User Guide to

ProDG Linker for PlayStation®3

SN Systems Limited
Version v470.1
February 27, 2015

Contents

1: Introduction

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 ProDG 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.

What's New

v420.1	*Added:* --oml switch. (B#99767) ADDRESS directive. Description of AFTER directive. "Code safety when using de-duplication" section. *Updated:* --sn-no-dtors switch. Removed --Ur switch. (B#99738)
v430.1	*Added:* "Limitations of de-duplication" and "Rewriting code for more effective de-duplication". "Trade-offs" added to "Dead-stripping and de-duplication" section. "Improving link-time performance" added to "Troubleshooting" section. *Updated:* Removed --no-remove-duplicate-inputs switch. (B#101254)
v460.1	*Updated:* Minor text corrections.

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"). 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://msdn.microsoft.com/en-us/library/ff556232.aspx).

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

- Ensure that the system has sufficient RAM. Since the link itself is I/O bound, it is 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.

For more information, see "Improving link-time performance".

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 are 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

The linker command-line syntax is as follows:

```
ps3ppuld <switches> <files>
ps3ppuld @<response filename>
```

<switches>	Any of the command-line switches described in the following section. These must be preceded with a hyphen '-'. See warning below.
<files>	Can be object files or libraries. The linker will use these files in addition to any files specified in the linker script. The wildcard characters "?" and "*" can be used to specify files. The question mark character will match a single character, and the asterisk will match multiple characters. For example to link all object files in the current folder, specify `*.o`.
<response filename>	The path of a file containing switches and filenames for the linker. The content of the response file is treated as if it were provided on the command line. However, options and filenames may be delimited with newlines as well as spaces within the response file.

Warning: 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.

Example: `--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. To invoke the linker using a link response file, call the linker executable with the argument '*@<link response filename>*'.

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.

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

Switch	Description
`-( | --start-group`	Start a group.
`-) | --end-group`	End a group.
`--32bit`	Forces the 32-bit linker to be used when using a 64-bit operating system.
`--callprof`	Add profiling code to the final output (for use with SN Tuner - see *User Guide to Tuner*)
`--comment`	Keep comment sections.
`--comment-report=`*`<file>`*	Write a report in *<file>* showing the comment for all the files in the link.
`--compress-output`	Compresses FSELF output. Must be used with `--oformat=fself` or `--oformat=fself_npdrm`.
`-d | --dc | --dp`	Force common symbols to be defined.
`--deep-search`	Enclose all libraries and object files within `--start-group`

	and `--end-group`.
`--default-paths`	Add default paths from the linker script LIB_SEARCH_PATHS element.
`--defsym=<symbol>=<value>`	Define a symbol *<symbol>* with value *<value>*.
`--disable-warning=<value>`	Disables a warning message. The argument is an error number without the leading 'L'. For example, "`--disable-warning=95`" will disable warning L0095. Several messages may be disabled with a single switch by providing a comma-separated list of error numbers. For example, "`--disable-warning=207,24`" will disable both warning L0207 and L0024.
`--discard-all`	Discard all local symbols.
`--discard-locals`	Discard temporary local symbols.

`--dont-strip-section=<section>`	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 `--dont-strip-section` switches.
	Example:
	`--dont-strip-section=.dont_strip1`
	`--dont-strip-section=.dont_strip2`
	The above switches will ensure that data in sections named ".dont_strip1" and ".dont_strip2" are not stripped.

`-dfl | --dump-file-layout`	Prints a view of the sections and segments in an ELF file based on file offsets.
`--eh-frame-hdr`	Silently ignored for compatibility with GNU compiler driver.
`--enable-warning=<value>`	Enables a disabled warning message. The argument is an error number without the leading 'L'. For example, "`--enable-warning=95`" will enable warning L0095. Several messages may be enabled with a single switch by providing a comma-separated list of error numbers. For example, "`--enable-warning=207,24`" will enable both warning L0207 and L0024.
`-e<symbol> | --entry=<symbol>`	Set start address.
`--exceptions`	Add linker default paths for libraries with exception handling (from the LIB_SEARCH_PATHS 'exceptions' key). See "Exceptions and RTTI".
`--external-prx-fixup`	Use the SDK ppu-lv2-prx-fixup tool rather than the internal mechanism. (Default is to perform the PRX fix-up internally for improved performance.) See "PRX fixup".
`--gc-sections`	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 `--strip-unused-data` 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 dead-stripping is enabled, the symbols listed in *<file>* will not be stripped. **Note:** When de-duplication is enabled, this switch does not inhibit de-duplication of the symbol and its underlying code/data.
--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, for example -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>*. **Tip**: Disable map file creation to improve link time.
--md=*<type>*	Perform "minidump" crash diagnostic. *<type>* is the type of diagnostic. **Note:** This switch is only supported under Windows.
-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".

`--no-default-paths`	Do not add default paths from the linker script LIB_SEARCH_PATHS element.
`--no-default-script`	Do not attempt to locate the default linker script if none was present on the command line.
`--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) (default). Enables the removal of exception-handling data from the final output file. See "Exceptions and RTTI".
`--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: LO154: 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".
`--no-required-files`	Do not add required files from the linker script REQUIRED_FILES 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-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. Formats other than those

	listed will yield an 'unrecognised output format' error: `'elf', 'fself', 'fself_npdrm', 'prx' or 'fsprx'`
`--oml`	Generate the additional data required for Object Module Loading (OML) support. For further information see "Loading object modules" in the Debugger user guide. **Caution:** Using this switch will cause a marginal reduction in link-time performance.
`--package`	Place all inputs and command-line options into a zip file. See "Input packager".
`--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`	Stores a hash of the program in the output file. This aids the debugger in locating debugging information for PRX modules. It is also required for automated core-dump support. **Note:** Enabling this feature will increase the link time.
`--print-embedded-symbols`	Prints the generated symbols for all binary data embedded in the final link. See "SPU Embedding command line options".
`--prx-fixup`	Perform the PRX fix-up step (default). See "PRX fixup".
`--prx-with-runtime`	Link the compiler runtime libraries with the PRX output. (Only effective if `--oformat` is "`prx`" or "`fsprx`".)
`-Qy`	Silently ignored for compatibility with GNU compiler driver.
`-r | --relocatable | --relocateable`	Generate relocatable output.
`--remap-source-paths=<file>`	Adds the list of file mappings in *<file>* to the output ELF file so that the debugger can use them for remapping its paths to the source code. See "Remapping source paths".
`--required-files`	Add required files from the linker script REQUIRED_FILES 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".
`--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".
`--sn-full-map`	Provides additional information in the map file, e.g. static variables. **Tip**: Disable map file creation to improve link time.
`--sn-no-dtors`	Do not call the destructors for global objects. If used in conjunction with `--strip-unused` or `--strip-unused-data` then the unused destructor code from global objects will be removed from the output file. **Note:** If you specify this switch when invoking the linker through the GCC driver it will be ignored and a warning will be issued.
`--sort-common`	Sort common symbols by size.
`--spu-format=<format>`	Specify embed format of SPU ELF files. Available options are `'binary'`, `'default'`. See "SPU Embedding command line options".
`--strip-duplicates`	This switch enables the de-duplication process. This must be used in conjunction with one of the dead-stripping options (`--strip-unused` or `--strip-unused-data`). See "Dead-stripping and de-duplication".
`--strip-report=<file>`	Creates *<file>* that shows the code and references within the program. Use this report to discover which functions and data are being dead-stripped. Example: `--strip-report=stripreport.txt` See "Strip report" for details.
`--strip-unused`	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 and de-duplication".
`--strip-unused-data`	Implicitly enables `--strip-unused`. Besides scanning for dead code, the linker will also locate unused data objects

	and delete them from the ELF file. See "Dead-stripping and de-duplication".
`--sysroot`	Sets the 'sysroot' directory prefix. This is used in the event that a library search path (set with the `--library-path/-L` option) begins with a '='. The '=' is replaced by the sysroot directory prefix.
`--Tbss=`*`<addr>`*	Set *`<addr>`* as starting address of bss section.
`--Tdata=`*`<addr>`*	Set *`<addr>`* as starting address of data section.
`--Ttext=`*`<addr>`*	Set *`<addr>`* as starting address of text section.
`-t | --trace`	Print names of files as they are opened.
`--temp-dir=`*`<path>`*	Specify a temporary directory to be used for temporary files.
`--toc-report=`*`<file>`*	Produces a dump detailing TOC data in *`<file>`*. See "Obtaining a TOC usage report".
`-u`*`<symbol>`* `|` `--undefined=`*`<symbol>`*	Link the inputs as if *`<symbol>`* were undefined. This switch has the same effect as the EXTERN linker script directive. When dead-stripping is enabled, *`<symbol>`* will not be stripped. **Note:** When de-duplication is enabled, this switch does not inhibit de-duplication of the symbol and its underlying code/data.
`--use-libcs`	Replace libc.a by libcs.a during linking.
`-V | -v | --version`	Print version information.
`--verbose`	Output lots of information during link.
`--Wall`	Produce all warnings.
`--warn-built-before`	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'.
`--warn-common`	Warn about duplicate common symbols.
`--warn-if-debug-found`	Warn if a section known to contain debug information is found. The following section names will produce the warning: .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
`--warn-once`	Warn only once per undefined symbol.
`--Werror`	Treat warnings as errors.
`--whole-archive`	Include all objects from following archives. For each archive mentioned on the command line after the `--whole-archive` option, include every object file in the archive in the link, even if no references are made to those object files. The behavior can be turned off using the `--no-whole-archive` switch, such that libraries following the second switch will then link normally. Example: `ps3ppuld main.o lib1.a --whole-archive lib2.a --no-whole-archive lib3.a` In this example `lib2.a` will have all its modules included even if not referenced. `lib1.a` and `lib3.a` will be linked normally. **Tip:** Dead stripping and deduplication will not be inhibited by use of this switch.

`--wrap=`*`<symbol>`*	Use wrapper functions for *`<symbol>`*.
	When a symbol `foo` is wrapped, the linker will redirect references to `foo` to `__wrap_foo` instead, and allow access to `foo` itself through `__real_foo`.
	For example, to intercept calls to `fopen()`, define a function `__wrap_fopen()` that performs any extra logic and then calls `__real_fopen()`, and pass `--wrap=fopen` to the linker.
	Within C++ code, the wrapper functions must have C linkage and the symbol to be wrapped will be the function's C++ mangled name.
	For example, to replace `int example (char const *)`, pass the linker `--wrap=_Z7examplePKc` when linking the following code:
	`extern "C" int __real__Z7examplePKc (char const *);`
	`extern "C" int __wrap__Z7examplePKc (char const * input)`
	`{`
	`  printf ("Calling example (%s)...\n", input);`
	`  return __real__Z7examplePKc (input);`
	`}`

	Tip: To replace or instrument memory allocation functions, the runtime library provides a simpler mechanism, which is preferred over the use of `--wrap`. For more information, see "Appendix F: Replacing The Memory Management Functions for PPU" in "C and C++ Standard Libraries - Overview and Reference".
`--write-args-to-output`	Writes the command-line arguments to the `.command_line` section of the output file.
`--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.)
`-X`	Discard temporary local symbols.
`-x`	Discard all local symbols.
`--zgc-sections`	PRX fix-up dead-stripping switch. The linker will select `--strip-unused-data` and issue a warning: `warning: L0153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping`
`--zgenentry`	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
`--zgenprx`	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
`--zgenstub`	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.
`--zlevel=`*`<openlevel>`*	Passed directly to the ppu-lv2-prx-libgen tool.

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.

Remapping source paths

The `--remap-source-paths=<file>` switch is used if you wish to build an ELF using code in one location, but debug it using code from another. If you try to do this without this switch then the paths to the source code stored in the ELF will be incorrect, and the debugger will not be able to find the source.

The file contains a list of mappings from original source paths, to new source paths.

For example:

```
c:\my_source\code_1 \\my_server\code_area_1
c:\my_source\code_2 \\my_server\code_area_2
```

In this case any source files with paths beginning `c:\my_source\code_1` will be searched for in `\\my_server\code_area_1`, and any source files found in `c:\my_source\code_2` will be searched for in `\\my_server\code_area_2`. Any other source files will retain their original path.

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".

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.
`ADDRESS`	Used with overlay definitions to force an overlay to a specific address.
`ADDR(<section>)`	Return the absolute address (the VMA) of the named section. Your script must previously have defined the location of that section.
`AFTER`	Used with overlay definitions to specify overlay output.
`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:** 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 cannot 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:** 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".
`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".
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".
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".
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.

> **Tip:** 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_FILES` directive in the default linker script looks like this:

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

```
    ecrtn.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 "`.`" 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 "`.`" 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-gb/library/7f0aews7(v=vs.100).aspx.

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

5: Dead-stripping and de-duplication

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.

> **Tip:** Disable dead-stripping on debug builds to improve link time.

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

`--dont-strip-section=<section>`	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 `--dont-strip-section` switches:
	`--dont-strip-section=.dont_strip1`
	`--dont-strip-section=.dont_strip2`
	The above switches will ensure that data in sections named ".dont_strip1" and ".dont_strip2" are not stripped.

`--gc-sections`	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 `--strip-unused-data` and issue a warning: `warning: LO153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping`
`--strip-duplicates`	This switch enables the de-duplication process. This must be used in conjunction with one of the dead-stripping options (`--strip-unused` or `--strip-unused-data`).
`--strip-report=<file>`	Creates *<file>* that shows the code and references within the program. Use this report to discover which functions and data are being dead-stripped. Example: `--strip-report=stripreport.txt` See "Strip report" for details.
`--strip-unused`	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.

`--strip-unused-data`	Implicitly enables `--strip-unused`. 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);
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 they point to the single remaining copy.

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

> **Tip:** Disable de-duplication on debug builds to improve link time.

Code safety when using de-duplication

Since de-duplication works by combining identical objects, code that relies on the address of global objects for control flow is unsafe.

Consider the following simplified code example:

```
const float a [] = {1.0f, 0.0f, 0.0f};
const float b [] = {1.0f, 0.0f, 0.0f};

void behaviour_a (float const *);
void behaviour_b (float const *);

void some_code (float const * param)
{
    if (param == a)
        behaviour_a (param);
    else
        behaviour_b (param);
}

int main ()
```

```
{
    some_code (a);
    some_code (b);
}
```

In the above code, 'a' and 'b' will be de-duplicated, so they will refer to the same data and therefore they will have the same address. When main is run, both executions of the if statement will take the same path.

> **Note:** It is undefined which branch the if statement takes, that is whether 'a' de-duplicates to 'b' or 'b' de-duplicates to 'a'.

Since functions can also be de-duplicated, code like the following is also dangerous:

```
typedef void (*fn_ptr) (int);
void f (int);
void g (int);

void behaviour_a ();
void behaviour_b ();

void some_code (fn_ptr fn)
{
    if (fn == f)
        behaviour_a ();
    else
        behaviour_b ();
}

int main ()
{
    some_code (f);
    some_code (g);
}
```

In the above code, 'f' and 'g' may be de-duplicated, in which case they will have the same address. Once again, the if statement will take the same path in both executions.

Instead of comparing pointers, using an auxiliary enumeration (or similar) is safer:

```
enum behaviour_type
{
    behaviour_type_a,
    behaviour_type_b,
};

const float a [] = {1.0f, 0.0f, 0.0f};
const float b [] = {1.0f, 0.0f, 0.0f};

void behaviour_a (float const *);
void behaviour_b (float const *);

void some_code (behaviour_type bt)
{
    if (bt == behaviour_type_a)
        behaviour_a (a);
    else
        behaviour_b (b);
}
```

```
int main ()
{
    some_code (behaviour_type_a);
    some_code (behaviour_type_b);
}
```

This code correctly invokes `behaviour_a ()` and then `behaviour_b ()`.

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.

Using de-duplication with Tuner

When using Tuner you are advised not to use de-duplication because issues may arise when functions identical to the sync function selected in Tuner are de-duplicated, which can cause Tuner to incorrectly detect a new frame. In some instances, the symbol for a de-duplicated function may be removed, making it unavailable for use as the sync function within Tuner.

If de-duplication is required when profiling, then it is recommended that the sync function is carefully selected to avoid the risk of de-duplication affecting profiling behavior.

Using de-duplication with TOC

The presence of multiple TOC regions within a program reduces the effectiveness of code de-duplication. In general, functions that use the TOC can only de-duplicate against functions within the same TOC region, since the TOC references make the code look different when the linker analyzes them. Additionally, calls to functions within the same source file inhibit de-duplication of the called functions, because the compiler's code generation assumes that both the callee and caller are in the same TOC region, which is an assumption that de-duplication could break.

To avoid these problems, link with the `--no-toc-restore` or `--no-multi-toc` switches when possible. See SN linker --notocrestore switch. Substantial improvements to the amount of code de-duplication are possible when these options are used.

Limitations of de-duplication

De-duplication works by identifying code or data objects that are identical in terms of their contents and the external objects they reference, if any.

For example, the following two functions would be considered duplicates:

```
extern int c ();

int a ()
{
    return c () + 5;
}

int b ()
{
    return c () + 5;
}
```

De-duplication only performs a single pass to determine code or data objects that are identical. This can lead to some code or objects that appear to be identical to be left in the output.

For example, if the following two external functions `'f ()'` and `'g ()'` are de-duplicated, the functions `'d ()'` and `'e ()'` would not be considered duplicates, because they reference distinct functions when the single de-duplication pass occurs.

```
extern int f ();
extern int g ();

int d ()
{
    return f () + 5;
}

int e ()
{
    return g () + 5;
}
```

The effectiveness of de-duplication is also limited when dealing with templated objects or data. In the following example the compiler will typically emit the string constant separately for each instantiation of the function template. As a result, the string objects will be de-duplicated, but none of the `'message<T>'` instantiations will be eligible for de-duplication.

```
template <typename T>
char const * message ()
{
    return "foo";
}
```

The same limitation also applies to constants that are members of templated classes:

```
template <typename T>
class Test
{
public:
    char const * message () const
    {
        return message_;
    }

private:
    static char const message_ [];
};

template <typename T>
char const Test::message_ [] = "bar";
```

The static data member 'message_' will be de-duplicated across instantiations of 'Test<T>', but the member function 'message ()' will not be.

Similarly, two functions returning constant values will not be de-duplicated if the constants are expressed by the compiler as separate data objects. For example:

```
__vector float vf1 ()
{
    return (__vector float) {1, 0, 0, 0};
}

__vector float vf2 ()
{
```

```
    return (__vector float) {1, 0, 0, 0};
}
```

The two functions above will not be de-duplicated, since the compiler must generally output code similar to the following, depending on the optimisation level and the values of the constants:

```
const __vector float __vf1 = {1, 0, 0, 0};
__vector float vf1 ()
{
    return __vf1;
}

const __vector float __vf2 = {1, 0, 0, 0};
__vector float vf2 ()
{
    return __vf2;
}
```

Note that the two vector constants will be de-duplicated.

Rewriting code for more effective de-duplication

It may be necessary to rewrite code that uses constants or delegated functions in order to gain the most benefit from de-duplication.

For example, the following functions will be de-duplicated as expected:

```
extern const __vector float x_axis; // A single definition is provided elsewhere

__vector float vf1 ()
{
    return x_axis;
}

__vector float vf2 ()
{
    return x_axis;
}
```

For templated classes, it may be necessary to extract constants to a non-template base class, or a global constant, as in the following example:

```
extern char const message []; // A single definition is provided elsewhere

template <typename T>
class Test
{
public:
    char const * message () const
    {
        return ::message;
    }
};
```

Note that this style of rewrite will still only enable the parent object to be de-duplicated, and does not help de-duplicate deeper trees of objects.

Strip report

The *strip report* details the effects of applying the various dead-stripping options.

The report is divided into four sections:

Unused objects

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

The number of bytes that could be stripped (Strip), the number of bytes of padding that remain (Pad), the name of the object, the section the object is in, and the file the object comes from will be displayed.

In the output, values within square brackets following an object file or archive name indicate the object's index within the symbol table.

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).

Example:

```
Legend
S = Referenced from synthesised (e.g. compiler-generated) code
G = Referenced via a global symbol
T = Referenced transitively from another known function
P = Exists within a 'problem section' (e.g. a KEEP section)

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:
    blr

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

    # The compiler may now emit a nop here to ensure the following
    # function is 8 byte aligned. This may not be part of the
    # symbol size.
    nop
.bar:
```

The compiler has aligned function "bar" to 8 bytes. To do this it has aligned the section to 8 bytes, and added a nop to the end of "foo". Unfortunately the symbol for "foo" does not include this nop in its size.

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.

Trade-offs

Both dead-stripping and de-duplication save space within the final binary image. However, they imply certain trade-offs in terms of:

- Loadable image size
- Link time
- Runtime performance
- Ease of debugging and profiling

Loadable image size

The effectiveness of dead-stripping depends on how many unused objects are present in the binary code, while de-duplication relies on identifying identical objects in the binary code. Therefore, their effectiveness depends on the structure of the inputs to the linker.

Link time

Enabling dead-stripping might increase link time, depending on the build scenario and code structure. When all of the input object files and archives are in the operating system file cache, the impact can be significant, for example, doubling the link time is possible. However, when the operating system file cache is empty, the cost of dead-stripping is usually insignificant compared to the impact of file I/O.

Similarly, de-duplication can further increase link time beyond that of dead-stripping. As with dead-stripping, the performance impact will depend on the operating system file cache, as well as the code structure.

Runtime performance

Both dead-stripping and de-duplication can affect the performance of the resulting program, since the physical layout of the code and data will vary depending on what is removed. Since dead-stripping removes unused objects while leaving the order of objects unchanged, any variation in runtime performance usually results from altering the memory layout and runtime cache behavior. As such it can produce either a positive or a negative change in runtime performance.

However, de-duplication alters the physical layout in a way that tends to reduce spatial locality, since any two identical objects anywhere in the program can be merged. For example, two functions in the same source file might not end up close together in memory if one is de-duplicated, leading to calls between those two functions missing the instruction cache when they did not previously, causing a reduction in runtime performance.

Ease of debugging and profiling

De-duplication merges identical objects at the binary level, which can lead to unpredictable results when using tools like Tuner and the debugger, as they rely on mapping between source- and binary-level constructs unambiguously.

For more information, see "De-duplication and debugging" **and** "Using de-duplication with Tuner"**.**

Summary

When loadable image size is not a concern, the best choice is to disable dead-stripping and de-duplication, which will give the fastest possible link times.

If your concerns are balanced between loadable image size and runtime performance, enabling dead-stripping provides the best balance between those issues and fast link times. However, if loadable image size is the critical factor, enabling de-duplication will save slightly more memory in exchange for longer link times.

When debugging, disabling de-duplication will improve the debugging experience. When profiling, disabling de-duplication might be required to interpret the results correctly; alternatively, ensure that the sync function is not de-duplicated. However, beware that de-duplication can affect the semantics of legal C/C++ programs so the dead-stripping/de-duplication settings used for testing builds should match those used for final submission. For more information, see "Code safety when using de-duplication".

Since the exact results depend on many factors, it is best to measure the impact of enabling dead-stripping or de-duplication to ensure that the trade-offs are appropriate for your situation.

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.

> **Tip:** 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".

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 *Register Saving and Restoring Functions* section of the PPU ABI Specifications for Cell OS Lv-2 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: Exceptions and RTTI

Linking with exception-handling support

You can add `--exceptions` to the command line to link the runtime support libraries required for C++ exception handling into the program. These libraries also include support for C++ runtime type information (RTTI).

Switch	Description
`--exceptions`	Add linker default paths for libraries with exception handling (from the LIB_SEARCH_PATHS 'exceptions' key).

Linking without exception-handling support

Alternatively, specifying `--no-exceptions` on the command line will link the program without support for C++ exception handling but note that support for RTTI is still enabled in this configuration.

Switch	Description
`--no-exceptions`	Add linker default paths for libraries without exception handling (from the LIB_SEARCH_PATHS 'no_exceptions' key) (default). Enables the removal of exception-handling data from the final output file.

Warning: The linker does not detect if the objects being linked require exception-handling support, so a mismatch between compiler and linker exception-handling options may cause the program to fail at runtime. Typically, throwing an exception in a program without exception-handling support enabled will result in the runtime calling the `abort()` function.

The SN linker defaults to linking with exception-handling support disabled.

Linking without RTTI support

The SN linker does not provide an option to link without RTTI support. This decision was made in order to provide a simpler interface and because removing RTTI support has little effect; the overhead of RTTI support is a small increase in the size of program, with no runtime penalty. Additionally, it is safe to link code compiled without exception-handling or RTTI support against runtime libraries that support RTTI.

In terms of the SDK libraries included in the link, this means that either the standard or the fno-exceptions libraries can be linked; the fno-exceptions/fno-rtti libraries cannot be used. However, due to the way the linking process works, RTTI support will only be linked in if it is used, so code that makes no use of RTTI will be largely unaffected.

8: 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.

Warning: 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".

SN linker command-line switches

Switch	Description
`--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".
`--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: LO154: there is too much TOC data (>64kB) for a single TOC region (consider removing both --no-multi-toc and --no-toc-restore)`
`--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`.

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
`-Xnotocrestore=0`	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. 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. No special linker switches are necessary for code built with this option to run correctly. This is the default value of the notocrestore control.
`-Xnotocrestore=1`	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.
`-Xnotocrestore=2`	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.

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.

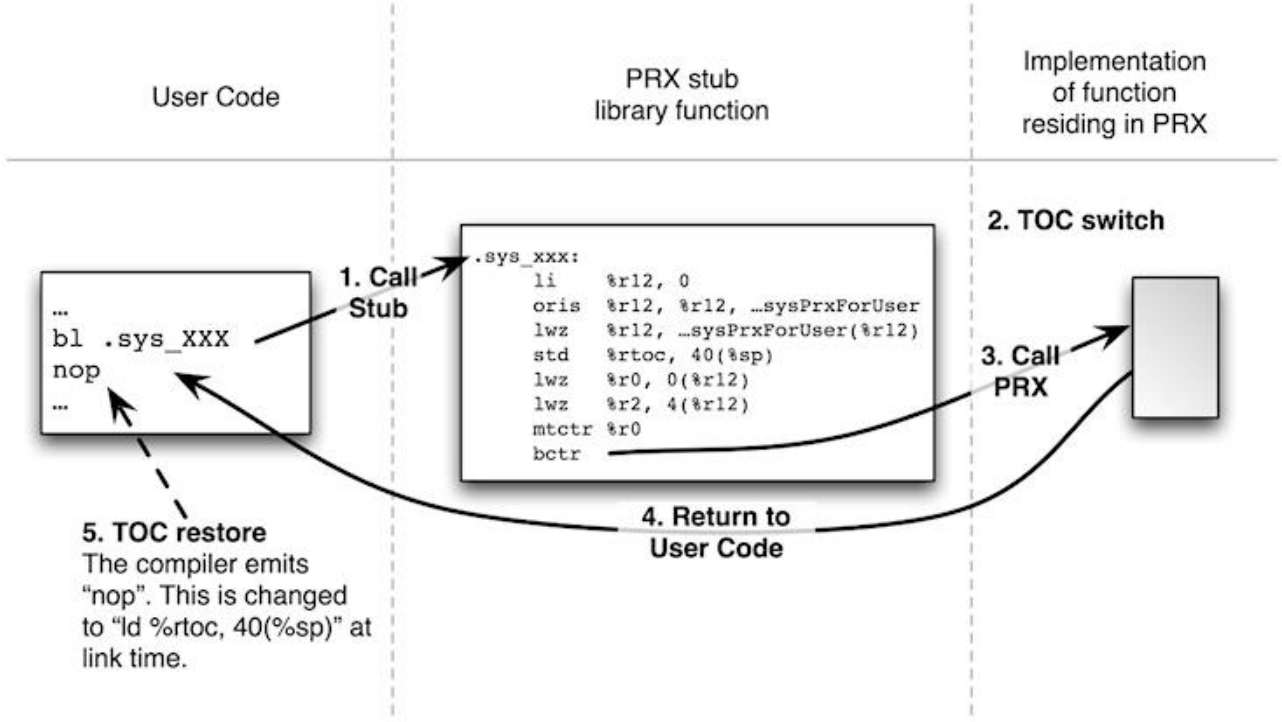
The tradeoff is that this code assumes that the TOC register (`%rtoc`/`%r2`) never needs to change before making a function call, or has to be restored after it. Enforcing that condition needs some assistance from the linker.

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.

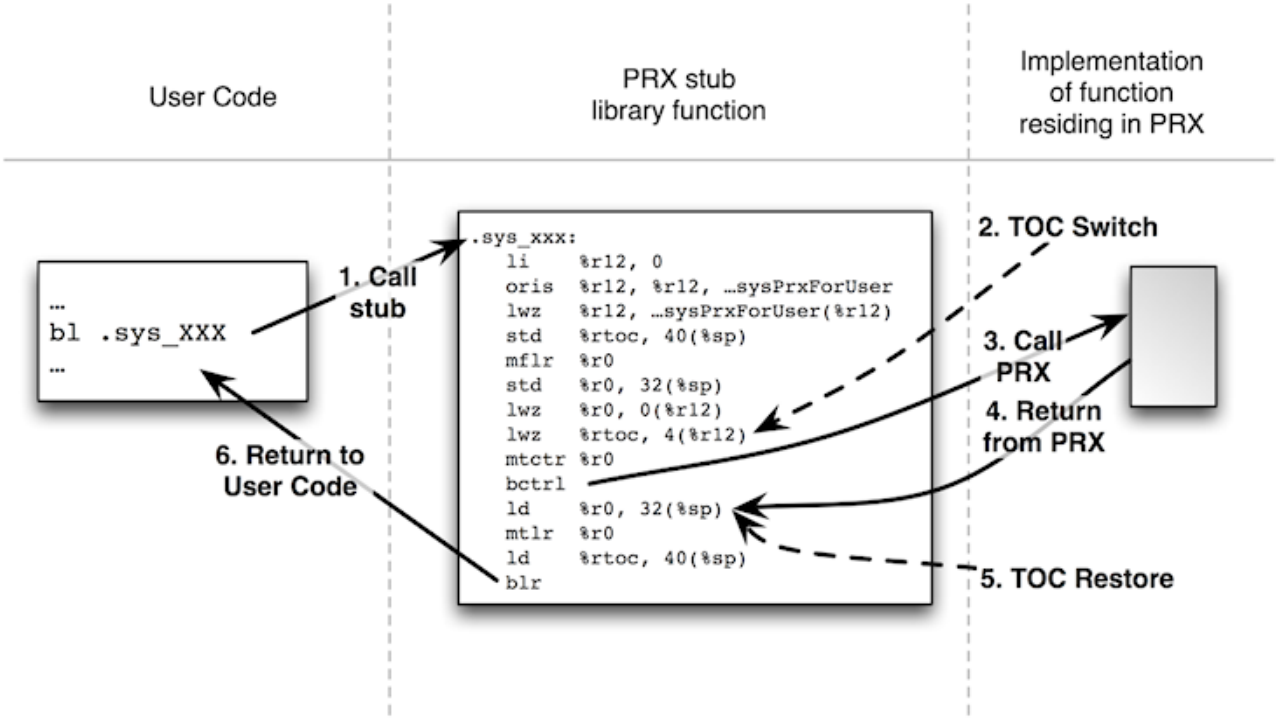
(2) The PRX stub code saves the current value of the TOC register and changes the TOC register so that it references the PRX TOC region.

(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" 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 demonstrates 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 correctly 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' 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=fself -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 noinline
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 noinline
```

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 noinline
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").

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
`--toc-report=<file>`	Writes a TOC assignment report to *<file>*.

9: Building PRX files

PRX generation

Command-line switches

Switch	Description
`-mprx`	Equivalent to `--oformat=prx`. For compatibility with GCC.
`-mprx-with-runtime`	Equivalent to `--oformat=prx --prx-with-runtime`. For compatibility with GCC.
`--oformat=prx`	Create PRX output.
`--oformat=fsprx`	Create signed PRX output.
`--prx-with-runtime`	Link the compiler runtime libraries with the PRX output. (Only effective if `--oformat` is "prx" or "fsprx".)
`--strip-unused`	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 and de-duplication".
`--strip-unused-data`	Implicitly enables `--strip-unused`. Besides scanning for dead code, the linker will also locate unused data objects and delete them from the PRX file. See "Dead-stripping and de-duplication".
`--zgc-sections`	Enable dead-stripping for PRX output. For compatibility with GCC. The linker will select `--strip-unused-data` and issue a warning: `warning: LO153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping`
`--zgenentry`	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
`--zgenprx`	Used when creating PRX output. See Cell OS Lv-2 PRX Programming Guide.
`--zgenstub`	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.

10: Embedding SPU programs

Embedding SPU ELF files using the SN Linker

SPU ELF files can be embedded in the final program by passing them directly to the linker. This removes the need for any intermediate steps between linking an SPU ELF and embedding it in a PPU ELF.

The improved workflow has no impact on the overall link time. SPU ELF files will be processed in the same order as if they were PPU object files on the command line. The linker also generates metadata for each embedded SPU ELF which allows Tuner and the debugger to identify the source. Finally, the linker will also generate the additional metadata required for embedding SPURS ELF files and libovis overlay tables. For more information about libovis see the "libovis Overview".

The linker can embed executable ELF files, shared libraries, SPURS Tasks, SPURS Job 2.0, SPURS Jobqueue jobs and custom SPURS policy modules.

Figures 1 to 3 illustrate the differences between the workflow when embedding an SPU ELF in the final PPU program.

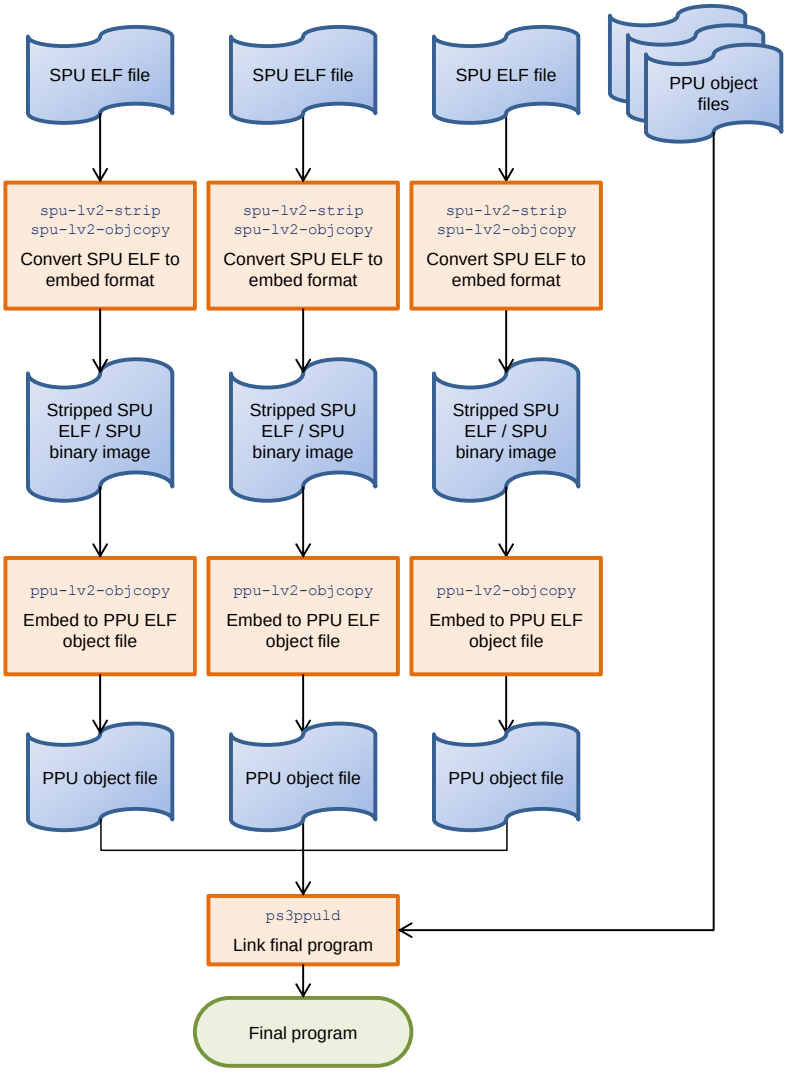


Figure 1: This example shows firstly, that GNU binary utilities are used to format the SPU ELF file data and secondly, that this is embedded into a PPU object ready for the final link.

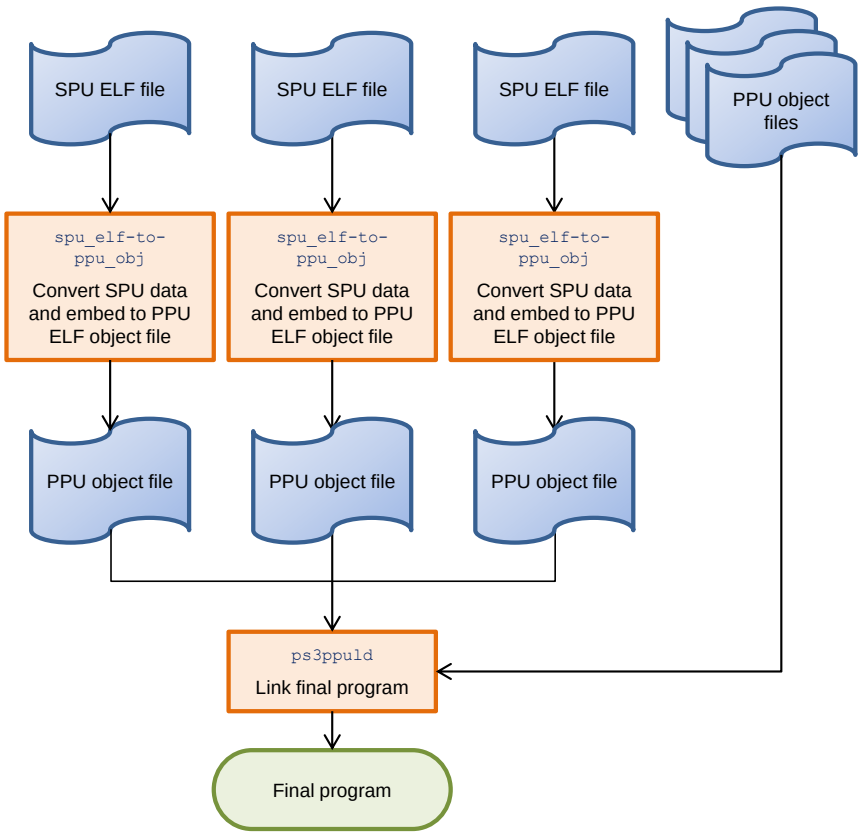


Figure 2: This example uses the single, external tool spu_elf-to-ppu_obj, to format and embed an SPU ELF file into a PPU object file, ready for the final link.

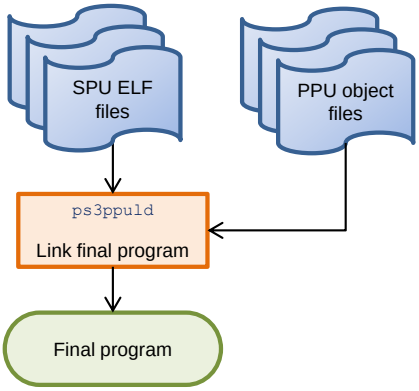


Figure 3: In this example no intermediary tools are necessary as the formatted and embedded SPU ELF files are specified directly on the linker command line and linked as if they were PPU object files.

SPU Embedding command line options

Switch	Description
`--spu-format=<format>`	Specify embed *<format>* of SPU ELF files. Available options are: `'binary'`, `'default'`.
`--print-embedded-symbols`	Prints the generated symbols for all binary data embedded in the final link.

Jobbin2 format support

The linker supports the embedding of SPURS job ELF files in jobbin2 format only.

For more information on embedded files see "SPU Programs Embedded in PPU Programs" in the SPU Program Support Tools user guide.

Explicitly specifying an embed format of binary will not change the format used to embed SPURS jobs. The jobbin2 format is preferred as it reduces the size of the embedded image by removing the zero-initialized data (.bss) section that is included in standard binary format images. SPURS jobs must meet the required binary specifications to be embedded correctly.

For more information on binary specifications see "Job Binary Specifications" in "libspurs Overview".

Requirements for embedding

SPU ELF files must be provided to the linker unstripped, because the linker uses the symbol table and section header table information to identify the type of SPU ELF file that is being embedded. All unnecessary data will be stripped from the SPU ELF by the linker when performing internal embedding.

Accessing embedded SPU ELF from main program

The linker will generate start and end symbols that reference the SPU image, and a size symbol that describes the size in bytes of the SPU image. Additional symbols may be generated for any metadata created. Due to the mangling of C++ symbol names, `extern "C"` must be used when declaring these symbols in C++ source files. The following table lists the symbols generated:

Symbol name	Description
`_binary_<program_type>_<filename>_start`	Start address of the SPU program.
`_binary_<program_type>_<filename>_end`	End address of the SPU program.
`_binary_<program_type>_<filename>_size`	Size of the embedded SPU program binary.
`_binary_<program_type>_<filename>_ovlytable`	References a statically generated overlay table required when using libovis. `cellOvisInitializeOverlayTable()` will then not need to be called by the program to dynamically generate the ELF overlay table.
`_binary_<program_type>_<filename>_taskbininfo`	References an instance of `CellSpursTaskBinInfo`, which is automatically generated when embedding a SPURS Task, with the values correctly initialized for the instance of the ELF file. See "CellSpursTaskBinInfo". For more information on libspurs see the "libspurs Task Reference manual".
`_binary_<program_type>_<filename>_jobheader`	References an instance of `CellSpursJobHeader` which has the `binaryInfo` and `JobType` members configured for the Embedded ELF.

<program_type> is determined by the SPU ELF type, and can be one of the following:

Embed type	Program type
Unknown Executable	spu
Policy module	pm
Spurs Task	task
SPURS Job 2.0	job
SPURS Jobqueue	jqjob
Shared library	so

<filename> is a modified version of the SPU ELF file name specified on the command line, in which characters that are invalid in C identifiers have been replaced by underscores. As with the external tool spu_elf-to-ppu_obj, the extension of the file will change to bin for binary format and jobbin2 for SPURS jobs.

Examples

The following code snippets demonstrate using the embedded SPU ELF data from the main PPU program. Full examples of accessing embedded SPU ELF data can be found in the PS3 SDK Samples $CELL_SDK/samples/sdk/spu_library/…

```
// example of using standard SPU embedded executable
extern const void * _binary_spu_spu_elf_start;
sys_spu_image_import (&spu_img, _binary_spu_spu_elf_start, SYS_SPU_IMAGE_DIRECT);
```

```
// example of using linker generated CellSpursTaskBinInfo
extern const CellSpursTaskBinInfo _binary_task_task_hello_spu_elf_taskbininfo;
CellSpursTaskId tid;
CellSpursTaskArgument arg;
int ret = taskset->createTask2 (&tid,
   &_binary_task_task_hello_spu_elf_taskbininfo,
   &arg,
   NULL,
   "hello task");
```

```
// example of using linker generated CellSpursJobHeader
extern "C" const CellSpursJobHeader _binary_job_job_hello_jobbin2_jobheader;
static void jobHelloInit (JobType *job)
{
    memset (job, 0, sizeof (JobType));
    job->header = _binary_job_job_hello_jobbin2_jobheader;
}
```

```
// example of using a shared library
extern char const _binary_task_task_main_spu_elf_start [];
extern char _binary_spuso_libsample_so_spu_so_start [];
CellSpurs *spurs = init_spurs ();
CellSpursTaskArgument arg;
arg.u32 [0] =  (uintptr_t) _binary_spuso_libsample_so_spu_so_start;
int ret = launch_task (spurs, _binary_task_task_main_spu_elf_start, &arg);
```

Embedding SPU ELF files using the binary format

By default, executable SPU ELF files are embedded as ELF format but in some cases it is desirable to embed the data in the raw binary format. For example a custom SPURS policy module may need the work units to be in binary format to execute but then the work units would have no distinguishing features that the linker can use to identify the embedding format that should be used. Therefore, you can change the embedding format by explicitly specifying the `--embed-format=binary` option. All executable SPU ELF files specified to the right of this option will be embedded in binary format.

In the following example, `unit1.elf` and `unit3.elf` will be embedded in ELF format, while `unit2.elf` will be embedded in the binary format:

```
ps3ppuld main.o unit1.elf --embed-format=binary unit2.elf --embed-format=default
unit3.elf -o out.self
```

Displaying SPU ELF embedded symbols

The generated symbols that point to the embedded data can be obtained by specifying the `--print-embedded-symbols` option when linking. This option also displays the identified SPU ELF type, the method of embedding, and any additional metadata generated. The following example shows the embedding information for a SPURS Task ELF:

```
C:\spurs_task_example>ps3ppuld --print-embedded-symbols task_hello_spu.elf main.o
-lspurs_jq_stub -lspurs_stub -lsysmodule_stub -o main.elf
Embedding SPU ELF File, "C:\spurs_task_example\task_hello_spu.elf" as ELF format
SPU ELF identified as SPURS Task
Generated symbols:
    _binary_task_task_hello_spu_elf_start
    _binary_task_task_hello_spu_elf_end
    _binary_task_task_hello_spu_elf_size
TaskBinInfo created as:
    _binary_task_task_hello_spu_elf_taskbininfo
```

11: 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
`--prx-fixup`	Perform the PRX fix-up step (default). Tells the linker to perform a PRX fixup and is the default unless `--gnu-mode` is specified.
`--no-prx-fixup`	Do not perform the PRX fix-up step. This disables the automatic 'PRX stub fix' phase. This is the default if `--gnu-mode` is specified.
`--external-prx-fixup`	Use the external 'ppu-lv2-prx-fixup --stub-fix-only' 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
`--compress-output`	Compresses FSELF output. Must be used with `--oformat=fself` or `--oformat=fself_npdrm`.
`--oformat=elf`	Create ELF output (default).
`--oformat=fself`	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.

12: Troubleshooting

Improving link-time performance

The linker includes a number of optional features which can affect link-time performance. If you are experiencing long link times, consider whether you can disable any of the following features:

Dead-stripping and de-duplication

Both dead-stripping and de-duplication can negatively affect link-time performance. Disabling those features when they are not required may improve link times substantially.

For more information, see "Trade-offs".

Map file generation

The linker includes options to generate a text-based map file containing a summary of the binary output. Map file generation can negatively affect link-time performance, so consider only using these options when there is a requirement to do so.

As a guide, the `--map` and `--sn-full-map` options increase link time by 30% on average.

Object Module Loading support

The Object Module Loading (OML) support feature adds metadata to the binary output file, which the OML system uses when updating the binary image in memory on the target. The generation of this data increases link time by 5-10% on average, so if you are not actively using OML, removing `--oml` is worth consideration.

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" 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" 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.

> **Caution:** 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.

13: Index