User Guide to

ProDG Build Tools for PlayStation®2

SN Systems Ltd
Version 2.00
March 2001

ProDGBuildToolsforPS2_UG-V2.00 / ProDG Build Tools for PlayStation 2 v 2.00 / March 2001

Document change control

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

Contents

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

Preface

Overview of ProDG for PlayStation 2

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

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

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

Currently ProDG for PlayStation 2 consists of:

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

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

This is the user guide to the ProDG Build Tools for PlayStation 2.

Working with the ProDG EE and IOP build tools

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

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

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

You can also invoke any of the binary utilities:

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

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

Building PlayStation 2 samples on your Win32 PC

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

Note: *Edit the demo* `makefiles` *and change the line that reads*

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

to

```
iop-path-setup
```

This is because Windows command line cannot execute the pipe command. Note also that the `iop-path-setup` *program is currently just a batch tool, which creates the correct path settings in the current directory. In a future release this tool will be replaced. However, if you do not change the positions of the build tools it will work perfectly.*

You can also build the PlayStation 2 samples by invoking the SN compiler driver `ps2cc` from the MS-DOS command line; see "Using the SN Systems compiler ps2cc" on page 5 for help with the `ps2cc` syntax.

Updates and technical support

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

http://www.snsys.com/ps2

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

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

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

Mail: SN Systems Ltd
4th Floor - Redcliff Quay
120 Redcliff Street
Bristol BS1 6HU
United Kingdom

Tel.: +44 (0)117 929 9733

Fax: +44 (0)117 929 9251

WWW: http://www.snsys.com/ps2

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

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

Chapter 1: **ProDG Compilers for PlayStation 2**

Using the SN Systems compiler ps2cc

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

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

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

```
[ps2cc]
....

#assembler_name=as
assembler_name=ps2eeas

opt_assembler_name=as
#opt_assembler_name=ps2eeas

#dvp_asm_name=ee-dvp-as
dvp_asm_name=ps2dvpas

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

iop_asm_name=as
#iop_asm_name=ps2iopas

#linker_name=ld
linker_name=ps2link

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

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

```
iop_linker_name=ld

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

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

....
```

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

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

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

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

How ps2cc interprets input files

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

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

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

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

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

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

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

Options

Once `ps2cc` has interpreted the relevant *file type*, any compiler *options* are specified in the command line – each one preceded by a minus sign (-).

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

Process control and output

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

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

C/C++ language options

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

Warning options

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

Debugging options

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

Optimization options

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

Preprocessor options

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

Assembler option

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

Linker options

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

Machine-dependent options

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

Using a response file

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

```
ps2cc @myresponsefile
```

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

Examples

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

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

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

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

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

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

Compiling C++ files for the IOP

The SN IOP C++ compiler is built using the latest 2.95.2 source code. It slots into and works directly with the debugger/VSI (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 11 and "C++ global constructors and destructors" on page 12.

Installing SN IOP C++ Compiler

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

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

   ```
   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) eg

   ```
   set PS2_DRIVE=c
   ```

   (Note that there is no colon-slash after the drive letter)

   And add the following two lines:

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

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

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

Setting up install directories

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

Building IOP demos

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

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

to

```
iop-path-setup
```

This is because DOS cannot execute the command as written.

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

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

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

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

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

C++ global constructors and destructors

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

```
...
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
extern "C" {
#endif
#include <stdio.h>
#include <kernel.h>
#include <sysmem.h>
#include <sif.h>
#if
defined(_LANGUAGE_C_PLUS_PLUS)||defined(__cplusplus)||define
d(c_plusplus)
}
#endif
#include <libsniop.h>  //Contains definitions for CTOR and
DTOR calls.
#define BASE_priority  32
class foo
{
public:
 int hoo;
 int bar;
 foo () {hoo = 40; printf("\nfoo Constructor");}
 ~foo () {hoo = 0; bar = 0; printf("\nfoo Destructor.");}
};
foo globfunc;  //Global object of class foo
ModuleInfo Module = { "cxx_fiddling", 0x0101 };
int thread1(void)
{
 printf("\nThread1 startup.");
 printf("\nCall Global Ctors.");
 SN_CALL_CTORS;  //Call all global CTORs
 globfunc.hoo = 32;
 globfunc.bar = 256;
 printf("\nCall global Dtors.");
 SN_CALL_DTORS;  //Call all global DTORs
 printf("\nStuff happening after global destructor cAll");
 return 0;
}
int start (void)
{
    struct ThreadParam param;
    int thread;
    printf("\nStartup thread init.");
    CpuEnableIntr();
    if (!sceSifCheckInit())
    {
 sceSifInit();
    }
    param.attr         = TH_C;
    param.entry        = thread1;
    param.initPriority = BASE_priority - 2;
    param.stackSize    = 64*1024;
    param.option       = 0;
    thread = CreateThread (&param);
    if (thread > 0)
    {
 StartThread (thread, 0);
 printf ("\nThread 1 started.");
 printf ("\nStartup thread terminated.");
 return 0;
    }
    else
    {
 printf ("\nStartup thread terminated.  Errors
encountered");
 return 1;
    }
}
...
```

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

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

IOPFIXUP error: unresolved symbols

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

Doubles and float software emulation

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

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

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

Using the ioplibdump utility

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

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

The syntax is as follows:

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

For example:

```
ioplibdump cxtmdm.irx
```

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

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

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

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

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

Using the ioplibgen utility

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

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

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

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

Chapter 2: **ProDG Assemblers for PlayStation 2**

Using the SN Systems Assemblers

The ProDG build tools for PlayStation 2 include the three SN Systems assemblers: `ps2eeas`, `ps2iopas` and `ps2dvpas`. The EE and IOP assemblers can be invoked directly from the EE or IOP (`ee-gcc`, or `iop-elf-gcc`) compilers using the `–snas` command line option. However, the SN Systems VU assembler `ps2dvpas` is used on the command line to assemble `.dsm` files that have been written in VU microcode (see “The ProDG VU assembler ps2dvpas” on page 23).

Specifying options

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

The command-line options and directives for the SN Systems' assemblers are compatible with the GNU assemblers except for some GNU command-line switches which are ignored or produce errors. See “Unsupported switches and directives” on page 19, "***Appendix A: Macro assembler*** *reference*" on page 53 and "***Appendix B:*** *Command line switches for assemblers*" on page 97.

In addition, if you specify the `-sn` option, you will have access to the SN Systems directives. See “SN Systems directives” on page 21.

Command line syntax

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

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

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

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

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

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

Input files

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

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

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

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

If you give the assembler no file names 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 17.

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 file names help error messages reflect the original source file, when the assembler source is itself synthesized from other files. See ".app-file string" on page 55.

Output (object) file

Every time you run the assembler it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`. You can give it another name by using the `-o` option. Conventionally, object file names end with `.o`.

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

Error and warning messages

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

Warning messages have the format

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

(where `NNN` is a line number). If a logical file name has been given (see ".app-file string" on page 55) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see ".line line-number" on page 73) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed.

Error messages have the format

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

The file name and line number are derived as for warning messages.

Symbol names

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

Case sensitivity is also significant: `foo` is not the same as `Foo`.

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

Local symbol names

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

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

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

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

`L`

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

digit

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

C-A

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

ordinal number

This is a serial number to keep the labels distinct. The first `0`: gets the number `1`; The 15th `0`: gets the number `15`; *etc.*. Likewise for the other labels `1`: through `9`:.

For instance, the first `1`: is named `L1C-A1`, the 44th `3`: is named `L3C-A44`.

The special dot symbol

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

Symbol attributes

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

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

Value

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

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

Type

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

Unsupported switches and directives

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

Switch	How it is handled
`-a`	Ignored
`-D`	Ignored
`-f`	Ignored
`--itbl`	Generate an error
`-J`	Ignored

-K	Ignored
--listing-lhs-width	Ignored
--listing-lhs-width2	Ignored
--listing-rhs-width	Ignored
--listing-cont-lines	Ignored
-membedded-pic	generate an error
-nocpp	Ignored
-R	generate an error
--traditional-format	Ignored
-w	Ignored
-X	Ignored
-Z	Ignored

See "***Appendix B:*** *Command line switches for assemblers*" on page 97 for a complete list of assembler switches.

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

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

`.noformat`	Ignored
`.nolist`	Ignored
`.nopage`	Ignored
`.org`	Ignored
`.plen`	Ignored
`.rva`	Ignored
`.sbttl`	Ignored
`.spc`	Ignored
`.struct`	Ignored
`.title`	Ignored

SN Systems directives

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

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

Directive	Description
`.endscope`	See `.scope` (below)
`.equr newreg, reg`	Specifies an alternative name for a register See "Naming registers and register fields" on page 22.
`.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 22.

See "***Appendix A:** Macro assembler reference*" on page 53 for a complete list of assembler directives.

Naming registers and register fields

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

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

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

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

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

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

Scope delimiting

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

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

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

```
Scopes can be nested.

e.g.

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

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

The ProDG VU assembler ps2dvpas

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

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

The following table lists all of the available options:

Options	Actions
`-a[list-options]`	Produce listing (not implemented)
`--defsym sym=value`	Define integer symbol
`-G<size>`	Set size of data items placed in `.sdata/.sbss`
`--gstabs`	Produce STABS debug info
`--help`	Print help
`-I <directory>`	Specify directory to search for include files
`-L`	Output local symbol information
`--keep-locals`	Same as –L
`-o <output file>`	Specify output filename
`-sn`	Activate SN Systems extensions See “SN Systems directives” on page 21 for further information
`--version`	Print version information
`-W`	Disable warnings

The following list briefly describes all of the default directives that are understood by `ps2dvpas`. This does not include the standard GNU directives or the VU opcodes, some of which can be found in "***Appendix A:** Macro assembler reference*" on page 53 or in the Sony documentation.

Directive	Description
`.data`	Switches to the `.vudata` section
`.dmadata`	Labels a block of DMA data
`.dmapackvif`	Flags whether the first part of DMA data should be packed in with the `dma` tag
`.enddirect`	Ends a VIF `direct` or `directhl` block
`.enddmadata`	Ends a DMA controller operation
`.endgif`	Ends a GIF operation
`.endmpg`	Ends a VIF `mpg` operation
`.endunpack`	Ends a VIF unpack operation
`.quad`	Specifies 128 bit data words
`.text`	Switches to the `.vutext` section
`.vu`	Switches to VU opcode mode
`.word`	Specifies 32 bit data word

DMA/VIF/GIF operations

The following sections list the DMA, VIF and GIF operations that are supported by `ps2dvpas`. See "***Appendix A:** Macro assembler reference*" on page 53 for further information.

DMA controller operations

`Dmacall`
`Dmacnt`
`Dmaend`
`Dmanext`
`Dmaref`
`Dmarefe`
`Dmarefs`
`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
Gifpacked
Gifreglist

[THIS PAGE IS LEFT BLANK INTENTIONALLY]

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

Introduction

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

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

Linkers included in ProDG for PlayStation 2

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

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

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

- `ee-gcc` cannot automatically call `ps2link`. However, our replacement compiler driver `ps2cc` can automatically call `ps2link`. See "Using the SN Systems compiler ps2cc" on page 5.

- You need to change your makefile to use this linker. See “Calling ps2link instead of ld“ on page 30.
- We include a standard `ps2link`-format linker script which works for all the samples. This script is installed as `ee/lib/ps2.lk`. See “Example linker script“ on page 41.

ProDG for PlayStation 2 also includes `ps2ld`, which will eventually unify the functionality of `ee-ld` and `ps2link`. This release only supports `ee-ld` command line options and script files but future releases will support `ps2link` syntax also. The major advantage in using `ps2ld` however, is that it is considerably faster than either `ee-ld` or `ps2link`. See "Using ps2ld" on page 50.

Benefits of ps2link

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

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

Features

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

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

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

ps2link system requirements

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

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

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

Under Windows 98:

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

Under Windows 2000:

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

Sections and groups

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

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

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

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

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

Replacing the GNU linker ld with ps2link

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

- You must write a new linker script, which may be done by modifying the example linker script `ps2.lk` (see "Example linker script" on page 41). This is installed in `\usr\local\sce\ee\lib` if you installed the ProDG EE build tools.
- You must ensure that every section is placed in a group in your linker script. For more information about the linker scripts (see "Linker control script language" on page 34).
- Write or modify your existing makefile so that `ps2link` is called instead of the GNU PlayStation 2 `ld` linker. This makefile automatically finds the `ps2.lk` file in its default location (see below).

Calling ps2link instead of ld

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

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

```
SHELL        = /bin/sh

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

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

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

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

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

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

.SUFFIXES: .c .s .cc .dsm

all: $(TARGET).elf

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

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

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

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

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

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

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

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

Invoking the linker

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

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

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

`ORG` start and end addresses and sizes. There is also a list of the symbols in your program. This list appears twice, once sorted by name and once by address.

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

ps2link command-line switches

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

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

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

Link case-insensitively (`-ci`)

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

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

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

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

For example:

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

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

Specify symbol as program entrypoint `-entry symbol`

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

Enable exceptions `-exceptions`

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

Specify small data threshold `-G n`

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

Add library search path `-l path`

The path that you specify using the `-l` switch is added to the set of paths that `ps2link` uses to search for libraries (see "INCLIB statement" on page 39). `ps2link` always tries to locate files relative to the current directory first.

Add include search path `-li path`

The path that you specify using the `-li` switch is added to the set of paths that `ps2link` uses to search for object files (see "INCLUDE statement" on page 39). `ps2link` always tries to locate files relative to the current directory first.

Produce static coverage report `-st`

This switch tells the linker to produce a report on unused functions and variables in your image. See "Dead-stripping" on page 49 for more details.

Strip unused elements from the program `-strip`

This switch tells the linker to remove unused functions and variables from your image. See "Dead-stripping" on page 49 for more details.

Treat warnings as errors `-we`

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

Detect duplicate C variable definitions `-wm`

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

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

Linker control script language

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

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

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

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

Sections and groups

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

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

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

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

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

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

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

Script syntax

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

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

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

These are the commands recognised in the script language:

`INCLUDE filename[,prefix]`

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

`INCLIB filename[,prefix]`

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

`ORG address`

Set the ORG address to `address`.

`Name EQU value`

Define a symbol with the given `value`.

`Name GROUP attributes`

Define `name` as a group with the specified attributes.

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

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

`SECTALIGN number`

Set the default section alignment for all subsequent sections.

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

Command reference

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

ORG statement

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

```
ORG <address>
```

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

GROUP statement

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

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

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

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

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

group

This directive creates a group.

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

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

group attributes

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

bss

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

SECTALIGN statement

This directive specifies the default alignment for subsequent section directives.

```
sectalign <alignment>
```

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

Sections are normally aligned using the following rules:

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

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

SECTION statement

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

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

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

`sectionname` is the name of the section to recognise.

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

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

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

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

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

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

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

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

INCLUDE statement

This directive includes an object file in your game.

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

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

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

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

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

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

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

You must write an `INCLUDE` statement for each object file in your game.

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

INCLIB statement

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

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

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

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

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

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

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

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

EQU statement

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

```
    newname    equ    expr
```

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

Section and group descriptors

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

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

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

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

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

Example linker script

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

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

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

;      The heap size and stack details are defined here.

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

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

        org 0x00100000

indata  group

        section *.indata,indata

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

        org 0x00200000

text    group

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

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

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

        section *,text

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

_gp     equ     __lit8_obj+0x7FF0
```

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

;       This group is for uninitialized data

bss     group   bss

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

_fbss   equ     _bss_obj

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

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

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

_end    equ     _bss_objend

;       This group is for the scratchpad.

        org 0x70000000

spad    group

        sectalign 4

        section .spad,spad
```

The linker and libraries

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

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

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

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

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

Building relocatable DLLs

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

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

Invoking the DLL linker

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

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

e.g.

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

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

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

Some restrictions that currently apply are :

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

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

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

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

Files required

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

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

Script file for ps2dlllk

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

The script file syntax for `ps2dlllk` is :

```
; comment
```

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

```
.main
```

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

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

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

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

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

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

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

`.reference symbol-name`

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

`.resolve dll-file-name`

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

`.nodebug`

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

`.index symbol-name index-number`

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

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

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

Associated library functions

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

`int snInitDllSystem (void** index_pointer);`

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

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

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

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

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

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

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

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

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

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

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

Example

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

Modifications to the makefile

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

```
OBJS = crt0.o \

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

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

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

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

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

4. In the build rule for the main program:

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

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

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

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

   The file `blow.lk` contains the two lines

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

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

   ```
   physics.elf: physics.o

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

   physics.o -nostartfiles
   ```

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

Modifications to blow.c

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

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

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

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

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

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

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

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

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

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

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

The DLL checker

The ProDG PlayStation 2 DLL checker program `ps2dllcheck` is used to check for undefined symbols in a DLL (`.rel` file) or PlayStation 2 EE executable (`.elf` file), which has been linked using the ProDG DLL Linker for PlayStation 2, `ps2dlllk` (see "Building relocatable DLLs" on page 43). Provided the DLL and `.elf` file have been built correctly, all of the undefined symbols in the `.elf` should refer to functions in the DLL, and *vice versa*.

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

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

ps2dllcheck command-line syntax

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

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

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

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

Displaying undefined symbols

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

```
Undefined symbols :
SetParticle Position
```

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

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

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

```
No undefined symbols
```

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

Dead-stripping

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

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

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

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

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

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

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

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

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

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

Additional notes

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

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

Using ps2ld

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

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

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

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

Building with ps2ld

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

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

Additional command line switches in ps2ld

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

Unsupported command line switches in ps2ld

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

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

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

Unsupported script file directives in ps2ld

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

```
OUTPUT_ARCH

OUTPUT_FORMAT

CONSTRUCTORS

HLL

SYSLIB

VERSION

TARGET
```

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

```
PHDRS

NOCROSSREFS

MEMORY

SORT

KEEP

CREATE_OBJECT_SYMBOLS
```

Appendix A: **Macro assembler reference**

Macro assembler directives

The assembler provides a wide range of directives, as well as built-in functions and variables, to control the assembly of the source code and its layout in the target machine. This section describes their use and operation.

Here are some general notes about this reference section.

1. Directives, functions and variables are listed in alphabetical order.
2. Directives can be specified in either all uppercase or all lowercase.
3. If the syntax for a directive does not make explicit reference to the format of the label field, then a normal label may appear there. There may be a special format for the label or a label might not be allowed at All
4. In directives requiring a quoted string as an operand, you can use either single-quote (`'`) or double-quote (`"`) characters to enclose the string but the same type of quote character must be used to mark both the start and end of each individual string.
5. You can use the sequence `\^x,` where `x` is any alphabetic character, to represent the appropriate control character in a string. For example, `\^A` is read as `<Ctrl+C>`, or ASCII code 0x01.
6. Where appropriate a description is given for each directive, together with one of the following categories: **All** – directive supports all PlayStation2 processors, **DVP** – directive supports Vector Unit processors, **MIPS** – directive supports MIPS processors, **SN Systems** – directive only supports SN Systems assemblers. In addition, the assembler(s) the directive supports are also shown.

.abicalls

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will generate an error if used.

.abort

Category All

Description This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive to tell the assembler to quit also.

Supported By All

See Also `.ABORT`

.ABORT

Category All

Description When producing COFF output, the assembler accepts this directive as a synonym for .abort. When producing `b.out` output, the assembler accepts this directive, but ignores it.

Supported By All

.align abs-expr, abs-expr, abs-expr

Description Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at All
You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.
The way the required alignment is specified varies from system to system. For the a29k, hppa, m68k, m88k, w65, sparc, and Hitachi SH, and i386

using ELF format, the first expression is the alignment request in bytes. For example .align 8 advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example '.align 3' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides .balign and .p2align directives, which have a consistent behavior across all architectures (but are specific to GAS).

Supported By	`ps2dvpas, ee-dvp-as`

.app-file string

Category	All
Description	`.app-file` (which may also be spelled '`file`') tells the assembler that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes ""; but if you wish to specify an empty file name, you must give the quotes--"".
Supported By	All

.ascii "string"...

Category	All
Description	`.ascii` expects zero or more string literals (see section Strings) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.
Supported By	All
See Also	`.aciz "string"`

.asciz "string"...

Category	All

Description	`.asciz` is just like .ascii, but each string is followed by a zero byte. The "`z`" in '.asciz' stands for "`zero`".
Supported By	All
See Also	`.ascii "string"`

.balign[wl] abs-expr, abs-expr, abs-expr

Category All

Description Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example .balign 8 advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at All You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value `0x368d` (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

Supported By All

.base

Category DVP

Description VIF operation that sets the base address of the double buffer. `base` sets the lower 10 bits of the `IMMEDIATE` field to the `VIF1_BASE` register. These bits become the base address of the double buffer.

Supported By `ps2dvpas ee-dvp-as`

.byte expressions

Category DVP

Description `.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

Supported By All

.comm symbol, length

Category All

Description `.comm` declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If the linker does not see a definition for the symbol--just one or more common symbols--then it will allocate length bytes of uninitialized memory. length must be an absolute expression. If the linker sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be a power of two. If the linker allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, the assembler will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

Supported By All

.cpadd

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will generate an error if used.

.cpload

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will generate an error if used.

.cprestore

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will generate an error if used.

.data

Description Switches to the `.vudata` section.

Supported By `ps2dvpas`

.data subsection

Description `.data` tells the assembler to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.

Supported By `ps2dvpas, ee-dvp-as`

.def name

Description Begin defining debugging information for a symbol name; the definition extends until the `.endef` directive is encountered. This directive is only observed when the assembler is configured for COFF format output; when producing `b.out`, `.def` is recognized, but ignored.

`.data` tells the assembler to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.

Supported By `ps2dvpas, ee-dvp-as`

See Also `.endef`

.desc symbol, abs-expression

Description This directive sets the descriptor of the symbol (see "Symbol attributes" on page 19) to the low 16 bits of an absolute expression. The `.desc` directive is not available when the assembler is configured for COFF output; it is only for `a.out` or `b.out` object format. For the sake of compatibility, the assembler accepts it, but produces no output, when configured for COFF. Begin defining debugging information for a symbol name; the definition extends until the `.endef` directive is encountered. This directive is only observed when the assembler is configured for COFF format output; when producing `b.out`, `.def` is recognized, but ignored.

Supported By `ps2dvpas, ee-dvp-as`

.dim

Description This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def/.endef` pairs. `.dim` is only meaningful when generating COFF format output; when the assembler is generating `b.out`, it accepts this directive but ignores it. This directive sets the descriptor of the symbol (see "Symbol attributes" on page 19) to the low 16 bits of an absolute expression.

Supported By `ps2dvpas, ee-dvp-as`

See Also `.def, .endef`

.direct

Category DVP

Description VIF operation that transfers data to the `GIF(GS)`. `direct` transfers the following `IMMEDIATE` pieces of 128-bit data to the `GS` via `GIF PATH2`. It is necessary to put the appropriate `GIFtag` to the data.

Supported By `ps2dvpas, ee-dvp-as`

.directhl

Category	DVP
Description	VIF operation that transfers data to the `GIF(GS)`. `directhl` transfers the following `IMMEDIATE` pieces of 128-bit data to the `GS` via `GIF PATH2`. The appropriate `GIFtag` must be attached to the data. Interruption of `PATH3 IMAGE` mode data does not occur during the data transfer via `PATH2` by `directhl`. Furthermore, when the `IMAGE` mode data is being transferred via `PATH3`, `directhl` stalls until the end of the transfer.
Supported By	`ps2dvpas, ee-dvp-as`

.dmacall

Category	DVP
Description	DMA controller operation
Supported By	`ps2dvpas, ee-dvp-as`

.dmacnt

Category	DVP
Description	DMA controller operation
Supported By	`ps2dvpas, ee-dvp-as`

.dmadata

Category	DVP
Description	Labels a block of DMA data.
Supported By	`ps2dvpas, ee-dvp-as`

.dmaend

Category	DVP
Description	DMA controller operation
Supported By	`ps2dvpas, ee-dvp-as`

.dmanext

Category	DVP
Description	DMA controller operation
Supported By	`ps2dvpas, ee-dvp-as`

.dmapackvif

Category	DVP
Description	Flags whether the first part of DMA data should be packed in with the dma tag.
Supported By	`ps2dvpas, ee-dvp-as`

.dmaref

Category	DVP
Description	DMA controller operation
Supported By	`ps2dvpas, ee-dvp-as`

.dmarefe

Category	DVP
Description	DMA controller operation

Supported By `ps2dvpas, ee-dvp-as`

.dmarefs

Category DVP

Description DMA controller operation

Supported By `ps2dvpas, ee-dvp-as`

.dmaret

Category DVP

Description DMA controller operation

Supported By `ps2dvpas, ee-dvp-as`

.eject

Description Force a page break at this point, when generating assembly listings.

Supported By `ee-dvp-as`

.else

Category All

Description `.else` is part of the assembler's support for conditional assembly; see section `.if absolute expression`. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

Supported By All

See Also `.if`

.enddirect

Category DVP

Description Ends a VIF direct or `directhl` block.

Supported By `ps2dvpas, ee-dvp-as`

.enddmadata

Category DVP

Description Ends a DMA controller operation

Supported By `ps2dvpas, ee-dvp-as`

.endef

Category All

Description This directive flags the end of a symbol definition begun with `.def`. `.endef` is only meaningful when generating COFF format output; if the assembler is configured to generate `b.out`, it accepts this directive but ignores it.

`.else` is part of the assembler's support for conditional assembly; see section `.if absolute expression`. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

Supported By All

See Also `.def, .if absolute expression`

.endfunc

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.endgif

Category	DVP
Description	Ends a GIF operation
Supported By	`ps2dvpas, ee-dvp-as`

.endif

Category All

Description `.endif` is part of the assembler's support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See section `.if absolute expression`. This directive flags the end of a symbol definition begun with `.def`.

`.endef` is only meaningful when generating COFF format output; if the assembler is configured to generate `b.out`, it accepts this directive but ignores it.

`.else` is part of the assembler's support for conditional assembly; see section `.if absolute expression.` It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

Supported By All

See Also `.if`

.endmpg

Category	DVP
Description	Ends a VIF mpg operation
Supported By	`ps2dvpas, ee-dvp-as`
See Also	`.mpg`

.endscope

Category SN Systems.

Description Used in conjunction with .scope, this directive delimits a scope for local labels. Any label beginning with the '@' character will be local to that scope. See "Naming registers and register fields" on page 22 for sample code.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

See Also `.scope`

.endunpack

Category DVP

Description Ends a VIF unpack operation

Supported By `ps2dvpas, ee-dvp-as`

.equr newreg, reg

Category SN Systems.

Description Specifies an alternative name for a register. See "Naming registers and register fields" on page 22 for sample code.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.equ symbol, expression

Description This directive sets the value of symbol to expression. It is synonymous with '`.set`'; see section .set symbol, expression. The syntax for equ on the HPPA is '`symbol  .equ expression`'.

Supported By `ps2dvpas, ee-dvp-as`

See Also `.set`

.equiv symbol, expression

Category All

Description The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if symbol is already defined. Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

Supported By All

See Also `.equ`

.err

Category All

Description If the assembler assembles a `.err` directive, it will print an error message and, unless the `-Z` option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

Supported By All

.extern

Category All

Description `.extern` is accepted in the source program--for compatibility with other assemblers--but it is ignored. The assembler treats all undefined symbols as external.

Supported By All

.file string

Category All

Description `.file` (which may also be spelled '`.app-file`') tells the assembler that we are about to start a new logical file. `string` is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '"'; but if you wish to specify an empty file name, you must give the quotes--"".

Supported By All

See Also `.app-file`

.fill repeat, size, value

Category All

Description repeat, size and value are absolute expressions. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, to make it compatible with other assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer the assembler is assembling for. Each size bytes in a repetition is taken from the lowest order size bytes of this number. `size` and `value` are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.

Supported By All

.float flonums

Category All

Description This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`. The exact kind of floating point numbers emitted depends on how the assembler is configured.

Supported By All

See Also	`.single`

.flush

Category	DVP
Description	VIF operation that waits for the end of the microprogram. `flush` waits for the state in which transfers to the GIF from `PATH1` and `PATH2` have ended after the end of the microprogram in `VU1`.
Supported By	`ps2dvpas, ee-dvp-as`

.flusha

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.flushe

Category	DVP
Description	VIF operation that waits for the end of the microprogram. `flushe` waits for the state when the microprogram in `VU0/VU1` has ended.
Supported By	`ps2dvpas, ee-dvp-as`

.func

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.gifimage

Category	DVP
Description	GIF operation
Supported By	`ps2dvpas, ee-dvp-as`

.gifpacked

Category	DVP
Description	GIF operation
Supported By	`ps2dvpas, ee-dvp-as`

.gifreglist

Category	DVP
Description	GIF operation
Supported By	`ps2dvpas, ee-dvp-as`

.global symbol,globl symbol

Category	All
Description	`.global` makes the symbol visible to the linker. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program. Both spellings (`.globl` and `.global`) are accepted, for compatibility with other assemblers.
Supported By	All

.gpword

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will generate an error if used.

.hword expressions

Category	All
Description	This expects zero or more expressions, and emits a 16 bit number for each.

This directive is a synonym for `.short`; depending on the target architecture, it may also be a synonym for `.word`.

Supported By All

See Also `.short`

.ident

Category All

Description This directive is used by some assemblers to place tags in object files. The assembler simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

Supported By All

.if absolute expression

Category All

Description `.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by `.endif` (see ".endif" on page 64); optionally, you may include code for the alternative condition, flagged by `.else` (see ".else" on page 62).

The following variants of `.if` are also supported:

```
.ifdef symbol
```

Assembles the following section of code if the specified symbol has been defined.

```
.ifndef symbol
.ifnotdef symbol
```

Assembles the following section of code if the specified symbol has not been defined. Both spelling variants are equivalent.

Supported By All

See Also `.else`

.include "file"

Category All

Description This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I` command-line option (see "***Appendix B:** Command line switches for assemblers*" on page 97. Quotation marks are required around the file.

Supported By All

.insn

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.int expressions

Category All

Description Expect zero or more expressions, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depend on what kind of target the assembly is for.

Supported By All

.irp symbol, values...

Description Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each value, symbol is set to value, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use `\symbol`.

For example, assembling

```
        .irp    param,1,2,3
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

Supported By `ps2dvpas, ee-dvp-as`

.irpc symbol, values...

Description Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the `.irpc` directive, and is terminated by an `.endr` directive. For each character in value, symbol is set to the character, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use `\symbol`.

For example, assembling

```
    .irpc    param,123
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
    move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

Supported By `ps2dvpas, ee-dvp-as`

.itop

Category DVP

Description VIF operation that sets the data pointer. `itop` sets the lower 10 bits of the `IMMEDIATE` field to the `VIFn_ITOPS` register. This value is read from the VU by the `xitop` instruction.

Supported By `ps2dvpas, ee-dvp-as`

.lcomm symbol, length…

Category	All
Description	Reserve `length` (an absolute expression) bytes for a local common denoted by `symbol`. The section and value of symbol are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. `Symbol` is not declared global (see ".global symbol,globl symbol" on page 69.), so is normally not visible to the linker. Some targets permit a third argument to be used with `.lcomm`. This argument specifies the desired alignment of the symbol in the bss section. The syntax for `.lcomm` differs slightly on the HPPA. The syntax is '`symbol .lcomm, length`'; symbol is optional.
Supported By	All

.lflags

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.line line-number

Category	All
Description	Change the logical line number `.line-number` must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number line-number - 1. Even though this is a directive associated with the `a.out` or `b.out` object-code formats, the assembler still recognizes it when producing COFF output, and treats `.line` as though it were the COFF `.ln` if it is found outside a `.def/.endef` pair. Inside a `.def`, .line is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.
Supported By	All
See Also	`.ln`

.linkonce [type]

Description

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensures that the linker will only include it once in the final output file. The .linkonce pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

This directive is only supported by a few object file formats; currently, the only object file format which supports it is the Portable Executable format used on Windows NT.

The type argument is optional. If specified, it must be one of the following strings. For example:

```
.linkonce same_size
```

Not all types may be supported on all object file formats.

`discard`

Silently discard duplicate sections. This is the default.

`one_only`

Warn if there are duplicate sections, but still keep only one copy.

`same_size`

Warn if any of the duplicates have different sizes.

`same_contents`

Warn if any of the duplicates do not have exactly the same contents.

Supported By `ee-dvp-as`

.list

Description

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the '`-a`' command line option; see "***Appendix B:*** *Command line switches for* *assemblers*" on page 97), the initial value of the listing counter is one.

Supported By	`ee-dvp-as`
See Also	`.nolist`

.livereg

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.llen

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

ln line-number

Category	All
Description	'`.ln`' is a synonym for '`.line`'.
Supported By	All
See Also	`.line`

.long expressions

Category	All
Description	`.long` is the same as '`.int`'; see ".int expressions" on page 71.
Supported By	All

.lsym

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.macro

Category All

Description The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, this definition specifies a macro sum that puts a sequence of numbers into memory:

```
    .macro  sum from=0, to=5
         .long   \from
         .if     \to-\from
         sum     "(\from+1)",\to
         .endif
         .endm
```

With that definition, '`SUM 0,5`' is equivalent to this assembly input:

```
    .long   0
         .long   1
         .long   2
         .long   3
         .long   4
         .long   5
```

```
.macro macname
.macro macname macargs ...
```

Begin the definition of a macro called macname. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with '`=deflt`'. For example, these are all valid .macro statements:

```
.macro comm
```

Begin the definition of a macro called `comm`, which takes no arguments.

```
.macro plus1 p, p1
.macro plus1 p p1
```

Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write '`\p`' or '`\p1`' to evaluate the arguments.

```
.macro reserve_str p1=0 p2
```

Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as '`reserve_str a,b`' (with '`\p1`' evaluating to `a` and '`\p2`' evaluating to `b`), or as '`reserve_str ,b`' (with '`\p1`' evaluating as the default, in this case '`0`', and '`\p2`' evaluating to `b`).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, '`sum 9,17`' is equivalent to '`sum to=17, from=9`'.

`.endm`

Mark the end of a macro definition.

`.exitm`

Exit early from the current macro definition.

`\@`

The assembler maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with '`\@`', but only within a macro definition. .

Supported By All

.mark

Category DVP

Description VIF operation that sets the `mark` value. `mark` sets the value of the `IMMEDIATE` field to the `VIFn_MARK` register. It is possible to use this value for synchronization with the EE core and debugging.

Supported By `ps2dvpas, ee-dvp-as`

.mpg

Category DVP

Description VIF operation that transfers the microprogram. `mpg` waits for the end of the microprogram and transfers the following `NUM` pieces of 64-bit data (microinstruction code) to the MicroMem address shown by the `IMMEDIATE` field.

Supported By `ps2dvpas, ee-dvp-as`

See Also `.endmpg`

.mri val

Category DVP

Description If `val` is non-zero, this tells the assembler to enter MRI mode. If `val` is zero, this tells the assembler to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See `–M` in "***Appendix B:*** *Command line switches for assemblers*" on page 97.

Supported By `ps2dvpas, ee-dvp-as`

.mscal

Category DVP

Description VIF operation that activates the microprogram. `mscal` waits for the end of the microprogram under execution and activates the micro program with the value of the `IMMEDIATE` field as the starting address.

Supported By `ps2dvpas, ee-dvp-as`

.mscalf

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.mscnt

Category DVP

Description VIF operation that activates the microprogram.`mscnt` waits for the end of the microprogram under execution and executes the next microprogram, from the address following the most recently ended one in the program counter.

Supported By `ps2dvpas, ee-dvp-as`

.mskpath3

Category DVP

Description	VIF operation that sets the `PATH3` mask. `mskpath3` enables/disables transfer processing via the `GIF PATH3`.
Supported By	`ps2dvpas, ee-dvp-as`

.noformat

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.nolist

Description	Controls (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.
Supported By	`ee-dvp-as`
See Also	`.list`

.nopage

Supported By	This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.octa bignums

Description	This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer. The term "octa" comes from contexts in which a "word" is two bytes; hence octa-word for 16 bytes.
Supported By	`ps2dvpas, ee-dvp-as`

.offset

Category	DVP

Description

VIF operation that sets the double buffer offset. `offset` sets the lower 10 bits of the `IMMEDIATE` field to the `VIF1_OFST` register. At the same time the DBF flag of the `VIF1_STAT` register is cleared to `0` and the `VIF1_BASE` register value is set to the `VIF1_TOPS`, i.e. the pointer for the double buffer points to the `BASE`.

Supported By

`ps2dvpas, ee-dvp-as`

.org

Supported By

This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.org new-lc, fill

Description

Advance the location counter of the current section to `new-lc`. `new-lc` is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if `new-lc` has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of `new-lc` is absolute, the assembler issues a warning, then pretends the section of `new-lc` is the same as the current subsection. `.org` may only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because the assembler tries to assemble programs in one pass, `new-lc` may not be undefined.

Note that the origin is relative to the start of the section, not to the start of the subsection.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with fill which should be an absolute expression. If the comma and fill are omitted, fill defaults to zero.

Supported By

`ps2dvpas, ee-dvp-as`

.p2align[wl] abs-expr, abs-expr, abs-expr

Description

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3` advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at All You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value `0x368d` (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

Supported By `ps2dvpas, ee-dvp-as`

.plen

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.psize lines, columns

Description Use this directive to declare the number of lines--and, optionally, the number of columns--to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and columns specification; the default width is 200 columns.

The assembler generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify lines as `0`, no formfeeds are generated save those explicitly specified with .eject.

Supported By `ps2dvpas, ee-dvp-as`

.quad

Description Specifies 128 bit data words.

Supported By `ps2dvpas`

.quad bignums

Category All

Description `.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term "quad" comes from contexts in which a "word" is two bytes; hence quad-word for 8 bytes.

Supported By All

.rept count

Category All

Description Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive count times.

For example, assembling

```
        .rept   3
        .long   0
        .endr
```

is equivalent to assembling

```
        .long   0
        .long   0
        .long   0
```

Supported By All

.rpalloc poolname, newreg, newreg,…

Category SN Systems.

Description Performs a register equate for each of the newregs by getting a register value from the specified register pool poolname.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rpfree poolname, reg, reg,…

Category SN Systems.

Description Releases each specified reg back into the pool poolname and marks the register name as undefined so you cannot use it again accidentally.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rpinit poolname, reg, reg,…

Category SN Systems.

Description Creates a register pool of the specified poolname and makes the list of registers available for allocation from that pool.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rsb name, count

Description Aligns the rs counter to the byte boundary, assigns the rs counter value to the name and advances the rs counter by count * size.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN

Systems directives" on page 21 for further details.

.rsd name, count

Description Aligns the rs counter to the doubleword boundary, assigns the rs counter value to the name and advances the rs counter by count * size.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rsh name, count

Description Aligns the rs counter to the halfword boundary, assigns the rs counter value to the name and advances the rs counter by count * size.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rsq name, count

Description Aligns the rs counter to the quadword boundary, assigns the rs counter value to the name and advances the rs counter by count * size.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rsreset (expression)

Description Sets the rs counter to 0 or the value of expression if it is specified.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `–sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rsw name, count

Description Aligns the rs counter to the word boundary, assigns the rs counter value to the name and advances the rs counter by count * size.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

.rva

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.sbttl

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.scl class

Description Set the storage-class value for a symbol. This directive may only be used inside a `.def/.endef` pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

The '`.scl`' directive is primarily associated with COFF output; when configured to generate `b.out` output format, the assembler accepts this directive but ignores it.

.scope

Category SN Systems.

Description Used in conjunction with endscope to delimit a sope for local labels. Any label beginning with the '@' character will be local to that scope.

See "Naming registers and register fields" on page 22 for sample code.

Supported By This directive is only supported by the SN Systems assemblers. It is activated by specifying the `-sn` switch on the command line. See "SN Systems directives" on page 21 for further details.

See Also `.endscope`

.section name

Description

Use the .section directive to assemble the following code into a section named name.

This directive is only supported for targets that actually support arbitrarily named sections; on a.out targets, for example, it is not accepted, even with a standard a.out section name.

For COFF targets, the .section directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b`
bss section (uninitialized data)
`n`
section is not loaded
`w`
writable section
`d`
data section
`r`
read-only section
`x`
executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the .section directive is not quoted, it is taken as a subsegment number (see section Sub-Sections).

For ELF targets, the .section directive is used like this:

```
.section name[, "flags"[, @type]]
```

The optional flags argument is a quoted string which may contain any combination of the following characters:

`a`
section is allocatable
`w`
section is writable

`x`
section is executable

The optional type argument may contain one of the following constants:

`@progbits`
section contains data
`@nobits`
section does not contain data (i.e., section only occupies space)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

For ELF targets, the assembler supports another type of .section directive for compatibility with the Solaris assembler:

```
.section "name"[, flags...]
```

Note that the section name is quoted. There may be a sequence of comma separated flags:

`#alloc section is allocatable`
`#write`
section is writable
`#execinstr`
section is executable

Supported By `ps2dvpas, ee-dvp-as`

.set symbol, expression

Category All

Description Set the value of symbol to expression. This changes symbol's value and type to conform to expression. If symbol was flagged as external, it remains flagged.

You may .set a symbol many times in the same assembly.

If you .set a global symbol, the value stored in the object file is the last value stored into it.

The syntax for set on the HPPA is '`symbol .set expression`'

Supported By All

.short expressions

Category All

Description `.short` is normally the same as '`.word`'. See ".word expressions" on page 95. In some configurations, however, `.short` and `.word` generate numbers of different lengths.

Supported By All

See Also `.hword`

.single flonums

Category All

Description This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`. The exact kind of floating point numbers emitted depends on how as is configured.

Supported By All

See Also `.float`

.size

Category All

Description This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs.
`.size` is only meaningful when generating COFF format output; when the assembler is generating `b.out`, it accepts this directive but ignores it.

Supported By All

.skip size, fill

Description This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as '`.space`'.

Supported By `ps2dvpas, ee-dvp-as`

.sleb 128 expressions

Description `sleb128` stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See ".uleb128 expressions" on page 94.

Supported By `ps2dvpas, ee-dvp-as`

.space size, fill

Description This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as '`.skip`'.

Warning: In most versions of the GNU assembler, the directive `.space` has the effect of `.block`.

Supported By `ps2dvpas, ee-dvp-as`

.spc

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.stabd, stabn, stabs

Description There are three directives that begin `.stab`. All emit symbols (see section Symbols), for use by symbolic debuggers. The symbols are not entered in the assembler's hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

`string`
This is the symbol's name. It may contain any character except `\000`, so is more general than ordinary symbol names. Some debuggers are used to code arbitrarily complex structures into symbol names using this field.

`type`
An absolute expression. The symbol's type is set to the low 8 bits of this

expression. Any bit pattern is permitted, but linkers and debuggers choke on silly bit patterns.

`other`
An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.

`desc`
An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

`value`
An absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd, .stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

```
.stabd type , other , desc
```

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings. The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

```
.stabn type , other , desc , value
```
The name of the symbol is set to the empty string "".

```
.stabs string , type , other , desc , value
```

All five fields are specified.

Supported By `ps2dvpas, ee-dvp-as`

.stcol

Category DVP

Description VIF operation that sets the filling data. `stcol` stores the following 4 words of data in the `VIFn_C0-VIFn_C3` registers. These are used as filling data when decompressed by the VIF code `unpack`.

Supported By `ps2dvpas, ee-dvp-as`

.stcycl

Category	DVP
Description	VIF operation that sets write recycle. `stcycl` sets the values of the `IMMEDIATE` field to the `VIFn_CYCLE` register.
Supported By	`ps2dvpas, ee-dvp-as`

.stmask

Category	DVP
Description	VIF operation that sets the data mask pattern. `stmask` stores the following 1 word of data in the `VIFn_MASK` register. This data becomes the mask pattern of the write data.
Supported By	`ps2dvpas, ee-dvp-as`

.stmod

Category	DVP
Description	VIF operation that sets the addition decompression mode. `stmod` sets the lower 2 bits of the `IMMEDIATE` field to the `VIFn_MODE` register. This becomes the addition decompression mode setting.
Supported By	`ps2dvpas, ee-dvp-as`

.string "str"

Category	All
Description	Copy the characters in `str` to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in section Strings.
Supported By	All

.strow

Category DVP

Description VIF operation that sets the filling data. `strow` stores the following 4 words of data in the `VIFn_R0`-`VIFn_R3` registers.

Supported By `ps2dvpas, ee-dvp-as`

.struct

Supported By This directive is not currently supported by the SN Systems asemblers or the GNU assembler. It will be ignored if used.

.subttl "subheading"

Description Use subheading as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

Supported By `ps2dvpas, ee-dvp-as`

.symver

Description Use the `.symver` directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. There are cases where it may make sense to use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library.

For ELF targets, the .symver directive is used like this:

```
.symver name, name2@nodename
```

In this case, the symbol name must exist and be defined within the file being assembled. The `.versym` directive effectively creates a symbol alias with the name `name2@nodename`, and the main reason that we do not try and create a regular alias is that the `@` character isn't permitted in symbol names. The `name2` part of the name is the actual name of the symbol by which it will be externally referenced. The name name itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The nodename portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, nodename should correspond to the nodename of the symbol you are trying to override.

Supported By `ps2dvpas, ee-dvp-as`

.tag structname

Description This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. Tags are used to link structure definitions in the symbol table with instances of those structures. `.tag` is only used when generating COFF format output; when the assembler is generating `b.out`, it accepts this directive but ignores it.

Supported By `ps2dvpas, ee-dvp-as`

.text

Description Switches to the `.vutext` section.

Supported By `ps2dvpas, ee-dvp-as`

.text subsection

Description Tells the assembler to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.

Supported By `ps2dvpas, ee-dvp-as`

.title "heading"

Description Use heading as the title (second line, immediately after the source file name and page number) when generating assembly listings. This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

Supported By `ee-dvp-as`

.type int

Category All

Description This directive, permitted only within `.def/.endef` pairs, records the integer int as the type attribute of a symbol table entry. `.type` is associated only with COFF format output; when the assembler is configured for `b.out` output, it accepts this directive but ignores it.

Supported By All

.uleb128 expressions

Description `uleb128` stands for "unsigned little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See ".sleb 128 expressions" on page 89.

Supported By `ps2dvpas, ee-dvp-as`

.unpack

Category DVP

Description VIF operation that transfers data to the VU Mem. `unpack` decompresses the following data in the format shown in the lower 4 bits (`vn, vl`) of the `CMD` field and transfers it to the VU Mem. The transfer destination address is the `IMMEDIATE` field value in `VPU0` and is the value obtained by adding the `VIF1_TOPS` register and `IMMEDIATE` field values, according to the specification by the `FLG` bit in `VPU1`.

Supported By `ps2dvpas, ee-dvp-as`

.val addr

Description This directive, permitted only within `.def/.endef` pairs, records the address `addr` as the value attribute of a symbol table entry.

`.val` is used only for COFF output; when the assembler is configured for `b.out`, it accepts this directive but ignores it.

Supported By `ps2dvpas, ee-dvp-as`

.vifnop

Category DVP

Description VIF operation

Supported By `ps2dvpas, ee-dvp-as`

.vu

Category DVP

Description Switches to VU opcode mode.

Supported By `ps2dvpas, ee-dvp-as`

.word expressions

Category All

Description This directive expects zero or more expressions, of any section, separated by commas. The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

Warning: Special Treatment to support Compilers

Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If your machine does 32-bit addressing you can ignore this issue.

In order to assemble compiler output into something that works, the assembler occasionally does strange things to '`.word`' directives. Directives of the form '`.word sym1-sym2`' are often emitted by compilers as part of jump tables. Therefore, when the assembler assembles a directive of the form '`.word sym1-sym2`', and the difference between `sym1` and `sym2` does not fit in 16 bits, it creates a secondary jump table, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to `sym2`. The original '`.word`' contains `sym1` minus the address of the long-jump to `sym2`.

If there were several occurrences of '`.word sym1-sym2`' before the secondary jump table, all of them are adjusted. If there was a '`.word sym3-sym4`', that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are adjusted to contain `sym3` minus the address of the long-jump to `sym4`; and so on, for as many entries in the original jump table as necessary.

Supported By All

Appendix B: **Command line switches for assemblers**

Command line switches

The following table lists the various command line switches that can be used to control assembly.

Switch	Description
`-a`	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`-a[cdhlmns]`	These options enable listing output from the assembler. By itself, `-a` defaults to `–ahls` and requests high-level, assembly and symbols listing. The following letters can be used to select specific options: `-ah` requests a high-level language listing `-al` requests an output-program assembly listing `-as` requests a symbol table listing `-am` includes macro expansions `=file` sets the name of the listing file (if this option is used it must be the last one). High-level listings require that a compiler debugging option like `-g` be used, and that assembly listings (`-al`) be requested also. Use the `-ac` option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing. Use the `-ad` option to omit debugging directives from the listing Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list`, `.nolist`, `.psize`,

	`.eject`, `.title`, and `.sbttl`. If you do not request listing output with one of the `-a` options, the listing-control directives have no effect. The letters after `-a` may be combined into one option, *e.g.*, `-aln` for assembly listing without forms processing. Supported by `as`.
`-D`	This switch is not supported by the SN Systems' or `as` and will be ignored if used.
`--defsym sym=value`	Define the integer symbol *sym* to be *value* before assembling the input file. *value* must be an integer constant. As in C, a leading '0x' indicates a hexadecimal value, and a leading '0' indicates an octal value. Supported by `ps2dvpas` and `as`.
`--EB`	Generate "big endian" format output.
`--EL`	Generate "little endian" format output.
`--emulation=`*`name`*	This option causes `as` to emulate `as` configured for some other target, in all respects, including output format (choosing between ELF and ECOFF only), handling of pseudo-opcodes which may generate debugging information or store symbol table information, and default endianness. The available configuration names are: `mipsecoff, mipself, mipslecoff, mipsbecoff, mipslelf, mipsbelf`. The first two do not alter the default endianness from that of the primary target for which the assembler was configured; the others change the default to little- or big-endian as indicated by the `b` or `l` in the name. Using `-EB` or `-EL` will override the endianness selection in any case. This option is currently supported only when the primary target `as` is configured for a MIPS ELF or ECOFF target. Furthermore, the primary target or others specified with `--enable-targets=...` at configuration time must include support for the other format, if both are to be available. For example, the Irix 5 configuration includes support for both.
`-f`	`-f` stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. *Warning:* if you use `-f` when the files actually need to be preprocessed (if they contain comments, for example), `as` does not work correctly.

	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`-G` *num*	This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format, such as a DECstation running Ultrix. The default value is 8.
`-G<size>`	Set size of data items placed in `.sdata/.sbss`. Supported by `ps2dvpas`
`--gstabs`	Produces STABS debugging information for each assembly line. Supported by `ps2dvpas` and `as`.
`--help`	Print a summary of the command line options and exit. Supported by `ps2dvpas` and `as`.
`-I <directory>`	Add directory *dir* to the search list for `.include` directives. `-I` can be used as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, the assemblers searche any `-I` directories in the same order as they were specified (left to right) on the command line. Supported by `ps2dvpas` and `as`.
`--itbl`	This switch is not supported by SN Systems' assemblers and will generate an error if used.
`-J`	This switch is supported by `as` to not warn about signed overflow. This switch is not supported by SN Systems' assemblers and will be ignored if used.
`-K`	Issue warnings when difference tables altered for long displacements. `as` sometimes alters the code emitted for directives of the form `.word` *sym1*-*sym2*. You can use the `-K` option if you want a warning issued when this is done. This switch is not supported by SN Systems' assemblers and will be ignored if used.
`--keep-locals`	Keep (in the symbol table) local symbols. On traditional `a.out` systems these start with `L`, but different systems have different local label prefixes. Supported by `ps2dvpas` and `as`.
`-L`	Labels beginning with 'L' (upper case only) are called *local labels*. See "Symbol names" on page 17. Normally you do not see such labels when debugging, because they are intended for the use of programs like compilers that compose assembler programs. This option tells the assemblers to retain the `L...`

	symbols in the object file. Usually if you do this you also tell the linker `ld` to preserve symbols whose names begin with `L`. By default, a local label is any label beginning with `L`, but each target is allowed to redefine the local label prefix. Local labels begin with `L$` on the HPPA and `;` for the ARM family. Supported by `ps2dvpas` and `as`.
`-[list-options]`	Produce listing (not implemented). Supported by `ps2dvpas`.
`--listing-lhs-width`	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`--listing-lhs-width2`	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`--listing-rhs-width`	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`--listing-cont-lines`	This switch is not supported by SN Systems' assemblers and will be ignored if used.
`mcpu=`*`cpu`*	Generate code for a particular MIPS cpu. This has little effect on the assembler, but it is passed by `gcc`.
`membedded-pic`	This switch is not supported by SN Systems' assemblers and will generate an error if used.

`mips1`
`mips2`
`mips3`
	Generate code for a particular MIPS Instruction Set Architecture level. `-mips1` corresponds to the R2000 and R3000 processors, `-mips2` to the R6000 processor, and `-mips3` to the R4000 processor.

`m4650` and `no-m4650`	Generate code for the MIPS R4650 chip. This tells the assembler to accept the `mad` and `madu` instruction, and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4650` turns off this option.
`-M or --mri`	The `-M` or `--mri` option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of `as` to make it compatible with the ASM68K or the ASM960 (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using `as`. The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. To support these you need to enhance each object file format individuallyas

	follows: global symbols in common section - The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. `as` handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them. complex relocations - The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not supported by other object file formats. END pseudo-op specifying start address - The MRI END pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the `-e` option to the linker, or in a linker script. IDNT, .ident and NAME pseudo-ops The MRI IDNT, .ident and NAME pseudo-ops assign a module name to the output file. This is not supported by other object file formats. ORG pseudo-op - The m68k MRI ORG pseudo-op begins an absolute section at a given address. This differs from the usual `as` .org pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script. There are some other features of the MRI assembler which are not supported by `as`, typically either because they are difficult or because they seem of little consequence. EBCDIC strings are not supported. Packed binary coded decimal is not supported. This means that the DC.P and DCB.P pseudo-ops are not supported. FEQU pseudo-op The m68k FEQU pseudo-op is not supported. NOOBJ pseudo-op The m68k NOOBJ pseudo-op is not supported. OPT branch control options The m68k OPT branch control options---B, BRS, BRB, BRL, and BRW---are ignored. `as` automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose. OPT list control options - The following m68k OPT list control options are ignored:

	C, CEX, CL, CRE, E, G, I, M, MEX, MC, MD, X. The following m68k OPT options are ignored: NEST, O, OLD, OP, P, PCO, PCR, PCS, R.OPT D option is default The m68k OPT D option is the default, unlike the MRI assembler. OPT NOD may be used to turn it off. XREF pseudo-op. The m68k XREF pseudo-op is ignored. .debug pseudo-op The i960 .debug pseudo-op is not supported. .extended pseudo-op The i960 .extended pseudo-op is not supported. .list pseudo-op. The various options of the i960 .list pseudo-op are not supported. .optimize pseudo-op The i960 .optimize pseudo-op is not supported. .output pseudo-op The i960 .output pseudo-op is not supported.
`MD`	`as` can generate a dependency file for the file it creates. This file consists of a single rule suitable for `make` describing the dependencies of the main source file. The rule is written to the file named in its argument. This feature is used in the automatic updating of makefiles.
`nocpp`	This switch is not supported by `as` and SN Systems' assemblers and will be ignored if used.
`-o <output file>`	Specify output filename. There is always one object file output By default it has the name `a.out` (or `b.out`, for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name. Whatever the object file is called, the assemblers overwrite any existing file of the same name. Supported by `ps2dvpas` and `as`.
`-R`	`-R` tells `as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment. Binary data is the same but data section parts are relocated differently. The data section part of the object file is zero bytes long because all its bytes are appended to the text section. When the assembler is configured for COFF output, this option is only useful if you use sections named `.text` and `.data`. `-R` is not supported for any of the HPPA targets. Using `-R` generates a warning from as. This switch is not supported by SN Systems' assemblers and will generate an error if used.
`-sn`	Activate SN Systems extensions. See "SN Systems

	directives" on page 21 for further information. Supported by `ps2dvpas`.
`--statistics`	Use `--statistics` to display two statistics about the resources used by `as`: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds)
`--strip-local-absolute`	Remove local absolute symbols from the outgoing symbol table.
`--traditional-format`	For some targets, the output of `as` is different from the output of some other assemblers. This switch requests `as` to use the traditional format instead. For example, it disables the exception frame optimizations which `as` normally does by default on `gcc` output. This switch is not supported by SN Systems' assemblers and will be ignored if used.

`--trap`
`--no-trap`
`--break`
`--no-break`
	Control how to deal with multiplication overflow and division by zero. `--trap` or `--no-break` (which are synonyms) take a trap exception (and only work for Instruction Set Architecture level 2 and higher); `--break` or `--no-trap` (also synonyms, and the default) take a break exception.

`--version` or `-v`	Print version information. Supported by `ps2dvpas` and `as`.
`-w`	This switch is not supported by `as` and SN Systems' assemblers and will be ignored if used.
`-W`	The assemblers should never give a warning or error message when assembling compiler output. But some programs cause the assemblers to issue a warning that a particular assumption was made. All such warnings are directed to the standard error file. This option will disable all warnings. It only affects the warning messages and does not change the assembly of your file. Errors, which stop the assembly, are still reported. Supported by `ps2dvpas` and `as`.
`-X`	This switch is not supported by `as` and SN Systems' assemblers and will be ignored if used.
`-Z`	After an error message, `as` normally produces no output. If for some reason you are interested in object file output even after `as` gives an error message on your program, use the `-Z` option. If there are any errors, `as` continues anyways, and writes an object file after a final warning message of the form '*n* `errors`, *m* `warnings`, `generating bad object`

	`file`.' This switch is not supported by SN Systems' assemblers and will be ignored if used.

Index