ProDG Linker for PlayStation®3

ユーザー ガイド

SN Systems Limited
バージョン470.1
2015年2月27日

目次

1: はじめに

SN Linker（リンカ）は、高性能なリンカとして設計されています。本リンカにより、SNC あるいは GCC ツール チェーンで作成されたオブジェクト ファイルやアーカイブのリンクが行えます。

以下が、SN Systems リンカの主要な機能です。

- ProDG Debugger と互換性のある ELF イメージとデバッグ情報を生成。
- テンプレート、グローバルなオブジェクトの構築・破壊、例外などを含む C++ をサポート。
- イメージから未使用および重複する関数を削除。
- イメージから未使用のデータを削除。
- 使用されないデストラクタをイメージから削除。
- メッセージおよび MAP ファイルに含まれるシンボル名をデマングル 。

更新情報

v420.1	*追加：* --oml スイッチ(B#99767) ADDRESS 命令 AFTER 命令の解説 「重複除外を使用する際のコードの安全性」セクション。 *更新：* --sn-no-dtors スイッチ。 --Ur スイッチを削除(B#99738)
v430.1	*追加：* 「重複除外の制限事項」および「より効果的な重複除外のためにコードを書き直すには」 「トレードオフ」を「デッド ストリッピングと重複除外」セクションに追加。 「リンク時のパフォーマンスを向上させる」を「トラブルシューティング」セクションに追加。 *更新：* --no-remove-duplicate-inputs スイッチを削除(B#101254)

パフォーマンス

リンク プロセスはほぼすべてが I/O バウンドです。すなわち、時間の大半が入力 ELF ファイルからのセクション読み込みや、最終 ELF または SELF 出力ファイルへの書き込みに費やされます (詳細は 「リンク後処理のステップ」を参照)。リンカでは、使用可能なプロセス アドレス領域を最大限に活用するように試みられます (Windows XP では最大 3GB に制限され、32 ビット Windows マシンでのプロセス仮想アドレス領域は Boot.ini ファイルで「/3GB」スイッチが使用されない限り 2GB に制限される - 詳細は、http://msdn.microsoft.com/ja-JP/library/ff556232.aspxを参照)。

リンク時間の向上を試みる場合、ホスト マシンの指定時に考慮すべき要素が 2 つあります。

- システムに十分な RAM を確保。リンク自体は I/O バウンドであるため、仮想メモリ スワップ ファイルの使用を絶対最低限にとどめておくことが重要です。
- スタティック ライブラリの過剰使用を避ける。スタティック ライブラリはシステム コンポーネントや、ライブラリの配布時には欠かせないものの、大きなビルド内での「サブプロジェクト」に対する便利なパッケージング メカニズムとして使用されることもあります。ただし、スタティック ライ

ブラリは、適切なオブジェクト ファイルがリンクに組み込まれていることを確認するため、リンカによって繰り返しスキャンされる必要があり、このスキャンのせいで実行時間は長くなります。

詳細は、「リンク時のパフォーマンスを向上させる」を参照してください。

必要メモリ

リンカ スワッピングを防ぐため、ご使用のシステムには最低 2 GB の RAM を確保してください。また、無理のないリンクのため経験則として、メモリはリンク中のオブジェクト、およびライブラリファイルのトータルサイズの最低半分に該当する量が必要といえます。PS3 プロジェクトのリンクには、基本メモリ サイズとして 2 GB が推奨されます。

2: リンカのコマンドライン構文

リンカのコマンドライン構文は以下のとおりです：

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

<switches>	以下のセクションで解説するすべてのコマンドライン スイッチ。これらのスイッチの前には、ハイフン(「-」)を付ける必要がある。下記の警告を参照。
<files>	オブジェクトファイルの場合と、ライブラリの場合がある。リンカは、リンカ スクリプトで指定されたファイルに加えて、これらのファイルを使用する。 ワイルドカード文字「?」と「*」はファイルの指定に使用でき、疑問符文字は 1 文字と、アスタリスクは複数の文字に相当する。現在のフォルダ内ですべてのオブジェクト ファイルをリンクする場合、*.o のように指定する。
<response filename>	リンカへのスイッチおよびファイル名を含むファイルへのパス。レスポンス ファイルの内容は、コマンドラインから提供されたように扱われる。しかし、レスポンス ファイル内では、オプションやファイル名をスペースだけでなく改行でも区切ることが可能。

警告：1 つの文字から成るスイッチには、1 つのハイフンを前に付ける必要があります。複数の文字から成るスイッチには、1 つまたは 2 つのハイフンを連続して前に付けます (曖昧性を避けるため、2 つのハイフンの使用を推奨)。

多くのスイッチには引数が必要です。

- 1 つの文字から成るスイッチの場合、引数はスイッチに直接付加 (例：-l*<library>*)、または間にスペースを入れた後付加します (例：-e *<entry>*)。
- 複数の文字から成るスイッチの場合、引数は等号で区切る (例：--entry=*<entry>*)、または間にスペースを入れる (例：--Map *<mapfile>*) 必要があります。

コマンドラインに、コメント (「/*」で始め「*/」で終わる) を挿入することもできます。コメント区切り文字間のスイッチと引数はすべて無視されます。

スイッチ処理の順番

スイッチは、以下のように段階ごとに処理されます。

(1) 後続段階 (2 から 5) で処理されないスイッチが、順番に処理されます。

(2) 明示的ライブラリ パスが追加されます。

(3) リンカ スクリプトが開かれ、解釈されます。

(4) リンカ スクリプトからの値を上書きするオプションが処理されます。

例：--entry=*<symbol>*

(5) 入力ファイルが開かれます。

リンカがオブジェクトまたはアーカイブ ファイルとして認識できないリンカ入力ファイルを指定した場合、そのファイルはリンカ スクリプトとして読み込まれます。このファイルがリンカ スクリプトとして解析不可能な場合は、リンカによってエラーが報告されます。

入力パッケージャ

SN Systems ではリンカのサポートや、発生し得る問題の解決を支援するために、入力パッケージング ツールが使用されます。このツールでは、`--package` スイッチをリンカに渡すことにより、使われたコマンドライン オプションと共にすべての入力が、1 つの Zip ファイルにパッケージされます。`--package-file-name` スイッチを使うと、パッケージのファイル名を指定できます。この Zip ファイルを使用すると、SN Systems では過去とまったく同じ方法でリンカの再実行が行えるため、バグを正確に再現することができます。

以下は、生成されるパッケージ内に含まれる各種ファイルのリストです。

オブジェクト ファイル

リンカへの入力であるすべてのオブジェクト ファイルが、パッケージ ファイルに収められます。これには、プログラムからのオブジェクト ファイル、使用される任意のライブラリ、および PS3 SDK から取られる必須ファイルが含まれます。なお、入力パッケージャでは、コードとデバッギング情報がそのまま残されます。

リンク レスポンス ファイル

このファイルには、実行時に行われるリンクの複製に必要とされるコマンドライン情報が含まれます。これは、実際に使用したコマンドラインとまったく同じというわけではありませんが、挙動は複製されます。リンク レスポンス ファイルを使ってリンカを起動するには、「`@<`*リンク レスポンス ファイル名*`>`」という引数を渡してリンカ実行ファイルを呼び出します。

ファイル名マッピング

パッケージ内のオブジェクト ファイル名は、可能な限り元のファイル名に類似したものとなります。ただし、名前が重複する場合は、固有の数字が該当ファイル名の 1 つに付加されます。このため、元のパス情報はこの新しいファイル名のせいで無効となります。これを解決するため、全ファイルの情報を確認できるように、元の名前とパス間のマッピング、およびパッケージ中に現れる新しい名前が出力されます。

バージョン ファイル

このファイルには、使用されるリンカのバージョン情報が含まれます。

スクリプト ファイル リスト

このファイルには、リンカで使用されるスクリプト ファイルのリストが含まれます。 これは、パッケージ ファイルを入力としてリンカを実行する際に、内部で使用されます。

リンカ スイッチ

多くのスイッチはコンソール プラットフォームには適していないため、GNUリンカのldがサポートしているスイッチの中には、リンカがサポートしていないものもあります。

コマンドライン スイッチは以下のいずれかになります。

スイッチ	詳細
`-( | --start-group`	グループを開始する。
`-) | --end-group`	グループを終了する。
`--32bit`	64 ビットのオペレーティング システム使用中に 32 ビットのリンカの使用を強制する。
`--callprof`	プロファイリング コードを最終出力に追加する (SN Tuner と共に使用。詳細は『*Tuner ユーザーガイド*』を参照)。
`--comment`	コメント セクションを保存する。

--comment-report=<*file*>	リンクの全ファイルに対するコメントを含むレポートを <*file*> に書き込む。
--compress-output	FSELF 出力を圧縮する。--oformat=fself または --oformat=fself_npdrm と共に使用しなければならない。
-d | --dc | --dp	リンカに、部分リンクであってもcommonデータを割り当てるよう指令。
--deep-search	すべてのライブラリとオブジェクト ファイルを、--start-group と --end-group 内に含める。
--default-paths	リンカ スクリプト LIB_SEARCH_PATHS エレメントからデフォルト パスを追加する。
--defsym=<*symbol*>=<*value*>	シンボル <*symbol*> を値 <*value*>で定義する。
--disable-warning=<*value*>	警告メッセージを無効にする。引数には、先行する「L」を除いたエラー番号を指定。たとえば、「--disable-warning=95」では警告「L0095」が無効化される。 エラー番号のリストをコンマで区切ることにより、複数のメッセージを 1 つのスイッチで無効化できる。たとえば、「--disable-warning=207,24」では警告「L0207」と「L0024」の両方が無効化される。
--discard-all	すべてのローカル シンボルを破棄する。
--discard-locals	一時的なローカル シンボルを破棄する。
--dont-strip-section=<*section*>	指定された <*section*> のデッドストリッピングを行わない。KEEP リンカ スクリプト命令を使用するのと同等。複数セクションのデッドストリッピングを行わないようにするには、複数の --dont-strip-section スイッチを使用する。 例： --dont-strip-section=.dont_strip1 --dont-strip-section=.dont_strip2 上記のスイッチを使うと、「.dont_strip1」および「.dont_strip2」と名付けられたセクションにあるデータはストリップされない。
-dfl | --dump-file-layout	ファイル オフセットに基づき、ELF ファイル中のセクションとセグメントのビューを出力する。
--eh-frame-hdr	GNU コンパイラ ドライバとの互換性のため、暗黙のうちに無視される。
--enable-warning=<*value*>	無効化された警告メッセージを有効にする。引数には、先行する「L」を除いたエラー番号を指定。たとえば、「--enable-warning=95」では警告「L0095」が有効化される。 エラー番号のリストをコンマで区切ることにより、複数のメッセージを 1 つのスイッチで有効化でき

	る。たとえば、「--enable-warning=207,24」では警告「L0207」と「L0024」の両方が有効化される。
-e<*symbol*> | --entry=<*symbol*>	開始アドレスを設定する。
--exceptions	例外処理ありのライブラリ用のリンカ デフォルト パスを追加する (LIB_SEARCH_PATHS "exceptions" キーから)。「例外と RTTI」を参照。
--external-prx-fixup	内部メカニズムではなく、SDK ppu-lv2-prx-fixup ツールを使用する。(デフォルトでは、パフォーマンス向上のため、PRX 修正を内部で実行する。)「PRX の自動修正」を参照。
--gc-sections	GNU ld のデッドストリッピング スイッチ。2.7.2782.0 より前のバージョンの SN Linker では、このスイッチには対応していないと単に警告が発せられる。より後のバージョンでは自動的に --strip-unused-data が選択され、警告が発せられる。 warning: LO153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping
--gnu-mode	GNU 互換機能を有効にする。このオプションは、GCC コンパイラ ドライバがリンカを呼び出す際に使用されることを意図したもの。
--help	オプション ヘルプを表示する。
--just-symbols=<*file*> | -R<*file*>	<*file*> からのシンボルのみをリンクする。
--keep=<*file*>	<*file*> にリストされているすべてのシンボルを保存。これによって、ライブラリで定義されているシンボルであっても、これらのシンボルを含むようにリンカに指示する。デッドコードストリッピングが有効になっている場合、<*file*> 内にリストされているシンボルはストリップされない。 **メモ：**重複除外が有効になっているときは、このスイッチを指定しても、そのシンボルや対応するコード/データは重複除外の対象になります。
--keep-eh-data	最終出力ファイルから例外処理データを削除しない。(--gnu-mode が指定されており、--exceptions または --no-exceptions のいずれも指定されていない場合、これがデフォルトの挙動となる。)
--keeptemp	リンク中に作成された任意の一時ファイルを削除しないよう、リンカに指示する。
-L<*path*> | --library-path=<*path*>	<*path*> をライブラリ検索パスに追加。
-l<*name*> | --library=<*name*>	ライブラリ ファイル「lib<name>.a」をリンクに含める。たとえば -lc と入力すると、リンカは libc.a を検索する。

`--linkonce-size-error`	別モジュールからの linkonce セクションのサイズが異なる場合、エラーを発する。`--linkonce-size-warning` に優先する。
`--linkonce-size-warning`	別モジュールからの linkonce セクションのサイズが異なる場合、警告する。
`--Map=<mapfile>`	*<mapfile>* 内にマップ ファイルを生成する。 **ヒント**：マップ ファイルの作成を無効化すると、リンク時間が短縮されます。
`--md=<type>`	「ミニダンプ」クラッシュ診断を実行する。*<type>* は診断のタイプ。 **メモ**：このスイッチは Windows でのみサポートされます。
`-mprx`	`--oformat=prx` と同等。GCC との互換性を目的とする。
`-mprx-with-runtime`	`--oformat=prx --prx-with-runtime` と同等。GCC との互換性を目的とする。
`--multi-toc`	`--multi-toc` は、64 KB より多い TOC データの使用を可能にするスイッチ (これがデフォルトの挙動)。 別の TOC 領域を使用するモジュール間でのコール時、リンカは TOC 領域調整を行うリンケージコードを自動的に生成する。「TOC を切り替える場合」を参照。
`--no-default-paths`	リンカ スクリプト LIB_SEARCH_PATHS エレメントからのデフォルト パスを追加しない。
`--no-default-script`	コマンドラインでリンカ スクリプトを指定しなかった場合に、デフォルトのリンカ スクリプトを検索しない。
`--no-demangle`	エラー メッセージやその他出力に含まれるシンボル名をデマングルしない。
`--no-exceptions`	例外処理なしのライブラリ用のリンカ デフォルト パスを追加 (LIB_SEARCH_PATHS "no_exceptions" キーから) (デフォルト)。最終出力ファイルから例外処理データの削除を行う。「例外と RTTI」を参照。
`--no-keep-eh-data`	`--exceptions` が指定されている場合を除き、最終出力ファイルから例外処理データを削除する。(`--gnu-mode` が指定されている場合を除き、`--exceptions` または `--no-exceptions` のいずれも指定されていない場合、これがデフォルトの挙動となる。)
`--no-multi-toc`	`--no-multi-toc` は、複数の TOC 領域をサポートするために使用されるリンケージコードをリンカが作成しないようにするもので、プログラムに 64KB を越える TOC データが含まれる場合、エラーを発する。

	このスイッチは GNU ld にも存在。 `error:L0154: there is too much TOC data (>64kB) for a single TOC region (consider removing both --no-multi-toc and --no-toc-restore)`
`--no-ppuguid`	PPU GUID の生成を無効にする (デフォルト)。
`--no-prx-fixup`	PRX 修正ステップを実行しない。「PRX の自動修正」を参照。
`--no-required-files`	リンカ スクリプト REQUIRED_FILES エレメントから必要なファイルを追加しない (`--required-files` も参照)。
`--no-sn-dwarf-string-pool`	オブジェクト ファイル内の不必要な文字列を削除する、重複文字列チェックを無効にする。
`--no-standard-libraries`	リンカ スクリプト STANDARD_LIBRARIES エレメントによって指定される標準ライブラリを追加しない。
`--notocrestore` | `--no-toc-restore`	SNC PPU C/C++ コンパイラの `-Xnotocrestore=2` スイッチと合わせて使用する。 `--notocrestore` スイッチにより、リンカでは PRX スタブ ライブラリ (OS などの PRX 関数へのコールを行うために使用される) の再記述が行われる。この機能は、複数 TOC 領域が存在する場合には使用できないため、`--notocrestore` 使用時は `--no-multi-toc` が有効になっていると仮定される。 `--no-toc-restore` は `--notocrestore` の別名。
`--no-whole-archive`	`--whole-archive` の効果を無効にする。
`--noinhibit-exec`	エラーが発生した場合でも出力ファイルを作成する。
`-o<file>` | `--output=<file>`	出力ファイルとして *<file>* を使用。
`--oformat=<format>`	出力ファイルのフォーマットを指定する。リストされた以外のフォーマットは、「unrecognised output format (認識されない出力フォーマット)」エラーの原因となる。 `'elf', 'fself', 'fself_npdrm', 'prx' or 'fsprx'`
`--oml`	Object Module Loading (OML) サポートに必要な付加的なデータを生成する。詳細については、デバッガのユーザー ガイドにある、「オブジェクト モジュールのロード」を参照。 **注意:**このスイッチを使用すると、リンク時のパフォーマンスがわずかに低下します。
`--package`	すべての入力とコマンドライン オプションを 1 つの

	Zip ファイルに収める。「入力パッケージャ」を参照。
`--package-file-name=<file>`	*<file>* をパッケージ作成時に使われる名前として使用する。
`--pad-debug-line=<value>`	各 `.debug_line` データを *<value>* バイトだけパッドし、再エンコーディング中の拡張を可能にする。デッドストリップが有効な場合 *<value>* はデフォルトで 1 になり、それ以外の場合はデフォルトで 0 になる。
`--ppuguid`	出力ファイル内のプログラムのハッシュを保存する。これはデバッガによる PRX モジュールに対するデバッギング情報の場所の特定を支援するもの。これはまた、自動コアダンプ サポートでも必要となる。 **メモ：**この機能を有効にするとリンク時間が長くなります。
`--print-embedded-symbols`	最終リンク内に埋め込まれたすべてのバイナリ データについて生成されたシンボルをプリントする。「SPU 埋め込みコマンド ライン オプション」を参照。
`--prx-fixup`	PRX 修正ステップを実行する (デフォルト)。「PRX の自動修正」を参照。
`--prx-with-runtime`	コンパイラのランタイム ライブラリと PRX 出力をリンクする。(`--oformat` が「`prx`」または「`fsprx`」の場合のみ有効。)
`-Qy`	GNU コンパイラ ドライバとの互換性のため、暗黙のうちに無視される。
`-r | --relocatable | --relocateable`	部分リンク実行をリンカに指令する。
`--remap-source-paths=<file>`	*<file>* 内のファイル マッピングのリストを出力 ELF ファイルに追加し、デバッガがそれを使用して、パスをソース コードへ再マッピングできるようにする。「ソース パスの再マッピング」を参照。
`--required-files`	リンカ スクリプト REQUIRED_FILES エレメントから必要なファイルを追加 (`--no-required-files` を参照)。
`--retain-symbols-file=<file>`	*<file>* にリストされているシンボルのみを保存する。
`-S | --strip-debug`	すべてのデバッグ情報を出力からストリップする。
`--S-lib`	ライブラリからのデバッグ情報を出力からストリップする。
`-s | --strip-all`	すべてのシンボルとデバッグ情報を出力からストリ

	ップする。
--s-lib	ライブラリからのシンボルとデバッグ情報を出力からストリップする。
--script=<*file*> | -T<*file*>	<*file*> をリンカ スクリプトとして使用する。
--show-messages	すべての可能なエラーと警告メッセージの一覧を表示する。
--sn-best	あるセクションに対して、リンカ スクリプト内に可能な場所が複数ある場合、そのセクションは最適マッチの場所に配置される。(下記の) --sn-first と比較すること。「リンカ スクリプト内でのセクションの場所の不確定性を解決する」を参照。
--sn-first	リンカは、リンカ スクリプト内で最初にマッチした場所にセクションを配置する。(上記の) --sn-best と比較すること。「リンカ スクリプト内でのセクションの場所の不確定性を解決する」を参照。
--sn-full-map	マップファイル中に追加の情報 (たとえばスタティック変数) を入れる。 **ヒント**：マップ ファイルの作成を無効化すると、リンク時間が短縮されます。
--sn-no-dtors	グローバル オブジェクトに対してデストラクタを呼び出さない。 --strip-unused または --strip-unused-data と合わせて使用された場合は、グローバル オブジェクトから未使用のデストラクタ コードは出力ファイルから削除される。 **メモ**：GCC ドライバ経由でリンカを呼び出したとき、このスイッチを指定した場合、このスイッチは無視され、警告が発行されます。
--sort-common	共通シンボルをサイズで並べ替える。
--spu-format=<*format*>	SPU ELF ファイルの埋め込み形式を指定する。利用できるオプションは、「binary」と「default」。「SPU 埋め込みコマンド ライン オプション」を参照。
--strip-duplicates	重複除外プロセスが有効化される。このスイッチはデッドストリッピング オプションのいずれか (--strip-unused または--strip-unused-data) と併用する必要がある。 「デッド ストリッピングと重複除外」を参照。
--strip-report=<*file*>	プログラム内のコードや参照を示す <*file*> を作成する。このレポートは、デッドストリップされている関数やデータを発見するために使用する。 例：--strip-report=stripreport.txt 詳細は、「ストリップ レポート」を参照。

`--strip-unused`	コードのデッドストリッピングを有効化する。リンカにより、プログラムの完全なコール ツリーが構築されるため、オブジェクト ファイルとアーカイブがスキャンされる。不必要と判断された関数はすべて最終 ELF ファイルから削除される。 「デッド ストリッピングと重複除外」を参照。
`--strip-unused-data`	暗黙的に `--strip-unused` を有効にする。デッド コードのスキャンに加え、リンカでは、使用していないデータ オブジェクトを検出し、ELF ファイルからこれらを削除する。 「デッド ストリッピングと重複除外」を参照。
`--sysroot`	「sysroot」ディレクトリのプレフィックスを設定する。これは、ライブラリ検索パス (`--library-path/-L` オプションで設定) が「=」で始まる場合に使用される。「=」は sysroot ディレクトリ プレフィックスで置換される。
`--Tbss=<addr>`	*<addr>* を bss セクションのアドレスとして設定する。
`--Tdata=<addr>`	*<addr>* を data セクションのアドレスとして設定する。
`--Ttext=<addr>`	*<addr>* を text セクションのアドレスとして設定する。
`-t | --trace`	ファイルが開かれると同時に名前を表示する。
`--temp-dir=<path>`	一時ファイルに使用される一時ディレクトリを指定する。
`--toc-report=<file>`	*<file>* 内に TOC データの詳細を記載したダンプを生成する。「TOC 使用レポート」を参照。
`-u<symbol> | --undefined=<symbol>`	*<symbol>* を未定義として入力をリンクする。このスイッチはリンカスクリプト命令 EXTERN と同じ効果がある。デッドコード ストリッピングが有効になっているときは、*<symbol>* はストリップされない。 **メモ：**重複除外が有効になっているときは、このスイッチを指定しても、そのシンボルや対応するコード/データは重複除外の対象になります。
`--use-libcs`	リンク時に libc.a を libcs.a に置換する。
`-V | -v | --version`	バージョン情報を表示する。
`--verbose`	リンク中に多量の情報を出力する。
`--Wall`	すべての警告を表示する。
`--warn-built-before`	リンカによって読み込まれるファイルの最終変更日付が、指定された日付や時刻よりも前の場合に警告する。日付は、「yyyy/mm/dd:hh:mm:ss」のように

	指定される。日付の後ろ部分が省略されている場合、ゼロと見なされる (例：「2007/06/21」は時刻「00:00:00」となる)。
--warn-common	共通オブジェクトで問題が発生した場合は警告する。
--warn-if-debug-found	デバッグ情報を含むことで知られるセクションが検出された場合、警告する。以下のセクション名によって警告が発せられる。 .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	未定義シンボルについて 1 度だけ警告をする。
--Werror	警告をエラーとして処理する。
--whole-archive	後続するアーカイブからのオブジェクトすべてをリンクに含める。--whole-archive オプション後のコマンドライン上で定義されたそれぞれのアーカイブに対し、オブジェクトファイルに対して参照が行われていなかった場合でも、リンク対象のアーカイブ内の全オブジェクトファイルをリンクに含める。 この設定を無効にするには --no-whole-archive スイッチを使用する。 このスイッチの後に定義されたライブラリは通常どおりリンクされる。 例： `ps3ppuld main.o lib1.a --whole-archive lib2.a --no-whole-archive lib3.a` この例では、lib2.a のすべてのモジュールが、たとえ参照されていない場合でもリンクに含められる。lib1.a と lib3.a は通常の方法でリンクされる。 **ヒント:**このスイッチを使用しても、デッドストリッピングと重複除外は利用できます。
--wrap=*<symbol>*	*<symbol>* に対してラッパー関数を使用する。 シンボル foo がラップされると、リンカは foo への

	参照を _wrap_foo にリダイレクトし、__real_foo からの foo へのアクセスを可能にする。
	たとえば、fopen() へのコールをインターセプトするには、任意の追加ロジックを実行して _real_fopen() をコールする関数 __wrap_fopen() を定義し、--wrap=fopen をリンカに渡す。
	C/C++ コード内では、ラッパー関数には C リンケージが必要で、ラップされるシンボルはその関数の C++ のマングルされた名前となる。
	たとえば、int example (char const *) を置き換えるときは、以下のコードをリンクする際にリンカに --wrap=_Z7examplePKc を渡す。
	`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);`
	`}`
	ヒント：メモリ割り当て関数の置き換えを行う場合、またはそれに計測用のコードを挿入する場合、ランタイム ライブラリではより簡単なメカニズムが利用できます。--wrap を使用する代わりに、こちらを使用してください。
	詳細については、『C/C++ 標準ライブラリ 概要・リファレンス』の「付録 F: PPU 用メモリ管理関数の置き換え」を参照してください。

--write-args-to-output	出力ファイルの .command_line セクションにコマンドライン引数を書き込む。
--write-fself-digest	SELF ヘッダーに SHA-1 要約を作成する。このスイッチを使用するとリンク時間が長くなる。(デフォルトでは無効で、--oformat=fself または --oformat=fself_npdrm が使用される場合にのみ有効となる。)
-X	一時的なローカル シンボルを破棄する。
-x	すべてのローカル シンボルを破棄する。
--zgc-sections	デッドストリップ スイッチの PRX での修正。リンカにより --strip-unused-data が選択され、以下の警告が発せられる。 `warning: LO153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping`
--zgenentry	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenprx	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。

--zgenstub	リンカに最初のリンクを実行させ、「--stub-archive」を libgen ツールに渡す。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zlevel=<openlevel>	ppu-lv2-prx-libgen ツールに直接渡される。

リンカ出力におけるコマンドライン順の影響

SN リンカと GNU リンカでは、同様のコマンドライン構文が使用されますが、コマンドライン引数の順序とリンカ出力に関しては違いが見られます。

もっとも重要な違いは、アーカイブの処理にあります。SN Linker では、コマンドラインにおけるアーカイブの配置によってシンボル解決が影響を受けることはありません。特定のインプットによって解決可能なシンボルはすべてリンク中に検出されます。結果として、SN Linker では --start-group と --end-group アーカイブ分類演算子に対応しているものの、これらを使用する必要がありません。アーカイブ分類演算子は、最終出力のレイアウトに影響を与え、分類されたアーカイブからの情報を互いに近くに配置する役割を果たします。一方、GNU リンカはデフォルトではコマンドラインで指定されたアーカイブを左から右への1回のパスで処理するため、未定義シンボルを参照するオブジェクトやアーカイブは、該当シンボルを定義するアーカイブの左側に配置する必要があります。

関連する差異として、コマンドラインにオブジェクトとアーカイブを交互に指定しても、SN リンカの出力レイアウトに影響を与えないことが挙げられます。そうではなく、すべてのオブジェクト ファイルが一緒に処理され、その後アーカイブが、未解決シンボルに対応するため必要に応じて処理されます。したがって、GNU リンカを使用した場合には可能であるとしても、オブジェクトとアーカイブの部分がリンカ出力内で混ざることに依存してはいけません。上級ユーザーは、必要に応じてカスタム リンカ スクリプトを使用することにより、求めるレイアウトを強制的に実現することも可能です。

ソース パスの再マッピング

--remap-source-paths=<file> スイッチは、ある場所に存在するコードを使用して ELF をビルドしたいが、それをデバッグするときに他の場所にあるコードを使用したい場合に使用します。このスイッチを指定せずにこれを行うと、ELF に保存されているソース コードへのパスが不正となり、デバッガはソースを見つけることができなくなります。

このファイルには元のソース パスから新しいソース パスへのマッピングのリストが保存されます。

例：

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

この場合、c:\my_source\code_1 で始まるパスを持つ任意のソースファイルは \\my_server\code_area_1 で検索され、c:\my_source\code_2 にある任意のソース ファイルは \\my_server\code_area_2 で検索されます。それ以外のソースファイルは、元のパスのままです。

3: リンカ スクリプト

デフォルトのリンカ スクリプト

コマンドライン オプション処理が --script オプションに出会うことなく完了すると、`--no-default-script` が使用されない限り、リンカはリンカ スクリプトのデフォルトの場所を合成し、ここで検出されるファイルの処理を試行します。使用されるデフォルトのパスは以下のとおりです。

`--relocatable` が指定された場合：「$CELL_SDK/target/ppu/lib/prx32.sn」

`--relocatable` が指定されない場合：「$CELL_SDK/target/ppu/lib/elf64_lv2_prx.sn」

リンカ スクリプト命令

リンカは、ほとんどの「ld format」リンカ スクリプト命令に対応しています。リンカ スクリプトのフォーマットに関する詳細情報は、GNU の Web サイト http://www.gnu.org/ で、GNU のリンカ ld のドキュメントを参照してください。

GNU のリンカ ld で使用できて、このリンカで使用できないスクリプト ファイル命令の詳細は、「対応しないスクリプト ファイル命令」を参照してください。

キーワード	詳細
`ABSOLUTE(<exp>)`	式 *<exp>* の絶対値 (負でないという意味ではなく、再配置不可能という意味) を戻します。主に、シンボル値が通常セクションに関連する場合に、セクション定義内のシンボルに絶対値を割り当てるのに役立ちます。
`ADDRESS`	オーバーレイ定義と共に使用し、指定されたアドレスへのオーバーレイを強制します。
`ADDR(<section>)`	指名されたセクションの絶対アドレス (VMA) を戻します。スクリプトでは、事前にこのセクションの場所を定義しておく必要があります。
`AFTER`	オーバーレイ定義と共に使用し、オーバーレイ出力を指定します。
`ALIGN`	
`ASSERT(<exp>, <message>)`	*<exp>* がゼロ以外であることを保証します。ゼロの場合、エラーコードと共にリンカが終了され、*<message>* が表示されます。
`AT`	
`BLOCK`	
`BYTE`	
`COPY`	
`CREATE_OBJECT_SYMBOLS`	各入力ファイルに対するシンボルを作成するよう、リンカに指示するコマンドです。各シンボルの名前は、対応する入力ファイルの名前になります。各シ

	ンボルのセクションは、CREATE_OBJECT_SYMBOLS コマンドに含まれる出力セクションになります。これは、a.out オブジェクト ファイル フォーマットの標準です。通常これは、他のオブジェクト ファイル フォーマットに使用されません。
DEFINED	
DSECT	
END	
ENTRY	
EXCLUDE_FILE	
EXTERN(*<symbol> <symbol> ...*)	未定義シンボルとして *<symbol>* を強制的に出力ファイルに含めます。これを行うことにより、標準ライブラリから追加モジュールのリンクを引き起こす場合などがあります。各 EXTERN に対して複数のシンボルをリストでき、EXTERN を複数回使用することもできます。このコマンドは、-u コマンドライン オプションと同じ働きをします。
FILEHDR	
FILL	
FLAGS	
FORCE_COMMON_ALLOCATION	このコマンドは、-d コマンドライン オプションと同じ働きをします。すなわち、再配置可能な出力ファイルが指定された場合でさえも (-r)、共通シンボルへのスペース割当てをリンカで実行します。
GLOBAL	
GROUP (*<file> <file> ...*)	GROUP コマンドは、指定されるファイルがすべてアーカイブであることを除き、INPUT と同様で、新しい未定義参照が作成されなくなるまで、繰り返し検索を行います。「--start-group」コマンドライン オプションの詳細を参照してください。 **メモ：** また、SN Linker では、コンマがファイル名の一部と見なされるため、コンマ区切りのファイル名には対応していません。
INCLUDE *<filename>*	この時点で、リンカ スクリプト *<filename>* がインクルードされます。ファイルは現在のディレクトリと、-L コマンドライン オプションで指定された任意のディレクトリで検索されます。
INFO	
INPUT (*<file> <file> ...*)	コマンド ラインで指定された場合と同様に、INPUT コマンドでは、指定されたファイルをリンクに含むよう、リンカに指示します。たとえば、リンクを実行するたびに subr.o をリンクに含めたいものの、毎

	回リンク コマンド ラインで設定するのは面倒な場合、リンカ スクリプトに「INPUT (subr.o)」を入れることができます。実際、すべての入力ファイルをリンカ スクリプトにリストし、そのリンカを -T オプション 1 つで呼び出せます。リンカでは、該当ファイルを現在のディレクトリで開こうとします。見つからない場合は、リンカによってアーカイブ ライブラリ検索パスを通じた検索が行われます。-L コマンドライン オプションの詳細を参照してください。INPUT (-lfile) を使用すると、リンカでは --library コマンドライン スイッチと同様に、その名前を libfile.a に変換します。INPUT コマンドを間接的なリンカ スクリプトで使用すると、リンカ スクリプト ファイルがインクルードされる時点において、該当ファイルがリンクに含められます。これは、アーカイブ検索に影響することがあります。 **メモ：** また、SN Linker では、コンマがファイル名の一部と見なされるため、コンマ区切りのファイル名には対応していません。
KEEP	
l	
len	
LENGTH	
LIB_SEARCH_PATHS	「LIB_SEARCH_PATHS」を参照してください。
LOADADDR	
LOCAL	
LONG	
MAP	
MAX(*<exp1>*, *<exp2>*)	*<exp1>* と *<exp2>* の最大を戻します。
MEMORY	
MIN(*<exp1>*, *<exp2>*)	*<exp1>* と *<exp2>* の最小を戻します。
NEXT	
NOCROSSREFS(*<section>* *<section>* ...)	このコマンドは、特定の出力セクション間における任意の参照に関するエラーを、リンカで発行するために使用されます。特定タイプのプログラム、特にオーバーレイを使用する埋め込み式システムにおいて、1 つのセクションがメモリにロードされる際、別のセクションはロードされません。2 つのセクション間における直接参照はすべてエラーとなります。たとえば、あるセクションで定義される関数を、別のセクションでコールするとエラーとなります。NOCROSSREFS コマンドでは、出力セクション名

	のリストが使用されます。セクション間の相互参照がリンカによって検出された場合、エラーが報告され、ゼロ以外の終了ステータスが返されます。NOCROSSREFS コマンドでは、入力セクション名ではなく、出力セクション名が使用されます。
NOLOAD	
NONE	
o	
org	
ORIGIN	
OUTPUT(*<filename>*)	OUTPUT コマンドでは、出力ファイルを指定します。リンカ スクリプトでの OUTPUT(*<filename>*) 使用は、コマンドラインで「-o*<filename>*」を使用するのとまったく同じです。両方が使用される場合、コマンド ライン オプションが優先します。OUTPUT コマンドを使用すると、出力ファイルの通常のデフォルト名「a.out」以外のデフォルト名を定義できます。
OVERLAY	
PHDRS	
PROVIDE	
PT_DYNAMIC	
PT_INTERP	
PT_LOAD	
PT_NOTE	
PT_NULL	
PT_PHDR	
PT_SHLIB	
PT_TLS	
QUAD	
REQUIRED_FILES	「REQUIRED_FILES」を参照してください。
SEARCH_DIR(*<path>*)	SEARCH_DIR コマンドでは、リンカによるアーカイブ ライブラリの検索場所となるパスのリストに、パスを追加できます。SEARCH_DIR(*<path>*) は、コマンドラインで「-L*<path>*」を使用した場合とまったく同じ働きをします。両方が使用される場合、リンカでは両方のパスが検索されます。その場合、コマンドライン オプションを使用して指定されたパスが、最初に検索されます。「REQUIRED_FILES」も

	参照してください。
SECTIONS	
SHORT	
SINGLE_TOC	
SIZEOF	
SIZEOF_HEADERS	
sizeof_headers	
SORT	
SQUAD	
STARTUP	STARTUP コマンドは、ファイル名がコマンドラインで最初に指定されたかのように、最初にリンクされる入力ファイルになる点を除き、INPUT コマンドとまったく同じように機能します。これは、エントリポイントが常に最初のファイルの始めであるシステムを使用する場合に便利です。
STANDARD_LIBRARIES	「STANDARD_LIBRARIES」を参照してください。
STRING	

対応しないスクリプト ファイル命令

以下のスクリプトファイル命令を使用すると、警告が表示されます。

```
OUTPUT_ARCH
OUTPUT_FORMAT
```

以下のスクリプト ファイル命令は、リンカで実行されず、使用された場合はエラーが発生します。

```
HLL
INHIBIT_COMMON_ALLOCATION
SYSLIB
TARGET
VERSION
```

以下の命令はリンカによって受け入れられますが、警告なく無視されます。

```
CONSTRUCTORS
FLOAT
NOFLOAT
ONLY_IF_RO
ONLY_IF_RW
```

セクション

コンパイラの出力によく出現するセクションの完全なリストは以下のとおりです。

セクション	用途
.text	プログラムコード
.data	初期化された変数
.rodata	文字列などの読み出し専用データ

.bss	初期化されていない変数
.sdata	初期化された変数 (小データ)
.sbss	初期化されていない変数 (小データ)

リンカ スクリプトでファイルを参照する

リンカ スクリプト内のオブジェクトファイル名にワイルドカードがない場合、リンク コマンド ラインに現れなくてもリンクに必要であるとみなされます。従って、もしリンカがオブジェクトを見つけられない場合はリンクが失敗します。たとえば、

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

この場合、foo.o にワイルドカード ('*' または '?') がないのでリンクに付加されます。もしリンカが foo.o を見つけられない場合はリンクが失敗します。

コマンドラインに明示的にリストされている場合だけ foo.o をリンクに加えたい場合は、スクリプトに若干手を加え、ワイルドカードを含むようにします。たとえば、

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

LIB_SEARCH_PATHS

2 つのリンカ スイッチ (--default-paths、--no-default-paths) により、デフォルトのライブラリ検索パスが、リンカ スクリプトの LIB_SEARCH_PATHS エレメントから取られたパスによって補足されたものであるかどうかが制御されます。

> ヒント： --default-paths/--no-default-paths スイッチに関係なく、現在のディレクトリは常にライブラリ検索パスに含まれます。

デフォルトのリンカ スクリプトにおける LIB_SEARCH_PATHS 命令は、以下のようになります。

```
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

REQUIRED_FILES エレメントには、ターゲットに正しくリンクするために通常必要な、ファイル名の配列が含まれます。これは、システム ライブラリで必要とされる GCC 起動関連ファイルをユーザーが忘れた場合に警告を発するため、PS3 リンカで使用されてきました。また、これらのファイルのいずれかが欠けている場合に、ユーザーがクラッシュの診断に時間を費やさないようにデザインされました。この機能により、必要なファイルがコマンドラインに明示的にリストされていない場合、リンカでは自動的にこれらをリンクに含めるようになります。

デフォルトのリンカ スクリプトにおける REQUIRED_FILES 命令は、以下のようになります。

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

STANDARD_LIBRARIES

STANDARD_LIBRARIES エレメントは、GROUP コマンドの機能と同じですが、これは、--no-standard-libraries コマンドライン スイッチによって無効にすることができます。

デフォルトのリンカ スクリプトにおける STANDARD_LIBRARIES 命令は、以下のようになります。

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

4: セクション シンボル

セクションの開始と終了擬似シンボル

リンカでは、ELF セクションの始まりと終わりに対する GNU ld スタイルの擬似シンボルがサポートされており、これは未定義の参照が検出された場合にインスタンス化されます。

__start_XXX または __stop_XXX と名付けられたシンボルを使用する場合、リンカでは、「XXX」と名付けられたセクションのアドレスが合成されます。セクション名は、C 名称として表せなければなりません (すなわち、英数字とアンダーライン)。

C 識別子として表せる名前を持つセクションがリンカで認識されると、それぞれセクションの始まりと終わりを示す「__start_NAME」と「__stop_NAME」という名前のシンボルが推測的に生成されます。これらのシンボルへの未解決参照がリンカ入力内で検出された場合、生成されるシンボルはリンカ出力内で個別に定義され、参照はそれらを指すように解決されます。この機能は、GNU リンカによって提供される拡張機能を模倣したものです。

```
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];
}
```

ドット セクション

さらなる拡張機能として、リンカでは「.」(ドット) で始まるセクションに対し、同様のプロセスを実行します。この場合シンボル名は、先頭の「.」をシーケンス「_z」に置換することによって作成されます。たとえば、「.text」 セクションの始まりは「__start__ztext」で表されます。続いて未解決参照に対して同様のチェックが行われ、シンボルを定義するべきかどうかが決定されます。

> **メモ：**先頭のドット後のセクション名の残りは、有効な C 識別子であることが求められます。このため、複数のドットを含むセクション名は無視されます。

セクション名が「.」で始まるものの、それ以外は有効な C 識別子であるとリンカで認識された場合、該当セクションの始まりと終わりを示すシンボルが推測的に生成されます。これらのシンボルへの未解決参照がリンカ入力内で検出された場合、生成されるシンボルはリンカ出力内で個別に定義され、参照はそれらを指すように解決されます。シンボル名は先頭の「.」をシーケンス「_z」に置換することによって作成され、変更されたセクション名の接頭辞に「__start_」または「__stop_」が付けられます。たとえば、「.text」 セクションの始まりは「__start__ztext」で表されます。

例 1:C 識別子として適切な文字を含む名前を持つセクションの開始アドレスと終了アドレスを検出するために、セクション シンボルを使用。

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

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

/* リンカによって生成されるシンボル*/
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;
}
```

例 2: ドットで始まる名前のセクションにアクセスするために、セクション シンボルを使用。

```
#include <stddef.h>

/* リンカによって生成されるシンボル*/
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

SN リンカでは、入力オブジェクト ファイルに含まれる「.linker_cmd」セクションの処理に対応しています。このセクションは、Microsoft スタイルの `#pragma comment ("lib","xxx")` の使用に対応して SNC 240.1 以降で出力されます。この機能の説明は、Microsoft の Web サイト (http://msdn.microsoft.com/ja-jp/library/7f0aews7(v=vs.100).aspx) に掲載されています。

このプラグマは、リンカのコマンドラインにファイルを自動追加するために使用できます。

5: デッド ストリッピングと重複除外

使用していないコードやデータのストリッピング

GNU リンカのデッドストリッピングは、各セクションの参照カウントを元に処理を行います。再配置エントリがシンボルを参照している場合、そのシンボルを含むセクションは参照済みとして記録 (マーク) されます。処理の終了時点でマークされていないセクションは必要ないと認識され、出力に書き込まれません。このアプローチのマイナス面は、これがセクション全体においてのみ機能する点であり、これは、特殊な GCC の -ffunction-sections スイッチと -fdata-sections スイッチを使用してコードがコンパイルされている必要があることを意味します。これらのスイッチの使用によって生じる ELF セクションの増加は、リンク時間に悪影響を与える可能性があります。

SN Linker のデッドストリッピングはまったく異なります。SN Linker のデッドストリッピングは、リンクするコードやデータの参照を正確に決定するために、再配置エントリ自体をスキャンします。この方法の利点は、リンクするオブジェクトが特殊なコンパイラ スイッチを用いてコンパイルされていない場合でも、セクションの中からコードやデータをストリップできることにあります。

> **ヒント:**デバッグ ビルドでデッドストリッピングを無効化すると、リンク時間が短縮されます。

コマンドライン スイッチ

SN Linker のデッドストリッピングの性質上、その制御は、GNU リンカで使用されるものとは異なる一連のスイッチで行われます。リンクするオブジェクトは、GCC の -ffunction-sections や -fdata-sections スイッチを用いてコンパイルされている必要はありません。

スイッチ	詳細
--dont-strip-section=*<section>*	指定された *<section>* のデッドストリップを行わない。KEEP リンカ スクリプト命令を使用するのと同等。複数セクションのデッドストリッピングを行わないようにするには、複数の --dont-strip-section スイッチを使用する。 --dont-strip-section=.dont_strip1 --dont-strip-section=.dont_strip2 上記のスイッチを使うと、「.dont_strip1」および「.dont_strip2」と名付けられたセクションにあるデータはストリップされない。
--gc-sections	GNU ld のデッドストリッピング スイッチ。2.7.2782.0 より前のバージョンの SN Linker では、このスイッチには対応していないと単に警告が発せられる。より後のバージョンでは自動的に --strip-unused-data が選択され、警告が発せられる。 warning: L0153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping
--strip-duplicates	重複除外プロセスが有効化される。このスイッチはデッドストリッピング オプションのいずれか (--strip-unused または--strip-unused-data) と併用する必要がある。
--strip-report=*<file>*	プログラム内のコードや参照を示す *<file>* を作成する。このレポートは、デッドストリップされている関数やデータを発見するために使用する。 例：--strip-report=stripreport.txt 詳細は、「ストリップ レポート」を参照。

--strip-unused	コードのデッドストリッピングを有効化する。リンカにより、プログラムの完全なコール ツリーが構築されるため、オブジェクト ファイルとアーカイブがスキャンされる。不必要と判断された関数はすべて最終 ELF ファイルから削除される。
--strip-unused-data	暗黙的に --strip-unused を有効にする。デッド コードのスキャンに加え、リンカでは、使用していないデータ オブジェクトを検出し、ELF ファイルからこれらを削除する。

未定義のシンボル

デッドストリップが有効化されている場合には、呼び出し側が最終的に参照されていない場合であっても、参照されているシンボルはすべて定義されていなければなりません。

例：

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

int main ()
{
}
```

bar() がプログラム内の他の場所で定義されておらず、foo() 以外の関数からコールされない場合、つまりこれがデッドストリップされるコードによってのみコールされる場合でも、ストリッピングの有効/無効に関わらず、リンク エラーが発生します。

```
error:LO039: reference to undefined symbol `bar()' in file "main.o"
```

重複除外

重複除外はリンカの機能で、同一コードや読み取り専用データの重複コピーを除去することにより、最終実行ファイル イメージのサイズをさらに削減することを目的としています。プログラムには、大量の重複コードや重複データが含まれていることがあります。コンテンツが読み取り専用である場合、リンカは重複したコンテンツを削除し、削除されたコンテンツへの参照を、残された単一コピーへの参照へ適切に変更することができます。

重複除外を有効にするには、--strip-unused または --strip-unused-data と合わせて、--strip-duplicates スイッチを使用します。

> **ヒント:** デバッグ ビルドで重複除外を無効化すると、リンク時間が短縮されます。

重複除外を使用する際のコードの安全性

重複除外は同一のオブジェクトをまとめることによって成立するため、制御フローがグローバル オブジェクトのアドレスに依存するコードは安全ではありません。

以下の簡略化されたコードの例を見てみましょう。

```
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);
}
```

上記のコードでは、'a' と 'b' が重複除外されるため、これらは同一のデータを参照し、それゆえに同じアドレスを持つことになります。main が実行されているとき、if ステートメントの両方の実行は、同じパスをとります。

> メモ：if ステートメントがとる分岐は未定義です。これには'a' が重複除外されて 'b' になる場合と、'b' が重複除外されて 'a' になる場合が考えられます。

関数も重複除外されることがあるため、次のようなコードも危険です。

```
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);
}
```

上記のコードでは、'f' と 'g' が重複除外される恐れがあり、その場合、この 2 つは同じアドレスを持つことになります。この場合も、if ステートメントは両方の実行において同じパスを辿ります。

ポインタを比較する代わりに、補助的な列挙 (またはそれに準じるもの) を使用する方が安全です。

```
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);
}
```

このコードなら、正しく、`behaviour_a ()` と `behaviour_b ()` を呼び出すことができます。

重複除外とデバッギング

重複除外機能を使用する際には、Debugger でプログラムの一部を表示できなくなる可能性が高くなります。残念ながら、DWARF デバッグ形式の制限により、重複除外の使用による副作用はやむを得ないものとなっています。特に、2 つの関数が重複除外される場合を考えてみると、1 つの実行コードに対応するソース コードが 2 つ以上存在するようになります。

一般的には、重複除外されていないプログラムの部分においては Debugger を使用できます。しかし、もし重複除外の使用時に必要なソース コードが Debugger で表示できない場合、重複除外を無効にするとデバッギング機能が再び使えるようになります。

Tuner での重複除外の使い方

Tuner を使うときは、重複除外 (de-duplication) を使用しないようにしてください。Tuner で選択した sync 関数と同一の関数を重複除外した場合、Tuner が誤って新しいフレームを検出してしまうことがあるためです。場合によっては、重複除外された関数のシンボルが削除され、Tuner 内の sync 関数として使用できなくなることがあります。

プロファイル中に重複除外を行う必要がある場合には、慎重に sync 関数を選び、プロファイルの動作に重複除外の影響が及ばないようにしてください。

TOC を使う場合の重複除外の使い方

1 つのプログラム内に複数の TOC 領域があると、コードの重複除外の効力が低下します。通常、TOC を使用する関数は、同一の TOC 領域内の関数に対してのみ、重複除外されます。これはリンカの解析時に TOC の参照がコードを異なるもののように見せてしまうため起こります。さらに、同一のソース ファイル内の関数呼び出しは、呼び出された関数の重複除外を制限します。なぜならこの場合、コンパイラのコード生成では呼び出し先と呼び出し元の両方が同じ TOC 領域にあるものと想定し、重複除外によりこの想定が破られる場合があるからです。

こういった問題を回避するため、できる限り、`--no-toc-restore` または `--no-multi-toc` スイッチを使ってリンクしてください。「SN リンカ --notocrestore スイッチ」を参照してください。これらのオプションを使うと、コード重複除外を大幅に改善できます。

重複除外の制限事項

重複除外は、コンテンツが同一であるコードやデータ オブジェクト、また、参照している外部オブジェクトがあれば、それが同一であるコードやデータ オブジェクトを識別することにより実行されます。

たとえば、以下の 2 つの関数は重複とみなされます。

```
extern int c ();

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

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

重複除外は、解析を 1 回のパスで実行するだけで、同一であるコードやデータ オブジェクトを特定します。このため、同一であるかのようにみえる一部のコードやオブジェクトは出力に置かれたままとなります。

たとえば、次のような 2 つの外部関数 'f ()' および 'g ()' が重複除外される場合、関数 'd ()' および 'e ()' は重複とはみなされません。これらの関数は、1 回のパスの重複除外解析が行われるとき、別個の関数を参照するためです。

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

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

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

重複除外の有効性は、テンプレート化されたオブジェクトやデータを扱う際にも制限されます。次の例では、コンパイラは、関数テンプレートの各インスタンス化ごとに文字列定数を別個に出力します。その結果、文字列オブジェクトは重複除外されますが、'message<T>' インスタンス化は、いずれも重複除外の対象になりません。

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

同様の制限はテンプレート化されたクラスのメンバである定数でも該当します。

```
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";
```

静的データ メンバ 'message_' は、'Test<T>' のインスタンス生成中に重複除外されますが、メンバ関数の 'message ()' は重複除外されません。同様に、コンパイラによって定数が別個のデータ オブジェクトとして表されている場合、定数値を返す 2 つの関数も重複除外されません。以下に例を示します。

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

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

コンパイラは通常、次のようなコードを出力しなければならないため、上の 2 つの関数は重複除外されません (最適化レベルと定数の値による)。

```
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;
}
```

ここでは 2 つのベクター定数は重複除外されます。

より効果的な重複除外のためにコードを書き直すには

重複除外を最も効果的に実行するためには、定数またはデリゲートされた関数を使用するコードを書き直す必要がある場合があります。

たとえば、次のような関数は、予想通り重複除外されます。

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

__vector float vf1 ()
{
    return x_axis;
}

__vector float vf2 ()
```

```
{
    return x_axis;
}
```

テンプレート化されたクラスの場合、非テンプレート ベースのクラス、またはグローバル定数に定数を抽出する必要がある場合があります。以下の例を参照してください。

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

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

このように書き直しても、親オブジェクトの重複除外を有効化するにとどまり、オブジェクトのより深いツリー構造までの重複除外を助けるものではないことに留意してください。

ストリップ レポート

ストリップ レポート には、各種デッドストリップ オプションの適用効果が詳しく記載されます。

ストリップ レポートは以下の 4 つのセクションに分けられています。

使用していないオブジェクト

最初のセクションには、使用されない、または余分と判断されたオブジェクト (重複コードやデータなど) がリストされます。

ストリップ可能なバイト数 (ストリップ)、残存するパディング数 (パッド)、オブジェクトの名前、オブジェクトのあるセクション、オブジェクトが属するファイルなどが表示されます。

表に含まれるオブジェクト ファイルやアーカイブ名に続く角かっこ内の値は、シンボル テーブル内のオブジェクトのインデックスを表します。

例：

```
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.
```

ストリップできないオブジェクトから参照されるオブジェクト

2 番目のセクションには、何らかの理由でストリップが認められないオブジェクトがリストされます。

このセクションの最初には凡例が記載され、その下の表で使用されるフラグの意味が説明されます。リストには、オブジェクトがストリップできない理由を説明したフラグと、オブジェクト名、そして出力ファイル内でこれが属するセクションが表示されます。ここでもっとも一般的なフラグは「G」で、これはオブジェクトが、ストリップされない他のグローバル シンボルによって参照されていることを意味します。一般的に、これは参照場所を特定のオブジェクトに帰属させられなかったことを意味します (サイズ 0 のシンボルに関連する位置が起源の場合など)。

例：

```
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)
```

ストリップされていないオブジェクトと、それが参照するオブジェクト

3 番目のセクションには、出力ファイルに対して決定される参照グラフが含まれます。参照グラフ内のリンクは、どのプログラムでも表示しにくいため、グラフの各エントリはこれが直接参照するオブジェクトと共にリストされます。

- エントリの前には一連のダッシュとソース オブジェクト ファイルが配置されます (かっこに囲まれている場合もある。オブジェクトがアーカイブに帰する場合は、その後にアーカイブ名)。オブジェクト名は、かっこ内に表示されるターゲット セクションと共にリストされます。
- オブジェクトがグラフ内のリーフでない場合、「`requires...`」が表示され、そのオブジェクトを参照するオブジェクトのリストが合わせて表示されます (1 行に 1 つの参照オブジェクト)。
- 参照オブジェクトの後に「[symbol]」が表示される場合、それは、これ以上解決できないシンボルへの参照を表します (このため、デッドストリッピングには適さない)。

以下はその例です。

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


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

デッドストリッピングに準拠したツールチェーンでビルドされていないオブジェクト モジュール

4 番目のセクションには、SN リンカで使用されるデッドストリッピング メカニズムとは互換性のない、オブジェクト ファイルのリストが表示されます (かっこに入っている場合もあり、続いて親アーカイブを表示)。

残念ながら、SDK 200 で使用されているものより前の GNU アセンブラのバージョンの最適化は、セクション内の分岐の再配置可能ファイルを生成するものではありませんでした。これは、分岐命令が出力先の相対値を使うためで、分岐命令はソースと出力先アドレスの違いを認知しています。これにより、リンカは呼び出し元の参照が存在することを知りえず、再配置エントリがないことから、リンカのデッドストリッパがうまく機能しませんでした。

SDK200 以降の GNU アセンブラの出力には必要なすべての再配置エントリが含まれ、かつ、必要な再配置エントリが存在していることを示すビットが ELF のファイル ヘッダーに設定されるため、安全にデッドストリップが行えます。当然ながら、そして残念ながらこれには完全な再ビルドが必要になります。

デッドストリッピングに安全ではないオブジェクト ファイルにリンカが遭遇すると、次のような警告が発行されます。

```
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>]).
```

関数「ゴースト」

リンカでは、ストリップされた関数の「ゴースト」が最終実行可能ファイルに残されるような状況が存在します。

デッドストリッピングや重複除外の際に、リンカは入力ファイル内で定義されたシンボルを元に処理を行います。これらのシンボルは、入力ファイルに含まれているデータ項目の名前、場所、アドレスの情報を持ちます。しかし、シンボルが対応する範囲にないデータが存在することはめずらしくありません。デッドストリッパは、このデータにはまったく干渉できません。データは削除してはなりませんし、その要件 (この場合はアラインメント) は引き続き遵守しなければなりません。

この現象が発生する例を以下に示します。

```
    .section .text

    # アラインメントは 2^3 (すなわち 8 バイト)。
    .align 3

    # 「.foo」シンボルを宣言し、これに .text セクション内の
    # アドレスを指定する。
.foo:
    blr

    # ここで、.foo シンボルのサイズを提供する。
    .size . - .foo

    # コンパイル時にここに nop が生成され、
    # 以下の関数が 8 バイトでアラインされることを保証する。これはシンボル サイズの一部
    # ではない場合がある。
    nop
.bar:
```

コンパイラは関数「bar」を 8 バイトでアラインします。これを行うために、コンパイラはセクションを 8 バイトでアラインし、nop 命令を「foo」の終わりに追加します。残念ながら「foo」のシンボルにはこの nop のサイズは含まれません。

デッドストリッピングを行う際、リンカは参照されていない関数やデータをすべて削除します。しかし、削除を行うときは、リンカはセクションの宣言されたアラインメントを順守し、残りのオブジェクトが引き続き正しくアラインされるようにしなければなりません。また、セクション中で ELF シンボルによってカバーされない部分は、そのデータが不必要かどうかを判断できないため、デッドストリッパが処理することはできません。

シンボルの定義によってカバーされていないため、リンカは nop をストリップできず、また、リンカは 8 バイトのアラインメントを維持しなければならないため、nop で占有される 4 バイトを単純に残すこともできません。結果として、上記の例では blr とパディングとして挿入された nop 命令の両方が、ストリップされた関数 foo の「ゴースト」として残ることになります。

GCC の場合、可能な回避策として -falign-functions=4 スイッチを渡します。これにより、関数が 4 バイト境界でアラインされ、不要な nop 命令が削除されます。パフォーマンスへの影響はほとんどあるいはまったくありませんが、アラインされていない関数の存在により、関数の初回命令発行時に余分な 1 CPU サイクルが追加される可能性はあります。ただし、この種の発行ストールが PPU において障害となることはめったにありません。

トレードオフ

デッドストリッピングと重複除外の両方が、最終的なバイナリ イメージ内のスペースの節約を行います。しかし、これには次のような点で、特定のトレードオフが伴います。

- 読み込み可能なイメージ サイズ

- リンク時間
- ランタイムのパフォーマンス
- デバッギングとプロファイリングのしやすさ

読み込み可能なイメージ サイズ

デッドストリッピングの有効性は、使用していないオブジェクトがバイナリ コード内にいくつ存在するかに左右されます。一方、重複除外の有効性は、バイナリ コード内の同一オブジェクトの識別に左右されます。このため、デッドストリッピングと重複除外の有効性は、リンカへの入力の構造に左右されます。

リンク時間

ビルド シナリオとコードの構造により、デッドストリッピングを有効化するとリンク時間が長くなることがあります。入力オブジェクト ファイルとアーカイブのすべてがオペレーティング システムのファイル キャッシュ内にある場合、リンク時間が 2 倍になるなど、その影響が顕著になることがあります。しかし、オペレーティング システムのファイル キャッシュが空である場合には、デッドストリッピングのコストは通常、ファイル入/出力の影響と比較して有意にはなりません。

これと同様、重複除外のリンク時間はデッドストリッピングのリンク時間を越え、さらに長くなることがあります。デッドストリッピングと同様、パフォーマンスへの影響はコード構造だけでなく、オペレーティング システムのファイル キャッシュにも左右されます。

ランタイムのパフォーマンス

デッドストリッピングと重複除外は共に、リンクされたプログラムのパフォーマンスに影響を及ぼします。コードとデータの物理レイアウトが、削除されるものにより異なるためです。デッドストリッピングはオブジェクトの順序を変化させずに使用していないオブジェクトを削除するため、ランタイム パフォーマンスにおける変化は、通常、メモリ レイアウトとランタイム キャッシュの挙動の変更によるものです。このため、ランタイム パフォーマンスにはプラスまたはマイナスの変化がもたらされます。

しかし、重複除外は空間的局所性を低減する方向に物理レイアウトを変更します。プログラム内の任意の場所に 2 つの同一オブジェクトがあれば、これらはマージ可能なためです。たとえば、ひとつのソース ファイルにある 2 つの関数の 1 つが重複除外された結果、これらの 2 つの関数がメモリ内の近い場所になくなってしまう場合があります。これらの 2 つの関数が互いに呼び出し合う場合には、以前に呼び出されていなければ命令キャッシュミスが起きるという結果になりえます。それによってランタイム パフォーマンスが低下します。

デバッギングとプロファイリングのしやすさ

重複除外は同一のオブジェクトをバイナリ レベルでマージします。Tuner や Debugger といったツールを使用しているときは、これは予期せぬ結果につながる場合があります。これらのツールは、ソースおよびバイナリ レベルの構成要素間の一義的な対応に依存するためです。

詳細は、「重複除外とデバッギング」および「Tuner での重複除外の使い方」を参照してください。

まとめ

読み込み可能なイメージ サイズには問題がない場合、デッドストリッピングと重複除外を無効化するのが最良の選択といえます。これによりリンク時間が最短になります。

読み込み可能なイメージ サイズとランタイム パフォーマンス両方で同じような問題が見受けられる場合、デッドストリッピングを有効化すると、これらの問題点や高速リンク時間の間で好ましいバランスが保てます。しかし、読み込み可能なイメージ サイズが重要な要因である場合、重複除外を有効化すると、リンク時間は延びますが、多少のメモリの節約ができます。

デバッグを行うときは、重複除外を無効化しておくとデバッグの動作がよりよいものになります。プロファイリングを行うときは、プロファイリング結果を正しく解釈するために、または sync 関数が重複除外されないようにするために、重複除外を無効化する必要がある場合があります。しかし、重複除外は正規の C/C++ プログラムのセマンティクスに影響を及ぼす可能性があります。このため、ビルドのテストに

使用するデッドストリッピング/重複除外の設定は、最終的に使用するものに一致させてください。詳細は、「重複除外を使用する際のコードの安全性」を参照してください。

実際の結果は多くの要因に左右されるため、デッドストリッピングや重複除外の有効化による影響を慎重に調整し、利用している状況にトレードオフがふさわしいものであることを確認してください。

6: リンケージコードの生成

「リンケージコード」とは、リンク時に作成されるコードのスニペットを指します。これらは、TOC などの ABI 仕様をサポートするために PS3 で使用されます。

リンケージコードは呼び出し元の近くに挿入され、オリジナルのコール命令がリンケージコードに確実にアクセスできるようにします。

TOC を切り替える場合

TOC を切り替える際に生成されるリンケージコードではTOC レジスタの値が変更されます。このリンケージコードには、呼び出される側の関数アドレスと、呼び出し側の TOC 領域からの相対距離の両方が含まれます。

> **ヒント:**合計 64KB 未満の TOC データをプログラム内で使用することにより、このリンケージコードの生成を阻止し、結果として生じるパフォーマンスへの影響を避けることができます。これを実行する方法には、リンカの `--no-multi-toc` または `--no-toc-restore` オプションの使用が挙げられます。「TOC オーバーヘッドの除去」セクションに記載されているテクニックを使用すると、さらに TOC オーバーヘッドを削減することが可能です。

分岐距離が短い場合

分岐距離が相対分岐命令で許可される最大値を越えない場合、以下のリンケージコードがリンカによって挿入されます (`R_PPC64_REL24` 再配置に相当)。

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

分岐距離が長い場合

分岐距離が相対分岐命令で許可される最大値を越える場合、以下のリンケージコードがリンカによって挿入されます。呼び出し側では、リンケージコードへの相対的な分岐を実行し、続いてリンケージコードが 32 ビットで表現されるアドレスに対して分岐を実行します。

TOC レジスタに加え、このリンケージコードによって `%r11` レジスタと `%ctr` レジスタが変更されます。これらは、ABI で volatile として定義されています。

```
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
```

TOC を切り替えない場合

分岐距離が相対分岐命令で許可される最大値を越える場合、リンカはリンケージコードを挿入します。

このリンケージコードによって `%r11` と `%ctr` レジスタが変更されます。これらは、ABI で volatile として定義されています。

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

レジスタの保存および復元をする場合

リンカは必要に応じて、Cell OS Lv-2 PPU ABI 仕様書の「レジスタの保存および復元関数」セクションに記載されている関数を作成します。ABI で定義される名前を持つ未定義関数への参照 (_savegpr0_32 や _restvr_20 など) が存在する場合、リンカはこのドキュメントに記載されている定義で関数を作成します。

リンカはこれらの関数の複数コピー作成をできる限り回避し、かつ、これらの関数へのリンケージコードを作成しないことを保証します。既存の「レジスタの保存および復元関数」に対してアクセスが不可能である場合、リンカは新しいコピーを生成します。

例:

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

「レジスタの保存および復元関数」の全リストについては、Cell OS Lv-2 PPU ABI 仕様書を参照してください。

7: 例外と RTTI

例外処理サポートを含むリンク

コマンドラインに「--exceptions」を追加すると、C++ 例外処理に必要なランタイム サポート ライブラリがプログラムにリンクされます。これらのライブラリには、C++ 実行時タイプ情報 (RTTI) に対するサポートも含まれます。

スイッチ	詳細
--exceptions	例外処理ありのライブラリ用のリンカ デフォルト パスを追加する (LIB_SEARCH_PATHS "exceptions" キーから)。

例外処理サポートを含まないリンク

コマンドラインに「--no-exceptions」を指定すると、C++ 例外処理に対するサポートを含まずにプログラムがリンクされます。ただし、RTTI のサポートはこの設定においても有効となります。

スイッチ	詳細
--no-exceptions	例外処理なしのライブラリ用のリンカ デフォルト パスを追加 (LIB_SEARCH_PATHS "no_exceptions" キーから) (デフォルト)。最終出力ファイルから例外処理データの削除を行う。

警告:リンカは、リンクされるオブジェクトが例外処理サポートを必要とするかどうかを検出しないので、コンパイラとリンカで例外処理オプションが一致していないと、プログラムが実行時に誤動作する可能性があります。一般的に、例外処理サポートが有効とされていない場合にプログラムで例外が発生すると、ランタイムによって abort() 関数がコールされます。

SN Linker のデフォルトでは、例外処理サポートが無効な状態でリンクが行われます。

RTTI サポートを含まないリンク

SN Linker には、RTTI サポートを含まずにリンクを行うオプションが用意されていません。これはインターフェイスをシンプルなものとするために決定されたもので、RTTI サポートを削除してもほとんど影響がないことに基づいています。RTTI サポートのオーバーヘッドは、プログラム サイズにおける若干の増加で、実行時間への影響はありません。また、例外処理と RTTI のいずれのサポートも含まずにコンパイルされたリンク コードは、RTTI をサポートするランタイム ライブラリと安全にリンクできます。

リンクに含まれる SDK ライブラリの観点から見ると、これは、標準のライブラリと fno- exceptions ライブラリのどちらかがリンク可能で、fno-exceptions/fno-rtti ライブラリは使用できないことを意味します。ただし、リンク プロセスでの処理方法の結果、RTTI サポートはそれが使用される場合にのみリンクされるため、RTTI を使用しないコードでは、概して影響を受けることがありません。

8: TOC 情報

バックグラウンド情報

Cell OS Lv-2 PPU ABI 仕様書には、コンパイラとリンカ両方の挙動に影響する TOC として知られる構造についての説明があります。

- 関数へのコールでは、リンカがコード修正を行えるように、コール命令自体の後にスペースを設けなければならない。
- ポインタによる関数へのコールでは、中間的な構造体である「.opd」エントリを使用しなければならない。この構造体は、ターゲット コードで使用される TOC 領域のアドレスと、ターゲット コード自体のアドレスから構成される。

以下は、両方の挙動を示した例です。

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

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

SNC-O3 スイッチでこのサンプルをコンパイルすると、以下のスニペットのようなコードが生成されます。この大半は ABI によって規定されているため、GCC でも非常に似たコード出力が行われます。明確にするために、関数のプロローグとエピローグは削除されています。

```
.foo:
     ...関数プロローグを削除...
     lwz     %r4, 0(%r3)
     std     %rtoc, 40(%sp)
     mtctr     %r4
     lwz     %rtoc, 4(%r3)
     bctrl
     ld     %rtoc, 40(%sp)
     ...関数エピローグを削除...

.qaz:
     ...関数プロローグを削除...
     bl     .bar
     nop
     ...関数エピローグを削除...
```

この命令の大半は TOC を機能させるために存在しています。GCC でコンパイルされるコードとの互換性を確保するため、ABI で規定されているルールに注意深く従っていますが、SNC ではこの TOC にデータを配置しません。

「TOC 復元なし」モードにより、直接コールと関数ポインタによるコールの両方の効率を上げることができます。

TOC オーバーヘッドの除去

このセクションでは、SNC コンパイラの -Xnotocrestore=2 コントロール変数設定、および SN リンカの --notocrestore (--no-toc-restore の別名) スイッチの使用による、「TOC 復元なし (no TOC restore)」モードの使用について説明します。これらのスイッチにより、TOC のオーバーヘッドをほぼ完全に削除し、全体的なコード サイズを大幅に削減できます。

- SNC の -Xnotocrestore=2 でコンパイルされたコードと、SNC でコンパイルされた他のコードや GCC でコンパイルされたコードは、リンク時に混合させることができます。ただし、アプリケーション内の TOC サイズが合計 64KB 以下である必要があります。この制限は、--no-toc-restore スイッチが使用される際に、リンカによって適用されます。

「TOC 復元なし」モードの使用方法は次のとおりです。

(1) SNC の -Xnotocrestore=2 コントロール変数を設定してコンパイルする。

(2) SN リンカの --notocrestore スイッチを有効にしてリンクする。

> **警告:**「TOC 復元なし」モードと互換性のない方法で、PRX コードを構築することも可能です。問題が発生した場合は、「制限事項」を参照してください。

SN Linker コマンドライン スイッチ

スイッチ	詳細
--multi-toc	--multi-toc は、64 KB より多い TOC データの使用を可能にするスイッチ (これがデフォルトの挙動)。 別の TOC 領域を使用するモジュール間でのコール時、リンカは TOC 領域調整を行うリンケージコードを自動的に生成する。 「TOC を切り替える場合」を参照。
--no-multi-toc	--no-multi-toc は、複数の TOC 領域をサポートするために使用されるリンケージコードをリンカが作成しないようにするもので、プログラムに 64KB を越える TOC データが含まれる場合、エラーを発する。このスイッチは GNU ld にも存在。 error:L0154: 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	SNC PPU C/C++ コンパイラの -Xnotocrestore=2 スイッチと合わせて使用する。 --notocrestore スイッチにより、リンカでは PRX スタブ ライブラリ (OS などの PRX 関数へのコールを行うために使用される) の再記述が行われる。この機能は、複数 TOC 領域が存在する場合には使用できないため、--notocrestore 使用時は --no-multi-toc が有効になっていると仮定される。 --no-toc-restore は --notocrestore の別名。

SNC PPU C/C++ コンパイラ コントロール変数

ここでは便宜のために、SNC PPU C/C++ コンパイラ -Xnotocrestore コントロール変数の説明をします。より厳密な説明については、『SNC PPU C/C++ コンパイラ ユーザーガイド』を参照してください。

コントロール変数	詳細
-Xnotocrestore=0	コンパイラは、ABI 完全準拠コードを生成する。ポインタによって関数をコールするコードでは、呼び出し先の TOC レジスタの値が、呼び出し元のものと異なる可能性があると仮定される。

	呼び出される側のコードがリンク時に別の TOC 領域に存在する場合に、リンカが TOC ポインタの復元を行えるように、外部関数へのコール後に nop 命令が生成される。 このオプションでビルドされたコードを適切に実行するために、特別なリンカ スイッチは必要ない。 この値が notocrestore 制御のデフォルト値。
`-Xnotocrestore=1`	コンパイラにより、外部関数へのコール後の nop 命令が省かれるが、ポインタを通じたコールは TOC-safe であることが保証される。プログラムは、SN リンカの `--notocrestore` スイッチをつけてリンクする必要がある。
`-Xnotocrestore=2`	コンパイラでは、外部関数へのコール後の nop 命令が省かれ、ポインタによるコールは常に同じ TOC 領域を使用すると仮定される。プログラムは、SN リンカの `--notocrestore` スイッチをつけてリンクする必要がある。

SNC コンパイラ -Xnotocrestore コントロール変数

`-Xnotocrestore=2` を使用して「バックグラウンド」セクションと同じコード スニペットを SNC でコンパイルすると、以下のようなコードが出力されます。

```
.foo:
     ...関数プロローグを削除...
     lwz     %r3, 0(%r3)
     mtctr     %r3
     bctrl
     ...関数エピローグを削除...
.qaz:
     ...関数プロローグを削除...
     bl      .bar
     ...関数エピローグを削除...
```

こちらの方がはるかに優れています。ここでは、最初のケースから 1 ストアと 2 ロードを削除し、2 番目のケースからは不必要な nop を削除しています。

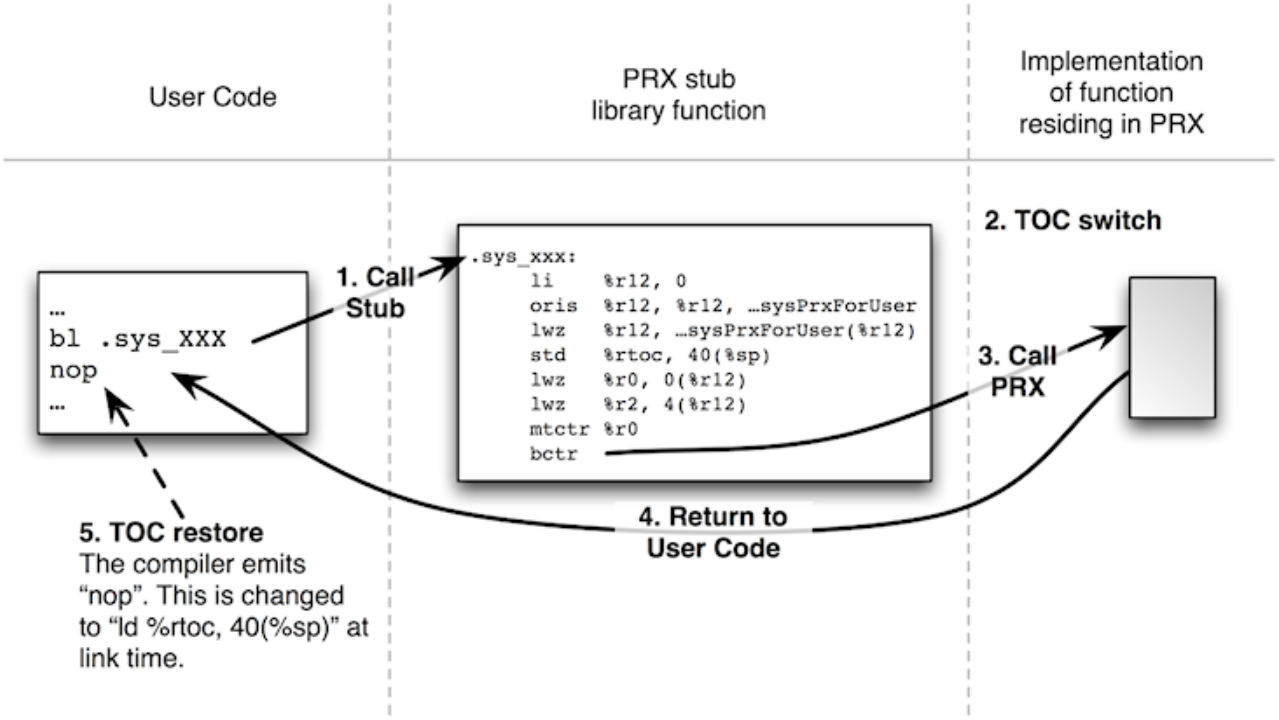
このコードでは、TOC レジスタ (`%rtoc`/`%r2`) は関数コールの実行前に変更する必要が一切ない、その後に復元される必要はないと仮定しています。この条件を満たすためには、リンカによる支援が必要となります。

SN Linker --notocrestore スイッチ

GCC でコンパイルされた PRX ライブラリは、そのままでは「TOC 復元なし」モデルとともに動作しません。これらの PRX ではメイン プログラムとは別 TOC 領域が必要であり、リンカではこの挙動を継続してサポートしなければなりません。

以下の図は、PRX スタブ ライブラリ内のコードを、「TOC 復元なし」モード に対応するために別の実装に置き換える、リンカで使用されるメカニズムを示したものです。

--notocrestore を使用しない PRX コール



PRX のコールに使用されるデフォルトのコード シーケンスには、ターゲット関数への分岐前に、PRX の TOC データをポイントするように TOC レジスタを変更するコードが含まれます。この関数は、オリジナルのコールした場所に直接戻ります。リンカは、オリジナル コール命令に続く nop 命令をパッチし、呼び出し側の TOC レジスタ値を復元します。

実行される処理は以下のとおりです。

(1) ユーザー コードにより、PRX スタブ コードがコールされます。これにより、PRX コールの実行に必要なセットアップが行われます。

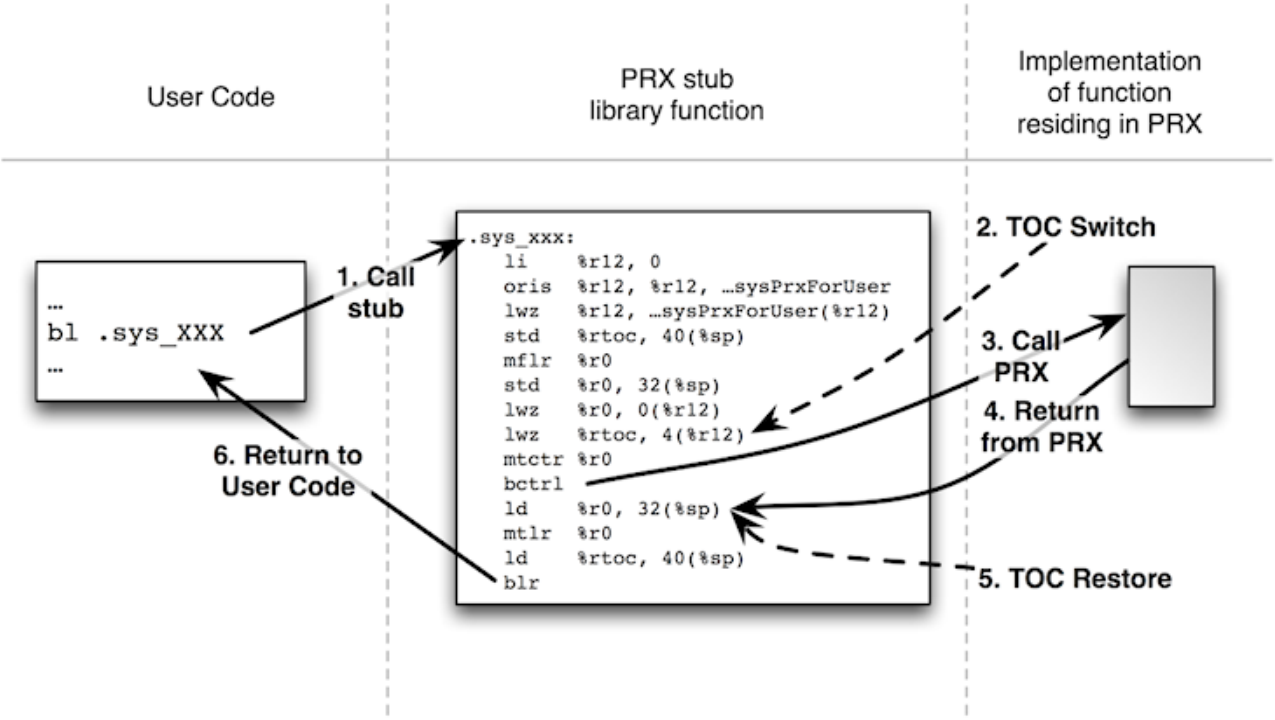
(2) PRX スタブ コードでは、TOC レジスタの現在値が保存され、PRX TOC 領域が参照されるように TOC レジスタを変更します。

(3) PRX コードが呼び出されます。

(4) コールが直接ユーザー コードに戻されます。

(5) ユーザー コードによってオリジナルの TOC 領域が復元されます。従来の外部関数コールと同じテクニックを使用し、コンパイラの nop が TOC 復元命令に置換されます。これが「PRX の修正」プロセスです。詳細は「PRX の自動修正」を参照してください。

--notocrestore を使用した PRX コール



「TOC 復元なし」モードが有効な場合、リンカは、PRX 関数コールの実行に使用されるコード シーケンスを認識すると、これを TOC 復元を直接行うバージョンに置換します (コール元のパッチに依存するのではなく)。

置換コードによって実行される処理は以下のとおりです。

(1) ユーザー コードにより、PRX スタブ コードがコールされます。これにより、PRX コールの実行に必要なセットアップが行われます。

(2) PRX スタブ コードでは、TOC レジスタの現在値が保存され、PRX TOC 領域が参照されるように TOC レジスタを変更します。

(3) PRX コードが呼び出されます。

(4) コールが PRX スタブ コードに戻ります。

(5) オリジナルの TOC 値が復元されます。

(6) ユーザー コードに戻ります。

このアプローチの利点は、PRX 関数へのコール命令の後に続く、コンパイラによって追加される nop 命令の生成に依存しないことにあります。

この方法で PRX スタブ ライブラリの置換を有効にするには、`--notocrestore` スイッチを使用してください。

TOC データの合計が 64KB を越える場合、リンカはエラーを発行します。

```
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 では、`-mminimal-toc` や `-mbase-toc` スイッチなどを使用し、TOC の領域使用を強制的に少なくすることが可能です。SNC は TOC データを生成することはありません。

制限事項

「TOC 復元なし」手法は、PRX モジュール内関数へのコールをインターセプトすることに依存しています。直接的なコールの場合、リンカでは、前述したようにスタブ コードの書き換えが行えます。

以下の例では、これが C と C++ プログラムの両方で示されており、それぞれで関数ポインタと仮想メソッドが個々に使用されています。

メモ：これらの例では、TOC への参照があることを確実にするために、PRX ライブラリ ソース コードのコンパイルには GCC を使用する必要があります。PRX のコンパイルに SNC が使用される場合、問題は発生しません。

TOC 復元なし C サンプル (prx1)

最初の例は、PRX 常駐ライブラリ (prx1.h、prx1.c) と、PRX を使用するアプリケーション (app1.c) で構成されます。PRX では、関数のアドレスを戻す関数 `get_callback()` がエクスポートされます。メインプログラムでは、`get_callback()` を使用して関数ポインタを取得し、その後にこれをコールします。予想される結果では、PPU stderr チャンネルに「in callback」というテキストが表示されます。

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);
/* get_callback 関数をエクスポート。*/
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"

/* PRX のコール準備 */
```

```
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;
}

/* PRX の後処理*/
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);

    /* PRX からコールバックを取得し、それをコール */
    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 .
# 「TOC 復元なし」モードを試みる場合は、
# 以下の 2 行のコメントを外す
#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

# 問題を再現させるため、TOC が確実に使用
# されるようprx1.o は GCC で
# コンパイルしなければならない。
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) $^
```

「TOC 復元なし」モードが使用されるとこの例は失敗します。リンカは、ポインタによってコールされる場合、callback() で認識される TOC ポインタの値を正しいものにするために行う、コールバック関数のインターセプトができません。

TOC 復元なし C++ サンプル (prx2)

2 番目の例も、PRX 常駐ライブラリ (prx2.h と prx2.cpp 内のソース コード) と、PRX を使用するアプリケーション (app2.cpp) で構成されます。PRX では、class foo のインスタンスを戻す関数 get_foo() がエクスポートされます。続いてメイン プログラムでは、クラスの仮想メソッドの 1 つが呼び出されます。予想される結果では、stdout に「in member_function」というテキストが出力されます。

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

# 「TOC 復元なし」モードを試みる場合は、
# 以下の 2 行のコメントを外す
#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

# ライブラリ PRX をリンクし、対応する
# スタブ ライブラリを作成。
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)
```

この例は、前述の C サンプルよりも一般的ではないでしょう。C++ クラス インターフェイスをライブラリから露出することは、「脆弱な基底クラス」問題 (http://en.wikipedia.org/wiki/Fragile_base_class) を参照) による悪影響を受けるため、めったに行われません。とはいえ、この例も同様な問題が発生します。この場合、リンカでは仮想関数コールのインターセプトができず、TOC 値に必要な調整が行えないため、PRX ライブラリで定義される仮想メソッドへのコールは失敗となる可能性があります。

解決策

この問題を解決するには、2 つのアプローチがあります。最初のアプローチでは、コードを変更する必要はないものの、「TOC 復元なし」モードの利点のいくつかを妥協しなければなりません。2 つめのアプローチではその妥協は削減されるものの、コード変更が必要となります。

SNC コンパイラ -Xnotocrestore=1

最初のアプローチでは TOC 復元モデルを使用します。このモデルではコンパイラにより、間接的な関数コールを実現するために TOC を意識したコードが生成されます。

これまでの全例において、コンパイラの -Xnotocrestore=2 モードが使用されてきました。このスイッチによってコンパイラでは、外部関数へのコールとポインタによるコールの両方から、TOC 関連の機構が削除されます。

リンカでは、リンク時にスタブ ライブラリ コードを置換することにより、呼び出される側の TOC レジスタ値がエントリ時に補正されると言う意味では、外部関数のコールが「安全である」と保証できます。ただし、ポインタによるコールについてはこの保証を行えません。

-Xnotocrestore=1 を使用すると、SNC コンパイラでは外部関数へのコール後の nop 命令が省略されますが、ポインタによるコールの実行には、TOC を意識したコードが引き続き使用されます。

この妥協により、上述の両方の例が正しく機能します。残念ながら、不必要な TOC コードが通常より除去されないことにより、「TOC 復元なし」モードの利点は減少します。

#pragma control notocrestore=0

2 つ目のアプローチでは、PRX ライブラリで実装される関数への間接的なコールを特定し、そのコールを実行する小さい関数を記述します。そしてこの新しい関数を、「TOC 復元なし」オプションがコンテキストで無効にされるべきであることを示す、コンパイラ プラグマでマークします。

上記 app1.c の例でのコード：

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

は、以下のようになります。

```
#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);
    ...
}
```

「TOC 復元なし」モードを無効にすること、および invoke_callback 関数がインライン化されないようにすることの両方が重要です。最適化制御は関数ごとに機能するため、invoke_callback() がインライン化された場合、「TOC 復元なし」モードの制御値を変更しても効果はありません。

このため、2 つのプラグマを使用する必要があります。1 つは「TOC 復元なし」モードを無効にするもの：

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

2 つ目は関数のインライン化を防ぐものです。

```
#pragma noinline
```

最後に、「TOC 復元なし」モードの前のステータスを復元します。

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

C++ 例の変更も同じ方針で行えます。ただし、今回は foo の「プロキシ」クラス (以下の例では foo_proxy と命名) と、スマート ポインタ クラス ntr_ptr (no-toc-restore ポインタ) が導入できるため、foo の仮想メソッドの各コールへの変更数を最小限に抑えることが可能です。

```
// アプリケーションから PRX へのコールに対し、
// プロキシとして作用するクラスを宣言。
class foo_proxy
{
public:
    typedef foo  proxied_type;
    explicit foo_proxy (foo * f) : f_ (f) { }

    void member_function () const;

private:
    foo * f_;
};

// foo::member_function をコールするスタブ関数。
// ABI に準拠した TOC 処理を使用するために、notocrestore 制御を 0 に
// 設定するプラグマと、この関数がインライン化されるのを防ぐための
// プラグマを使用。
#pragma control %push notocrestore=0
#pragma control noinline
void foo_proxy::member_function () const
{
    f_->member_function ();
}
#pragma control %pop notocrestore

// メンバ関数コールの実行時に、対応するプロキシ クラスが
// 確実に使用されるようにする、「スマート ポインタ」タイプの
// テンプレート クラス。
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_;
    };
```

上記のプロキシ クラスとテンプレート クラスを使用するため、main() の実装を以下のように多少変更します。変更前 :

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

```
    ...
}
```

は、以下のようになります。

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

TOC 使用レポート

リンカは、リンクに必要とされる TOC の割り当てやリンケージコードが表示されたレポートを出力することができます。これは、リンケージコードのパフォーマンスやコード サイズにあたえる影響を把握する際に役立ちます (詳細は、「TOC を切り替える場合」を参照)。

TOC 使用レポートは 3 つのセクションで構成されます。

- 「TOC モジュール サイズ」セクション：各モジュールと、そこに含まれる TOC データの量が示されます。
- 「TOC 割り当て」セクション：リンクされているオブジェクト モジュール内の各セクションに割り当てられている TOC 領域が示されます。また、リンカでは、別の TOC 領域を使用する関数 (つまりリンケージコードが必要となる関数) の名前を発見するため、静的解析が行われます。これらは各セクション内にリストされます。
- 「TOC 統計」セクション：ここには、それぞれの TOC 領域がリストされ、そのサイズとアドレスが表示されます。

このレポートは、他ツールによる直接的な解析を可能にするため、タブ区切りテキストとして記述されます。

コマンドライン スイッチ

スイッチ	解説
`--toc-report=<file>`	*<file>* に対して TOC 割り当てレポートを書き込む。

9: PRX ファイルのビルド

PRX 生成

コマンドライン スイッチ

スイッチ	解説
-mprx	--oformat=prx と同等。GCC との互換性を目的とする。
-mprx-with-runtime	--oformat=prx --prx-with-runtime と同等。GCC との互換性を目的とする。
--oformat=prx	PRX 出力を作成。
--oformat=fsprx	サイン付き PRX 出力を作成。
--prx-with-runtime	コンパイラのランタイム ライブラリと PRX 出力をリンクする。(--oformat が「prx」または「fsprx」の場合のみ有効。)
--strip-unused	コードのデッドストリッピングを有効化する。リンカにより、プログラムの完全なコール ツリーが構築されるため、オブジェクト ファイルとアーカイブがスキャンされる。不必要であると判断された関数はすべて最終 PRX ファイルから削除される。 「デッド ストリッピングと重複除外」を参照。
--strip-unused-data	暗黙的に --strip-unused を有効にする。デッド コードのスキャンに加え、リンカでは、使用していないデータ オブジェクトを検出し、PRX ファイルからこれらを削除する。 「デッド ストリッピングと重複除外」を参照。
--zgc-sections	PRX 出力に対するデッドストリップを有効にする。GCC との互換性を目的とする。リンカにより --strip-unused-data が選択され、以下の警告が発せられる。 warning: LO153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping
--zgenentry	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenprx	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenstub	リンカに最初のリンクを実行させ、「--stub-archive」を libgen ツールに渡す。Cell OS Lv-2 PRX プログラミング ガイドを参照。

10: SPU プログラムの埋め込み

SN Linker を使用した SPU ELF ファイルの埋め込み

リンカに直接 SPU ELF ファイルを渡すことによって、SPU ELF ファイルは最終プログラムに埋め込むことができます。この方法ならば、SPU ELF へのリンクと、それの PPU ELF への埋め込みの間の中間手順が不要になります。

改良されたワークフローは、リンク時間全体への影響もありません。SPU ELF ファイルは、コマンド ライン上で PPU オブジェクト ファイルであるかのように、同じ順序で処理されます。リンカはそれぞれ埋め込まれた SPU ELF に対するメタデータも生成します。このため、Tuner や Debugger でソースの特定ができます。なお、リンカは埋め込み SPURS ELF ファイルや libovis オーバーレイ テーブルに必要とされる追加のメタデータも生成します。libovis に関する詳細は、「libovis 概要」を参照してください。

リンカは実行可能な ELF ファイル、共有されているライブラリ、SPURS タスク、SPURS ジョブ 2.0、SPURS Jobqueue ジョブ、およびカスタマイズされた SPURS ポリシー モジュールなどを埋め込むことができます。

図 1 から図 3 は、SPU ELF を最終PPU プログラムに埋め込む際のワークフローの違いを表したものです。

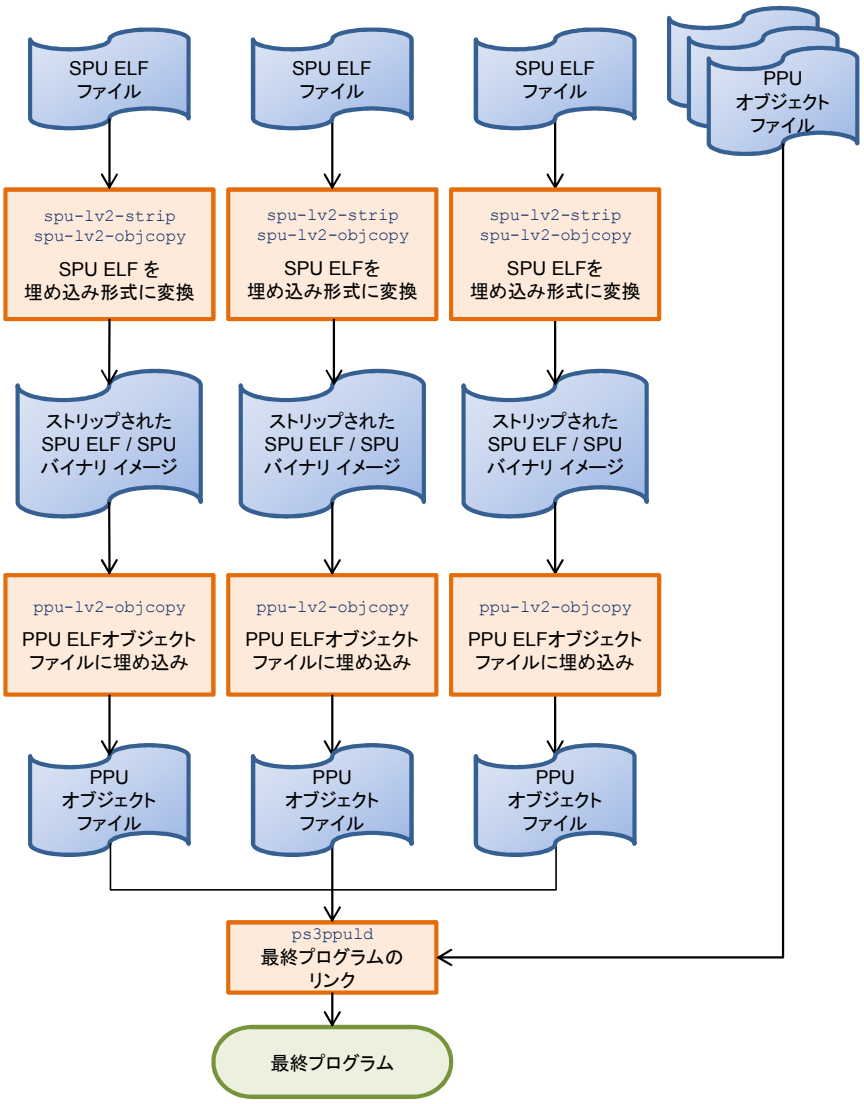


図 1：この図は、GNU バイナリ ユーティリティを使って SPU ELF ファイル データをフォーマットし、次にそれを最終リンクで利用できるような PPU オブジェクトに埋め込んでいるところを示す例です。

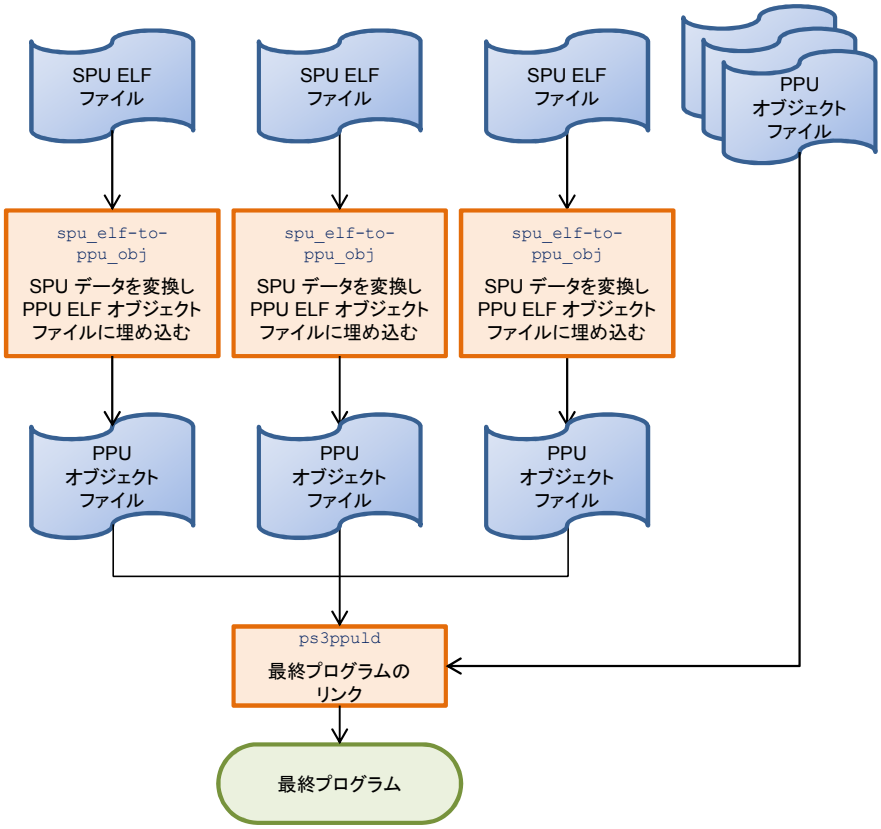


図 2：この図は、外部ツールである spu_elf-to-ppu_obj を使って SPU ELF ファイルをフォーマットし、PPU オブジェクト ファイルに埋め込んでいるところを示した例で、最終リンクですぐ使えます。

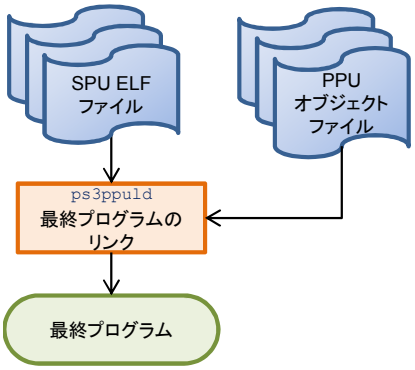


図 3：この例では、フォーマットされて埋め込まれる SPU ELF ファイルが直接リンカ コマンド ライン上で指定され、それらが PPU オブジェクトであるかのようにリンクされるため、中間ツールは必要とされません。

SPU 埋め込みコマンド ライン オプション

スイッチ	解説
`--spu-format=<format>`	SPU ELF ファイルの埋め込み *<format>* を指定する。利用できるオプションは次のとおり。「`binary`」、「`default`」。
`--print-embedded-symbols`	最終リンク内に埋め込まれたすべてのバイナリ デ

	ータについて生成されたシンボルをプリントする。

Jobbin2 フォーマット サポート

リンカでは、jobbin2 形式 のSPURS ジョブ ELF ファイルの埋め込みのみがサポートされています。

ファイルの埋め込みに関する詳細は、SPU プログラム作成支援ツール ユーザガイド「SPU プログラムを PPU プログラムに埋め込む手順」 を参照してください。

バイナリの埋め込み形式を明示的に指定しても、埋め込み SPURS ジョブの形式が変更されることはありません。jobbin2 形式は、標準バイナリ形式のイメージに含まれる、ゼロで初期化されたデータ (.bss) セクションを削除することにより埋め込みイメージのサイズを低減させるため、利用が推奨されています。SPURS ジョブを正しく埋め込むためには、必要なバイナリ仕様に適合させる必要があります。

バイナリ仕様の詳細は、「libspurs 概要」の、「ジョブバイナリの仕様」を参照してください。

埋め込みの要件

SPU ELF ファイルはストリップされていない状態でリンカに提供しなければなりません。リンカはシンボル テーブルとセクション ヘッダー テーブルの情報を使って埋め込みを行っている SPU ELF ファイルの種類を特定するためです。内部埋め込みを実行しているときは、リンカによって SPU ELF から不要なデータがすべてストリップされます。

メイン プログラムから埋め込まれた SPU ELF にアクセスするには

リンカは SPU イメージを参照する開始シンボルと終了シンボルと、SPU イメージのサイズをバイト数で表すサイズ シンボルを生成します。作成されたメタデータについては付加的なシンボルが生成されます。C++ シンボル名のマングル化を行うために、上記のようなシンボルを C++ ソース ファイルで宣言する際は、`extern "C"` を必ず使用してください。以下の表に、生成されるシンボルをまとめます。

シンボル名	詳細
`_binary_<program_type>_<filename>_start`	SPU プログラムの開始アドレス。
`_binary_<program_type>_<filename>_end`	SPU プログラムの終了アドレス。
`_binary_<program_type>_<filename>_size`	埋め込み SPU プログラム バイナリのサイズ。
`_binary_<program_type>_<filename>_ovlytable`	libovis 使用時に必要となる、スタティックに生成されたオーバーレイ テーブルの参照。ELF オーバーレイ テーブルをダイナミックに生成するプログラムによって `cellOvisInitializeOverlayTable()` をコールする必要はありません。
`_binary_<program_type>_<filename>_taskbininfo`	`CellSpursTaskBinInfo` のインスタンスを参照します。`CellSpursTaskBinInfo`は、SPURS タスクを埋め込むと、ELF ファイルのインスタンス用に値が正しく初期化された状態で自動的に生成されます。「CellSpursTaskBinInfo」を参照してください。

	libspurs の詳細は、「libspurs タスク リファレンス」を参照してください。
binary<program_type>_<filename>_jobheader	埋め込み ELF に対して設定された binaryInfo メンバと JobType メンバを持つ CellSpursJobHeader のインスタンスを参照します。

<program_type> は SPU ELF タイプによって決定され、次のいずれかの値をとります。

埋め込みタイプ	プログラム タイプ
未知の実行ファイル	spu
ポリシー モジュール	pm
Spurs タスク	task
SPURS Job 2.0	job
SPURS ジョブキュー	jqjob
共有ライブラリ	so

<filename> は、コマンド ライン上で指定された SPU ELF ファイル名を表し、C 識別子として無効な文字がアンダースコアで置き換えられます。外部ツールの spu_elf-to-ppu_obj と同様に、ファイルの拡張子はバイナリ形式の場合 bin に、SPURS ジョブの場合 jobbin2 に変更されます。

例

以下のコード スニペットは、メインの PPU プログラムから埋め込み SPU ELF データを使用したところを示したものです。埋め込み SPU ELF データへのアクセス方法を示す詳細例は、PS3 SDK Samples $CELL_SDK/samples/sdk/spu_library/… で確認することができます。

```
// 標準的な SPU 埋め込み実行ファイルの使用例
extern const void * _binary_spu_spu_elf_start;
sys_spu_image_import (&spu_img, _binary_spu_spu_elf_start, SYS_SPU_IMAGE_DIRECT);
```

```
// リンカにより生成された CellSpursTaskBinInfo の使用例
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");
```

```
// リンカにより生成された CellSpursJobHeader の使用例
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;
}
```

```
// 共有ライブラリの使用例
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);
```

バイナリ フォーマットを使用した SPU ELF ファイルの埋め込み

デフォルトでは、実行可能な SPU ELF ファイルは ELF 形式で埋め込まれていますが、データを生のバイナリ形式で埋め込まねばならない場合もあります。たとえば、カスタマイズされた SPURS ポリシー モジュールを実行するためには作業ユニットをバイナリ形式とする必要がある場合がありますが、この作業ユニットは、使用すべき埋め込み形式を特定するためにリンカが使用できる、識別機能を持ちません。このため、そのような場合には、`--embed-format=binary` オプションを明示的に指定することにより、埋め込み形式を変更してください。このオプションの右側で指定するすべての実行可能な SPU ELF ファイルはバイナリ形式で埋め込まれます。

次の例では、`unit1.elf` と `unit3.elf` は ELF 形式で埋め込まれますが、`unit2.elf` はバイナリ形式で埋め込まれます：

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

SPU ELF 埋め込みシンボルの表示

リンク中は、埋め込みデータをポイントする生成されたシンボルを `--print-embedded-symbols` オプションを指定することで入手できます。このオプションでは、識別された SPU ELF タイプ、埋め込み方法、生成された付加的なメタデータも表示されます。以下の例は、SPURS タスク ELF の埋め込み情報を示すものです。

```
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: リンク後処理のステップ

PS3 でのプログラム実行を可能にするには、以下の 2 つのリンク後処理を実行する必要があります。

- ppu-lv2-prx-fixup --stub-fix-only
- make_fself または make_fself_npdrm

リンクの実行に GCC を使用すると、2 つの処理のうち最初の 1 つがコンパイラ ドライバによって実行され (つまりユーザーにはこの処理が見えない)、make_fself は手動で実行する必要があります。

SN Linker では、この両ステップをリンキング プロセスの一部として実行できます。ELF ファイルの後処理の必要性、また、make_fself の場合は完全なコピー作成の必要性をなくすことにより、ディスク I/O の量を大幅に削減します。結果として、開発の反復時間が大幅に削減されます (当社テストでは平均 30% の削減)。

PRX の自動修正

有効な PS3 プログラムを作成するために必要な「ppu-lv2-prx-fixup --stub-fix-only」が、リンカに実装されています。これにより、リンカの ELF ファイルの後処理を行うプログラムによって実行されねばならない、付加的なディスク アクセスを避けられるため、リンク処理にかかる時間が改善されます。この処理を制御するスイッチがいくつか用意されています。

スイッチ	詳細
`--prx-fixup`	PRX 修正ステップを実行する (デフォルト)。リンカに PRX fixup 処理の実行を指示する。`--gnu-mode` が指定されない限りはこれがデフォルト。
`--no-prx-fixup`	PRX 修正ステップを実行しない。これにより、自動「PRX スタブ修正」段階が無効化される。`--gnu-mode` が指定されている場合、これがデフォルト。
`--external-prx-fixup`	内部メカニズムではなく、外部 「ppu-lv2-prx-fixup --stub-fix-only」ツールが使用される。(デフォルトでは、パフォーマンス向上のため、PRX 修正を内部で実行する。)

FSELF の作成

最高のパフォーマンスを得るためには、`--oformat=fself` スイッチを使用します (コンパイラ ドライバの使用時にはこれがデフォルトです)。

スイッチ	詳細
`--compress-output`	FSELF 出力を圧縮する。`--oformat=fself` または `--oformat=fself_npdrm` と共に使用しなければならない。
`--oformat=elf`	ELF 出力を作成する (デフォルト)。
`--oformat=fself`	SELF (署名付き ELF) 出力を作成する。
`--oformat=fself_npdrm`	ネットワーク SELF 出力を作成する。
`--write-fself-digest`	SELF ヘッダーに SHA-1 要約を作成する。このスイッチを使用するとリンク時間が長くなる。(デフォルトでは無効で、`--oformat=fself` または `--oformat=fself_npdrm` が使用され

	る場合にのみ有効となる。)

12: トラブルシューティング

リンク時のパフォーマンスを向上させる

リンカには、リンク時のパフォーマンスに影響を及ぼす恐れのあるオプションの機能がたくさん含まれています。リンク時間が長くなるなどの場合には、次の機能を無効化してみてください。

デッド ストリッピングと重複除外

デッド ストリッピングと重複除外はともに、リンク時のパフォーマンスに悪影響を及ぼすことがあります。これらの機能が必要ないときには、これを無効化することで、リンク時間が大幅に改善できます。

詳細は、「トレードオフ」を参照してください。

マップ ファイル生成

リンカには、バイナリ出力をまとめる、テキストベースのマップ ファイルを生成するオプションがあります。マップ ファイル生成は、リンク時のパフォーマンスに悪影響を及ぼす恐れがあります。このため、このオプションの使用は必要なときのみに限定するようにしてください。

ちなみに、`--map` オプションおよび `--sn-full-map` オプションは、リンク時間を平均 30% 延長させます。

オブジェクト モジュール ローディングの対応

OML (オブジェクト モジュール ローディング) のサポート機能は、ターゲット上のメモリ内のバイナリ イメージを更新するとき、OML システムが使用するメタデータをバイナリ出力ファイルに追加します。このデータの生成により、リンク時間が平均で 5 ～ 10 % 長くなります。このため、OML をアクティブに使用していない場合には、`--oml` の削除を検討してみてください。

エラーと警告

リンカのエラーコードおよびその説明は、`--show-messages` コマンドライン スイッチをお使いください。

「L0065 Another location found for ...」警告

この警告は、リンクするセクションに対して、リンカ スクリプト内に可能な場所が複数見つかった場合に生成されます。リンカ スクリプトを編集するか、`--sn-first` または `--sn-best` のコマンド ライン スイッチを使ってこの問題に対処してください。

詳細は、「リンカ スクリプト内でのセクションの場所の不確定性を解決する」を参照してください。

「L0280 Definition of symbol ... overrides definition from ...」警告

この警告は、1 シンボルに対する多重定義がリンカで検出されたものの、その定義の 1 つが他よりも高い優先度を保持していることを示します。リンカでは、優先度の高いシンボル定義を使用してリンクが継続されます。

ELF 仕様に従うと、多重定義はエラーとなります。ただし、SN Linker では 1 つの定義が明らかに優先されるという特定の状況において、多重定義が許可されます。

詳細は、「シンボルの上書き」を参照してください。

リンカ スクリプト内でのセクションの場所の不確定性を解決する

不確定性は、リンカ スクリプト内に、リンク済みのいずれかのオブジェクトからのセクションに対して複数の場所が可能である場合に発生します。そのような場合、次のスタイルの警告が生成されます。

```
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
```

この場合、ファイル o1.o からのテキストをどこに置くかについて不確定性が生じます。リンカ スイッチの --sn-first と --sn-best を使用すると、リンカ スクリプトをどのように解釈するか制御できます。--sn-first は、リンカ スクリプト内で最初にマッチした場所を使用します。--sn-best は、そのセクションに対する最適な場所にマッチし、警告を生成せずにこれを行います。デフォルト動作では最適な場所にマッチします。この場所がスクリプト内の最初の場所でない場合は、警告が生成されます。

シンボルの上書き

一般的に、多重定義は ELF 仕様で定められるところにより、エラー L0019 の原因となります。ただし、SN リンカでは、1 つの定義がオブジェクト ファイル内で検出され、別の定義がアーカイブで検出される場合において、多重定義が許可されます。この場合、オブジェクト ファイルからのシンボルに、より高い優先度が与えられ、エラーは発生しません。ただし、警告 L0280 が発行されます。例：

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

この振舞いを利用すれば、標準ライブラリからの関数を置換するなど、アーカイブからのシンボルを選択的に上書きできます。たとえば、標準ライブラリ版の malloc と free を上書きできます。

> **注意**： 標準ライブラリからの関数を置換するには、注意が必要です。上書きされる関数への全コール (標準ライブラリ内からのものでさえも) がリダイレクトされるため、結果が予測不可能となる場合があります。特に、free を置換せずに malloc を置換すると、メモリ割り当て問題が発生する可能性が高くなります。

多重定義がオブジェクト ファイル内で検出された場合、またはアーカイブ内にのみ検出された場合、エラー L0019 が発生します。

13: インデックス