

User Guide to

ProDG Linker for PlayStation®3

SN Systems Limited
Version 310.1
November 24, 2009

Copyright © Sony Computer Entertainment Inc. / SN Systems Ltd, 2003-2009.

'ProDG' is a registered trademark and the SN logo is a trademark of SN Systems Ltd.
'PlayStation' is a registered trademark of Sony Computer Entertainment Inc. 'GNU' is a trademark of the Free Software Foundation. Other product and company names mentioned herein may be the trademarks of their respective owners.



Contents

1: Introduction	5
Document version history	5
Overview of the linker	5
Performance	6
Memory requirements	6
2: Linker command-line syntax	7
Command-line syntax	7
Switch processing order	7
Input packager	8
Linker switches	8
Unimplemented linker switches	15
Ignored linker switches	16
Effect of command line order on linker output	16
3: Linker scripts	18
Default linker script	18
Linker script directives	18
Unsupported script file directives	21
Sections	22
Referencing files in linker scripts	22
LIB_SEARCH_PATHS	22
REQUIRED_FILES	23
STANDARD_LIBRARIES	24
4: Section symbols	25
Section start and end pseudo-symbols	25
Dot sections	25
Pragma comment	26
5: Dead-stripping	27
Stripping unused code and data	27
Command-line switches	27
Undefined symbols	28
De-duplication	28
De-duplication and debugging	28
Strip report	29
Unused objects	29
"Objects that are referenced from objects that cannot be stripped"	29
"Objects that have not been stripped and the object they reference"	30
"Object modules that were not built with a dead-stripping compatible toolchain"	30
Function "ghosts"	31
6: Shim generation	33
TOC shims	33

Short-branching TOC shims	33
Long-branching TOC shims	33
Branch shims	34
Millicode	34
7: TOC information	35
Background	35
Eliminating TOC overhead	36
SN linker command-line switches	36
SNC PPU C/C++ compiler control-variable	36
SNC compiler -Xnotocrestore control-variable	37
SN linker --notocrestore switch	37
Limitations	39
Obtaining a TOC usage report	47
Command-line switch	47
8: Building PRX files	48
PRX generation	48
Command-line switches	48
9: Post-link processing steps	49
PRX fixup	49
Make FSELF	49
10: Troubleshooting	51
Errors and warnings	51
'L0065 Another location found for ...' warning	51
'L0280 Definition of symbol ... overrides definition from ...' warning	51
Resolving ambiguous locations for sections in the linker script	51
Overriding symbols	52
11: Index	53

1: Introduction

Document version history

Ver.	Date	Changes
310.1	Nov. 2009	Added --comment and --package-file-name switches. Moved ONLY_IF_RO and ONLY_IF_RW script directives to 'silently ignored' list (documentation error). Bz73545: Removed '--sn-no-write-buffer' switch. Bz77567: Removed '--no-keep-memory' switch. Bz78517: Updated '--compress-output' switch.
300.1	Jun. 2009	Bz67408: Added note to --md=<type> switch. Bz71141: Added --temp-dir=<path> switch. Bz71316: Removed --external-make-fself. Bz71368: Updated description for --pad-debug-line switch. Bz71822: Updated 'Ignored linker switches'. Bz71830: Removed --sn-compress-debug=<value> switch. Bz72833: Added --zgc-sections switch. Bz72922: Updated 'PRX generation' (dead-stripping switches). Bz74636: Updated 'SNC compiler -Xnotocrestore control-variable'.
280.1	Apr. 2009	Bz68814: Added 'Input packager'. Updated 'Linker switches'. Bz70338: Minor corrections. Bz70494/70496: Updated 'Linker switches' and 'Make FSELF'. Bz70601: Added 'De-duplication and debugging'. Bz70704: Added 'L0280 Definition of symbol ... overrides definition from ...' warning' and 'Overriding symbols'. Bz70338/71353: Removed 'Error messages'. Other minor corrections.
270.1	Mar. 2009	Added 'Effect of command line order on linker output'.. Bz67110/67316/67406: Updated 'Linker switches'. Bz67453: Minor corrections. Bz69709: Updated 'Linker switches'.
250.3	Dec. 2008	Updated 'Linker switches' and 'LIB_SEARCH_PATHS'.
250.1	Oct. 2008	Major chapter reorganisation. Updated 'Linker switches'. Added 'Section symbols', 'Dead-stripping', 'Shim generation', 'TOC information', 'Post-link processing steps' and 'Troubleshooting'.
220.1	Apr. 2008	Updated 'Linker switches' and 'Linker scripts'. Added 'PRX fixup'.

Overview of the linker

The SN linker is designed to be a high performance linker. It is capable of linking object files and archives created by either the SNC or GCC tool chains.

These are the main features of the SN Systems linker:

- Produces ELF images and debug information compatible with the debugger
- Supports C++, including templates, global object construction/destruction, and exceptions.
- Removes unused and duplicate functions from the image.
- Removes unused data from the image.
- Removes unused destructors from the image.
- Demangles symbol names in messages and MAP file.

Performance

The linking process is almost entirely I/O bound; that is, most of the time is spent reading sections from the input ELF files and writing the final ELF or SELF output file (see "Post-link processing steps" on page 49). The linker will try to make the best possible use of the available process address space (limited to a maximum of 3GB on Windows XP; the virtual address space of processes on 32-bit Windows is limited to 2 GB unless the "/3GB" switch is used in the Boot.ini file - see <http://msdn2.microsoft.com/en-us/library/ms791558.aspx>).

When looking to improve link times, there are three factors to consider when specifying the host machine:

- Ensure that the system has sufficient RAM. Since the link itself is I/O bound, it's important to keep use of the virtual memory swap file to an absolute minimum.
- Avoid excessive use of static libraries. Whilst static libraries are essential for system components and when distributing a library, it is sometimes used as a convenient packaging mechanism for "sub-projects" within a larger build. However, static libraries must be repeatedly scanned by the linker to ensure that the correct object files are incorporated into the link and this scanning costs execution time.

Memory requirements

Your system should have at least 2 GB RAM to avoid linker swapping. A rule of thumb for linker 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. 2 GB is recommended as a base memory size for linking of PS3 projects.

2: Linker command-line syntax

Command-line syntax

The linker command-line syntax is as follows:

```
ps3ppuld <switches> <files>
```

<switches>	Any of the command-line switches described in the following section. These must be preceded with a hyphen '-'. See note (below).
<files>	Can be object files or libraries. The linker will use these files in addition to any files specified in the linker script.

Note: switches that consist of a single letter must be preceded by a single hyphen. Switches that consist of more than one letter may be preceded by either one or two consecutive hyphens; two hyphens are recommended to avoid any ambiguity.

Many switches require arguments.

- For switches that consist of a single letter, the argument may either be appended directly to the switch, e.g. -l<library>, or after an intervening space, e.g. -e <entry>.
- For switches that consist of more than one letter, arguments must either be separated by an equals sign, e.g. --entry=<entry> or after an intervening space, e.g. --Map <mapfile>.

Comments may be inserted in the command line, starting with '/' and ending with '*/'. This will cause all switches and arguments to be ignored between the comment delimiters.

Switch processing order

Switches are processed in stages, as follows:

1. Switches other than those handled during subsequent stages (2 to 5) are processed in-order.
2. Explicit library paths are added.
3. The linker script is opened and interpreted.
4. Options that override values from the linker script are processed:
--entry=<symbol>
5. The input files are opened.

If you specify a linker input file which the linker cannot recognize as an object or archive file, it will try to read the file as a linker script. If the file cannot be parsed as a linker script, the linker will report an error.

Input packager

The input packaging tool is used to aid SN Systems in supporting the linker and solving any problems you may be experiencing. By passing the `--package` switch to the linker you can package all of your inputs, along with the command-line options you have used, into a single zip file. You can specify a filename for the package by using the `--package-file-name` switch. With this zip file SN Systems can re-run the linker in exactly the same way you have done, allowing us to exactly reproduce any bug you may be experiencing.

This is a list of the different files that will appear in the package generated.

Object files

All the object files that are input to the linker will be placed in the package file. This includes the object files from your program, any libraries used and the required files taken from the PS3 SDK. It is worth noting that the input packager will leave the code and debugging information intact.

Link response file

This file contains the command-line information needed to replicate the link as you performed it. This will not exactly match the command line you have used, but will replicate the behavior.

File name mapping

The names of the object files in the package will be as similar as possible to the original file names. However, in the event of name clashes one of the files will have a unique number appended to its name. The original path information is also lost in the new filename. So that we have a good idea of where all the files have come from we output a mapping between the original name and path, and the new name as it appears in the package.

Version file

This file contains the version of the linker used.

Script file list

This file contains a list of script files used by the linker and is used internally when running the linker with a package file as the input.

Linker switches

The linker does not support all of the switches supported by the GNU linker, `ld`, as many of these are not appropriate for console platforms. For a complete list of switches which are not supported by the linker, see "Unimplemented linker switches" on page 15 and "Ignored linker switches" on page 16.

The command-line switches can be any of the following:

Switch	Description
<code>-(--start-group</code>	Start a group.
<code>-) --end-group</code>	End a group.
<code>--callprof</code>	Add profiling code to the final output (for use with SN Tuner - see <i>User Guide to Tuner</i>)
<code>--comment</code>	Keep comment sections.
<code>--comment-report=<file></code>	Write a report in <file> showing the comment for

	all the files in the link.
<code>--compress-output</code>	Compresses FSELF output. Must be used with <code>--oformat=fself</code> or <code>--oformat=fself_npdrm</code> .
<code>-d --dc --dp</code>	Force common symbols to be defined.
<code>--deep-search</code>	Enclose all libraries and object files within <code>--start-group</code> and <code>--end-group</code> .
<code>--default-paths</code>	Add default paths from the linker script <code>LIB_SEARCH_PATHS</code> element.
<code>--defsym=<symbol>=<value></code>	Define a symbol <code><symbol></code> with value <code><value></code> .
<code>--disable-warning=<value></code>	Disables a warning message. The argument is an error number without the leading 'L'. For example, " <code>--disable-warning=95</code> " will disable warning L0095. Several messages may be disabled with a single switch by providing a comma-separated list of error numbers. For example, " <code>--disable-warning=207,24</code> " will disable both warning L0207 and L0024.
<code>--discard-all</code>	Discard all local symbols.
<code>--discard-locals</code>	Discard temporary local symbols.
<code>--dont-strip-section=X</code>	This switch suppresses dead-stripping of a named section; it is equivalent to using the <code>KEEP</code> linker script directive. To suppress dead-stripping for multiple sections, use multiple <code>--dont-strip-section</code> switches. Example: <code>--dont-strip-section=.dont_strip1</code> <code>--dont-strip-section=.dont_strip2</code> The above switches will ensure that data in sections named ".dont_strip1" and ".dont_strip2" are not stripped.
<code>--enable-warning</code>	Enables a disabled warning message. The argument is an error number without the leading 'L'. For example, " <code>--enable-warning=95</code> " will enable warning L0095. Several messages may be enabled with a single switch by providing a comma-separated list of error numbers. For example, " <code>--enable-warning=207,24</code> " will enable both warning L0207 and L0024.
<code>-e<symbol> --entry=<symbol></code>	Set start address.
<code>--exceptions</code>	Add linker default paths for libraries with exception handling (from the <code>LIB_SEARCH_PATHS</code> 'exceptions' key).
<code>--external-prx-fixup</code>	Use the SDK <code>ppu-lv2-prx-fixup</code> tool rather than the internal mechanism. (Default is to perform the PRX fix-up internally for improved performance.) See "PRX fixup" on page 49.
<code>--gc-sections</code>	This is the GNU ld dead-stripping switch. Versions of the SN linker before 2.7.2782.0 will simply warn that the switch is unsupported. Later versions will automatically select <code>--strip-unused-data</code> and issue a warning:

	warning: L0153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping
--gnu-mode	Enable GNU compatibility features. This option is intended to be used by the GCC compiler driver when it invokes the linker.
--help	Print option help.
--just-symbols=<file> -R<file>	Just link symbols from <file>.
--keep=<file>	Keep all symbols listed in <file>. This tells the linker to include these symbols even if defined in libraries. If the command line includes '--strip-unused' or '--strip-unused-data' the symbols listed in <file> will not be stripped. Include this switch when function stripping modules to prevent the stripping of the entry points.
--keep-eh-data	Suppress the removal of exception-handling data from the final output file. (If --gnu-mode is specified, this is the default behavior when neither --exceptions nor --no-exceptions is specified.)
--keeptemp	This option instructs the linker not to delete any temporary files created during the link.
-L<path> --library-path=<path>	Add <path> to the library search path.
-l<name> --library=<name>	Include library file 'lib<name>.a' when linking, e.g. -lc causes the linker to look for libc.a.
--linkonce-size-error	Emit an error if the linkonce sections from different modules are different sizes. Overrides --linkonce-size-warning.
--linkonce-size-warning	Warn if the linkonce sections from different modules are different sizes.
--Map=<mapfile>	Generate a map file in <mapfile>.
--md=<type>	Perform "minidump" crash diagnostic. <type> is the type of diagnostic. <div>Note: This switch is only supported under Windows.</div>
-mprx	Equivalent to --oformat=prx. For compatibility with GCC.
-mprx-with-runtime	Equivalent to --oformat=prx --prx-with-runtime. For compatibility with GCC.
--multi-toc	The --multi-toc switch enables the use of more than 64 KB of TOC data (this is the default behavior). The linker will automatically generate TOC shims that perform TOC region adjustment when calling between modules that use different TOC regions. See "TOC shims" on page 33.
--no-default-script	Do not attempt to locate the default linker script if none was present on the command line.
--no-default-paths	Do not add default paths from the linker script

	LIB_SEARCH_PATHS element.
--no-demangle	Do not demangle symbol names in error messages and other output.
--no-exceptions	Add linker default paths for libraries without exception handling (from the LIB_SEARCH_PATHS 'no_exceptions' key). Enables the removal of exception-handling data from the final output file.
--no-keep-eh-data	Remove exception-handling data from the final output file, unless --exceptions is specified. (This is the default behavior when neither --exceptions nor --no-exceptions is specified, unless --gnu-mode is specified.)
--no-multi-toc	--no-multi-toc will prevent the linker from creating the TOC shims that are used to support multiple TOC regions, and it will issue an error if the program contains more than 64 KB of TOC data. This switch is also present in GNU ld. error: L0154: there is too much TOC data (>64kB) for a single TOC region (consider removing both --no-multi-toc and --no-toc-restore)
--no-ppuguid	Disables the generation of the PPU GUID (default).
--no-prx-fixup	Do not perform the PRX fix-up step. See "PRX fixup" on page 49.
--no-remove-duplicate-inputs	Normally a duplicated input file within a start/end group is ignored. This switch suppresses removal of duplicates.
--no-required-files	Do not add required files from the linker script REQUIRED_PATHS element (see also --required-files).
--no-sn-dwarf-string-pool	Disable duplicate string checking that removes redundant strings found in object files.
--no-standard-libraries	Do not add standard libraries specified by the linker script STANDARD_LIBRARIES element.
--notocrestore --no-toc-restore	Use in combination with the SNC PPU C/C++ compiler's -Xnotocrestore=2 switch. The --notocrestore switch will cause the linker to rewrite the PRX stub libraries that are used to make calls to PRX functions such as the OS. The feature is not compatible with the presence of multiple TOC regions, so --notocrestore implies --no-multi-toc. --no-toc-restore is an alias for --notocrestore.
--no-warn-mismatch	Do not warn about mismatched input files.
--no-whole-archive	Turn off the effect of --whole-archive.
--noinhibit-exec	Create an output file even if errors occur.
-o<file> --output=<file>	Use <file> as the output file.
--oformat=<format>	Specify format of output file. Combinations of format and platform other than those listed will yield an 'unrecognised output format' error:

	PS3: 'elf', 'fself', 'fself_npdrm', 'prx' or 'fsprx'
--package	Place all inputs and command-line options into a zip file. See "Input packager" on page 8.
--package-file-name=<file>	Use <file> as name to be used when creating a package.
--pad-debug-line=<value>	Pads each .debug_line contribution by <value> bytes to allow for expansion during re-encoding. When dead-stripping is enabled, <value> defaults to 1; otherwise, it defaults to 0.
--ppuguid	Enables the generation of the PPU GUID. This is a unique value, introduced in SDK 240, that identifies the PPU ELF file and may be used by the debugger to simplify the experience of debugging PRX modules.
--print-toc-info	Produces a dump of the TOC section requirements and assignment of each input object file and a summary of the overall number of TOC sections and their sizes. See "Obtaining a TOC usage report" on page 47.
--prx-fixup	Perform the PRX fix-up step (default). See "PRX fixup" on page 49.
--prx-with-runtime	Link the compiler runtime libraries with the PRX output. (Only effective if --oformat is "prx" or "fsprx".)
-r --relocatable --relocateable	Generate relocatable output.
--report-unused	Reports symbols that will be stripped (in file statcov.txt).
--required-files	Add required files from the linker script REQUIRED_PATHS element (See also --no-required-files).
--retain-symbols-file=<file>	Keep only symbols listed in <file>.
-S --strip-debug	Strips all debug information from output.
--S-lib	Strips debug information supplied by libraries from output.
-s --strip-all	Strips all symbols and debug information from output.
--s-lib	Strips symbols and debug information supplied by libraries from output.
--script=<file> -T<file>	Use <file> as the linker script.
--show-messages	Display a list of all possible error and warning messages.
--sn-best	If the linker script contains more than one possible location for a section, the section is placed in the location with the best match. Compare --sn-first (below). See "Resolving ambiguous locations for sections in the linker script" on page 51.
--sn-first	The linker places sections in the first matching location found in the linker script. Compare --sn-

	best (above). See "Resolving ambiguous locations for sections in the linker script" on page 51.
<code>--sn-full-map</code>	Provides additional information in the map file, e.g. static variables.
<code>--sn-no-dtors</code>	Linker marks destructors as unused. If used in conjunction with <code>--strip-unused</code> or <code>--strip-unused-data</code> then the unused destructor code will be removed from the game.
<code>--sort-common</code>	Sort common symbols by size.
<code>--strip-duplicates</code>	This switch enables the de-duplication process. This must be used in conjunction with one of the dead-stripping options (<code>--strip-unused</code> or <code>--strip-unused-data</code>). See "Dead-stripping" on page 27.
<code>--strip-report=<file></code>	Creates a file that shows the code and references within the program. Use this report to discover which functions and data are being dead-stripped. Example: <code>--strip-report=stripreport.txt</code> See "Strip report" on page 29 for details.
<code>--strip-unused</code>	Enables the dead-stripping of code. The linker will scan the object files and archives to build the program's complete call tree. Any functions that are found to be unnecessary will be removed from the final ELF file. See "Dead-stripping" on page 27.
<code>--strip-unused-data</code>	Implicitly enables <code>--strip-unused</code> . Besides scanning for dead code, the linker will also locate unused data objects and delete them from the ELF file. See "Dead-stripping" on page 27.
<code>--sysroot</code>	Sets the 'sysroot' directory prefix. This is used in the event that a library search path (set with the <code>--library-path/-L</code> option) begins with a '='. The '=' is replaced by the sysroot directory prefix.
<code>--Tbss=<addr></code>	Set <addr> as starting address of bss section.
<code>--Tdata=<addr></code>	Set <addr> as starting address of data section.
<code>--Ttext=<addr></code>	Set <addr> as starting address of text section.
<code>-t --trace</code>	Print names of files as they are opened.
<code>--temp-dir=<path></code>	Specify a temporary directory to be used for temporary files.
<code>--Ur</code>	Build global constructor/destructor tables in a relocatable output file.
<code>-u<symbol> --undefined=<symbol></code>	Start with undefined reference to <symbol>. This switch has the same effect as the EXTERN linker script directive.
<code>--use-libcs</code>	Replace libc.a by libcs.a during linking.
<code>-V -v --version</code>	Print version information.
<code>--verbose</code>	Output lots of information during link.

<code>--Wall</code>	Produce all warnings.
<code>--warn-built-before</code>	Warn if the last modification date of any file read by the linker is prior to the specified date and time. Dates are specified as 'yyyy/mm/dd:hh:mm:ss'. If trailing parts of the date are omitted then zeroes are assumed, e.g. '2007/06/21' implies a time of '00:00:00'.
<code>--warn-common</code>	Warn about duplicate common symbols.
<code>--warn-if-debug-found</code>	Warn if a section known to contain debug information is found. The following section names will produce the warning: <pre>.debug .debug_abbrev .debug_aranges .debug_frame .debug_funcnames .debug_info .debug_line .debug_loc .debug_macinfo .debug_pubnames .debug_pubtypes .debug_ranges .debug_sfnames .debug_srcinfo .debug_str .debug_typenames .debug_varnames .debug_weaknames .line</pre>
<code>--warn-once</code>	Warn only once per undefined symbol.
<code>--Werror</code>	Treat warnings as errors.
<code>--whole-archive</code>	Include all objects from following archives. For each archive mentioned on the command line after the <code>--whole-archive</code> option, include every object file in the archive in the link, rather than searching the archive for the required object files. This option may be used more than once. The behavior can be turned off using the <code>--no-whole-archive</code> switch, such that libraries following the second switch will then link normally.
<code>--wrap=<symbol></code>	Use wrapper functions for <symbol>. If the linker is passed the argument ' <code>--wrap=foo</code> ', it will resolve references to the symbol <code>foo</code> to <code>__wrap_foo</code> ; in addition, a new symbol <code>__real_foo</code> will be created which will resolve to the original <code>foo</code> . For example, by defining a function <code>__wrap_fopen()</code> that implements any desired wrapping logic and that invokes the real <code>fopen()</code> call through the <code>__real_foo</code> symbol, we can give the linker the argument ' <code>--wrap=fopen</code> ' and easily intercept calls to <code>fopen()</code> . <pre>extern FILE *__real_fopen (const char * restrict filename, const char * restrict mode); FILE *__wrap_fopen (const char * restrict filename, const char * restrict mode)</pre>

	<pre>{ printf ('open %s\n', filename); return __real_fopen (filename, mode); }</pre>
<code>--write-fself-digest</code>	Creates an SHA-1 digest in the SELF header. Using this switch will increase link time. (Default is off; only effective if <code>--oformat=fself</code> or <code>--oformat=fself_npdrm</code> is used.)
<code>-X</code>	Discard temporary local symbols.
<code>-x</code>	Discard all local symbols.
<code>--zgc-sections</code>	PRX fix-up dead-stripping switch. The linker will select <code>--strip-unused-data</code> and issue a warning: warning: L0153: <code>--zgc-sections</code> is deprecated: using <code>--strip-unused-data</code> for dead-stripping
<code>--zgenentry</code>	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
<code>--zgenprx</code>	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
<code>--zgenstub</code>	Causes the linker to simply perform the first link, and pass ' <code>--stub-archive</code> ' to the libgen tool. See Cell OS Lv-2 PRX Programming Guide.
<code>--zlevel=<openlevel></code>	Passed directly to the ppu-lv2-prx-libgen tool.

Unimplemented linker switches

The following GNU ld linker command-line switches will cause an 'option not implemented' error if the switches are used:

Switch	Description
<code>-A<arch> --architecture=<arch></code>	Set architecture
<code>--aarchive</code>	Shared library control for HP/UX compatibility (equivalent to <code>-Bstatic</code>)
<code>--adefault --ashared</code>	Shared library control for HP/UX compatibility (equivalent to <code>-Bdynamic</code>)
<code>-assert</code>	
<code>-b<target> --format=<target></code>	Specify target for following input files
<code>-Bdynamic -call_shared -dy</code>	Link against shared libraries
<code>-Bshareable -shared</code>	Create a shared library
<code>-Bstatic -non_shared -dn</code>	Do not link against shared libraries
<code>-Bsymbolic</code>	Bind global references locally
<code>-c<file> --mri-script=<file></code>	Read MRI format linker script
<code>--dynamic-linker=<file></code>	Set the dynamic linker to use

-EB	Link big-endian objects
-EL	Link little-endian objects
--embedded-relocs	Generate embedded relocations
-F<name> --filter=<name>	Filter for shared object symbol table
-f<file> --auxiliary=<file>	Auxiliary filter for shared object symbol table
--force-exe-suffix	Force generation of file with .exe suffix
-h<name> -soname=<name>	Set internal name of shared library
-i	Incremental link
-q --emit-relocs	Leave relocation sections and contents in fully linked executables
--relax	Relax branches on certain targets
-rpath=<dir>	Adds a directory to the runtime library search path
-rpath-link=<dir>	Try to locate required shared library files in the specified directory
--split-by-file	Split output sections for each file
--split-by-reloc=<count>	Split output sections every <count> relocations
--task-link	Do task level linking
--traditional-format	Use same format as native linker
--version-script=<scriptfile>	Read version information script from <scriptfile>

Ignored linker switches

The following GNU ld linker command-line switches will cause an 'option ignored' warning if the switches are used:

Switch	Description
-g	
-O	
-qmagic	For Linux compatibility
-Y<path>	Default search path for Solaris compatibility

The following switches, passed from the GNU compiler driver, are silently ignored: --eh-frame-hdr and -Qy.

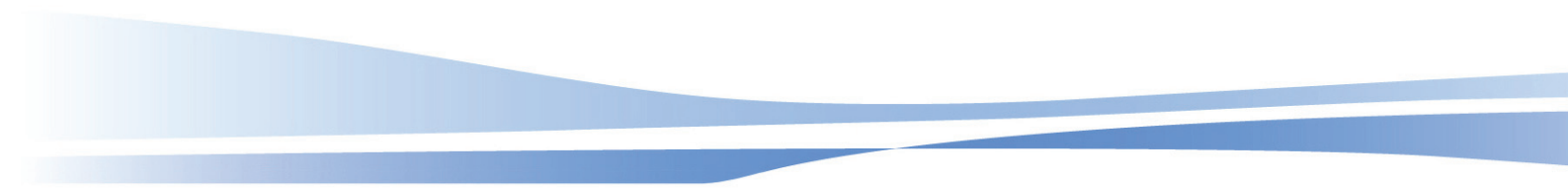
Effect of command line order on linker output

The SN linker and GNU linker have a similar command-line syntax but they differ with regard to the ordering of the command-line arguments and the linker output.

The most important difference is in the processing of archives. In the SN linker, the placement of archives on the command line does not affect symbol resolution. All symbols that can be resolved by a given set of inputs will be found during the link. As a result, the SN linker does not require use of the --start-group and --end-group

archive grouping operators, although they are supported; the archive grouping operators influence the layout of the final output, acting to keep contributions from the grouped archives near to one another. By contrast, the GNU linker defaults to performing a single left-to-right pass of the archives specified on the command line, such that an object or archive referring to an undefined symbol must appear to the left of the archive defining that symbol.

A related difference is that interleaving objects and archives on the command line does not affect the layout of the SN linker's output. Instead, all the object files are processed together, with archives processed subsequently as needed to satisfy unresolved symbols. Therefore, users cannot rely on parts of objects and archives being mixed in the linker output, even though this is possible using the GNU linker. If necessary, advanced users can use a custom linker script to achieve the required layout in a robust manner.



3: Linker scripts

Default linker script

If command-line option processing completes without a `--script` option being encountered, the linker will attempt to synthesize the default location of the linker script and process the file encountered there unless `--no-default-script` is used. The default path used is:

- If `--relocatable` is specified: '\$CELL_SDK/target/ppu/lib/prx32.sn'
- If `--relocatable` is not specified: '\$CELL_SDK/target/ppu/lib/elf64_lv2_prx.sn'

Linker script directives

The linker supports most of the 'ld format' linker script directives. For detailed information on the format of linker scripts, see the GNU linker ld documentation at <http://www.gnu.org/>.

For a complete list of script file directives supported by the GNU linker ld, which are not supported by the linker, see "Unsupported script file directives" on page 21.

Keyword	Description
ABSOLUTE(<exp>)	Return the absolute (non-relocatable, as opposed to non-negative) value of the expression <exp>. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative.
ADDR(<section>)	Return the absolute address (the VMA) of the named section. Your script must previously have defined the location of that section.
AFTER	
ALIGN	
ASSERT(<exp>, <message>)	Ensure that <exp> is non-zero. If it is zero, then exit the linker with an error code, and print <message>.
AT	
BLOCK	
BYTE	
COPY	
CREATE_OBJECT_SYMBOLS	The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the CREATE_OBJECT_SYMBOLS command appears. This is conventional for the a.out object file format. It is not normally used for any other object file format.

DEFINED	
DSECT	
END	
ENTRY	
EXCLUDE_FILE	
EXTERN(<symbol> <symbol> ...)	Force <symbol> to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. You may list several symbols for each EXTERN, and you may use EXTERN multiple times. This command has the same effect as the -u command-line option.
FILEHDR	
FILL	
FLAGS	
FORCE_COMMON_ALLOCATION	This command has the same effect as the -d command-line option: to make the linker assign space to common symbols even if a relocatable output file is specified (-r).
GLOBAL	
GROUP (<file> <file> ...)	The GROUP command is like INPUT, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of the '--start-group' command line option. Note that the SN linker considers comma to be part of a filename, so comma-delimited filenames are not supported.
INCLUDE <filename>	Include the linker script <filename> at this point. The file will be searched for in the current directory, and in any directory specified with the -L command-line option.
INFO	
INPUT (<file> <file> ...)	The INPUT command directs the linker to include the named files in the link, as though they were named on the command line. For example, if you always want to include subr.o any time you do a link, but you can not be bothered to put it on every link command line, then you can put 'INPUT (subr.o)' in your linker script. In fact, if you like, you can list all of your input files in the linker script, and then invoke the linker with nothing but a -T option. The linker will try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of the -L command line option. If you use INPUT (-lfile), the linker will transform the name to libfile.a, as with the --library command-line switch. When you use the INPUT command in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching. Note that the SN linker considers comma to be part of a filename, so comma-delimited filenames are not supported.

KEEP	
l	
len	
LENGTH	
LIB_SEARCH_PATHS	See "LIB_SEARCH_PATHS" on page 22.
LOADADDR	
LOCAL	
LONG	
MAP	
MAX(<exp1>, <exp2>)	Returns the maximum of <exp1> and <exp2>.
MEMORY	
MIN(<exp1>, <exp2>)	Returns the minimum of <exp1> and <exp2>.
NEXT	
NOCROSSREFS(<section> <section> ...)	This command may be used to tell the linker to issue an error about any references among certain output sections. In certain types of programs, particularly on embedded systems when using overlays, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors. For example, it would be an error if code in one section called a function defined in the other section. The NOCROSSREFS command takes a list of output section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. Note that the NOCROSSREFS command uses output section names, not input section names.
NOLOAD	
NONE	
o	
org	
ORIGIN	
OUTPUT(<filename>)	The OUTPUT command names the output file. Using OUTPUT(<filename>) in the linker script is exactly like using '-o<filename>' on the command line. If both are used, the command line option takes precedence. You can use the OUTPUT command to define a default name for the output file other than the usual default of a.out.
OVERLAY	
PHDRS	
PROVIDE	
PT_DYNAMIC	
PT_INTERP	
PT_LOAD	

PT_NOTE	
PT_NULL	
PT_PHDR	
PT_SHLIB	
PT_TLS	
QUAD	
REQUIRED_FILES	See "REQUIRED_FILES" on page 23.
SEARCH_DIR(<path>)	The SEARCH_DIR command adds path to the list of paths where the linker looks for archive libraries. Using SEARCH_DIR(<path>) is exactly like using '-L<path>' on the command line. If both are used, then the linker will search both paths. Paths specified using the command line option are searched first. See also "REQUIRED_FILES" on page 23.
SECTIONS	
SHORT	
SINGLE_TOC	
SIZEOF	
SIZEOF_HEADERS	
sizeof_headers	
SORT	
SQUAD	
STARTUP	The STARTUP command is just like the INPUT command, except that filename will become the first input file to be linked, as though it were specified first on the command line. This may be useful when using a system in which the entry point is always the start of the first file.
STANDARD_LIBRARIES	See "STANDARD_LIBRARIES" on page 24.
STRING	

Unsupported script file directives

The following script file directives will generate a warning if used:

OUTPUT_ARCH
OUTPUT_FORMAT

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

HLL
INHIBIT_COMMON_ALLOCATION
SYSLIB
TARGET
VERSION

The following directives are accepted by the linker but will be silently ignored:

CONSTRUCTORS
FLOAT
NOFLOAT

```
ONLY_IF_RO  
ONLY_IF_RW
```

Sections

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

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

Referencing files in linker scripts

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

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

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

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

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

LIB_SEARCH_PATHS

Two linker switches (`--default-paths`, `--no-default-paths`) control whether the default library search paths are supplemented by paths taken from the `LIB_SEARCH_PATHS` element of the linker script.

Note: the current directory is always included in the library search paths regardless of the `--default-paths/--no-default-paths` switches.

The `LIB_SEARCH_PATHS` directive in the default linker script looks like this:

```
LIB_SEARCH_PATHS  
{
```

```

exceptions :
{
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2'

    '$THIS_CELL_SDK/target/ppu/lib'

    '$THIS_CELL_SDK/host-win32/sn/ppu/lib/eh'
    '$THIS_CELL_SDK/host-win32/sn/ppu/lib'
    '$SN_PS3_PATH/ppu/lib/sn'
}
no_exceptions :
{
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1/fno-
exceptions'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1/noeh'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2/fno-
exceptions'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2/noeh'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2'
    '$THIS_CELL_SDK/host-win32/ppu/lib/gcc'
    '$THIS_CELL_SDK/host-win32/ppu/ppu-lv2/lib'

    '$THIS_CELL_SDK/target/ppu/lib/fno-exceptions'
    '$THIS_CELL_SDK/target/ppu/lib/noeh'
    '$THIS_CELL_SDK/target/ppu/lib'

    '$THIS_CELL_SDK/host-win32/sn/ppu/lib'
    '$SN_PS3_PATH/ppu/lib/sn'
}
}

```

REQUIRED_FILES

The `REQUIRED_FILES` element contains an array of filenames which are normally required to successfully link on the target. It has been used in the PS3 linker to warn if the user has omitted any of the GCC startup-glue files that are needed by the system libraries, and was designed to avoid the user from spending time diagnosing crashes when one of these files is missing. This functionality enables the linker to automatically include the required files in a link if they have not already been explicitly listed on the command line.

The `REQUIRED_PATHS` directive in the default linker script looks like this:

```

REQUIRED_FILES
{
    ecrti.o
    crt0.o
    crt1.o
    crtbegin.o
    crtend.o
    ecrti.o
}

```

STANDARD_LIBRARIES

The STANDARD_LIBRARIES element is identical in function to the GROUP command, but it may be disabled by the --no-standard-libraries command-line switch.

The STANDARD_LIBRARIES directive in the default linker script looks like this:

```
STANDARD_LIBRARIES (-lc -lgcc -lstdc++ -lsupc++ -lm -lsyscall  
-llv2_stub -lsnc)
```


4: Section symbols

Section start and end pseudo-symbols

The linker supports GNU ld-style pseudo-symbols for ELF section beginning and end, which are instantiated if an undefined reference is found.

If you use symbols named `__start_XXX` or `__stop_XXX`, the linker will synthesize the address of a section named "XXX". The section name must be representable as a C name (i.e. alphanumeric characters and underscores).

If the linker sees a section whose name can be represented as a C identifier, it will speculatively generate symbols with the names `__start_NAME` and `__stop_NAME` that mark the beginning and end of that section respectively. If unresolved references to these symbols are found within the linker inputs, the generated symbols will be defined in the linker output individually and the references will be resolved to point to them. This functionality mimics an extension provided by the GNU linker.

```
int foo __attribute__((section('bar')));
extern const unsigned char __start_bar [];
extern const unsigned char __stop_bar [];
const unsigned char * start_of_bar (void)
{
    return &__start_bar [0];
}
const unsigned char * end_of_bar (void)
{
    return &__stop_bar [0];
}
```

Dot sections

As a further extension, the linker performs a similar process for sections that start with '.' (dot). In this case, symbol names are created by replacing the leading '.' with the sequence `__Z`. For example, the start of the `.text` section would be indicated by `__start__Ztext`. The same check for unresolved references is then applied to determine whether the symbols should be defined.

Note: The remainder of the section name after the leading dot must be a valid C identifier. Therefore, section names containing multiple dots will be ignored.

If the linker sees a section whose name starts with `__Z` but is otherwise a valid C identifier, it will speculatively generate symbols that mark the beginning and end of that section. If unresolved references to these symbols are found within the linker inputs, the generated symbols will be defined in the linker output individually and the references will be resolved to point to them. The symbol names are created by replacing the leading `__Z` with the sequence `__Z` and prefixing the modified section name with `__start_` or `__stop_`. For example, the start of the `.text` section would be indicated by `__start__Ztext`.

Example 1: Use of section symbols to locate the start and end addresses of a section whose name contains characters that are legal for a C identifier.

```
#include <stdio.h>
#include <stdlib.h>

#define SECTION(x) \
    __attribute__((section (x)))
int bar_var_1 SECTION ("bar");
int bar_var_2 SECTION ("bar");

/* symbols generated by the linker! */
extern void const * __start_bar;
extern void const * __stop_bar;

int main ()
{
    printf ( "__start_bar=%p, __stop_bar=%p\n",
            &__start_bar,
            &__stop_bar
    );
    printf ( "&bar_var_1=%p, &bar_var_2=%p\n",
            &bar_var_1,
            &bar_var_2
    );
    return EXIT_SUCCESS;
}
```

Example 2: Use of section symbols to access a section whose name begins with a dot.

```
#include <stddef.h>

/* symbols generated by the linker! */
extern unsigned char const
    __start__Ztext [];
extern unsigned char const
    __stop__Ztext [];

ptrdiff_t size_of_dot_text (void)
{
    return    &__stop__Ztext [0]
            - &__start__Ztext [0];
}
```

Pragma comment

The SN linker supports the processing of a ".linker_cmd" section in the input object files. This section is emitted by SNC 240.1 and later in response to use of the Microsoft-style #pragma comment ("lib","xxx"). A description of this feature can be found on Microsoft's web site at: [http://msdn.microsoft.com/en-us/library/7f0aews7\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/7f0aews7(VS.71).aspx).

This pragma can be used to automatically add files to the linker's command line.

5: Dead-stripping

Stripping unused code and data

GNU ld's dead-stripping works by reference counting each section. When a relocation references a symbol, it marks the section containing the symbol as referenced. At the end of the process, any unmarked sections are known not to be needed and are not written to the output. The downside of this approach is that it only works on entire sections, which means that you must compile your code with the special GCC `-ffunction-sections` and `-fdata-sections` switches. The proliferation of ELF sections that result from the use of these switches is likely to adversely affect link times.

The SN linker's dead-stripping is quite different. It works by scanning the relocations themselves to determine exactly what each piece of code and data references. The advantage of this process is that it can strip pieces of code and data from the middle of sections without requiring special compiler switches.

Command-line switches

Because of the nature of the SN linker's dead-stripping, it is controlled by a set of switches that are different from those used by the GNU linker. There is no need to compile with the GCC `-ffunction-sections` or `-fdata-sections` switches.

Switch	Description
<code>--dont-strip-section=X</code>	<p>This switch suppresses dead-stripping of a named section; it is equivalent to using the KEEP linker script directive. To suppress dead-stripping for multiple sections, use multiple <code>--dont-strip-section</code> switches.</p> <p>Example: <code>--dont-strip-section=.dont_strip1 --dont-strip-section=.dont_strip2</code></p> <p>The above switches will ensure that data in sections named ".dont_strip1" and ".dont_strip2" are not stripped.</p>
<code>--gc-sections</code>	<p>This is the GNU ld dead-stripping switch. Versions of the SN linker before 2.7.2782.0 will simply warn that the switch is unsupported. Later versions will automatically select <code>--strip-unused-data</code> and issue a warning:</p> <pre>warning: L0153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping</pre>
<code>--strip-duplicates</code>	<p>This switch enables the de-duplication process. This must be used in conjunction with one of the dead-stripping options (<code>--strip-unused</code> or <code>--strip-unused-data</code>).</p>
<code>--strip-report=<file></code>	<p>Creates a file that shows the code and references within the program. Use this report to discover which functions and data are being dead-stripped.</p> <p>Example: <code>--strip-report=stripreport.txt</code></p> <p>See "Strip report" on page 29 for details.</p>

<code>--strip-unused</code>	Enables the dead-stripping of code. The linker will scan the object files and archives to build the program's complete call tree. Any functions that are found to be unnecessary will be removed from the final ELF file.
<code>--strip-unused-data</code>	Implicitly enables <code>--strip-unused</code> . Besides scanning for dead code, the linker will also locate unused data objects and delete them from the ELF file.

Undefined symbols

Even when dead-stripping is enabled, all referenced symbols must be defined even if the caller is not ultimately referenced.

For example:

```
extern void bar (void);
static void foo (void)
{
    bar ();
}

int main ()
{
}
```

If `bar()` is not defined elsewhere in the program and is not called by any function other than `foo()`, then even though it is only called by code that will be dead-stripped, you will receive a link error regardless of whether stripping is enabled:

error: L0039: reference to undefined symbol .bar in file main.o

De-duplication

De-duplication is a feature of the linker that is intended to further reduce the size of the final executable image by eliminating duplicated copies of identical code and read-only data. Programs sometimes contain a great deal of duplicated code and data. When the linker knows that content is read-only, it is able to remove the duplicates and change each of the references to the originals so that point to the single remaining copy.

To enable de-duplication, use the `--strip-duplicates` switch in conjunction with `--strip-unused` or `--strip-unused-data`.

De-duplication and debugging

When using de-duplication, it is likely that it will not be possible to view some parts of the program in the debugger. Unfortunately, the limitations of the DWARF debugging format make this an unavoidable side effect of using de-duplication. In particular, consider the case where two functions are de-duplicated: there are now two or more source code representations of the same executable code.

In general, it is possible to use the debugger in parts of the program that have not been de-duplicated. However, if necessary source code cannot be viewed in the debugger when using de-duplication, disabling de-duplication will restore debugging functionality.

Strip report

The *strip report* details the effects of applying the various dead-stripping options. If it is specified alongside one of the dead-stripping options, it will show the outcome of the dead-stripping process; otherwise, it will show the expected results.

The report is divided into four sections, each which provide details about some facet of the dead-stripping process.

Unused objects

The first section lists the objects which were determined as unused or otherwise extraneous (e.g. duplicated code or data).

- If dead-stripping is enabled, the names of the objects will be tabulated against the number of bytes that were stripped (under the column heading "Strip") and the number of bytes of padding that were left (under the column heading "Pad") to ensure correct alignment.
- If dead-stripping is not enabled, the object named will be tabulated against the virtual address of the object (under the column heading "Address") and the number of bytes that would have been stripped (under the column heading "Strip").

Within the table, values within square brackets following an object file or archive name indicate the object's index within the symbol table. Additionally, a small summary of the dead-stripping process will be written following the table.

For example:

```
212      0 .memset (.text) in ...\\target\\ppu\\lib\\fno-exceptions\\  
libc.a(memset.o) [8]  
      8      0 memset (.opd) in ...\\target\\ppu\\lib\\fno-exceptions\\  
libc.a (memset.o) [7]  
      8      0 main (.opd) in ...\\test.o[9]
```

A total of 13836 bytes can be saved by stripping 180 unused objects.
116 padding bytes will be left.

"Objects that are referenced from objects that cannot be stripped"

The second section lists those objects that are disallowed from being stripped for some reason.

At the top of this section, a legend is printed detailing the meaning of the flags used in the table below. The table itself shows the object name, followed by the section to which it belongs in the output file, against the flags which describe why the object cannot be stripped. The most common flag here is 'G', which indicates that an object is referenced through a global symbol. Typically, this indicates that the site of the reference could not be attributed to a given object (e.g. it originates from a position associated with a zero-sized symbol).

For example:

Flags	Object name	(Section name)
-------	-------------	----------------

```
-G--      ._start (.text)
```

"Objects that have not been stripped and the object they reference"

The third section contains the reference graph determined for the output file. Since links within the reference graph are awkward to visualize for most programs, each entry in the graph is listed along with the objects that it directly references.

- An entry is preceded by a series of dashes and the source object file (possibly in parentheses, following an archive name, if the object originated from an archive). The name of the object is listed, along with its target section in parentheses.
- If the object is not a leaf within the graph, the text "requires..." follows, along with the list of referenced objects (with one referenced object per line).
- If any referenced object is followed by the text "[symbol]", it is a reference to a symbol which cannot be further resolved (and is therefore ineligible for dead-stripping).

For example:

```
-----
...\target\ppu\lib\fno-exceptions\crt0.o
._start (.text)  requires...
    _start [symbol]
    _initialize

-----
...\test.o
.main (.text)  requires...
    <.toc.0>
    .puts
```

"Object modules that were not built with a dead-stripping compatible toolchain"

The fourth section is a list of object files (possibly parenthesized, following the parent archive) that are incompatible with the dead-stripping mechanism used by the SN linker.

Unfortunately, versions of the GNU assembler prior to the one being used for SDK 200 had an optimization that did not emit relocations for branches within a section, because the branch instruction uses a relative value for the destination and it knows the difference between the source and destination addresses. This meant that the lack of a relocation broke the linker's dead-stripper because the linker did not know that the reference to the callee existed.

The modified assembler now both emits all the required relocations and sets a bit in the ELF file header to indicate that the relocations are present, so that it is safe to dead-strip. Unfortunately, of course, this requires that everything is rebuilt.

When the linker encounters one or more object files that are not marked as safe to dead-strip, it will issue a warning:

```
warning: L0134: 11 of 67 files were not dead-stripped because they were not
built with a dead-stripping compatible toolchain (for details, see the strip
report [--strip-report <file>]).
```

Function "ghosts"

There are conditions under which the linker can appear to leave "ghosts" of stripped functions in the final executable.

When dead-stripping and de-duplicating, the linker is guided by symbols defined in the input files. These symbols describe the name, location, and address of the data items contained by those files. However, it is also not uncommon for there to be data that is not within the bounds of a symbol. This data is strictly off-limits to the dead-stripper: it must not be removed and its requirements – alignment in this case – must continue to be respected.

Below is an example that shows when this can occur:

```
.section .text

# Alignment is 2^3 i.e. 8 bytes
.align 3

# Declare the ".foo" symbol and give it an address within
# the .text section
.foo:
    stdu    %sp, -112(%sp)
    mflr    %r0
    std     %r0, 128(%sp)
    ...
    ld      %r0, 128(%sp)
    mtlr    %r0
    addi    %sp, %sp, 112
    bclr    20, 0

# Now provide the size of the .foo symbol
.size      . - .foo

# Guarantee 8 byte alignment of the following symbol.
# The compiler may elide this instruction: the assembler
# will insert padding to align the next function
nop

.bar:
```

The `.text` section of this object module has an alignment attribute of 8. This means that the section, and all the symbols within it, must be placed on an 8-byte boundary. The compiler has ensured this by adding a `nop` instruction after the `blr` instruction if the function does not contain a multiple of 8 bytes of code. Unfortunately, the function's symbol record in the object file does not include this padding instruction.

When dead-stripping, the linker removes any unreferenced functions or data. However, when doing so it must respect the declared alignment of the section and ensure that the remaining objects within it continue to be correctly aligned. In addition, any portion of a section that is not covered by an ELF symbol cannot be touched by the dead-stripper since it cannot know that the data is unneeded.

The net effect is that the linker cannot strip the `nop` (since it is not covered by a symbol definition), and it cannot simply leave behind the four bytes occupied by the `nop` because it must maintain the 8-byte alignment. This results in both the `blr` and `nop` instructions remaining as a "ghost" of the original function.

In the case of GCC, a possible workaround is to pass the `-falign-functions=4` switch. This will align the functions on 4-byte boundaries and eliminate the unwanted `nop` instructions. There should be little or no performance impact. The presence of

unaligned functions may add one extra CPU cycle on the first instruction issue of the function. However, this kind of issue stall is rarely the bottleneck on the PPU.



6: Shim generation

"Shims" are little snippets of code that are created at link time. They are used on the PS3 to support ABI features such as the TOC.

Shims are inserted close to the caller to guarantee that the shim code itself can be reached by the original call instruction.

TOC shims

TOC shims currently perform a relative adjustment of the TOC register. This has the effect that a TOC shim includes both the address of the callee function and the relative distance from the caller's TOC region.

Note: You can prevent TOC shims from being generated, and avoid the resulting performance impact, by using less than 64 KB of total TOC data in your program. This may be enforced using the linker's `--no-multi-toc` or `--no-toc-restore` options. You can further reduce the TOC overhead using the techniques described in the section "Eliminating TOC overhead" on page 36.

Short-branching TOC shims

This code is inserted by the linker when the callee and caller are separated by a distance less than or equal to the maximum branch distance permitted by the PowerPC branch instruction (which corresponds to the `R_PPC64_REL24` relocation).

```
std    %rtoc, 28(%sp)
addis  %rtoc, %rtoc, toc_difference (hi)
addi   %rtoc, %rtoc, toc_difference (lo)
b      R_PPC64_REL24 (callee)
```

Long-branching TOC shims

This type of shim is inserted by the linker when the callee and caller are separated by a distance greater than the maximum branch distance permitted by the PowerPC branch instruction. The caller performs a relative branch to the shim which then performs a branch to a full 32 bit address.

Besides the TOC register, the code modifies the `%r11` and `%ctr` registers; they are defined as volatile by the ABI.

```
std    %rtoc, 28(%sp)
```

```
addis %rtoc, %rtoc, toc_difference (hi)
addi  %rtoc, %rtoc, toc_difference (lo)
lis   %r11, R_PPC64_ADDR16_HA (callee)
addi  %r11, %r11, R_PPC64_ADDR16_LO (callee)
mtctr %r11
bctr
```

Branch shims

Branch shims are used to avoid errors when a PowerPC relative branch instruction is performed to an address whose distance from the caller exceeds the maximum distance permitted by the instruction.

The code modifies the %r11 and %ctr registers; they are defined as volatile by the ABI.

```
lis   %r11, R_PPC64_ADDR16_HA (callee)
addi  %r11, %r11, R_PPC64_ADDR16_LO (callee)
mtctr %r11
bctr
```

Millicode


The linker creates the functions described in the PPU ABI Specifications for Cell OS Lv-2 as "Register Saving and Restoring Functions" as required. A reference to an undefined function with one of the names defined by the ABI (such as _savegpr0_32 or _restvr_20) will result in an implementation of the function being created with the definition described in this document.

Although the linker will avoid creating multiple copies of these functions wherever possible, it also guarantees not to create a branch shim to them. If there is an existing copy of a millicode function that cannot be reached, a new copy will then be generated.

Example:

```
_savegpr0_30:
    std      %r30, -16(%sp)
_savegpr0_31:
    std      %r31, -8(%sp)
    std      %r0, 16(%sp)
    blr
```

For a complete list of millicode functions, see the PPU ABI Specifications for Cell OS Lv-2.



7: TOC information

Background

The PPU ABI Specifications for Cell OS Lv-2 describes a structure known as the TOC that has some ramifications for the behavior of both the compiler and the linker:

- A call to a function must have room after the call instruction itself for the linker to patch up the code.
- A call through a pointer to a function must use an intermediate structure: the ".opd" entry. This structure consists of the address of the TOC region used by the target code, and the address of the target code itself.

Here are is an example showing both behaviors:

```
typedef void (*func_ptr) (void);
void foo (func_ptr p)
{
    (*p) ();
}

extern void bar (void);
void qaz (void)
{
    bar ();
}
```

Compiling this sample with the SNC -O3 switch results in code that looks something like the snippet below. Most of this is mandated by the ABI, so GCC produces very similar looking output. The function prologues and epilogues have been trimmed for the sake of clarity.

```
.foo:
...snip function prologue...
lwz    %r4, 0(%r3)
std     %rtoc, 40(%sp)
mtctr   %r4
lwz     %rtoc, 4(%r3)
bctrl
ld      %rtoc, 40(%sp)
...snip function epilogue...

.qaz:
...snip function prologue...
bl      .bar
nop
...snip function epilogue...
```

However, much of this machinery is present in order to make the TOC work and, although it carefully follows the rules mandated by the ABI to ensure compatibility with GCC compiled code, SNC does not place data in the TOC.

The 'no TOC restore' mode allows us to improve the efficiency of both direct calls and calls through function pointers.

Eliminating TOC overhead

This section describes the use of a 'no TOC restore' mode through the use of the SNC compiler `-Xnotocrestore=2` control-variable setting and the SN linker `--notocrestore` (alias `--no-toc-restore`) switch. These switches together enable the overhead of the TOC to be almost entirely eliminated and can provide a significant reduction in overall code size.

- It is safe to freely mix code compiled with SNC `-Xnotocrestore=2` with other SNC-compiled code and with code compiled by GCC. The only restriction is that there is no more than a total of 64 KB of TOC data in the application. This limit will be enforced by the linker when its `--no-toc-restore` switch is used.

To use the 'no TOC restore' mode:

1. Compile with the SNC `-Xnotocrestore=2` control-variable setting.
2. Link with the SN linker `--notocrestore` switch.

Note: It is possible to construct PRX code in a way that is not compatible with the 'no TOC restore' mode. If you encounter difficulties, see "Limitations" on page 39.

SN linker command-line switches

Switch	Description
<code>--multi-toc</code>	The <code>--multi-toc</code> switch enables the use of more than 64 KB of TOC data (this is the default behavior). The linker will automatically generate TOC shims that perform TOC region adjustment when calling between modules that use different TOC regions. See "TOC shims" on page 33.
<code>--no-multi-toc</code>	<code>--no-multi-toc</code> will prevent the linker from creating the TOC shims that are used to support multiple TOC regions, and it will issue an error if the program contains more than 64 KB of TOC data. This switch is also present in GNU ld. error: L0154: there is too much TOC data (>64kB) for a single TOC region (consider removing both <code>--no-multi-toc</code> and <code>--no-toc-restore</code>)
<code>--notocrestore</code> <code>--no-toc-restore</code>	Use in combination with the SNC PPU C/C++ compiler's <code>-Xnotocrestore=2</code> switch. The <code>--notocrestore</code> switch will cause the linker to rewrite the PRX stub libraries that are used to make calls to PRX functions such as the OS. The feature is not compatible with the presence of multiple TOC regions, so <code>--notocrestore</code> implies <code>--no-multi-toc</code> . <code>--no-toc-restore</code> is an alias for <code>--notocrestore</code> .

SNC PPU C/C++ compiler control-variable

A description of the SNC PPU C/C++ compiler `-Xnotocrestore` control-variable is provided here for convenience. For the most accurate documentation, please see the *User Guide to SNC PPU C/C++ Compiler*.

Control-variable	Description
<code>-Xnotocrestore=0</code>	The compiler generates fully ABI compliant code. The code to call a function through a pointer assumes that the value of the TOC register at the callee may be different from that of the caller.

	<p>A nop instruction is generated after a call to an external function to allow the linker to restore the TOC pointer if the callee code resides in a different TOC region at link time.</p> <p>No special linker switches are necessary for code built with this option to run correctly.</p> <p>This is the default value of the notocrestore control.</p>
-Xnotocrestore=1	<p>The compiler elides the nop instruction after a call to an external function but calls through pointers are guaranteed to be TOC-safe. The program must be linked with the SN linker --notocrestore switch.</p>
-Xnotocrestore=2	<p>The compiler elides both the nop instruction after a call to an external function and assumes that a call through a pointer will always use the same TOC region. The program must be linked with the SN linker --notocrestore switch.</p>

SNC compiler -Xnotocrestore control-variable

Using SNC to compile the same code snippet as in the "Background" section with the -Xnotocrestore=2 setting produces output like this:

```
.foo:
    ...snip function prologue...
    lwz    %r3, 0(%r3)
    mtctr  %r3
    bctrl
    ...snip function epilogue...
.qaz:
    ...snip function prologue...
    bl     .bar
    ...snip function epilogue...
```

This is much better: we have eliminated one store and two loads in the first case, and removed the unnecessary nop in the second.

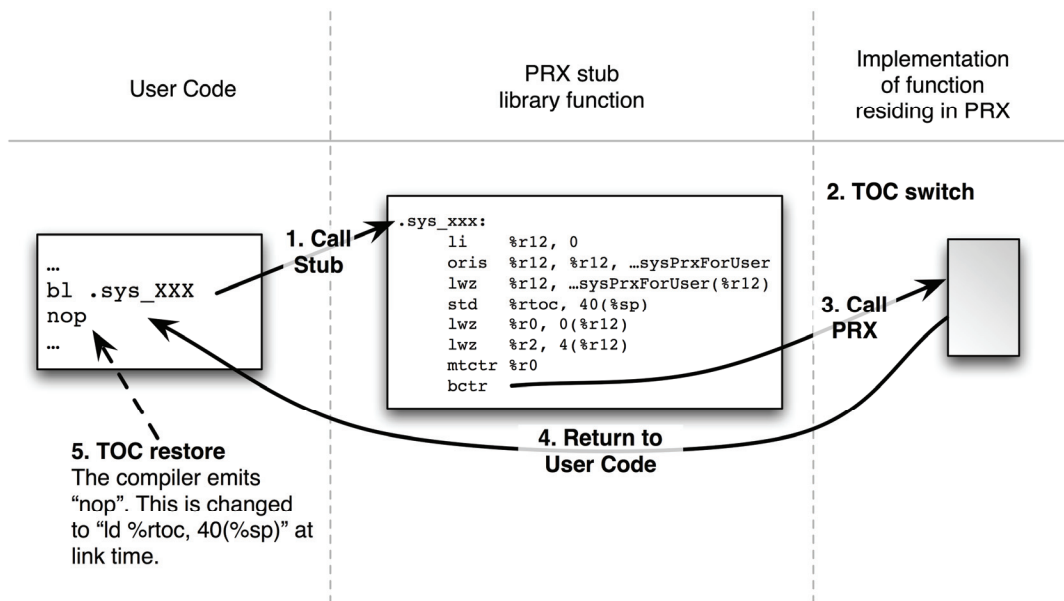
The tradeoff is that this code assumes that the TOC register (%rtoc/%r2) never needs to change before making a function call, or has to be restored after it. Enforcing that condition needs some assistance from the linker.

SN linker --notocrestore switch

PRX libraries compiled with GCC are not immediately compatible with the 'no TOC restore' model. These PRXs require a different TOC region from the main program, and the linker must continue to support this behavior.

The following illustrates the mechanism that the linker uses to replace the code in a PRX stub library with an alternative implementation that supports 'no TOC restore' mode.

PRX calls without --notocrestore

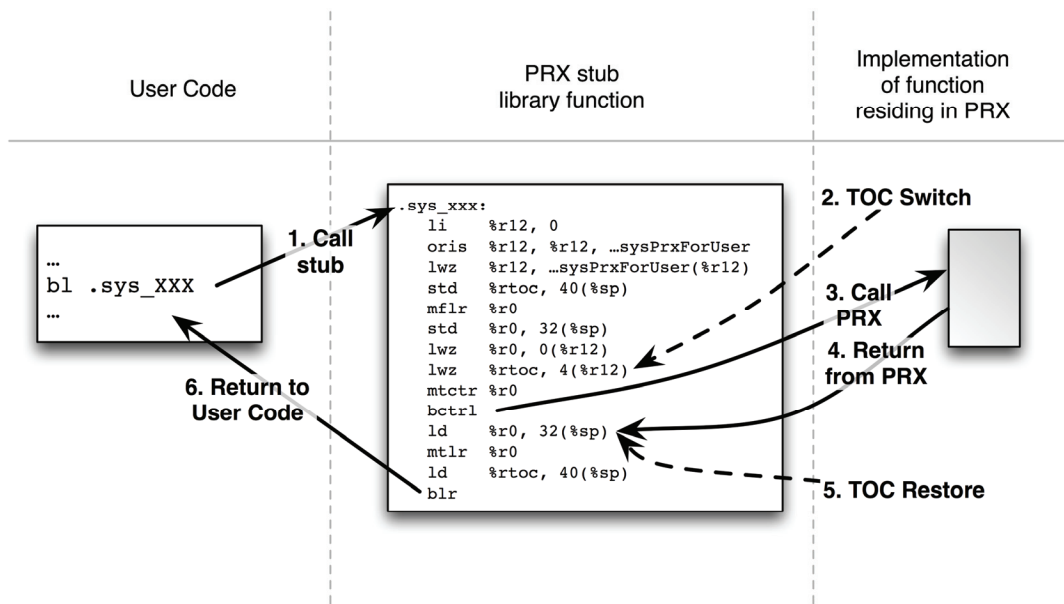


The default code sequence used to call a PRX contains code that changes the TOC register to point at the PRX's TOC data before branching to the target function. That function returns directly to the original call site. The linker patches the nop instruction that follows the original call to restore the caller's TOC register value.

The operations performed are:

1. User code calls the PRX stub code. This performs the setup required to make the PRX call.
2. The PRX stub code saves the current value of the TOC register and changes the TOC register so that it references the PRX TOC region.
3. The PRX code is invoked.
4. The call returns directly to user code.
5. User code restores the original TOC region. Using the same technique as a conventional external function call, the compiler's nop is replaced with a TOC restore instruction. This is the "PRX fixup" process; see "PRX fixup" on page 49 for more information.

PRX calls with --notocrestore



When 'no TOC restore' mode is enabled, the linker recognizes the code sequence that is used to perform PRX function calls and replaces it with a version that performs the TOC restore directly rather than relying on patching the original call site.

The operations performed by the replacement code are:

1. User code calls the PRX stub code. This performs the setup required to make the PRX call.
2. The PRX stub code saves the current value of the TOC register and changes the TOC register so that it references the PRX TOC region.
3. The PRX code is invoked.
4. The call returns to the PRX stub code.
5. The original TOC value is restored.
6. We return to user code.

The advantage of this approach is that it does not rely on the compiler having generated the additional nop instruction after the call to the PRX function.

To enable the replacement of PRX stub libraries in this manner, use the `--notocrestore` switch.

If the total amount of TOC data exceeds 64 KB, the linker will issue an error:

```
error: L0154: there is too much TOC data (>64kB) for a single TOC region
(consider removing both --no-multi-toc and --no-toc-restore)
```

GCC can be forced to consume less space in the TOC with switches such as `-mminimal-toc` and `-mbase-toc`. SNC never generates TOC data.

Limitations

The 'no TOC restore' scheme relies on intercepting the calls that the program makes to functions in PRX modules. For direct calls, the linker can rewrite the stub code as has been described.

The following examples demonstrate this in both a C and C++ program, each using function pointers and virtual methods respectively.

Note: GCC must be used to compile the PRX library source code in these examples simply to ensure that there are references to the TOC. The problem does not occur if SNC is used to compile the PRX.

No TOC restore C sample (prx1)

The first example consists of a PRX resident library (prx1.h, prx1.c), and an application (app1.c) that uses the PRX. The PRX exports a single function, `get_callback()`, that returns the address of a function. The main program uses `get_callback()` to get the function pointer and then calls it. The expected result is that the text "in callback" appears on the PPU stderr channel.

prx1.h:

```
#ifndef PRX1_H
#define PRX1_H

typedef void (*callback_ptr) (void);
callback_ptr get_callback (void);

#endif /* PRX1_H */
```

prx1.c:

```
#include "prx1.h"
#include <sys/prx.h>
#include <sys/tty.h>
SYS_MODULE_INFO (prx1, 0, 1, 0);
SYS_LIB_DECLARE (prx1, SYS_LIB_AUTO_EXPORT |
                 SYS_LIB_WEAK_IMPORT);
/* export the get_callback function. */
SYS_LIB_EXPORT (get_callback, prx1);

static void write_message (char const * message)
{
    unsigned int write_length;
    char const * end;
    for (end = message; *end != '\0'; ++end)
        ;

    sys_tty_write (SYS_TTYP_PPU_STDERR, message,
                  end - message, &write_length);
}

void callback (void)
{
    write_message ("in callback");
}

callback_ptr get_callback (void)
{
    return &callback;
}
```

app1.c:

```
#include <stdlib.h>
#include <cell/error.h>
#include <sys/prx.h>
#include <sys/paths.h>
#include "prx1.h"

/* get the PRX ready to call */
```



```
static sys_prx_id_t load_start (char const * path)
{
    int module_result;
    sys_prx_id_t id =
        sys_prx_load_module (path, 0, NULL);
    sys_prx_start_module (id, 0, NULL,
        &module_result, 0, NULL);
    return id;
}

/* clean up after the PRX */
static void stop_unload (sys_prx_id_t id)
{
    int module_result;
    sys_prx_stop_module (id, 0, NULL,
        &module_result, 0, NULL);
    sys_prx_unload_module (id, 0, NULL);
}

int main ()
{
    char const * path = SYS_APP_HOME "/prx1.sprx";
    sys_prx_id_t prx_id = load_start (path);

    /* get the callback from the PRX and call it */
    callback_ptr cb = get_callback ();
    (*cb) ();

    stop_unload (prx_id);
    return EXIT_SUCCESS;
}
```

prx1 sample Makefile:

```
# Makefile for prx1 example
CC      = ps3ppusnc
CFLAGS  = -g -O0
LD       = ps3ppuld
LDFLAGS =
RUN      = ps3run
RUNFLAGS = -p -q -r -f . -h .
# uncomment the two following lines to experiment
# with "no toc restore" mode.
#CFLAGS += -Xnotocrestore=2
#LDFLAGS += --notocrestore

.PHONY : all
all : prx1.sprx app1.self

.PHONY : clean
clean :
    -rm -f prx1.o prx1.sprx
    -rm -f prx1_stub.a prx1_verlog.txt
    -rm -f app1.o app1.self

.PHONY : run
run : app1.self
    $(RUN) $(RUNFLAGS) $^

app1.o : app1.c prx1.h
```

```
# To demonstrate the problem, prx1.o must be
# compiled with GCC to ensure that it contains
# use of the TOC.
prx1.o : prx1.c prx1.h
    ppu-lv2-gcc -o $@ -c -g -O0 prx1.c

prx1.sprx prx1_stub.a : prx1.o
    $(LD) --oformat=fsprx -o prx1.sprx \
        $(LDFLAGS) $^
app1.self : app1.o prx1_stub.a
    $(LD) --oformat=fself -o app1.self \
        $(LDFLAGS) $^
```

This example fails if 'no TOC restore' mode is used. The linker cannot intercept the invocation of the callback to correct ensure that the value of the TOC pointer is seen by callback() when it is called through a pointer.

No TOC restore C++ sample (prx2)

The second example consists again of a PRX resident library (the source code for which is in prx2.h and prx2.cpp) and an application (app2.cpp) that uses the PRX. The PRX exports a single function, get_foo(), that returns an instance of class foo. The main program then invokes one of the class's virtual methods. The expected result is that the text "in member_function" is written to stdout.

prx2.h:

```
#ifndef PRX2_H
#define PRX2_H

class foo
{
public:
    virtual ~foo ();
    virtual void member_function () const;
};

extern "C" foo * get_foo ();

#endif // PRX2_H
```

prx2.cpp:

```
#include "prx2.h"
#include <cstdio>
#include <sys/prx.h>

SYS_MODULE_INFO (prx2, 0, 1, 1);
SYS_LIB_DECLARE (prx2, SYS_LIB_AUTO_EXPORT |
    SYS_LIB_WEAK_IMPORT);
SYS_LIB_EXPORT (get_foo, prx2);

foo::~foo ()
{
}

void foo::member_function () const
{
    std::puts ("in foo::member_function");
}

extern "C" foo * get_foo ()
```

```
{
    return new foo;
}
```

app2.cpp:

```
#include <cstdlib>
#include <sys/prx.h>
#include <sys/paths.h>
#include "prx2.h"

class prx_loader
{
public:
    prx_loader (char const * path)
    {
        id_ = sys_prx_load_module (path, 0, NULL);
        int module_result;
        sys_prx_start_module (id_, 0, NULL,
                               &module_result, 0, NULL);
    }

    ~prx_loader ()
    {
        int module_result;
        sys_prx_stop_module (id_, 0, NULL,
                              &module_result, 0, NULL);
        sys_prx_unload_module (id_, 0, NULL);
    }
private:
    sys_prx_id_t id_;
};

extern "C" int sys_libc;
extern "C" int sys_libstdcxx;

int main ()
{
    sys_prx_register_library (&sys_libc);
    sys_prx_register_library (&sys_libstdcxx);
    prx_loader loader (SYS_APP_HOME "/prx2.sprx");

    foo * f = get_foo ();
    f->member_function ();

    return EXIT_SUCCESS;
}
```

prx2 sample makefile:

```
# Makefile for prx2 example
CXX      = ps3ppusnc
CXXFLAGS = -g -O0
LD       = ps3ppuld
LDFLAGS  =
RUN      = ps3run
RUNFLAGS = -p -q -r -f . -h .

APP_LIBRARIES = libc_libent.o libstdc++_libent.o
PRX_LIBRARIES = -lc_stub -lstdc++_stub
```

```

# uncomment the two following lines to experiment
# with "no toc restore" mode.
#CFLAGS += -Xnotocrestore=2
#LDFLAGS += --notocrestore

.PHONY : all
all : prx2.sprx app2.self

.PHONY : clean
clean :
    -rm -f prx2.o prx2.sprx
    -rm -f prx2_stub.a prx2_verlog.txt
    -rm -f app2.o app2.self

.PHONY : run
run : app2.self
    $(RUN) $(RUNFLAGS) $^

prx2.o : prx2.cpp prx2.h
    ppu-lv2-g++ -o $@ -c -g -O0 \
        -fno-exceptions -fno-rtti prx2.cpp

app2.o : app2.cpp prx2.h

# links the library PRX and creates the
# corresponding stub library.
prx2.sprx prx2_stub.a prx2_verlog.txt : prx2.o
    $(LD) --oformat=fsprx -o prx2.sprx \
        $(LDFLAGS) $^ $(PRX_LIBRARIES)

app2.self : app2.o prx2_stub.a
    $(LD) --oformat=fsself -o app2.self \
        $(LDFLAGS) $^ $(APP_LIBRARIES)

```

This example is likely to be less common than the preceding C example: exposing C++ class interfaces from a library is rarely done because they suffer from the "fragile base class" problem (see http://en.wikipedia.org/wiki/Fragile_base_class). Nonetheless it suffers from the same problem. In this case, calls to virtual methods that are defined in the PRX library may fail because the linker cannot intercept the virtual function call and perform the necessary adjustment of the TOC value.

Solutions

There are two approaches to solving this problem. The first of these requires no code changes, but compromises some of the advantages of the 'no TOC restore' mode whilst the second reduces the compromise, but requires code changes.

SNC compiler -Xnotocrestore=1

The first approach is to use a TOC-restore model in which the compiler emits fully TOC-aware code to implement indirect function calls.

All of the examples thus far have used the compiler's -Xnotocrestore=2 mode. This switch causes the compiler to eliminate the TOC-related machinery from both calls to external functions and calls through pointers.

The linker is able to guarantee that calls to external functions are "safe" in the sense that value of the callee's TOC register will be correct on entry by replacing the stub library code at link time. However, it is not able to make this guarantee for calls through pointers.

Using `-Xnotocrestore=1`, the SNC compiler elides the nop instruction after a call to an external function, but continues to use the full TOC-aware code to perform a call through a pointer.

This compromise allows both of the examples above to function correctly. Unfortunately, it reduces the advantage of 'no TOC restore' mode by eliminating less of the normally unnecessary TOC code.

#pragma control notocrestore=0

The second approach is to identify any indirect calls to a function that is implemented in a PRX library and write a small function that performs the call. This new function is marked with compiler pragmas to indicate that the 'no TOC restore' option should be disabled in that context.

In the example of `app1.c` above, the code:

```
int main ()
{
    ...
    callback_ptr cb = get_callback ();
    (*cb) ();
    ...
}
```

becomes:

```
#pragma control %push notocrestore=0
#pragma noline
void invoke_callback (callback_ptr cb)
{
    (*cb) ();
}
#pragma control %pop notocrestore

int main ()
{
    ...
    callback_ptr cb = get_callback ();
    invoke_callback (cb);
    ...
}
```

It is important to disable both the 'no TOC restore' mode and to prevent the `invoke_callback` function from being inlined. The optimization controls work on a per-function basis so if `invoke_callback()` was inlined, the attempt to change the no-TOC-restore control value would be ineffective.

For this reason we must use two pragmas: one to disable 'no TOC restore' mode:

```
#pragma control %push notocrestore=0
```

the second to prevent the function from being inlined:

```
#pragma noline
```

Finally, we restore the previous state of the 'no TOC restore' mode:

```
#pragma control %pop notocrestore
```

A modification of the C++ example is along the same lines. This time, however, we are able to introduce a "proxy" class for `foo` (named `foo_proxy` in the snippet below) and a smart pointer class, `ntr_ptr` (no-toc-restore pointer) to enable us to minimize the number of changes to each call of `foo`'s virtual methods.

```
// Declare a class that will act as a proxy for
```

```
// calls from the application to the PRX.
class foo_proxy
{
public:
    typedef foo proxied_type;
    explicit foo_proxy (foo * f) : f_ (f) { }

    void member_function () const;

private:
    foo * f_;
};

// The stub function that will call foo::member_function.
// We use pragmas to set the notocrestore control to 0 for
// ABI-compliant TOC handling and to prevent this function from
// being inlined.
#pragma control %push notocrestore=0
#pragma control noline
void foo_proxy::member_function () const
{
    f_->member_function ();
}
#pragma control %pop notocrestore
```

```
// A "smart pointer"-type template class that
// will ensure that the corresponding proxy class
// is used when performing member function calls.
template <class Proxy>
class ntr_ptr
{
public:
    typedef typename Proxy::proxied_type
        proxied_type;

    explicit ntr_ptr (proxied_type * p)
        : proxy_ (p) { }
    Proxy const * operator-> () const
    {
        return &proxy_;
    }
    Proxy * operator-> ()
    {
        return &proxy_;
    }

private:
    Proxy proxy_;
};
```

To use the proxy class and template class above, we change the implementation of main() a little so that:

```
int main ()
```

```
{  
    ...  
    foo * f = get_foo ();  
    f->member_function ();  
    ...  
}
```

becomes:

```
int main ()  
{  
    ...  
    ntr_ptr<foo_proxy> f (get_foo ());  
    f->member_function ();  
    ...  
}
```

Obtaining a TOC usage report

The linker can optionally emit a report which shows the TOC assignments and shims required by the link. This can be useful when trying to gain an understanding of the performance and code size impact of TOC shims (see "TOC shims" on page 33).

The TOC usage report is broken into three sections:

- The "TOC Module Sizes" section. This shows each module along with the amount of TOC data that it contains.
- The "TOC Assignments" section. This shows the TOC region assigned to each of sections within the object modules being linked. The linker also performs a static analysis to discover the names of any functions called that lie in different TOC regions (and hence will require a TOC shim). These are listed within each of the sections.
- The "TOC Stats" section. This lists each of the TOC regions, and shows their sizes and addresses.

The report is written as tab-separated text to enable straightforward analysis by other tools.

Command-line switch

Switch	Description
<code>--print-toc-info</code>	Writes a TOC assignment report to stdout.

8: Building PRX files

PRX generation

Command-line switches

Switch	Description
<code>-mprx</code>	Equivalent to <code>--oformat=prx</code> . For compatibility with GCC.
<code>-mprx-with-runtime</code>	Equivalent to <code>--oformat=prx --prx-with-runtime</code> . For compatibility with GCC.
<code>--oformat=prx</code>	Create PRX output
<code>--oformat=fsprx</code>	Create signed PRX output
<code>--prx-with-runtime</code>	Link the compiler runtime libraries with the PRX output. (Only effective if <code>--oformat</code> is "prx" or "fsprx".)
<code>--strip-unused</code>	Enables the dead-stripping of code. The linker will scan the object files and archives to build the program's complete call tree. Any functions that are found to be unnecessary will be removed from the final PRX file. See "Dead-stripping" on page 27.
<code>--strip-unused-data</code>	Implicitly enables <code>--strip-unused</code> . Besides scanning for dead code, the linker will also locate unused data objects and delete them from the PRX file. See "Dead-stripping" on page 27.
<code>--zgc-sections</code>	Enable dead-stripping for PRX output. For compatibility with GCC. The linker will select <code>--strip-unused-data</code> and issue a warning: warning: L0153: <code>--zgc-sections</code> is deprecated: using <code>--strip-unused-data</code> for dead-stripping
<code>--zgenentry</code>	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
<code>--zgenprx</code>	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
<code>--zgenstub</code>	Causes the linker to simply perform the first link, and pass '--stub-archive' to the libgen tool. See Cell OS Lv-2 PRX Programming Guide.

9: Post-link processing steps

Before it is possible to run a program on the PS3, two post-link processes must be performed:

- `ppu-lv2-prx-fixup --stub-fix-only`
- `make_fself` or `make_fself_npdrm`

When using GCC to perform links, the first of these is performed by the compiler driver (which means that it is normally transparent to the user) and the `make_fself` must be performed manually.

The SN linker is able to perform both of these steps as part of the linking process itself. By eliminating the need to post-process the ELF file and, in the case of `make_fself`, make a complete copy, this eliminates a significant amount of disk I/O. This should result in significantly reduced iteration times: an average of 30% in our tests.

PRX fixup

The '`ppu-lv2-prx-fixup --stub-fix-only`' phase of creating a valid PS3 program has been fully integrated into the linker. This improves the time taken for the linking phase by avoiding the additional disk accesses that must be performed by a program that post-processes the linker's ELF file. There are a number of switches that provide complete control over this process:

Switch	Description
<code>--prx-fixup</code>	Perform the PRX fix-up step (default). Tells the linker to perform a PRX fixup and is the default unless <code>--gnu-mode</code> is specified.
<code>--no-prx-fixup</code>	Do not perform the PRX fix-up step. This disables the automatic 'PRX stub fix' phase. This is the default if <code>--gnu-mode</code> is specified.
<code>--external-prx-fixup</code>	Use the external ' <code>ppu-lv2-prx-fixup --stub-fix-only</code> ' tool rather than the internal mechanism. (Default is to perform the PRX fix-up internally for improved performance.)

Make FSELF

For the best performance, use the `--ofORMAT=fself` switch (this is the default when using the compiler driver).

Switch	Description
<code>--compress-output</code>	Compresses FSELF output. Must be used with <code>--ofORMAT=fself</code> or <code>--ofORMAT=fself_npdrm</code> .
<code>--ofORMAT=elf</code>	Create ELF output (default).
<code>--ofORMAT=fself</code>	Create SELF (make signed-ELF) output.

`--oformat=fself_npdrm`

Create network SELF output.

`--write-fself-digest`

Creates an SHA-1 digest in the SELF header. Using this switch will increase link time. (Default is off; only effective if `--oformat=fself` or `--oformat=fself_npdrm` is used.

10: Troubleshooting

Errors and warnings

For a list of linker error codes and their meanings, use the `--show-messages` command-line switch.

'L0065 Another location found for ...' warning

This warning occurs when there is more than one possible location found in the linker script for a section being linked. Edit your linker script or use the `-sn-first` or `-sn-best` command line switches to address this issue.

See "Resolving ambiguous locations for sections in the linker script" on page 51 for more information.

'L0280 Definition of symbol ... overrides definition from ...' warning

This warning indicates that the linker found multiple definitions for a symbol, but that one of the definitions had a higher precedence than the others. The linker will continue linking using the preferred symbol definition.

According to the ELF specification, multiple definitions result in an error. However, the SN linker allows multiple definitions in certain circumstances where a single definition has clear precedence.

See "Overriding symbols" on page 52 for more information.

Resolving ambiguous locations for sections in the linker script

Ambiguities occur when there is more than one possible location in the linker script for a section from one of the linked objects. The following style of warning will be produced in such cases.

```
Command line : warning: L0065:Another location found for .text section from
file C:\so\o1.o
Command line : warning: First location '*(.text)', second 'o1.o(.text)'
Command line : warning: Linker will use the best location for this section
Command line : warning: Use -sn-best to remove warnings. Use -sn-first to
use first location in linker script
```

In this case there is an ambiguity about where the text from the file `o1.o` should go. The linker switches `-sn-first` and `-sn-best` allow control over how the linker script is interpreted. `-sn-first` will use the first matching location found in the linker script. `-sn-best` will match the best location for the section; it will do this without producing any warnings. The default behavior is to match the best location. If this location in the script is not the first location, then a warning is produced.

Overriding symbols

Typically, multiple definitions cause error L0019, as required by the ELF specification. However, the SN linker permits multiple definitions in the situation where one definition is found in an object file and another is found in an archive. In this case, symbols from object files are given higher precedence and there is no error. However, warning L0280 will be emitted. For example:

```
Command line : warning: L0280: definition of symbol `.foo' from  
"foo_replace.o" overrides definition from "libx.a(foo.o)"
```

This behavior supports selectively overriding symbols from archives, such as replacing functions from the standard libraries. For example, the standard library versions of malloc and free could be overridden.

Note: caution is advised when replacing functions from the standard library. Since all calls to the overridden function—even those from within the standard library—will be redirected, the results may be unpredictable. In particular, replacing malloc without replacing free is likely to cause memory allocation problems.

In cases where multiple definitions are found in object files or are found only in archives, error L0019 will be emitted.

11: Index

B

Background 35
Branch shims 34
Building PRX files 48

C

Command-line switch 47
Command-line switches 27, 48
Command-line syntax 7

D

Dead-stripping 27
De-duplication 28
De-duplication and debugging 28
Default linker script 18
Document version history 5
Dot sections 25

E

Effect of command line order on linker output 16
Eliminating TOC overhead 36
Errors and warnings 51

F

Function 31

I

Ignored linker switches 16
Input packager 8
Introduction 5

,

'L0065 Another location found for ...'
warning 51
'L0280 Definition of symbol ... overrides
definition from ...' warning 51

L

LIB_SEARCH_PATHS 22
Limitations 39
Linker command-line syntax 7
Linker script directives 18
Linker scripts 18
Linker switches 8

Long-branching TOC shims 33

M

Make FSELF 49
Memory requirements 6
Millicode 34

N

No TOC restore C sample (prx1) 40
No TOC restore C++ sample (prx2) 42

O

Obtaining a TOC usage report 47
Overriding symbols 52
Overview of the linker 5

P

Performance 6
Post-link processing steps 49
Pragma comment 26
PRX calls with --notocrestore 39
PRX calls without --notocrestore 38
PRX fixup 49
PRX generation 48

R

Referencing files in linker scripts 22
REQUIRED_FILES 23
Resolving ambiguous locations for sections
in the linker script 51

S

Section start and end pseudo-symbols 25
Section symbols 25
Sections 22
Shim generation 33
Short-branching TOC shims 33
SN linker command-line switches 36
SN linker --notocrestore switch 37
SNC compiler -Xnotocrestore control-
variable 37
SNC PPU C/C++ compiler control-variable 36
Solutions 44
STANDARD_LIBRARIES 24
Strip report 29
Stripping unused code and data 27
Switch processing order 7

T

TOC information 35
TOC shims 33
Troubleshooting 51

U

Undefined symbols 28
Unimplemented linker switches 15
Unsupported script file directives 21
Unused objects 29

