to the next error.

Show Tag when checked this option shows the DMA tag display in the left hand two columns.

Show VIF packets when checked expands the display one more level to include disassembly of the VIF codes within the DMA packets.

Show VU disasm when checked expands the display two levels to include VU disassembly of code within MPG VIF packets.

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:



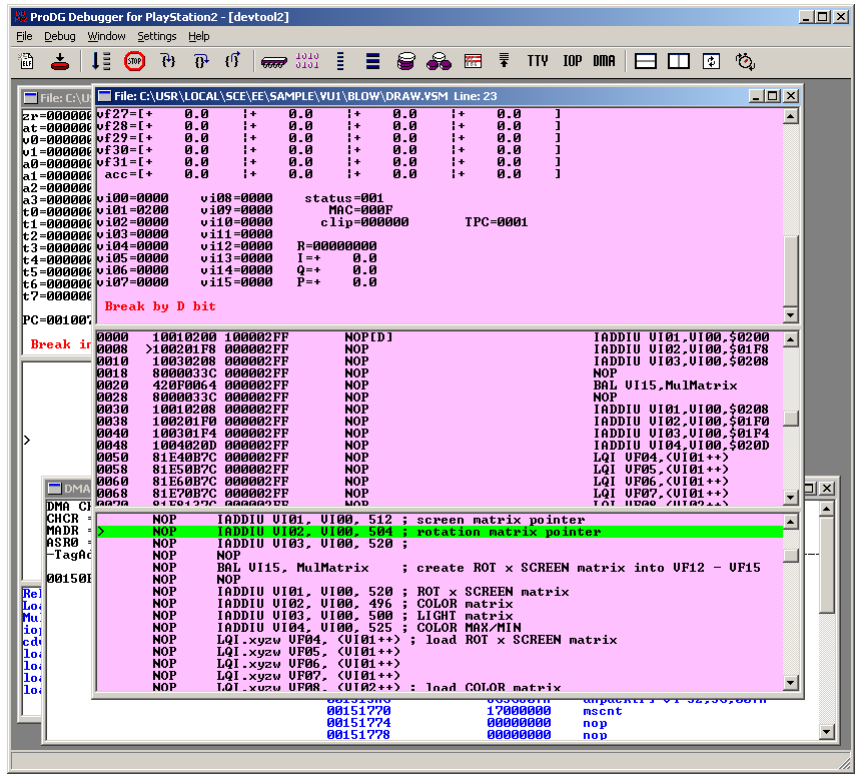
9. Select the first line of VU microcode (as above) and right-click to bring back the context menu and select **Set D bit breakpoint** – or just press the breakpoint button set-break accelerator key. This will set the [D] bit of that instruction so that when it executes it will cause the VU to halt.

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 which 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 which you just set. If you select the VU window now you will see something like this:



12. 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.*

13. 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.

14. 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.

15. 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).

Appendix: ***ProDG Debugger reference***

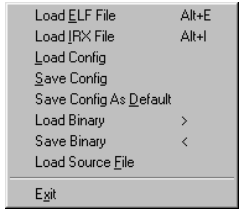
Overview

This appendix contains reference information for the ProDG Debugger menus, as well as for each of the different panes that are available in the Debugger to view target information. Each section contains information on the purpose of a pane and the commands available in the pane shortcut menu.

The panes are documented in the order in which they appear in the Debugger user interface. For more information on using panes in the Debugger see "Creating new Debugger windows" on page 103.

The last section in this appendix contains a keyboard shortcuts reference table.

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 a `.irx` file to the PlayStation 2 IOP processor. A dialog is presented to allow you to select a file and set file load options.

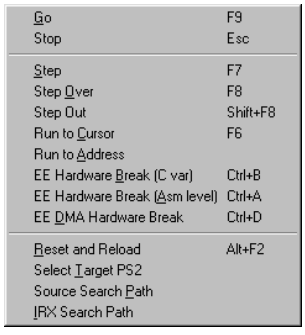
Load Config reload the Debugger configuration file from the current working directory and overwrites your current configuration.

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`

	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 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.

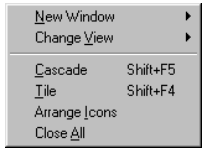
Source Search Path allows you to set or modify the search path for finding source files.

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 using this submenu you can change the type of the current pane to another pane type.

Cascade arranges all windows in standard Windows cascade format (overlapping).

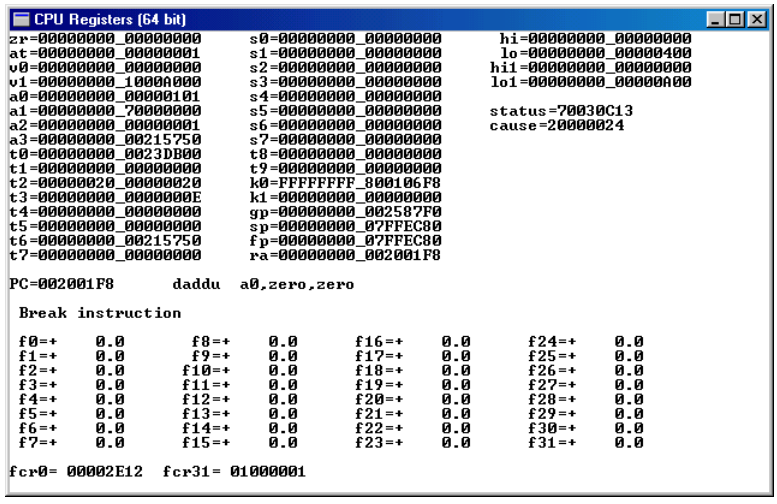
Tile arranges all Debugger windows in standard Windows tile format (non-overlapping).

Arrange Icons

Close All closes all open windows.

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).



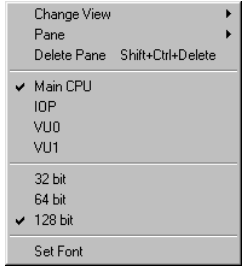
The registers pane is updated automatically whenever the target stops. If you click in the registers pane, the cursor will only be displayed on elements that can be modified.

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.

Shortcut menu



Change View using this submenu you can change the type of the

current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

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.

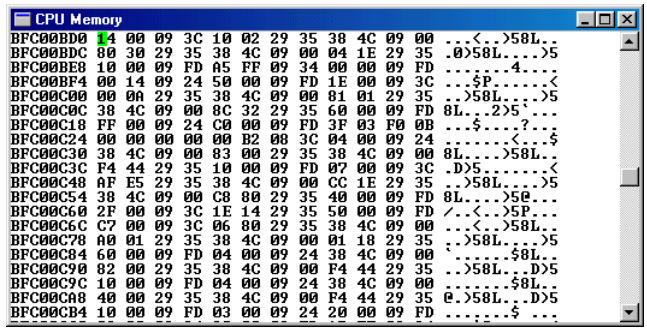
32 bit, 64 bit, 128 bit these options enable you to change the way in which the registers are displayed in the registers pane. The checked option is the one current;y in effect. This option only applies to the main CPU registers pane.

Set Font enables you to change the display font for information in the current registers pane only. If you wish to change the display font globally for all registers panes use the Application Settings dialog (see "Pane colors and fonts" on page 107).

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).

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 main CPU, IOP, Vector unit 0 or Vector unit 1 (by selecting the required unit in the shortcut menu).



The memory information is displayed in three distinct columns:

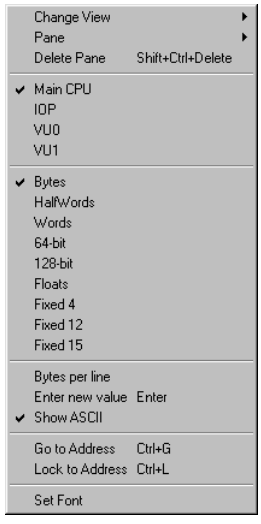
Address — This first column contains the address in memory on the target represented by the rest of the line. It is in hexadecimal format, and by default is 32 bits.

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->`.

Shortcut menu



Change View using this submenu you can change the type of the current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

Main CPU changes to show the memory on the main CPU. This is reflected in the window title. The background color of the memory pane is reset to white.

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.

Bytes changes the memory pane so that memory is displayed in bytes.

HalfWords changes the memory pane so that the memory is displayed in half words.

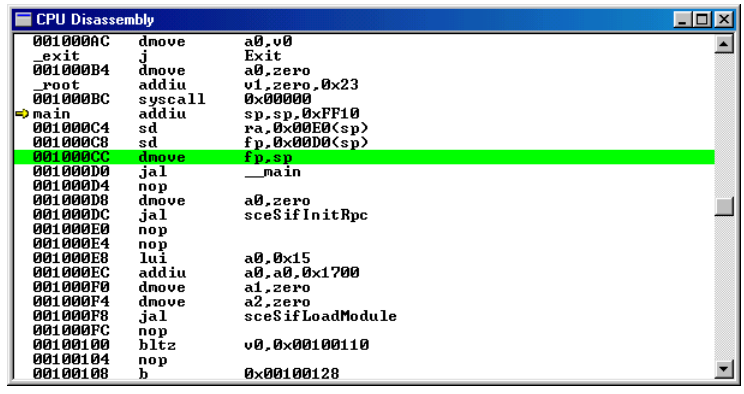
Words changes the memory pane so that the memory is displayed in words.

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

64-bit, 128-bit	changes the memory pane so that the memory is displayed as 64-bit or 128-bit integer values.
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.
Show ASCII	when checked, the third column shows the memory data in column two represented as ASCII characters.
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.
Lock to Address	enables you to specify an address expression which 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.
Set Font	enables you to change the display font for information in the current memory pane only. If you wish to change the display font globally for all memory panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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.



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

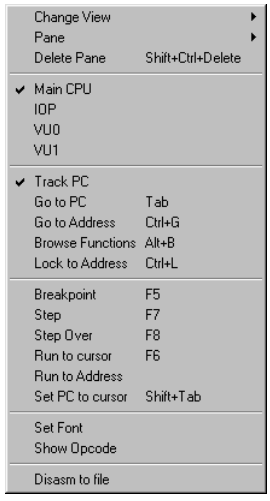
- the first column contains the address or label
- the second column displays the opcode in hexadecimal
- the third and fourth columns show the disassembled program instruction

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.

Shortcut menu



Change View using this submenu you can change the type of the current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

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.

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 which 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 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.

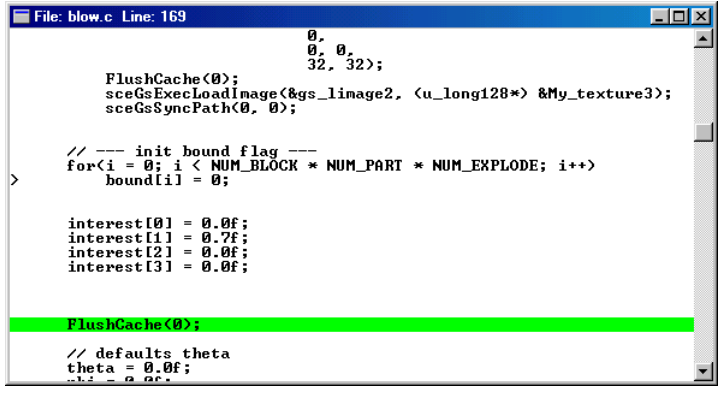
Set Font — enables you to change the display font for information in the current disassembly pane only. If you wish to change the display font globally for all disassembly panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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`.

Source pane

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



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 which the Debugger can search to locate source files using the **Source Search Path** option in the **Debug** menu.

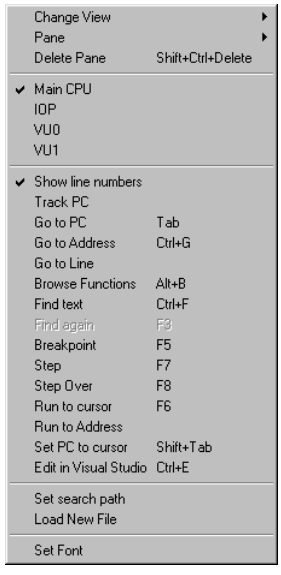
When you first open this pane it will normally show the current position of the program counter in your source file. However, if the program counter cannot be viewed in source then the pane shows a message saying that no source is available.

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.

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.

Shortcut menu



Change View using this submenu you can change the type of the current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

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.

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.

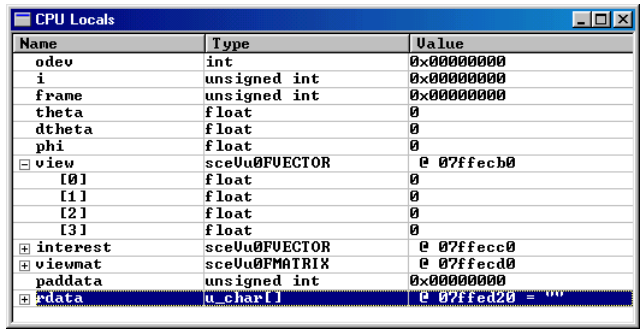
Set search path enables you to add directories to the search path for your application source files. If you add more than one path it must be separated by a semi-colon. This is useful if you do not have the source files stored in the same position as when you built the `.elf` file, or if you didn't build 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 which will always open the selected source file in a new source pane.

Set Font enables you to change the display font for information in the current source pane only. If you wish to change the display font globally for all source panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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.



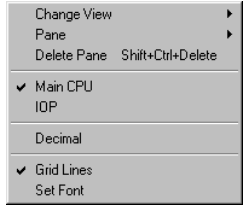
Each variable on the target is displayed with its current data **Type** and **Value**.

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.

Shortcut menu



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

Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected 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.

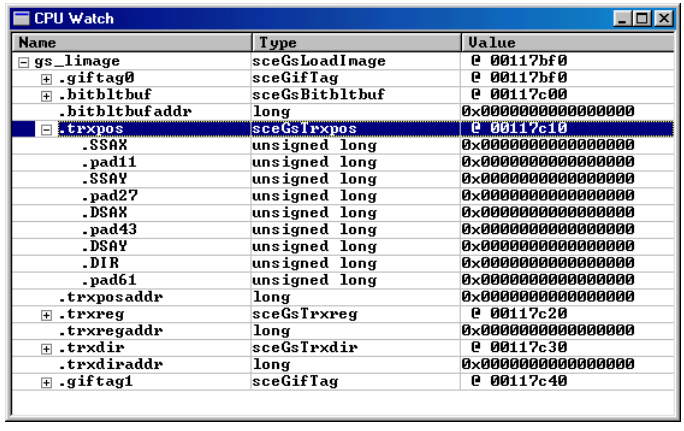
Decimal / Hex enables you to toggle the display of the currently selected local variable between decimal and hexadecimal.

Grid Lines when checked, displays a table grid separating rows and columns in the locals pane, to improve readability.

Set Font enables you to change the display font for information in the current locals pane only. If you wish to change the display font globally for all locals panes use the Application Settings dialog (see "Pane colors and fonts" on page 107).

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.



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.

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.*

Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected 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 which 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.

Decimal / Hex enables you to toggle the display of the currently selected watch variable between decimal and hexadecimal.

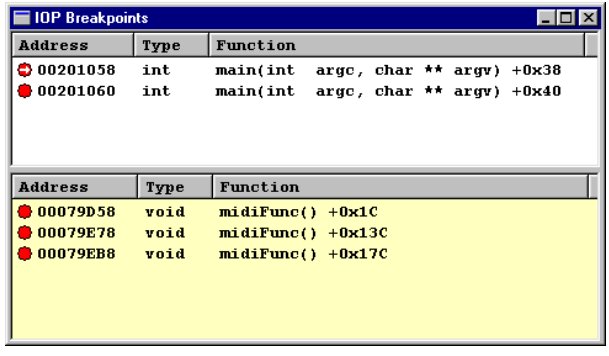
Grid Lines when checked, displays a table grid separating rows and columns in the watch pane, to improve readability.

Set Font enables you to change the display font for information in the current watch pane only. If you wish to change the display font globally for all watch panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

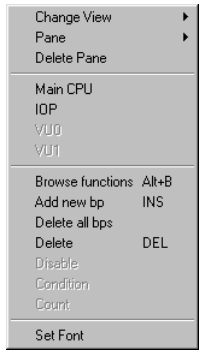
Breakpoints pane

The breakpoints pane enables you to see at a glance all the breakpoints set in your code. 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.



Shortcut menu



Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected pane.

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.

Browse functions — causes a Function Browser dialog to be displayed, in which you can select from dropdown listboxes 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.

Delete all bps — causes all breakpoints to be removed for the selected processor.

Delete — causes the selected breakpoint to be deleted.

Disable / Enable — causes the selected breakpoint to be disabled / enabled.

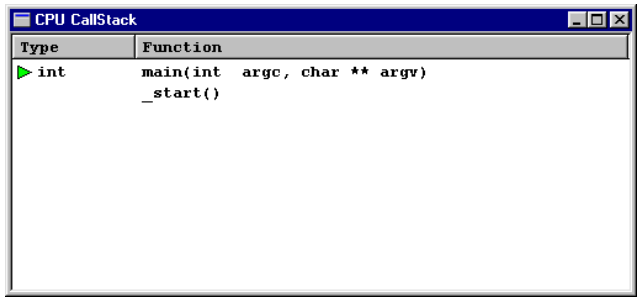
Set Font — enables you to change the display font for information in the current breakpoints pane only. If you wish to change the display font globally for all breakpoints panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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.

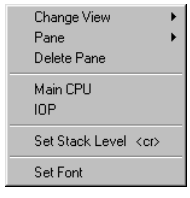
Whenever your application stops running, the call stack pane is updated with the functions that have been called by the application to arrive at the current program counter location.

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



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.

Shortcut menu



Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected pane.

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.

Set Font enables you to change the display font for information in the current call stack pane only. If you wish to change the display font globally for all call stack panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

TTY pane

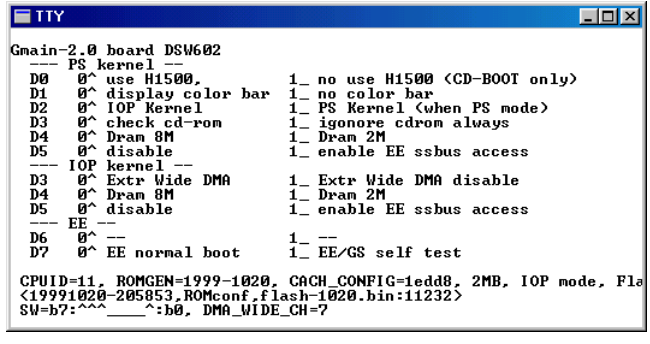
The TTY pane shows any standard output generated by the PlayStation 2. If you have inserted any `printf` commands in your source code, the output from these can be viewed in the TTY pane.

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

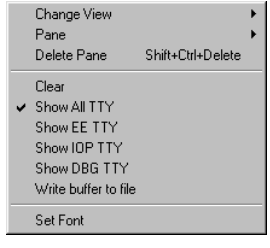
By default the TTY pane shows the output from the PlayStation 2 EE CPU and the IOP. However you can change the TTY pane properties to view one or the other output stream if required.

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.*



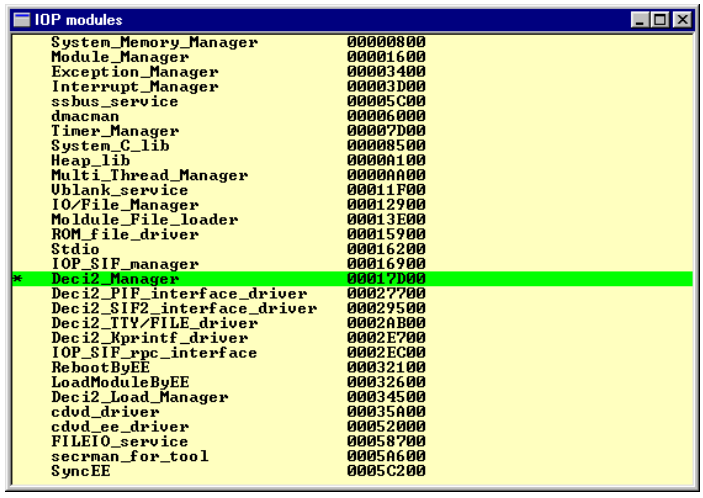
Shortcut menu



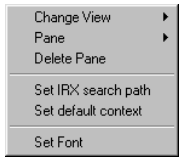
Change View	using this submenu you can change the type of the current pane to another pane type.
Pane	using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).
Delete Pane	deletes the currently selected pane.
Clear	clears the TTY buffers completely.
Show All 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 information about what the ProDG Debugger is doing, such as what IRX files it is able to find at start up, what IOP modules it can see on the target, etc.
Set Font	enables you to change the display font for information in the current TTY pane only. If you wish to change the display font globally for all TTY panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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.



Shortcut menu



Change View using this submenu you can change the type of the current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

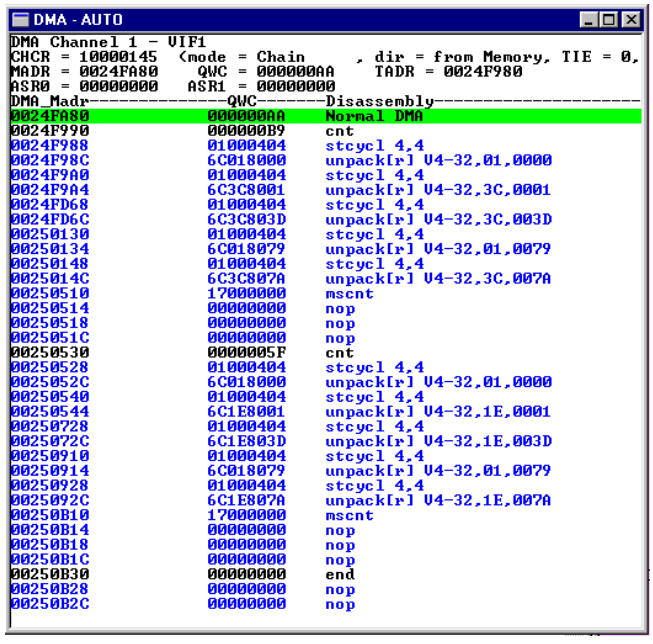
Set IRX search path enables you to set the search path for IRX files.

Set default context sets the selected IOP module (highlighted in green) as the default scope for expression evaluation.

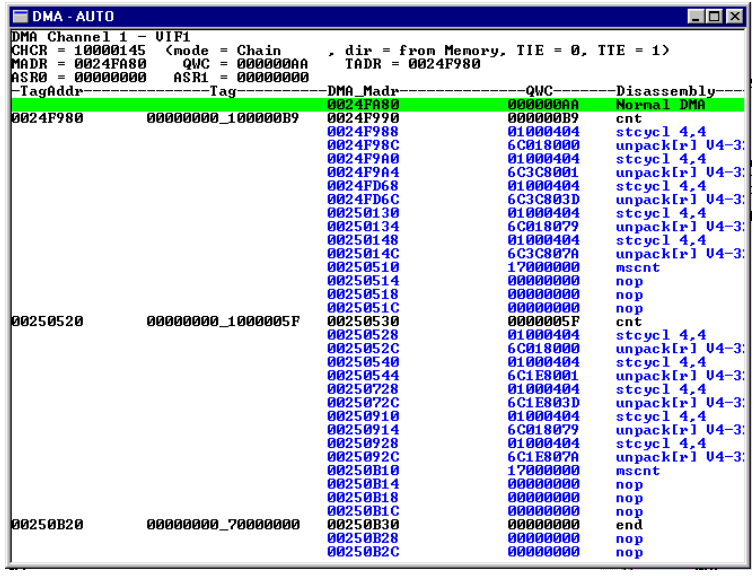
Set Font enables you to change the display font for information in the current IOP modules pane only. If you wish to change the display font globally for all IOP modules panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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 pane with tags hidden and VIF packets shown

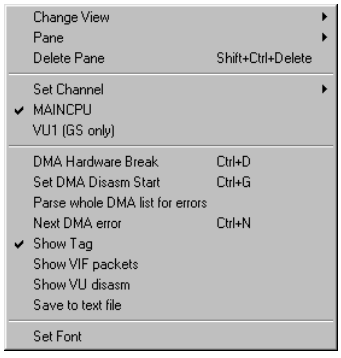


DMA pane with tags and VIF packets shown

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.

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

Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected pane.

Set Channel — using this submenu you can 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).

DMA Hardware Break — brings up the Hard breakpoint on DMA start dialog (see "To set a DMA channel hardware breakpoint" on page 152) 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 — will progress 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.

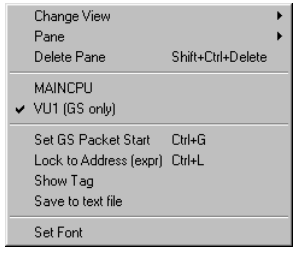
Show Tag when checked, tags are displayed.

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.

DMA packet / VIF packet / Set D bit breakpoint enables you to set a breakpoint on the currently selected line of DMA code. The shortcut menu option changes according to which detail level you have selected: DMA packets / VIF packets / VU microcode.

Set Font enables you to change the display font for information in the current DMA pane only. If you wish to change the display font globally for all DMA panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .



VU1 (GS only) DMA pane shortcut menu

Change View using this submenu you can change the type of the current pane to another pane type.

Pane using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane deletes the currently selected pane.

Main CPU when checked, the DMA pane displays DMA chains in EE RAM and the shortcut menu changes to the Main CPU shortcut menu (see above).

VU1 (GS only) when checked, the DMA pane displays GS packets in VU1 RAM.

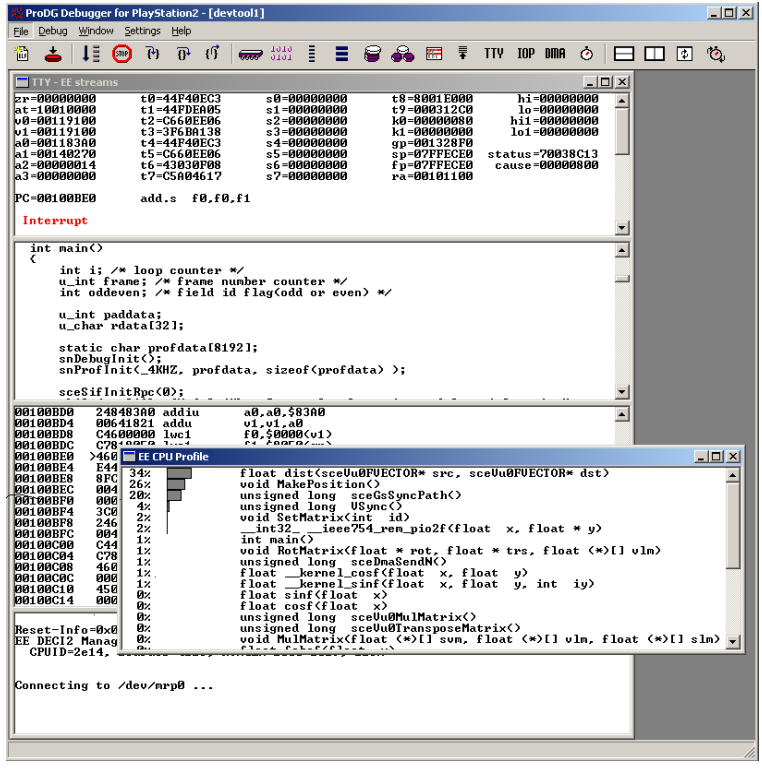
DMA Hardware Break brings up the Hard breakpoint on DMA start dialog ("To set a DMA channel hardware breakpoint" on page 152) which enables you to set a hardware breakpoint on a particular DMA channel.

Set GS packet Start brings up the Enter address dialog allowing you to

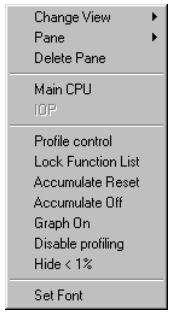
	specify the GS packet start address.
Lock to Address (expr)	brings up the Enter expression dialog allowing you to specify an address or expression to lock the display to.
Show Tag	when checked, tags are displayed.
Save to text file	enables you to save the disassembly to a DMA disassembly (`.dma`) file of your choice.
Set Font	enables you to change the display font for information in the current DMA pane only. If you wish to change the display font globally for all DMA panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

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.



Shortcut menu



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

Change View — using this submenu you can change the type of the current pane to another pane type.

Pane — using this submenu you can access the commands to manipulate split pane views (split, delete, etc.).

Delete Pane — deletes the currently selected pane.

Main CPU — changes to show a profile of the program being executed on the main CPU.

IOP — changes to show a profile of the program being executed on the PlayStation 2 IOP unit.

Profile Control — brings up a dialog which 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 which you can select at runtime see details of the PlayStation 2 EE-side API (see "EE profiler API" on page 133).*

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…

Accumulate Reset this option 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.

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.

Enable Profiling when checked, allows you to freeze the profile display for closer examination.

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 signifigant and should really be ignored.

Set Font enables you to change the display font for information in the current profile pane only. If you wish to change the display font globally for all profile panes use the Application Settings dialog (see "Pane colors and fonts" on page 107) .

Keyboard shortcut reference

This sections describes the shortcut keys for ProDG Debugger for PlayStation 2 and the ProDG Target Manager for PlayStation 2.

There are two possible keyboard shortcut maps available in the Debugger. You can switch between them using the options in the **Settings** menu. The following table shows the keyboards shortcuts that are available in each keymap.

ProDG Debugger for PlayStation 2 shortcut keys

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	
`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
F3	F3	Find search string again.	Source pane
Shift+F4	Shift+F4	Tile all visible panes in the main window.	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
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

Glossary of Terms

EE

The Sony 'Emotion Engine' 128-bit PlayStation 2 CPU.

ELF

Filename extension for a program built to run on the EE unit.

IOP

Input/Output Processor - acts as an interface to external devices.

IRX

Filename extension for a program built to run on the IOP unit.

VU

PlayStation 2 vector units.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Index