Power User's Guide to

ProDG for PlayStation®2 Build Tools

SN Systems

SN Systems Ltd
Version 3.0
December 2002

ProDGPS2_PU(BT)-E / Power User Guide to ProDG for PlayStation 2 Build Tools / Dec. 2002

Document change control

Ver.	Date	Changes
3.0	Dec. 2002	Released with ProDG for PlayStation 2 v3.0. Changes since v2.2g: added info on source browsing and viewing dependencies in VSI chapter; added several 'Troubleshooting' sections; new chapter 'Optimizing your builds'; split out info on 'Relocatable DLLs' to new chapter; new 'Utilities' chapter with info on blankelf, make, and usage of IOP utilities; unsupported assembler/linker options and error messages relegated to appendices; amended table of compiler C/C++ switches and warnings; corrected format for assembler error and warning messages; corrected -x<type> ps2cc switch; more info on -fopt-stack ps2cc switch; #pragma mathwarn added; new 'Additional compiler features' section; new ps2ld switches: -sn-full-map, -sn-overlays and -whole-archive.

Contents

Chapter 1: Introduction

Overview of ProDG for PlayStation 2

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

ProDG for PlayStation 2 consists of:

- **ProDG for PlayStation 2 Build Tools**: including Win32 port of the GNU PlayStation 2 tools; Microsoft Visual Studio integration option; ProDG compiler driver; ProDG linker and ProDG DLL linker; SN Systems utilities and library; ProDG assemblers.

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

- **ProDG Debugger** for PlayStation 2, a fully featured Win32 debugger for debugging your PlayStation 2 applications. The Debugger is described in *Power User's Guide to ProDG for PlayStation 2 Debugger.*

This manual covers the ProDG Build Tools.

Upgrading to ProDG Plus

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

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

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

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

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

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

Updates and technical support

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

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

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

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

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

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

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

SN Systems Limited

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

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

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

Updating ProDG over the web

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

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

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

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

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 2: Using the command-line build tools

ProDG build tools

The ProDG Build Tools for PlayStation 2 are installed to the directory <drive>:\usr\local\sce.

The following table details the ProDG build tools available for the PlayStation 2, with a brief description of what each does.

Command	**Description**
ps2cc	ProDG PlayStation 2 (EE and IOP) compiler driver. See "Using the ps2cc compiler driver" on page 25.
ps2dlllk	ProDG DLL linker. See "ps2dlllk command-line syntax" on page 60.
ps2dllcheck	ProDG DLL checker. See "ps2dllcheck command-line syntax" on page 62.
ps2dvpas ps2eeas ps2iopas	ProDG PlayStation 2 (DVP, EE and IOP) assemblers. See "Using the SN Systems assemblers" on page 35.
ps2ld	ProDG PlayStation 2 linker. See "Using the ps2ld linker" on page 45.

ProDG utilities

The following ProDG utilities are also available:

blankelf	Blank ELF utility. See "blankelf - blank ELF utility" on page 67.
make	A tool which controls the generation of executables and other files of a program.; make takes its input from a command file, with default name makefile. See "make" on page 69.
snarl	SN Archive Librarian. See "snarl - SN Archive Librarian" on page 69.
iopfixup	Performs relocations, patches up debug entries, adds symbols required for IOP execution and adds the .iopmod section. This contains information about loading the IRX file and running it. It only has one input, the partially linked ELF object file.

ioplibdump	Determines the external functions that are called from a particular IRX module. See "ioplibdump" on page 68.
ioplibgen	Produces ILB files and library stubs from Sony IOP TBL table files. See "ioplibgen" on page 68.
iopliblld	Creates an assembly stub source file by locating undefined symbols in the object file, then searching for the corresponding entries in resident library files. When it resolves a symbol it adds jump code pointing to it in the stub. See "iopliblld" on page 69.

GNU build tools

ProDG for PlayStation 2 also ships with GNU C and C++ compilers and PlayStation 2 variants of a number of GNU utilities.

The following table details the GNU build tools available for the PlayStation 2, with a brief description of what each does. For detailed information on using individual GNU tools, refer to the online GNU documentation at **www.gnu.org**.

Command	**Description**
ee-addr2line	Translates program addresses into source file names and line numbers. Addresses are specified on the command line after any options.
ee-c__filt	Decodes (demangles) C++ mangled names.
cc1	GNU C compiler. Normally called by a compiler driver such as gcc or ps2cc. Files sent to cc1 should be preprocessed with cpp first.
cc1plus	GNU C++ compiler. Normally called by a compiler driver such as gcc or ps2cc. Files sent to cc1plus should be preprocessed with cpp first.
cpp	GNU compiler pre-processor. Normally called by a compiler driver such as gcc or ps2cc. Files sent to cc1 and cc1plus should be preprocessed with cpp first.
ee-nm iop-nm	List the symbols in a symbol table, from the specified object files.
ee-objcopy	Copies the contents of one object file to another. Can write the destination object in a different format to the source.
ee-objdump iop-objdump	Displays information about one or more object files or archives.
ee-ranlib iop-ranlib	Generates an index to the contents of an archive and stores it in the archive. An archive with such an index speeds up linking. Equivalent to invoking ar with the –s flag.

ee-readelf	Displays information about one or more ELF format object files. Does not support examining archives or 64 bit ELF files.
ee-size	List the section sizes and total size for each object or archive in its argument list.
ee-strings	For each file specified strings.exe displays each sequence of printable characters greater than four characters in length. For object files only strings from initialized and loaded sections are displayed.
ee-strip iop-strip	Strips all symbols from the specified object or archive files

sn.ini - SN Systems configuration 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 specify which ProDG for PlayStation 2 components you are using, particularly for the benefit of the Visual Studio integration and the ps2cc compiler driver.

These are the default contents of sn.ini:

```
[ps2cc]
##
## EE
##
COMPILER_1=2
COMPILER_2=95
COMPILER_3=3
COMP_VERS=$(COMPILER_1).$(COMPILER_2).$(COMPILER_3)

SN_BASE=C:\Program Files\ProDG for PlayStation2
SCE_BASE=C:\usr
EE_BASE=$(SCE_BASE)\local\sce\ee

## EE Compiler

compiler_path=$(EE_BASE)\gcc\lib\gcc-lib\ee\$(COMP_VERS)
c_include_path=$(EE_BASE)\include;$(EE_BASE)\gcc\ee\include;$(E
E_BASE)\gcc\lib\gcc-lib\ee\$(COMP_VERS)\include
cplus_include_path=$(SCE_BASE)\STLport-
4.5.3\stlport;$(EE_BASE)\include;$(EE_BASE)\gcc\ee\include;$(EE
_BASE)\gcc\lib\gcc-
lib\ee\$(COMP_VERS)\include;$(EE_BASE)\gcc\ee;$(EE_BASE)\gcc\li
b\gcc-lib\ee\$(COMP_VERS)

# Use this cplus_include_path if you wish to use the GCC STL
implementation
```

```
#cplus_include_path=$(EE_BASE)\include;$(EE_BASE)\gcc\ee\includ
e;$(EE_BASE)\gcc\lib\gcc-
lib\ee\$(COMP_VERS)\include;$(EE_BASE)\gcc\ee;$(EE_BASE)\gcc\in
clude\g++-2

# Use this cplus_include_path if you do not use any STL (build
speed will be slightly faster)
#cplus_include_path=$(EE_BASE)\include;$(EE_BASE)\gcc\ee\includ
e;$(EE_BASE)\gcc\lib\gcc-lib\ee\$(COMP_VERS)\include

gcc_major_version=$(COMPILER_1)
gcc_minor_version=$(COMPILER_2)

## EE Assembler

assembler_name=ps2eeas
opt_assembler_name=ps2eeas
assembler_path=$(EE_BASE)\gcc\ee\bin

## DVP Assembler

dvp_asm_name=ps2dvpas
dvp_assembler_path=$(EE_BASE)\gcc\bin
dvp_include_path=$(EE_BASE)\include

## EE Linker

linker_name=ps2ld
linker_path=$(SN_BASE)
linker_script=$(EE_BASE)\lib\app.cmd
library_path=$(EE_BASE)\lib;$(EE_BASE)\gcc\ee\lib;$(EE_BASE)\gc
c\lib\gcc-lib\ee\$(COMP_VERS)
startup_module=$(EE_BASE)\lib\crt0.s


## stdlib used for autolinking e.g. 'ps2cc main.cpp support.cpp
-O2 -g -o main.elf' it links with these libs by default

stdlib=$(EE_BASE)\lib\libgraph.a $(EE_BASE)\lib\libdma.a
$(EE_BASE)\lib\libdev.a $(EE_BASE)\lib\libpkt.a
$(EE_BASE)\lib\libvu0.a $(EE_BASE)\gcc\ee\lib\libm.a

## Additional global compiler options e.g. -ffast-math -fno-
common

compiler_options=-ffast-math -fno-common

##
## IOP
##
IOP_COMPILER_1=2
IOP_COMPILER_2=95
IOP_COMPILER_3=2
IOP_COMP_VERS=$(IOP_COMPILER_1).$(IOP_COMPILER_2).$(IOP_COMPILE
R_3)
```

```
IOP_BASE=$(SCE_BASE)\local\sce\iop

## IOP Compiler

iop_compiler_path=$(IOP_BASE)\gcc\lib\gcc-lib\mipsel-scei-
elfl\$(IOP_COMP_VERS)
gcc_iop_major_version=$(IOP_COMPILER_1)
gcc_iop_minor_version=$(IOP_COMPILER_2)
iop_c_include_path=$(IOP_BASE)\gcc\lib\gcc-lib\mipsel-scei-
elfl\$(IOP_COMP_VERS)\include;$(IOP_BASE)\gcc\mipsel-scei-
elfl\include;C:\usr\local\sce\common\include;$(IOP_BASE)\gcc\mi
psel-scei-elfl
#iop_cplus_include_path=$(IOP_BASE)\gcc\lib\gcc-lib\mipsel-
scei-elfl\$(IOP_COMP_VERS)\include;$(IOP_BASE)\gcc\mipsel-scei-
elfl\include;C:\usr\local\sce\common\include;$(IOP_BASE)\gcc\mi
psel-scei-elfl

## IOP Assembler

iop_asm_name=ps2iopas

## IOP Intermediate Linker

iop_linker_name=ld
iop_linker_path=$(IOP_BASE)\gcc\mipsel-scei-elfl\bin

# Path to IOP build binaries, iopfixup, etc
iop_bin_path=$(IOP_BASE)\gcc\mipsel-scei-elfl\bin

# Path to IOP static libraries, libm.a, libsniop.a, etc
iop_lib_path=$(IOP_BASE)\gcc\lib\gcc-lib\mipsel-scei-
elfl\$(IOP_COMP_VERS)

# Path to ilb table files
ilb_lib_path=$(IOP_BASE)\gcc\mipsel-scei-elfl\lib

# These ilb's are referenced by default when you autolink, this
is like the stdlib= line for EE
iop_stdilb=iop_stdilb=iop.ilb cdvdman.ilb modhsyn.ilb
modmidi.ilb ilsock.ilb ilink.ilb
```

Lines starting with a '#' are treated as comments and are ignored, so you can easily swap between different versions of the build tools, assemblers and linker by changing which lines of sn.ini are preceded by the '#' symbol.

Using variables in sn.ini

Variables can be used in the sn.ini file. These operate in the same way as internal variables in makefiles. For example:

```
[ps2cc]
##Vars
SCE=c:\usr\local\sce\ee\gcc\ee\bin
```

```
COMPILER_1=2
COMPILER_2=96
COMPILER_3=0

COMP_VERS=$(COMPILER_1).$(COMPILER_2).$(COMPILER_3)

assembler_path=$(SCE)
## EE

## EE Compiler
compiler_path=c:\usr\local\sce\ee\gcc\lib\gcc-
lib\ee\$(COMP_VERS)

gcc_major_version=$(COMPILER_1)
gcc_minor_version=$(COMPILER_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\$(COMP_VERS)\include
cplus_include_path=c:\usr\STLport-
4.5.1\stlport;c:\usr\local\sce\ee\include;c:\usr\local\sce\ee\g
cc\ee\include;c:\usr\local\sce\ee\gcc\ee;C:\usr\local\sce\ee\gc
c\lib\gcc-lib\ee\$(COMP_VERS);c:\usr\local\sce\ee\gcc\lib\gcc-
lib\ee\$(COMP_VERS)\include

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\$(COMP_VERS)
```

In this setup, to change compilers, you only need to change the COMPILER_(X) variables at the start of the sn.ini.

Environment variables can also be used as expected. For example:

```
Assembler_path=%SN_PATH%\bin
```

libsn.a - SN Systems library

The archive libsn.a is the main repository for EE target-system code written by SN Systems. This includes DLL support and profiling code. You should always include libsn.a as the first item in your list of EE libs as this is where SN Systems will place other target-system code as and when necessary.

Obtaining the version for libsn.a

The __SN_Libsn_version global symbol allows you to obtain the library version at run time, e.g.

```
printf("libsn version = %d\n",__SN_Libsn_version[0]);
```

Chapter 3: The Visual Studio 6.0 integration

Overview of the Visual Studio 6.0 integration

This chapter describes how to integrate ProDG for PlayStation 2 with Microsoft Visual Studio 6.0. 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.

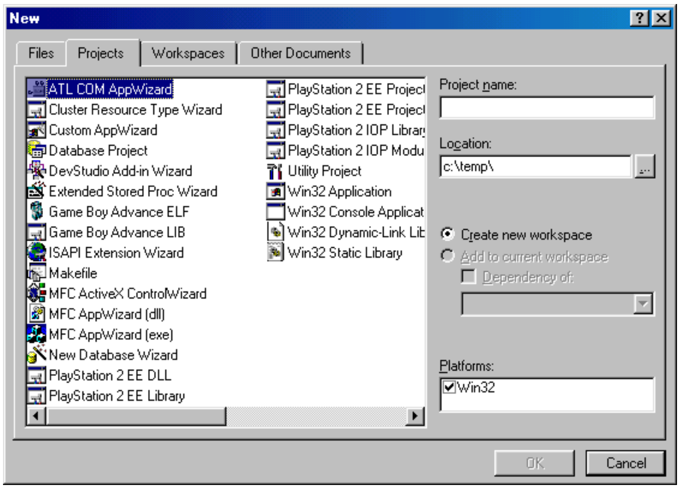
The Visual Studio 6.0 integration has the following main features:

- Builds your project using the ProDG compiler driver ps2cc, which invokes ee-gcc, iop-elf-gcc, dvpasm and ps2ld
- 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

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:



Project name:

Location: c:\temp\

[✓ Create new workspace]

Platforms: ✓ Win32

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 a relocatable DLL" on page 59 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.

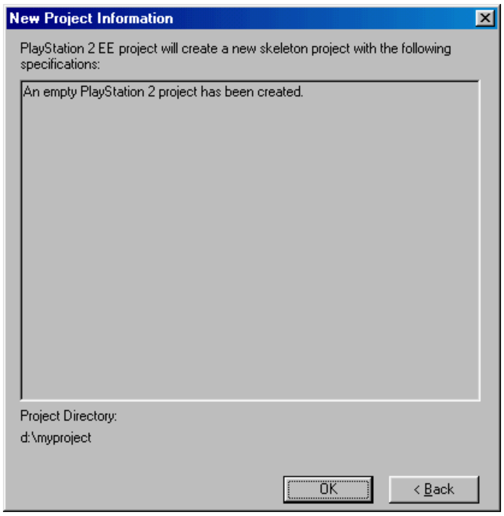
   If the project source files are in a different location to the Visual Studio project file, then Visual Studio will not be able to locate any included files *even if they are in the same directory as the source file*. The solution to this is either:

   - (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. The Win32 checkbox must be checked in the **Platforms** dialog, to create a PlayStation EE/IOP project. 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:


New Project Information

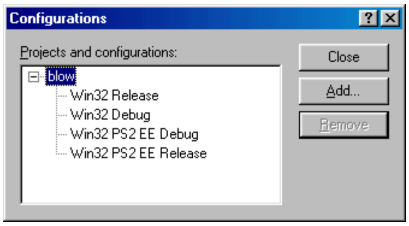
PlayStation 2 EE project will create a new skeleton project with the following specifications:

An empty PlayStation 2 project has been created.

Project Directory:
d:\myproject

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:


Configurations

Projects and configurations:

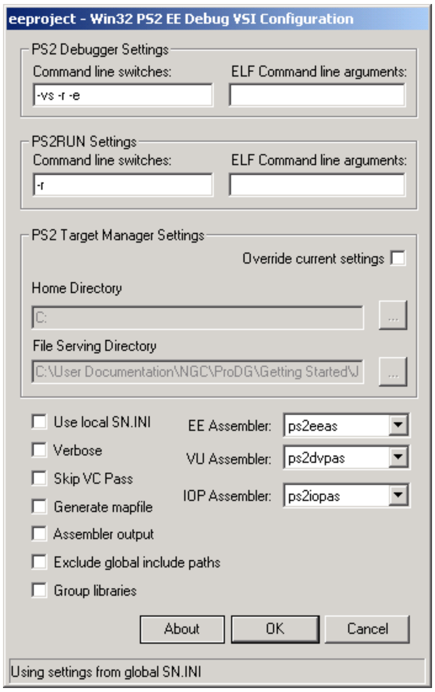
blow
- Win32 Release
- Win32 Debug
- Win32 PS2 EE Debug
- Win32 PS2 EE Release

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

Configuring the Visual Studio 6.0 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:


eeproject - Win32 PS2 EE Debug VSI Configuration

PS2 Debugger Settings

Command line switches: -vs -r -e

ELF Command line arguments:

PS2RUN Settings

Command line switches: -r

ELF Command line arguments:

PS2 Target Manager Settings

Override current settings ☐

Home Directory: C:

File Serving Directory: C:\User Documentation\NGC\ProDG\Getting Started\J

☐ Use local SN.INI

☐ Verbose

☐ Skip VC Pass

☐ Generate mapfile

☐ Assembler output

☐ Exclude global include paths

☐ Group libraries

EE Assembler: ps2eeas

VU Assembler: ps2dvpas

IOP Assembler: ps2iopas

Using settings from global SN.INI

PS2 Debugger Settings	Enables you to set the command-line switches and ELF command-line arguments passed to the ProDG Debugger program, ps2dbg.

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.
PS2 Target Manager Settings	Enables you to set the home and fileserving directories as used by ProDG Target Manager, ps2tm. To use different settings, click the **Override current settings** 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 "sn.ini - SN Systems configuration file" on page 7). 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 "Using the ps2cc compiler driver" on page 25). Check this option to skip the Visual C compiler pass and so speed up the compile. Do NOT check this option if you need to view dependencies (see "Viewing dependencies" on page 21).
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.
Group libraries	Check this option to cause the "-start-group" and "-end-group" commands to be added before and after the list of libraries to be used on the linker command line. This tells the linker to scan the list of libraries contained between the two tags multiple times to resolve any undefined symbols (see "'Reference to undefined symbol' error" on page 24).
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 a relocatable DLL" on page 59 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 preprocessor 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	-Os	Optimize for size
/Ot	-O2	Optimize for speed
/O2	-O2	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 > Settings > C/C++ tab > Project Options** window 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++ tab > C++ Language** category or add /GX to the project settings box.

Linker options can be added the same way by including them in **Project > Settings > Link tab** section.

Note: Visual Studio will link with whichever linker is pointed to by the linker_name environment variable in the sn.ini file (see "sn.ini - SN Systems configuration file" on page 7).

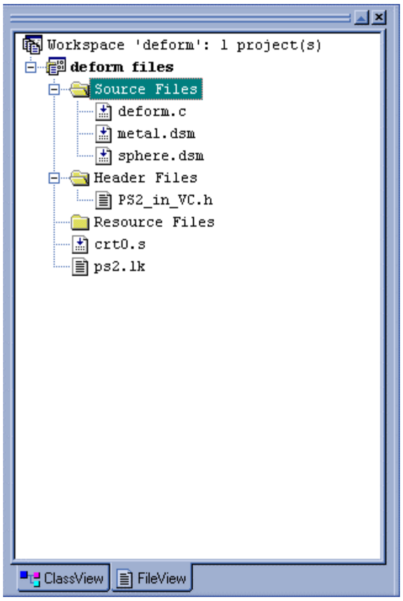
If you are planning to do VU debugging then the GNU linker ld MUST be used. The app.cmd supplied is the default linker script for the GNU linker ld.

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**.
2. Click **New** in the **File** menu and click the **Projects** tab. The New Projects dialog is displayed.
3. Select the PlayStation 2 AppWizard **PlayStation 2 EE Project (ELF)**.
4. Enter 'deform' in the **Project name** box.
5. Specify the location as: \usr\local\sce\ee\sample\vu1.

   Note that the directory deform should be automatically appended to the end of the directory path.
6. 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:



Workspace 'deform': 1 project(s)
deform files
Source Files
deform.c
metal.dsm
sphere.dsm
Header Files
PS2_in_VC.h
Resource Files
crt0.s
ps2.lk

7. Click **Build deform.elf** in the **Build** menu, and deform.elf 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. These can be overridden by creating a [ps2run] section in your sn.ini file (see "sn.ini - SN Systems configuration file" on page 7), 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 the *Power User Guide to ProDG for PlayStation 2 Debugger* for more information on ps2dbg switches.

The ps2dbg command-line switches can be overridden by creating a [ps2dbg] section in your sn.ini file (see "sn.ini - SN Systems configuration file" on page 7), 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 called from the Visual Studio Integration, ProDG Debugger automatically imports any breakpoints that have been set in your source in Visual Studio.

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 19).

Source browsing

The Visual Studio integration features the ability to use Visual C++'s built-in source browsing. However, there are some changes and workarounds needed to allow Visual C++ to generate the maximum amount of browse data.

How Visual C++ generates browse information

The browse data generation feature of Visual C++ relies on the PC compiler to generate browse data for each compilation unit. When called with the command parameter '/FR', the compiler generates an output SBR (.sbr) file. At the end of the build process, it calls BSCMAKE to collect all the individual SBR files into one .bsc browse database.

The problem is that the use of the PC compiler requires the source must be compatible with PC compilations. If the compiler encounters errors it will not generate an SBR file. Constructs in the SDK header files, and user code that is perfectly legal in GNU compilations can cause the browser information generation to fail.

Improving browse information generation

The Visual Studio integration only needs the source to be syntactically correct for the PC compiler, for browse information to be generated. To this end, it calls the PC compiler with a forced include of a special header file, which fixes most of the problems encountered in the SDK headers. This file (PS2_in_VC.h) is automatically created by the application wizard when you create a Visual Studio integration project. While this file can use the pre-processor to fix most of the SDK syntax problems, there are a few that require patches to the SDK headers themselves. These patches use the definition of the symbol __SNVSI__ to alter the header syntax to be PC compatible. These changes will have no effect on the normal use of the SDK headers.

User code and source browsing

Care must also be taken in user code to allow the PC compiler to generate browse data. If you find a compilation unit is not producing an SBR file, the simplest solution is to add your own re-definitions to the PS2_in_VC.h. The most common technique is to simply #define the offending construct to be null.

Using source browsing with makefiles

If you are using external makefiles to build your projects, you can also generate browse information from them. The Visual C++ help library contains an article (Q102326) that describes how to generate browse files outside of the Visual C++ IDE.

Viewing dependencies

The Visual Studio integration supports viewing dependencies. This section describes the changes you need to make in order to get reliable dependency information.

How Visual C++ generates dependency information

The method Visual C++ uses to generate and maintain source file dependency is controlled via the /FD option to the PC compiler. If /FD is not specified 'quick dependency generation' is generated by the Visual C IDE itself. This is a very simple scan through the project's source files. It simply looks for any '#include' directive and builds a dependency graph based on that. However, as it takes no heed of the context of the '#include' (it even finds them in strings and comments), so for more than trivial projects, it's likely that the generated dependencies will be incomplete or wrong.

In addition, a parameter /P allows the PC compiler to solely derive dependency information from the pre-processor phase, rather than a full syntax scan. However, the PC compiler cannot generate browse information in this mode. If you do not require browsing information, the use of /FD and /P will normally be sufficient to maintain accurate dependency data.

If browse information is required, once the PC compiler can process your source files and generate correct source browse information, you should ensure that the /FD option is specified (but not /P).

Getting dependency information to work

Dependency information is generated by the Microsoft Visual C++ 6.0 pass of your build so you must make sure that the "Skip VC Pass" option is NOT selected in the Visual Studio integration control panel.

You should also make sure that the '/FD' option is always specified in **Project > Settings > C/C++ > Project options**. Otherwise Microsoft Visual Studio will perform a quick dependency check rather than letting the Microsoft Visual C++ 6.0 pre-processor do it. Since this quick dependency check is not done by the Microsoft Visual C++ 6.0 pre-processor then

#defines will not be taken into consideration so you'll get numerous errors about header files not being found.

Try adding a "showvcerrors=1" entry to the "[VSI]" section of your sn.ini file. This will allow you to see any errors produced by the Microsoft Visual C++ 6.0 pass. If there are any errors then dependency information will not be produced for that source file.

When browse information is disabled your code is only passed through the Microsoft Visual C++ 6.0 pre-processor for the Microsoft Visual C++ 6.0 pass so you're only likely to get #include errors (which are easily fixed by adding the missing include path(s) to your project settings). However, when browse information is enabled your code must be passed through the Microsoft Visual C++ 6.0 compiler itself. Unless your code can be parsed without any syntax errors then you will not get browse or dependency information.

We have provided a PS2_in_VC.h file in VSI projects to aid you in fixing some of these problems. These files are only included in the Microsoft Visual C++ 6.0 pass so you can use these files to #define out any areas of code that the Microsoft Visual C++ 6.0 compiler cannot compile correctly. Sometimes it may be necessary to actually modify your source code when modifying the PS2_in_VC.h file cannot fix a syntax error in the Microsoft Visual C++ 6.0 pass (such as with multi-line asm statements). Since _WIN32 and SN_TARGET_PS2 are only defined together during the Microsoft Visual C++ 6.0 pass then this is just a simple case of doing something like this:

```
#ifdef SN_TARGET_PS2
#ifndef _WIN32
asm("
...
...
");
#endif
#endif
```

Troubleshooting

Installing breaks other third-party integrations

After installing the Visual Studio integration you may find your existing Visual Studio build setup (e.g. Microsoft XBox SDK) no longer functions correctly.

The standard Microsoft Visual C++ build tools (cl.exe, link.exe, lib.exe) are by default located in the C:\Program Files\Microsoft Visual Studio\VC98\Bin directory. When you install the ProDG Visual Studio integration, our own wrapper tools (sncl.exe, snlink.exe, snlib.exe) are also placed in that directory and the following registry keys modified so that our tools are run instead of the Microsoft ones:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Platforms\Win32 (x86)\Tools\32-bit C/C++
Compiler for 80x86\Executable Path=sncl.exe

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Platforms\Win32 (x86)\Tools\COFF Linker for
80x86\Executable Path=snLink.exe

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Tools\Library Manager\Executable
Path=snlib.exe
```

However when you install the XBox SDK these registry keys are changed back to:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Platforms\Win32 (x86)\Tools\32-bit C/C++
Compiler for 80x86\Executable Path=cl.exe

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Platforms\Win32 (x86)\Tools\COFF Linker for
80x86\Executable Path=link.exe

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Build
System\Components\Tools\Library Manager\Executable
Path=link.exe /lib
```

The XBox SDK usually places its executables (also called cl.exe, link.exe, lib.exe) in the C:\Program Files\Microsoft XBOX SDK\VC7 directory. The problem is that if our Visual Studio integration installation detects a build that is not a PlayStation 2 project, it will pass any commands onto the default executables. So if an XBox project is being built the wrong files will be called (the default Win32 files in C:\Program Files\Microsoft Visual Studio\VC98\Bin, rather than the XBox ones in C:\Program Files\Microsoft XBOX SDK\VC7).

There are three ways to remedy this problem:

1. Delete/rename the cl.exe, link.exe and lib.exe programs in the C:\Program Files\Microsoft Visual Studio\VC98\Bin directory and add the path of the XBox versions to **Options > Directories > Executable Files**.

2. Move the sncl.exe, snlink.exe and snlib.exe programs to the C:\Program Files\Microsoft XBOX SDK\VC7 directory and ensure that this directory is specified in **Options > Directories > Executable Files**.

3. Move the sncl.exe, snlink.exe and snlib.exe programs to a dedicated directory on their own. Add this directory to **Options > Directories > Executable Files**. Ensure that both the C:\Program Files\Microsoft Visual Studio\VC98\Bin and C:\Program Files\Microsoft XBOX SDK\VC7 directories are specified in **Options > Directories > Executable Files**. Our Visual Studio integration

executables should then run the Microsoft executables from whichever directory is highest in the list.

'Reference to undefined symbol' error

The ps2ld linker is dependent on the order in which the libraries are specified on the command line. If a library references a symbol in another library that is specified before it on the command line then you will get "Reference to undefined symbol..." errors.

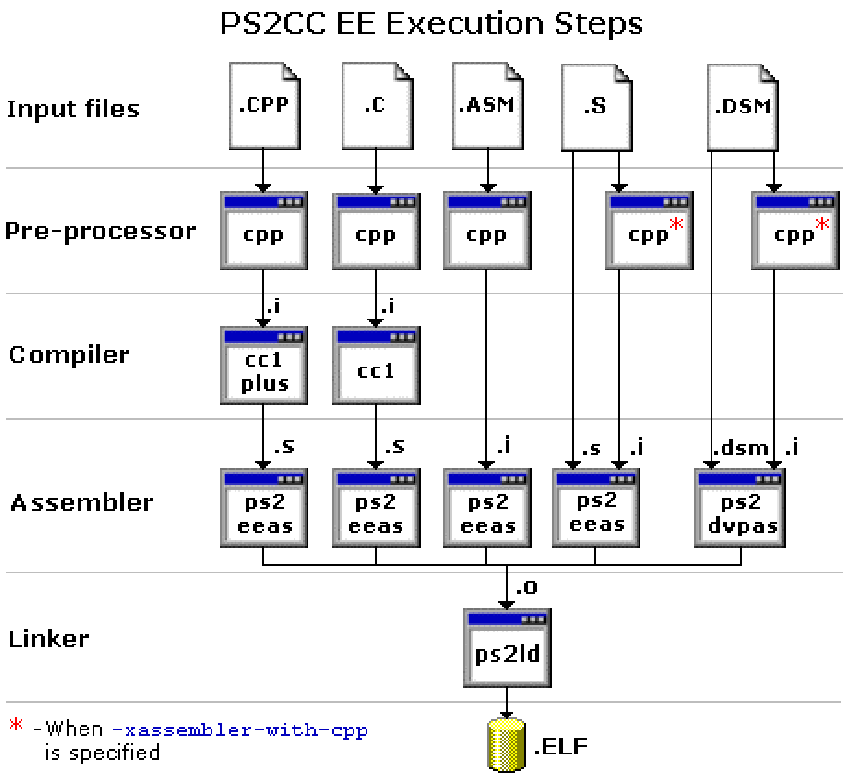
You can remove this order dependency by selecting the 'Group libraries' option in the Visual Studio integration control panel. Please note that grouping libraries will increase the build time. The most efficient method to resolve this problem is to order the libraries so that no library references a symbol from a library listed before it on the command line.

Chapter 4: The ProDG compiler driver

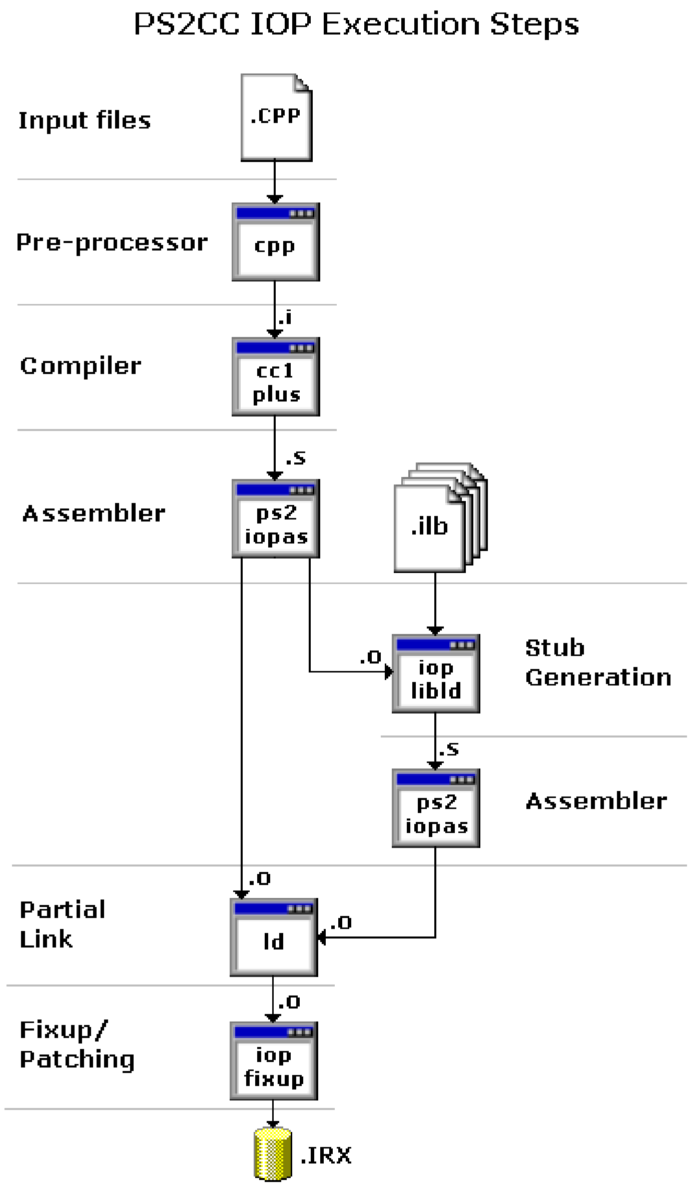
Using the ps2cc compiler driver

The compiler driver ps2cc is a tool to automate the calling of the ProDG for PlayStation 2 compiler, assemblers and linker to produce executable code. It uses command-line switches and filename extensions to determine which compiler or assembler to use for each source file, and then passes the resulting output to the subsequent linker phase. This means you don't have to invoke each tool separately.

The following diagram shows the relationship between these different steps when compiling for the EE:



while the next diagram shows the different steps called by ps2cc when compiling for the IOP (note that .c files can also be used as source, but the compiler driver will call cc1.exe instead of cc1plus.exe):



The compiler driver automatically generates many parameters to the other tools. To see exactly what is being passed to each tool, specify the '-v' (verbose) output option. This will write to standard output all the parameters as each tool is called.

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=.... See "sn.ini - SN Systems configuration file" on page 7 for details.

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=ps2eeas
opt_assembler_name=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=ps2ld
linker_path=c:\usr\local\sce\ee\gcc\bin
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
....
```

ps2cc command-line syntax

The ps2cc compiler driver command-line syntax is as follows:

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

If you are porting a GNU project you can replace calls to ee-gcc in your makefiles with calles to ps2cc, but you must check that all the options you specify are supported by ps2cc.

ps2cc options

Compiler driver options precede the filename list. The following tables group the various compiler options according to type:

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.
-E	Pre-process only. If no output file is specified, output is sent to the screen.
-iop	Compiles for the IOP (default=EE).

-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).
-S	Compile to assembler source. If no output file is specified, the output file is the original filename with a new extension of .S.
-save-temps	Preserve intermediate temporary files such as pre-processor output and compiler-generated assembler source.
-v	Verbose mode – print all commands before execution.
-x<type>	Treat subsequent input files as being of type <type>. See "Overriding default file extension actions" on page 33.

C/C++ language options

Options	**Actions**
-ansi	Check code for ANSI compliance.
-f…	Specify a compiler option / optimization (full list in GNU documentation at **www.gnu.org**).
-fabslineno	In warnings and errors, refer to the absolute line number in the pre-processed input to the compiler rather than the line number in the input file to the pre-processor (i.e. report line number in the .i file not the input .c/.cpp file). This can be useful when attempting to find errors in complex pre-processor macros. When used in conjunction with -save-temps, the .i file can be examined to find the exact place of the warning or error.
-fbyte-bool	Make the C++ 'bool' type 1 byte long rather than the default of 4 bytes. **Note:** this switch must be used project wide (i.e. don't mix bool sizes between compilation units).
-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".
-fdevstudio	Produce warning and error messages in Visual C++ development studio format.
-fforce-link-once	Force template instances and implied inline methods to be placed in link once sections even when generating debug data (the default is to not use link once sections when debugging). This allows automatic removal of repeated template instances and implied inline methods at link time (see below).

-fno-implement-inlines	Do not generate non-inlined instances of inlined functions. See "Preventing generation of non-inlined function copies" on page 53 for details.
-fno-exceptions	Disable the C++ language exception features. Enabling the C++ exception feature can result in generating a large amount of extra code and data, even if your program does not use C++ exceptions.
-fno-inline-debugging	Do not generate debug data for inlined functions. This is useful when debugging code with a lot of inlined methods where the jumping to inlined source code is distracting when stepping code.
-fno-static-dtors	Suppresses generation of destructor code for static global class instances. See "Removing global destructors" on page 53 for details.
-fopt-stabs	Perform an optimization pass to remove unreferenced types when generating STABS debug data. When enabled the compiler only includes debug data for types that are actually used in that compilation module. Using this option can result in slightly longer compile times, but should supply much faster linking and loading into the debugger.
-fopt-stack	When the compiler is generating code to save / restore clobbered GP registers on function entry / exit it automatically saves all 128 bits of the clobbered registers. When this switch is enabled the compiler will check the actual register usage, and if only the lower 64 bits of a register are clobbered then the compiler will generate code to save and restore only the lower 64 bits, thus reducing the size of the stack frame. Note that asm statements that alter scratch registers will cause 128-bit saves/restores; this is due to the fact that the compiler cannot determine the mode the user asm code uses to access the output/clobber registers.
-frelax-128	Allow maths expressions on 128 bit types. However, the maths operations will only be performed on the lower 64 bits of the variables.
-frel-stabs	Do not include any source level information when generating debug data. This is useful when you wish to be able to examine variables in an optimized build, where the source information would be of limited use due to code removal/reordering.

For non-debug C++ builds, the compiler generates template instances in special 'link_once' sections. The linker uses these sections to ensure that only one copy of each template instance is linked into the final program. For debug builds the compiler, by default, uses 'weak' symbols for multiple template instances as this allows full debugging, but it does bloat the final code size. Use the compiler switch -fforce-link-once to generate link_once sections even for debug builds. However, note that full debugging of template code may not be possible. Note also that use of this feature requires the latest style app.cmd linker script as supplied in the Sony libraries.

Warning options

Options	Actions
-W…	Enable (or disable using -Wno-...) individual warnings (full list in GNU compiler documentation at **www.gnu.org**).
-W	Enable extra warnings not enabled by -Wall. See "C compiler not giving some warnings" on page 34.
-Wall	Enable all warnings.
-Wno-missing-enum-case	Do not produce warnings for missing cases of a switch based on an enumeration.
-Wpromote-doubles	Warn of generation of promotion to double of float types. This warning can also be set on a function by function basis using the #pragma mathwarn (see below).
-Wsoftware-extern-library	Warn on generation of calls to software maths library routines.
-Wsoftware-math-library	Warn on generation of calls to software maths runtime routines.
-w	Disable all warnings.

We also provide the #pragma mathwarn to allow toggling of
the -Wpromote-doubles warning setting on a function by function basis:

Usage:

```
#pragma mathwarn on
#pragma mathwarn off
```

When set to on, a function will warn about the compiler promoting floats to doubles or calling software floating pouint maths routines.

Debugging options

Options	Actions
-g	Generate debug information for source-level debugging. **Note:** this option is required if using 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 at **www.gnu.org**).

Assembler option

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

The assembler arguments must be separated from each other by commas. You do not usually need to use this mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler.

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 at **www.gnu.org**).

Filenames

ps2cc can accept any of the following types of file as input and applies the following actions according to the filename 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	.DSM	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.

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 and 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.

Overriding default file extension actions

It is also possible to use the -x<type> 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.

<type>	**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

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 new lines. 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.

Additional compiler features

This section describes some additional features of the compiler.

Unaligned load/store built-in calls

Two built-in compiler functions allow the reading and writing of integers to memory regardless of their alignment:

```
int __builtin_getunaligned ( void* ptr );
int __builtin_setunaligned ( void* ptr, int value );
```

Troubleshooting

C compiler not giving some warnings

The C compiler features some extra warning settings that are not set on by default. In addition, setting the "warning all (-Wall)" option also does not set them on! The 'extra' warnings are enabled by providing the '-W' switch on your build command, e.g.:

```
ps2cc -W -Wall myfile.c
```

'Integer constant out of range' error

C code that uses u_long128s will compile without any problem, but the same code in a C++ file will generate the compiler error "integer constant out of range" if you assign large numbers to a u_long128 variable like this:

```
u_long128 gBigNumbers[] = {
0x000000656e6f74536c6c614679646f42,
0x0000646e756f72476c6c614679646f42,
0x00000000646f6f576c6c614679646f42
};
```

In C++ you need to put 'LL' at the end of long long integer literals.

'Unknown escape sequence' error

The following warning may occur when compiling source code that contains Shift-JIS encoded characters:

```
"unknown escape sequence: '\' followed by char code X"
```

This is because some Shift-JIS multi-byte characters contain a 0x5C ('\') character, which the C preprocessor interprets as a control code.

This problem can be resolved in two ways:

1. Insert an extra 0x5C ('\') character next to S-JIS 0x5C ('\') character. This causes the C preprocessor to interpret it as a character code rather than a control code.

2. Create the following Windows or makefile environment variable:

   LANG=C-SJIS

Chapter 5: The ProDG assemblers

Using the SN Systems assemblers

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

The command-line options and directives for the PlayStation 2 assemblers are compatible with the GNU assemblers except for some GNU options and assembler directives, which are ignored or produce errors. See "Unsupported assembler options" on page 87 and "Unsupported assembler directives" on page 87 for further details.

In addition, if you specify the -sn option, you will have access to the SN Systems directives, which provide excellent features like scope delimiting and register naming. See "SN Systems directives" on page 38.

Command-line syntax

The SN Systems PlayStation 2 assemblers directly replace the GNU assemblers ee-as, iop-as and ee-dvp-as, and are completely compatible with them. To use them you need to invoke the assembler directly on the command line, for example:

```
ps2eeas <options> <input file> ...
```

After the program name the command line may contain options and filenames.

Options are instructions to change the behavior of the assembler. An option is specified as a hyphen ('-') followed by one or more letters. Note that the case of the letter is important. Options may appear in any order, and may be before, after, or between filenames.

Two hyphens together ('--') are used to name the standard input file explicitly, as the source of a file to be assembled.

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:

```
ps2eeas -o my-object-file.o mumble.s
```

```
ps2eeas -omy-object-file.o mumble.s
```

The following table lists all of the available options:

Options	Actions
--defsym sym=value	Define integer symbol
-G<size>	Set size of data items placed in .sdata/.sbss
-g	Produce STABS debug info
--help	Print help
-I <directory>	Specify directory to search for include files
-L	Output local symbol information
-keep-constants	Keep constants in symbol table. Default is to exclude constants from symbol table.
--keep-locals	Same as –L
-MD<filename>	Produces GNU make dependency file.
-no-align	Turns off data alignment.
-no-align-warn	Turns off data alignment and warns when writing unaligned data.
-o <output file>	Specify output filename
-sn	Activate SN Systems extensions. See "SN Systems directives" on page 38 for further information
--stdout	Print error messages on stdout rather than stderr
--version	Print version information
-W	Disable warnings
-Wdivbug-on	Enable warnings about auto-padding of illegal div instructions

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 SN Systems EE and IOP assemblers ps2eeas and ps2iopas can be invoked directly from the EE or IOP (ee-gcc, or iop-elf-gcc) compiler drivers using the -snas command line option, or from ps2cc by setting your sn.ini file to use the appropriate assemblers.

- The SN Systems VU assembler ps2dvpas is used on the command line to assemble .dsm files that have been written in VU microcode. 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.

Assembler directives

The following list briefly describes all of the default directives that are understood by the PlayStation 2 assemblers. This does not include the standard GNU directives, which can be found in the GNU documentation.

Directive	Description
.data	Switches to the .data section
.quad	Specifies 128 bit data words
.word	Specifies 32 bit data word

The following list briefly describes all of the PlayStation 2 specific directives that are understood by the SN Systems VU assembler `ps2dvpas`. This does not include the VU opcodes, which can be found in the Sony documentation.

Directive	Description
.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
.text	Switches to the .vutext section
.vu	Switches to VU opcode mode
.vu0	Same as .vu, but enables VU0 syntax checking
.vu1	Same as .vu, but enables VU1 syntax checking

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	

SN pseudo opcodes (use -sn)

LOIB	Same as LOI, but encodes as a binary.
LOIF	Same as LOI, but encodes as a float.

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 39 for sample code.
.incbin "filename"	Includes a file "filename" without actually assembling it.
.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 40 for sample code
.startmicro <dest_instruction> .startmicroa <dest_addr> .endmicro	Enables VU source-level debugging. See "VU source-level debugging" on page 40 for further details.

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
```

Register labels can also be redefined in the same source file, for example:

```
.equr AREGISTER, VF01
...
.equr AREGISTER, VF02
```

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.

VU source-level debugging

This section describes how to enable VU source-level debugging for microcode that has been DMA'd without using the MPG assembler opcode. It refers to the ps2dvpas assembler.

- Any VU microprogram you wish to enable source-level debugging for, must be wrapped in a .startmicro/.endmicro pair.

- You must also specify the intended destination in VU micromem as a .startmicro parameter. If you do not provide a parameter the assembler will default to 0x0.

There are two variations of the .startmicro directive:

`.startmicro`	This takes a <destination instruction> parameter, i.e. equivalent to the VIF packet encoding (VU micromem address / 8).
`.startmicroa`	This takes a <destination address> parameter, i.e. equivalent to the assembler MPG opcode specification (standard VU micromem address).

An example of their use is shown in the following examples:

```
.vu
VU1_1_MICRO_START:
.microstart      ;defaults to 0x0
.include "swapped1.vsm"
.microend
VU1_1_MICRO_END:
VU1_2_MICRO_START:
.microstart 100  ;equivalent to .microstarta 0x320
.include "swapped2.vsm"
.microend
VU1_2_MICRO_END:
```

- The directive address parameters can be expressions but they must evaluate to a constant.
- If you are DMAing the same microprogram to different places on the VU during your program, then you must specify the destination address of the transfer you wish to debug.
- If at some point in your program you wanted to transfer VU1_1_MICRO_START to VU1_2_MICRO_END, you would need to change your .microstart/.microend pairs to wrap these only. In this case you would not be able to debug any of the occasions when you transferred VU1_1_MICRO_START and VU1_2_MICRO_START separately.

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 42.

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 object file. Conventionally, object filenames end with the .o extension, so that when assembling <file>.s the default name of the object file is <file>.o. You can give it another name by using the -o option.

The object file becomes input to the linker. It contains assembled program code, information to help the linker integrate the assembled program into an executable 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 the compiler driver 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>:(<line_number>) : Warning: <message>, "<line of
code>"
```

- If a logical filename has been given (using the directive .app-file string) it is used for the filename <file_name>, 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 <line_number> printed, otherwise the actual line in the current source file is printed.
- The text "<line of code>" will show the actual line of source that contains the problem.

Error messages have the format

```
<file_name>:(<line_number>) : Error: <message>, "<line of
code>"
```

- The filename and line number are derived in the same way as for warning messages.

Symbol names

For detailed information on the formation of symbol names in the GNU assembler as, refer to the GNU documentation at http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html - SEC45.

Local symbols

There are several types of local symbols that can be used when writing assembler. It can be confusing distinguishing the one you need to use.

There are symbols that are local to the file scope. These are equivalent to variables or functions declared as "static" in C code. Other object files cannot refer to these symbols. Other files may have their own local symbols of the same name. This will not cause problems when linking.

All labels are assumed to be local by default and are converted to a global symbol by compiler directives such as ".global".

The assembler and linker normally keeps these symbols as they are in scope when considering a context associated with the file. For example when debugging code within the object the local symbols are in scope and a debugger will evaluate them.

There is also a naming convention for symbols that you want to use for code flow control but which are not relevant to debugging. These symbols have a ".L" prefix. Alternatively, you can use an "@" prefix when SN extensions are enabled using the command line option -sn. These symbols will be used for the purpose of assembly or linking but will be discarded when no longer needed. These symbols do not appear in the final executable. The symbols must still have a unique name within an assembly file. The compiler uses labels with a local prefix for loops where there is no logical name associated with the address of the beginning of a loop. Use "-L" or "--keep-locals" on the command line to keep symbols with a local prefix, if you so desire. When writing assembler by hand you often want to keep the labels relating to flow control for debugging. In this case use a normal label without a prefix. However, like the compiler, you may use the local label prefix to remove these labels from the final symbol table.

Chapter 6: The ProDG linker

Using the ps2ld linker

This chapter describes the ProDG for PlayStation 2 linker, ps2ld. These are the main features of the linker:

- 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.
- Demangles symbol names in messages and MAP file.
- Removes unused and duplicate functions from the image.
- Requires a ProDG tools license (the same license as the ProDG Debugger).
- Much faster than the GNU linker ld.

Memory requirements

Your system should have at least 256 MB RAM to avoid linker swapping. A rule of thumb for ps2ld 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.

Building with ps2ld

The Sony sample programs use the GNU linker, ld, by default. There are two ways to invoke the ProDG linker ps2ld.exe instead:

- call ps2ld in place of ld.
- call ps2ld in place of collect2 and ld.

To call ps2ld in place of ld only

1. Rename the original GNU ld.exe file to something else, e.g. ld-orig.exe.

2. Rename ps2ld.exe to ld.exe and place it in the standard directory for binary files \usr\sce\local\sce\ee\gcc\ee\bin.

The compiler driver ee-gcc will then use collect2.exe, which in turn invokes ld.exe.

To call ps2ld in place of collect2

The ps2ld linker does not require collect2.exe at all, so it is possible to bypass collect2 altogether, using one of the following methods:

- Call ps2ld with the -collect2 option on the command line and then pass it the standard collect2 command-line options and linker script. Note that the -T <linker script> option must be placed at the end of the command line. The ps2ld linker will perform the same tasks as collect2 and ld combined, thereby reducing the overall link time.

- If using the ps2cc compiler driver (see "Using the ps2cc compiler driver" on page 25) the choice of linker can be specified by setting the linker_name and linker_path environment variables in the sn.ini file (see "sn.ini - SN Systems configuration file" on page 7). For example:

```
[ps2cc]
linker_name=ps2ld
linker_path=c:\PROGRA~1\PRODGF~1
linker_script=c:\usr\local\sce\ee\lib\app.cmd
```

Later versions of ps2cc will detect that ps2ld is configured as the linker and automatically add the -collect2 argument to the linker options. This allows for seamless migration from a project using ps2cc and collect2 as the linker, to one using ps2ld as the linker.

Linker scripts

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. The ProDG linker, ps2ld, supports most of the 'ld format' linker script directives.

For detailed information on the format of linker scripts, see the GNU linker ld documentation at **www.gnu.org**.

For a complete list of script file directives supported by the GNU linker ld, which are not supported by ps2ld, see "Unsupported script file directives in ps2ld" on page 90.

Sections

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)

Referencing files in linker scripts

If an object filename in a linker script does not contain any wild cards then it is assumed that this object is required for the link even if it does not appear on the link command line. Thus if the linker is unable to find the object the link will fail, e.g.:

```
mysection :
{
     foo.o(.text)
}
```

In this case foo.o does not contain any wild cards ('*' or '?') so is added to the link. If the linker cannot find foo.o then the link would fail.

If you wish foo.o to only be added to the link if it is explicitly listed on the command line then modify the script slightly so that the name contains a wild card, e.g.:

```
mysection :
{
     foo.o*(.text)
}
```

ps2ld command-line syntax

The ps2ld linker command-line syntax is as follows:

```
ps2ld [switches] [files]
```

`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.

ps2ld switches

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

Switches	Actions
-collect2	Perform a 'collect2' linker pass.
-comment	Keep comment sections.
-d, -dc or -dp	Tells the linker to allocate common data even though it is only doing a partial link.
-defsym <symbol>=<value>	Define <symbol> with <value>.
-e <entry> or --entry=<entry>	Set entry point to <entry>.
-G<size> or --gpsize=<size>	Sets maximum size of gp relative variables. Sets the threshold for small data objects to be <size> bytes, i.e. objects <size> bytes and below will be placed in the small data section.
-keep <file>	Keep all symbols listed in <file>. This tells the linker to include these symbols even if defined in libraries. If the command line includes '-strip-unused' or '-strip-unused-data' the symbols listed in <file> will not be stripped. Include this switch when function stripping modules to prevent the stripping of the entry points.
-l<library>	Include library file "lib<library>.a" when linking, e.g. -lc causes the linker to look for libc.a.
-L <dir>	Add this path to the list of directories searched for libraries.
-Map <mapfile>	Generate a map file in <mapfile>.
-no-keep-memory	Use less memory when linking.
-o <file>	Use <file> as the destination file. This file should have an ELF extension. The symbol table information used by the debugger is also written to the output file.
-print-map or -M	Print map file to stdout.
-r	Tells the linker to perform a partial link.
-report-unused	Reports symbols that will be stripped (in file statcov.txt).
-S or –strip-debug	Strips debug info from output.
-S-lib	Strips debug info from libs.
-s or	Strips symbols and debug info from

-strip-all	output.
-s-lib	Strips symbols and debug info from libs.
-sn-force-section <section>	Output this section even if empty.
-sn-full-map	Provides additional information in the map file, e.g. static variables.
-sn-no-dtors	Linker marks destructors as unused. If used in conjunction with -strip-unused or -strip-unused-data then the unused destructor code will be removed from the game.
-sn-overlays	Allows debugger to debug overlay code.
-start-group <libs> -end-group	Allows symbols defined in any library in <libs> to be resolved, irrespective of the order in which the symbols are defined and used. See "'Reference to undefined symbol' error" on page 49.
-strip-unused	Removes unused function code.
-strip-unused-data	Removes unused function code and data.
-T <file>	Use <file> as the linker script.
-t or –trace	Print names of files as they are opened.
-Tbss <addr>	Set <addr> as address of bss section.
-Tdata <addr>	Set <addr> as address of data section.
--warn-stabs	Warn if there were STABS problems.
-whole-archive	Linker includes all the contents of a library without regard to whether the objects are referenced. The switch affects all libs that appear on the command line after the switch. The behavior can be turned off using the -no-whole-archive switch, such that libs following the second switch will then link normally.
-x or -discard-all	Discard local symbols.

The ProDG linker ps2ld does not support all of the switches supported by the GNU linker, ld, as many of these are not appropriate for the PlayStation 2 platform. For a complete list of switches which are not supported by ps2ld, see "Unsupported ld switches" on page 89.

Troubleshooting

'Reference to undefined symbol' error

The ps2ld linker is dependent on the order in which the libraries are specified on the command line. If a library references a symbol in another library that is specified before it on the command line then you will get "Reference to undefined symbol..." errors.

You can remove this order dependency by specifying -start-group <libs> -end-group on the linker command line. Please note that grouping libraries will increase the build time. The most efficient method to resolve this problem is to order the libraries so that no library references a symbol from a library listed before it on the command line.

Chapter 7: Optimizing your builds

Introduction

This chapter gives an overview of methods to optimize building and running code on the Sony Computer Entertainment Inc. PlayStation®2 using ProDG. In addition check the technical support section of the SN Systems website (see "Updates and technical support" on page 2) for further in-depth details of some of these optimizations.

When optimizing your code size, build speed or code execution speed there will always be trade-offs. It is important to measure whatever you are optimizing and see what effect the change has. Also target your efforts in the areas that are most relevant. It can seem very tempting to turn on all the features everywhere. However a more targeted approach is often much more effective.

Optimizing for small ELF size

If you need to reduce the size of your ELF file so that it takes up less memory you have little option but to try and reduce the amount of code, data and debug information in your program build.

Apart from taking up less memory a smaller ELF is also likely to lead to faster build times and may lead to a performance improvement at run time due to improved cache usage. Because it has these additional optimization benefits, your primary aim must be to make your ELF as small as possible. This section describes some ways you can try to achieve this.

Compiler options

The following table lists the compiler switches, which are effective in reducing the size of the ELF file:

Option	Effects
-fbyte-bool	Make the C++ 'bool' type 1 byte long rather than the default of 4 bytes. **Note:** this switch must be used project wide (i.e. don't mix bool sizes between compilation units).
-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".

-fforce-link-once	Force template instances and implied inline methods to be placed in link once sections even when generating debug data (the default is to not use link once sections when debugging). This allows automatic removal of repeated template instances and implied inline methods at link time.
-fno-exceptions	Do not generate code for handling C++ exceptions.
-fno-implement-inlines	Do not generate non-inlined instances of inlined functions. See "Preventing generation of non-inlined function copies" on page 53 for details.
-fno-inline-debugging	Do not generate debug data for inlined functions. This is useful when debugging code with a lot of inlined methods where the jumping to inlined source code is distracting when stepping code.
-fno-rtti	Do not generate code for handling C++ run time type information (RTTI).
-fno-static-dtors	Suppresses generation of destructor code for static global class instances. See "Removing global destructors" on page 53 for details.
-frel-stabs	Do not include any source level information when generating debug data. This is useful when you wish to be able to examine variables in an optimized build, where the source information would be of limited use due to code removal/reordering.

If some areas of your C++ code use run time type information (RTTI) or exceptions then it may be worth considering redesigning these areas of code, in order to eliminate them from your game.

Linker options

The following linker options can help reduce generated ELF size:

Option	**Effects**
-s or -strip-all	Remove all symbol information when generating the ELF. See "Removing symbol information" on page 53.
-S or -strip-debug	Remove debug information (but not symbols) from the generated ELF.
-s-lib	Removes debug info from the libraries but leaves it in the object files. See "Removing unnecessary debug information" on page 53 for more information.
-sn-no-dtors	Do not link global destructors. See "Removing global destructors" on page 53 for details.
-strip-unused	Removes unused code from the ELF. This can be very beneficial on debug builds of code with a lot of template usage, as this will remove the duplicate instances of template methods (this is achieved

	automatically on non-debug builds).
-strip-unused-data	Removes unused code and data from the ELF. Compared to using -strip-unused, in addition to removing unused data this option may strip additional code that was only referenced through data structures that have been stripped.

Removing symbol information

Using the -s or -strip-all option will remove all symbol information and so make the ELF smaller.

Removing unnecessary debug information

You should only produce debug information for the files you are interested in debugging. Use the -g compiler switch on the files you want to debug and -g0 for files you do not need to debug.

Note: If you are using the Visual Studio integration then you cannot set the -g option on a file by file basis. However you can split your project up into libraries and turn off debug info generation in the library project, which you are not debugging. Move game code you do not want to debug out of the main project and into a library.

If all the code you wish to debug is linked as objects then you can remove debug information from any libraries you are linking by using the -s-lib linker option.

Preventing generation of non-inlined function copies

If you are inlining a lot of functions, the compiler switch -fno-implement-inlines can be used to prevent the generation of non-inlined copies of the function. However, if your code does call non-inlined versions of these functions, at least one compilation module must include instances of them or you will get link-time errors.

Note that although use of this option will reduce ELF size it can increase the build time.

The GNU compiler also uses an extension to the language to indicate that no non-inlined instances should be generated. If functions are declared 'external inline' in a header they will not generate a non-inline instance.

Removing global destructors

Global destructor code calls the destructors for global instances of classes. Compiler and linker options are available to remove global destructor code, where for example the program never exits (the console is simply switched off) and there is no requirement to tidy up. Removing global destructors can be particularly effective when used in conjunction with stripping. Note that this option should not be used for DLLs as they need to tidy up global instances of classes when they are unloaded.

The compiler option -fno-static-dtors removes global destructor code from the object files, whereas the linker option -sn-no-dtors stops global destructors from being linked. The linker option can remove destructors from library code that was not built using the -fno-static-dtors compiler option. For best results use both these options.

Optimizing for fast execution

If you are trying to optimize your code for fast execution you must target your efforts on finding which sections of code consume the most CPU time. You can use the basic EE profiler built in to the ProDG Debugger for PlayStation 2, or the more advanced Tuner for PlayStation 2 to help you find less than optimal areas of code.

Optimizing for small ELF size can in itself lead to a performance improvement at run time due to improved cache usage. See "Optimizing for small ELF size" on page 51 for more information.

Compiler option

The following compiler switch may help increase execution speed:

Option	**Effects**
-fopt-stack	When the compiler is generating code to save / restore clobbered GP registers on function entry / exit it automatically saves all 128 bits of the clobbered registers. When this switch is enabled the compiler will check the actual register usage, and if only the lower 64 bits of a register are clobbered then the compiler will generate code to save and restore only the lower 64 bits, thus reducing the size of the stack frame.

Using the compiler optimization switches

The compiler offers many optimization switches. For simplicity there are some predefined levels of optimization available using the -Ox switch. This switch can take four values by specifying the suffix to be any number from 0 to 3, thus -O0 to -O3. Generally the greater the number, the more options are used and the faster the code executes. However you need to be aware that using the -O3 flag for all your files can reduce performance compared to a game compiled with -O2.

The major difference between -O2 and -O3 is that -O3 turns on automatic inlining. The compiler will inline functions even if inlining is not implied in their declaration. However, this tends to produce large code. In this situation the caching becomes much more inefficient with many more cache misses, making overall execution slower. Therefore, you will usually find that using -O2 for all files makes the code run faster compared to using -O3.

You may find that occasional targeted use of the -O3 optimization on a few key modules may improve performance. The basic EE profiler in the ProDG Debugger for PlayStation 2 may help find these. If you find this is the case then you can target particular functions within a module for inlining by declaring them inline and then going back to using the -O2 optimization switch.

Optimizing for build speed

Optimizing your game for small ELF size is the main way you can improve build times, by requiring less code (especially debug code) to be compiled. See "Optimizing for small ELF size" on page 51. This section describes some additional ways to optimize for faster build times.

Compiler debug optimization

Use the compiler debug optimization switch '-fopt-stabs' to remove debug information about unreferenced types. Although this option can increase compilation time slightly, usually the faster link time offsets this penalty.

Avoid unnecessary include files

Taking some care when planning your source project can reduce build times. The most critical area is in the usage of include files. If your project is structured so that compilation modules only include the minimum number of headers this can drastically reduce compilation time, link time and debugger load time for debug builds. For every type contained in header files, the compiler generates debug information. If your modules include a lot of unnecessary headers this will enlarge the debug data. This in turn leads to longer link times and slower loading of program into the debugger.

Another header area that can cause problems in both debug and release builds is the use of templates in C++. If your compilation modules include a lot of inter-dependent template code, each compilation can incur an overhead generating the instances of template methods. Again, where possible, keeping the use of headers containing templates to a minimum can improve build speed.

Use of the map or debug log linker options can increase build time.

Maximize system resources

- Always make a backup of your system before making hardware or software changes.

For fast builds the memory usage should not be greater than the amount of physical RAM you have installed. If it does go above this value then it would be worth installing more RAM.

If you are using a recent Windows version (e.g. Windows 2000) you can check how much memory is being used by following the instructions below:

1. Run the Task Manager (Press <Ctrl+Alt+Del> then click on **Task Manager**) during your build.

2. Click the **Performance** tab and look at the memory usage.

Disabling other applications and services can greatly speed up build times. For example:

- Close all file browser and web browser windows. File browsing windows that view files being updated or created can slow down a build considerably.

- Close your e-mail client and personal web server.

- Close any other applications not required during the build.

- Disable sharing of your drives on the network.

- Disable virus checkers.

Note: Always ensure any changes you make are acceptable to your system administrator, especially if disabling a virus checker.

Good disk housekeeping may help reduce build times. As the disk gets fuller or more defragmented, build times can increase.

- Wherever possible, delete all unused files, especially object files in old projects. Using a search to find all files of type "Intermediate files" does this best.

- Search for all files above a particular size or older than a particular date and see if any can be deleted.

- Delete any files in temporary folders that are no longer needed. There can be a number of these folders so search for any folder with 'temp' in the name.

- Try to keep at least 25% free space on your hard disk.

Once you have tidied your disk it is worth defragmenting it. This can take a long time and is best run overnight. Before starting defragmentation, reboot your machine. Reboot again after defragmentation has completed.

Technical support

If you have any problems using ProDG, your first port of call should be the support area of the SN Systems website (see "Updates and technical support" on page 2).

You should first try searching the Technical Library, which contains numerous documents and FAQs relating to ProDG.

Feedback to SN Systems

If your builds appear to be taking a long time then let us know. Try removing map generation, function stripping and debug information. If you get a significant speed improvement please include this in your report to us. Please provide examples wherever possible.

The following information would help us to diagnose any problems:

- Full description of PC hardware and operating system
- ProDG tools versions being used (-v option)
- Whether using Visual Studio or make; any special make options.
- Whether debug info / mapfile generation / stripping is used.

If the website Technical Library does not contain the answer to your problem, you can also send an e-mail to the SN Systems Support Team (see "Updates and technical support" on page 2).

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 8: Relocatable DLLs

Building a relocatable DLL

Using relocatable DLLs 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.exe (See "ps2dlllk command-line syntax" on page 60) 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 an 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.

Checking your DLL and ELF

The ProDG PlayStation 2 DLL checker program ps2dllcheck (see "ps2dllcheck command-line syntax" on page 62) is used to check for undefined symbols in a DLL (.rel file) or an ELF file which has been linked using ps2dlllk. 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.

- You can use ps2dllcheck to find function calls which may have been accidentally misspelt in your source, as these will also be listed as undefined symbols.

ps2dlllk command-line syntax

The ps2dlllk DLL linker command-line syntax is as follows:

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

Example:

```
ps2dlllk physics.lk ps2cc -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.

Limitations

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.

  Note that the relapp.cmd must include all of the sections used, otherwise ps2dlllk will generate a warning.

Files required

ps2dlllk.exe	The ProDG PlayStation 2 DLL linker
libsn.a	The SN Systems library: 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

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

Script files

The script file is a sequence of commands. White space (spaces and tabs) is 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 "DLL Management" on page 63.

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 an 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.

DLL Management

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 bytes 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.
SN_DLL_INVALID_SYMTYPE	7 - Symbol type invalid - DLL file is corrupt or libsn.a needs updating.
SN_DLL_INVALID_DEFINE_ GLOBAL_FAILED	8 - Failure defining a global symbol. Most likely cause is that there is a breakpoint on an address that needs to be patched.
SN_DLL_INVALID_DEFINE_ ABS_FAILED	9 - Failure defining an absolute symbol. Most likely cause is that there is a breakpoint on an address that needs to be patched.
SN_DLL_INVALID_PATCH_ FAILED	10 - Invalid patch type - DLL file is corrupt or libsn.a needs updating.
SN_DLL_INVALID_REMOVE_ RELMOD_FAILED	11 - The DLL has not been loaded so can't be moved.

DLL 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 c:\usr\local\sce\ee\sample\vu1\blow\physics.rel
       ; make sure we supply all routines
       ; needed by physics.rel
       ; - need full path so debugger knows
       ; where to look for the ELF file
```

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.

Chapter 9: Utilities

blankelf - blank ELF utility

The purpose of BLANKELF is to fill the code sections with zero within an ELF file. It is used when customers want to send us an ELF file that has something wrong with its debug or symbol data, but they don't want to let a fully working ELF out of the office. You can use BLANKELF to zero all the code, which renders the ELF useless. However, even with no runnable code, the SN Systems support team can load the ELF and see what is wrong with its debug or symbol data.

Note: The BLANKELF utility overwrites the code in the ELF file in place. It does not make a copy of the file!

To blank an ELF file

1. Copy the BLANKELF.EXE program to your development directory.
2. Copy the ELF file to somewhere safe - IMPORTANT!
3. From the command prompt type BLANKELF <filename>, where <filename> is the name of your ELF.
4. The ELF file will now have only debug and symbol information.

iopfixup

The iopfixup utility performs the last part of an IRX link process - doing relocations, patching up the debug entries, adding the symbols needed for IOP execution, and adding the .iopmod section. This section contains information for loading the IRX file and running it. The program only has one input, which is the partially linked ELF object file.

Usage: `iopfixup <options> <outputfile> <inputfile>`

where options can be:

```
       -v                  Verbose mode
       -o <outputfile>     Convert without debug symbols
       -r <outputfile>     Convert with debug symbol info
       -e entry_symbol     Entry point specification symbol
       -m <.irx file>      Display module name and version
                           number
```

ioplibdump

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

Usage: `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.

ioplibgen

The ioplibgen utility produces ILB files and library stubs from Sony IOP TBL (.tbl) table files.

Usage: `ioplibgen <inputfile(.tbl)> <options>`

where <options> can be:

```
        -e <entry_table_source(.s)>
        -d <stub_ilb_data(.ilb)>
        -s <stub_source>
```

To create the entry table source for the TBL file the syntax is as follows:

```
ioplibgen <inputfile(.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 <inputfile(.tbl)> -d <stub_ilb_data(.ilb)>
```

ioplibld

The ioplibld utility creates an assembly stub source file by locating undefined symbols in the object file, then searching for the corresponding entries in resident library files (IRX modules already loaded into the IOP, e.g. stdio, cdvd-driver). When it resolves a symbol it adds jump code pointing to it in the stub.

Usage: `ioplibld -s <stub_source> <objectfiles>:<stub_ilb_data>`

The resident libraries are searched by using ILB tables. These are supplied by Sony for the standard modules but for custom modules there is the ioplibgen utility which generates both ILB tables and the library calling stub, which is linked in during the partial link (see "ioplibgen" on page 68). For Sony supplied modules these calling stubs are located in libraries linked during the partial link.

make

The program make.exe is the standard build utility. It interprets the input / output dependency relationships specified in a makefile.

The default make.exe provided with ProDG is case sensitive, i.e. filenames specified in the makefile and filenames on your development PC must match in a case-sensitive way. However you can download from the SN Systems website (downloads page for ProDG for PlayStation 2) a case-insensitive version of the make utility v3.78.1. This enables you to use make with files whose names do not match the case of filenames specified in the makefile.

For detailed information on makefiles and the make utility, see the GNU documentation at **www.gnu.org**.

snarl - SN Archive Librarian

The **SN AR**chive **L**ibrarian (`snarl`) follows the same usage as GNU ar.

You can find general information on controlling ar from the command line and from MRI scripts in the GNU documentation at **www.gnu.org**.

The following sections document some of the additional features that snarl offers.

Library reading

The latest version is now capable of reading Metrowerks Codewarrior format libraries. You can display information on these libraries (e.g. display the contents, etc.) and preserve the Metrowerks format, but performing any manipulation commands will convert the library into the GNU standard which is fully compatible with our tools.

If you wish to just convert the library from Metrowerks format to GNU format without having to change its contents, then just use the **rebuild symbol table** option ('s'), e.g.:

```
snarl s mwformat.a
```

Cross-platform

Snarl is capable of reading and creating libraries for most platforms (with the exception of Win32). It has been tested with Nintendo GameCube, Nintendo Game Boy Advance, and Sony PlayStation 2.

Libraries can also be created that contain object files built for different platforms. For example, a library sky.a could contain the objects ngcsky.o, agbsky.o and ps2sky.o. These could then contain functions such as NGC_getskycoords(), AGB_getskycoords(), and PS2_getskycoords() respectively, making library maintenance in a cross-platform game easier to manage.

Command-line syntax

Snarl allows you to create, modify and extract from archives. In this context an archive is usually a library, for example libc.a.

Usage:

```
snarl [-]key[mod [relpos] [count]] archive
[file(s)...] [symbol(s)...]
```

`snarl --help` Print list of commands

Where *`key`* must be one of the following:

d[NWT]	Delete [file(s)] from the archive
m[abNWT]	Move [file(s)] found in the archive
p[N]	Print [file(s)] found in the archive
q[cfSWT]	Quick append [file(s)] to the archive
r[abucfSWT]	Replace existing or insert new [file(s)] into the archive
s[WT]	Rebuild symbol table (ranlib) (performed by default)
t	Print table of contents for [file(s)] in the archive
h[N]	Set mod times of [file(s)] in the archive to current time
w[l]	Display the archive symbol table
F[lN]	Display complete symbol table for [file(s)]
x[oCN]	Extract [file(s)] from the archive

Or one of the following symbol manipulation commands:

G	Make [symbol(s)] global
W	Make [symbol(s)] weak
L	Make [symbol(s)] local

And the available 'mods' are:

a	Add file(s) after [relpos] archive member
b	Add file(s) before [relpos] archive member
c	Do not warn if the archive had to be created
C	Do not allow extracted files to overwrite existing files
f	Truncate inserted filenames into 8.3 format
l	Demangle c++ symbol names (if demangle.dll is available)
N	Specify instance [count] of same filename entries in archive
o	Preserve original date
S	Suppress building of symbol table (it is built by default)
T	Warn if symbols are multiply defined
u	Only replace archive members if [file(s)] are newer
v	Verbose mode
V	Show version

Verbose mode

Appending a 'v' onto most snarl keyletters will enable verbose mode. One of the most useful ways of using this is with the **print table of contents** ('`t`') command; this will force extended information to be displayed about the library's contents.

Print archive files

Snarl can be used to create archives of any file type (obviously only object files will be included in the archive symbol table that is used at link time). So for example a text file containing the version history can easily be stored within the library.

This could then be viewed by using the **print archive files** ('p') command, e.g.:

```
snarl p test.lib versions.txt
```

Printing object files

Other implementations of library archivers do not handle **print archive files** ('`p`') very gracefully when object files are specified. Snarl will still display text-based files as usual, but will automatically switch to a formatted hex dump if it detects an object file. This is sent to stdout so it can be easily redirected to a file using the '>' DOS redirect command, e.g.:

```
snarl p test.lib obj1.o obj2.o > out.txt
```

Displaying the symbol table

The 'w' keyletter dumps the archive symbol table to stdout. This shows which symbols are visible to the linker and what object file they reside in within the library.

Versions 1.3.3.95 onwards support demangling C++ symbol names.

The 'F' keyletter dumps out the entire object file symbol tables to stdout (from version 1.4.2.9). This enables all symbols in the library to be viewed along with their type and scope. It is recommended you pipe this output to a file as it can be quite substantial:

```
snarl F test.lib > out.txt
```

Super fast append

If you are building a library in stages (i.e. not just adding all of the object files at once) then you can take advantage of the "Super Fast Append" feature. This is activated when you specify a quick append ('q') and suppress the building of the symbol table ('S'). This is only effective when working with large libraries (>20MB). It will provide no real benefit with smaller ones.

It works by appending the new file to the library without loading in the existing data. Note that to make the library functional, the symbol table must be built once you have completed all of the desired operations.

The following example shows how to speed up the appending of four object files to a large library:

Usual implementation:

```
snarl q test.lib obj1.o obj2.o
...
snarl q test.lib obj3.o
...
snarl q test.lib obj4.o
```

Super Fast Append implementation:

```
snarl qS test.lib obj1.o obj2.o
...
snarl qS test.lib obj3.o
...
snarl qS test.lib obj4.o
snarl s test.lib            // rebuild symbol table
```

In the first example the time taken following each append is *N* (where *N* is some time delay proportional to the size of the existing library). Making a total time of 3*N* seconds.

In the second example the time taken following each append is ≈0 seconds, and the time taken following the final rebuild of the symbol table is *N*. Making a total time of *N* seconds.

Note: If any of the filenames of the object files to be added are more than 15 characters in length, then the fast append will be cancelled as the extended filename section will have to be rebuilt. You can get around this by using the 'f' modifier which will shorten all inserted filenames into 8.3 DOS format, e.g.: `snarl qSf test.lib "filename that is longer than 15 characters.obj"`

Warning of multiply-defined symbols

The 'T' modifier warns if multiply-defined symbols are present. It can be simply added to any existing key command. For example to perform a quick append and warn for multiply-defined symbols use:

```
snarl qT test.lib obj1.o obj2.o obj3.o obj4.o obj5.o
```

If you just want to list any multiply-defined symbols without actually manipulating the library, then you can specify 's' as your key command and 'T' as your modifier. This will rebuild the archive symbol table and warn (without changing the library contents). e.g.:

```
snarl sT test.lib
```

This feature is automatically activated if you are building a library with the Visual Studio Integration, and any warnings are displayed in the Visual Studio build window.

Symbol manipulation commands

There are three Symbol Manipulation Commands: "Make Global" (G), "Make Weak"(W), and "Make Local"(L), which can be used to modify symbol properties within a library (from version 1.4.2.9)

For example, the following changes 'sym1' in test.lib to a weak symbol:

```
snarl W test.lib sym1
```

You can specify several symbols on one command line, for example:

```
snarl G test.1 sym1 sym2 sym3
```

NOTE that if these commands are used on their own then they will just alter the symbol properties in the object file containing the symbol within the library. They will not update the archive symbol table, i.e. if you change a local symbol to a global then it will be not be visible to the linker until the archive symbol table is rebuilt. You can of course do this at the same time by specifying the 's' argument:

```
snarl Gs test.1 sym1 sym2 sym3
```

Response file scripting

Snarl supports simple response files, for example:

```
snarl lib.a @response.txt
```

where response.txt is a text file in the format

```
object1.o
object2.o
object3.o
object4.o
```
[etc.]

Note that using a response file will automatically delete any existing archive of the same name and build a new one (appending is not possible). If you want to do anything more complex use MRI scripting (see "MRI scripting" on page 74).

MRI scripting

Snarl supports ar's MRI scripting format documented at:

http://www.gnu.org/manual/binutils/html_chapter/binutils_1.html - SEC3

It also offers a number of additional features (from version 1.4.2.9).

SN Systems extra features

GLOBAL symbol, symbol, ... symbol	Make each listed symbol from the current archive global; equivalent to 'snarl G archive symbol ... symbol'. Requires prior use of OPEN or CREATE.
WEAK symbol, symbol, ... symbol	Make each listed symbol from the current archive weak; equivalent to 'snarl W archive symbol ... symbol'. Requires prior use of OPEN or CREATE.
LOCAL symbol, symbol, ... symbol	Make each listed symbol from the current archive local; equivalent to 'snarl L archive symbol ... symbol'. Requires prior use of OPEN or CREATE.
$(ENV)	Environment variable macro expansion. Any macros used in this format will be expanded into the value of the specified environment variable upon execution, e.g.: open $(LIB_DIR)\lib.a

Running MRI scripts from the command line

```
snarl -M [<mri-script>]
```

Use snarl -M? to display MRI script commands

If you omit the <mri-script> argument, snarl displays a command window that allows you to run the program interactively. The commands that can be entered in interactive mode are described at:

http://www.gnu.org/manual/binutils/html_chapter/binutils_1.html - SEC3

Running MRI scripts from Windows Explorer

snarl also allows users to run MRI style scripts directly from Windows Explorer.

To run MRI scripts from Windows Explorer

1. Enable file associations with snarl and the .sns (snarl script) extension, using the command 'snarl -E'.
2. Save all scripts in plain text with the .sns filename extension.
3. Double-click the saved script file to automatically execute it.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Chapter 10: Building for EE profiling

How to support EE profiling in your game

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 ProDG Debugger profile pane will do the rest. Your PlayStation 2 program should run at pretty much full speed.

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 initialize 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 significant 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);
```

```
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 that 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 i.e. "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 will be accepted but flag values of 4, 5, 6 etc. will be rejected.

```
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 that 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.

```
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 that can be selectively filtered by the profiler. This function can only clear additional bits, it will never set them. Bits that 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.

Profiling example

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 <drive>:\usr\local\sce\iop\modules directory along with your usual IOP modules).

2. Edit your main source file, i.e. the one that 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.

```
main()
{
static u_long128 profdata[2048]; // quadword aligned, can be 2K
                                // to 64K bytes
sceSifInitRpc(0);
// Load the SNProfil module
if(sceSifLoadModule("host0:/usr/local/sce/iop/modules/SNProfil.
irx", 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 and debug your program.

Chapter 11: Building IOP modules

Building an IRX

Use the ProDG compiler driver's -iop switch to build an IRX file instead of the default ELF file.

The compilation and assembly to object (.o) files are standard MIPS/R3000. All generated object files are scanned by ioplibld (see "ioplibld" on page 69), which then generates a library stub. This library stub, which resolves dynamic library references made from the initial .c source (e.g. printf etc.), is then assembled to generate a stub object file.

The stub object file generated by ioplibld (_ilb_stub.o) is then linked with the original object file(s) from the C source to form the intermediate rel object file. This is almost identical to an ELF file but has not had any address fixup performed.

The iopfixup utility then performs all the standard link operations (i.e. fixup) needed to create an executable, plus some special operations, mainly involving inserting an IOP header section needed to create an IRX file (see "iopfixup" on page 67).

Compiling C++ files for the IOP

The SN IOP C++ compiler is built using the 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 83 and "C++ global constructors and destructors" on page 83.

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 has 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/mipsel-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 libraries, 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 that 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)||defined(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)||defined(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)||defined(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)||defined(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 initialize 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.

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: This is software emulation and is not really suitable for use in release code that needs to run quickly. Note that the IOP printf does not support floating point output. If you use these types, you must link with libsniop.a, by adding '-lsniop' to your makefile or Visual Studio integration link stage.

Troubleshooting

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.

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Appendix A: Assembler reference

Unsupported assembler options

The following assembler options 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

Unsupported assembler directives

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

Appendix B: Linker reference

Unsupported ld switches

All the standard ld command line switches are recognised by ps2ld but it 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 will generate a warning if used:

```
OUTPUT_ARCH
OUTPUT_FORMAT
```

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

```
HLL
SYSLIB
VERSION
TARGET
```

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

```
CONSTRUCTORS
NOCROSSREFS
MEMORY
SORT
KEEP
CREATE_OBJECT_SYMBOLS
```

Error messages

The ProDG linkers produce numbered error messages. Some messages are generic to linkers built for all consoles, while others are specific to one platform.

Generic error messages

The following errors may be generated by all ProDG linkers.

Error	Description

L0001	ReadFile() failed when reading from "<FILE>"
L0002	Failed to SetFilePointer("<OFFSET>")
L0003	Failed to SetFilePointer(<OFFSET>,0,NULL,FILE_CURRENT)
L0004	Symbol '<SYMBOL>' was not found in <FILE>(<OBJECT>)
L0005	Unexpected token
L0006	Failed to WriteFile()
L0007	-N option not supported
L0008	-omagic not supported
L0009	-n not supported
L0010	-nmagic not supported
L0011	-export_dynamic_symbols not supported
L0012	-E not supported
L0013	Failed to read .line info in file <FILE>
L0014	Did not find <SYMBOL> in Elf sym table in file <FILE>
L0015	System error <NUMBE>:<MESSAGE>
L0016	Symbol <SYMBOL> not defined in a small data section but could be short referenced
L0017	Short reference to symbol "<SYMBOL>" from <FILE>
L0018	Short reference to symbol "<SYMBOL>" from <FILE>
L0019	Symbol '<SYMBOL>' multiply defined
L0020	Common symbol <SYMBOL> is multiply defined
L0021	Second start-group without end-group
L0022	SORT(CONSTRUCTORS) illegal as a KEEP operand - ignored
L0023	OUTPUT_FORMAT ignored
L0024	OUTPUT_ARCH ignored
L0025	Lots of sections are not found in linker script e.g. <SECTION NAME>
L0026	Unable to find a similar section to "<SECTION NAME>". Adding it to end of link
L0027	Debug file specified twice. Ignoring "-debug <FILE>"
L0028	The <OPTION> option has been sent directly to the linker
L0029	end-group without start-group
L0030	Ignoring bss attribute
L0031	Could not find script file <FILE>
L0032	Could not open file <FILE>
L0033	Entry point label <SYMBOL> not defined
L0034	Error while outputting section "<SECTION NAME>"
L0035	Expecting (after file specification

L0036	Failed to open command line file <FILE>
L0037	Missing {
L0038	No input files
L0039	Reference to undefined symbol <SYMBOL> in file <FILE>
L0040	No input objects containing code
L0041	Syntax error
L0042	Syntax error in expression
L0044	<NAME> is not a recognised program header
L0045	Bad patch type 0x<NUMBER> in file <FILE>
L0046	Could not make reference to local symbol <SYMBOL>
L0047	Could not make reference to global symbol <SYMBOL>
L0048	Failed to open Elf file '<FILE>'
L0049	Reference to undefined section number 0x<NUMBER> in file <FILE>
L0050	Attempt to write outside buffer
L0051	Unknown Relocation type
L0052	Error reading lib. Expected a "/" char in module name "<MODULE>" in lib <FILE>
L0053	Could not find symbol table in library <FILE>
L0054	Could not find required string table in library <FILE>
L0055	Could not open output file <FILE>
L0056	Unknown global symbol type <NUMBER>
L0057	Could not open debug output file <FILE>

ps2ld specific error messages

The following table lists error messages specific to the ps2ld linker.

Error	Description
L2000	Failed to find storage class for section <NAME> in file <FILE>
L2003	Failed to find symtab

Index