GNU コンパイラと
Nios II Software Build Tool

ver.11

2012 年 4 月 Rev.1

GNU コンパイラと Nios II Software Build Tool

目次

1. はじめに

この資料は、Nios® II Software Build Tool（ver 9.1 より導入）を使用する際、そのベースとなる GNU コンパイラやリンカの初歩的な概念を体系的に説明した資料です。Nios II 以外の CPU アーキテクチャで GNU コンパイラ未経験のソフトウェア設計者や、同ツールを使用して組込みソフトウェアを開発するハードウェア設計者（ALTERA® Quartus® II ユーザ）に有用な内容です。4 章以降は、ハードウェア設計者から見て高度な内容も含まれていますが、参考レベルに留めてください。

対象バージョン: Quartus II v11.1

Nios II Software Build Tool v11.1

Nios II v11.1

2. 組込みソフトウェアの開発フロー

最初に、組込みソフトウェアの開発フローの概略を紹介します。

① ソフトウェア・プログラムの作成

ソフトウェア開発環境やテキスト・エディタを使用して、プログラミング言語 (C/C++) を記述します。

② C/C++ コンパイル

C/C++ コンパイラが、プログラミング言語をアセンブリ言語に変換します。Nios II Software Build Tool では、GNU ベースの C/C++ クロス・コンパイラ (GCC) を使用します。

③ アセンブル

アセンブラが、アセンブリ言語をオブジェクト・ファイルに変換します。

※ FPGA 開発フローの Quartus II コンパイル・フローのアセンブラとは異なり、FPGA / CPLD の書込みファイル (.sof / .pof) の生成は行いません。

④ リンク

リンカが、ライブラリやオブジェクト・ファイルの情報を集約・結合して、最終的に実行ファイル（ELF ファイル）を生成します。Nios II Software Build Tool では、ELF ファイルは、拡張子 .elf で表記されます。

⑤ ダウンロード・ファイル（ROM 化ファイル）の生成

GNU バイナリ・ユーティリティが、ELF ファイルをモトローラ S レコード・フォーマットに変換します。

Nios II Software Build Tool では、S レコード・ファイルは、拡張子 .flash で表記します。

⑥ ダウンロードおよび実機動作

Nios II Software Build Tool が、JTAG ポート (JTAG_UART ペリフェラル) を経由して

S レコード・ファイルを RAM 上に格納して、プログラムを実行します。FPGA 開発フローでは、SOF ファイルの書込み・デバッグに相当します。

⑦ ROM 化および書込み

Nios II Software Build Tool の Flash Programmer が、JTAG ポートを経由して S レコード・ファイルを ROM に書込みます。FPGA 開発フローでは、POF ファイルの書込みに相当します。

3. GNU コンパイラと make ユーティリティ

3-1. GNU コンパイル・フローと GCC

GCC は、GNU Compiler Collection の先頭 3 文字から取った「GNU の C コンパイラ」を意味する略語です。組込み用途のクロス・コンパイラは様々なコンパイラが存在しますが、昨今最も普及しているのが GCC です。図 3-1 は、前章のコンパイル・フローを GNU ベースに少し踏み込んだ内容です。

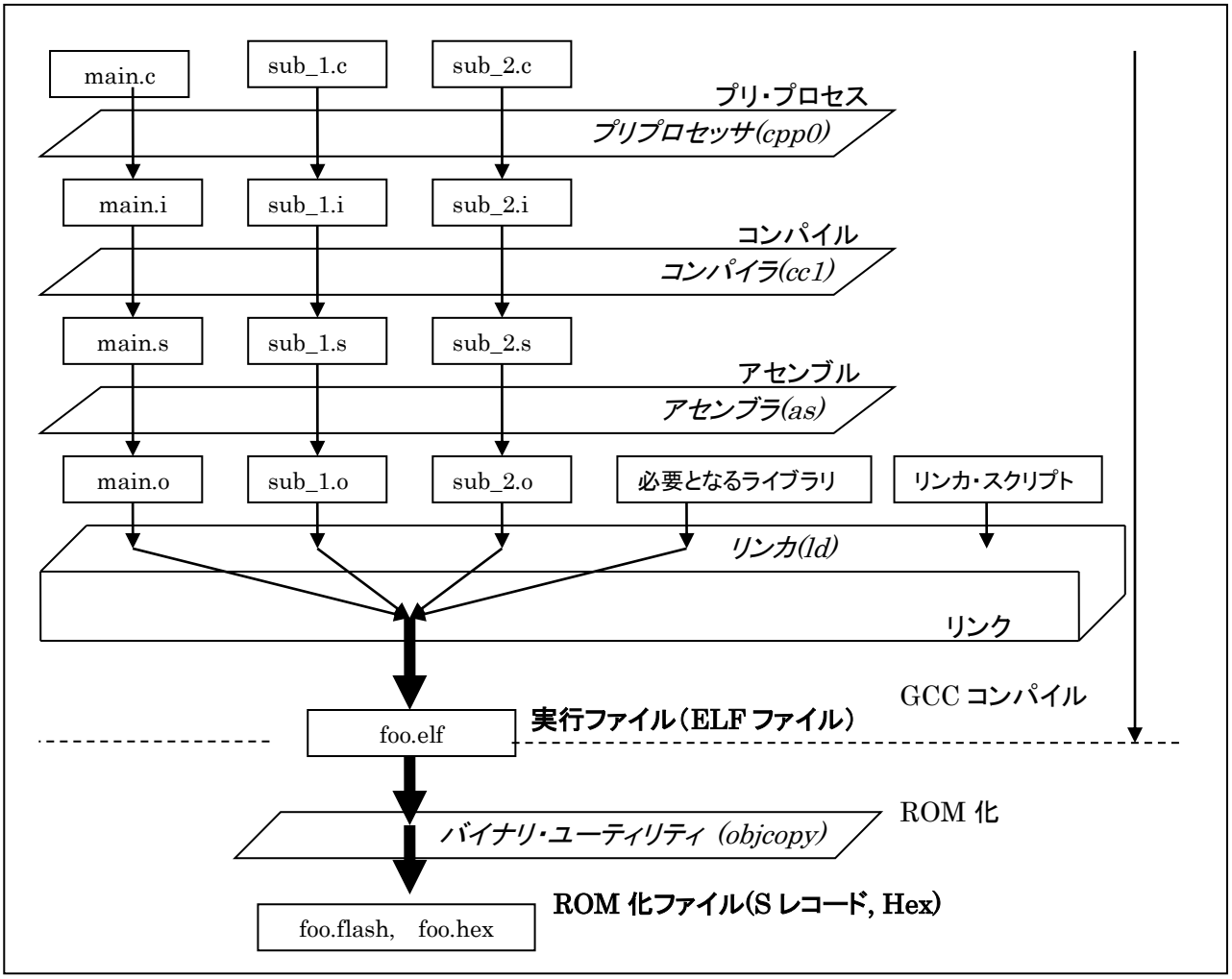

図 3-1. GCC コンパイル・フローと ROM 化フロー

GNU のコンパイラでも、Quartus II と同様にコンパイル・フロー順(プリプロセッサ cpp0 → コンパイラ cc1 → アセンブラ as → リンカ ld) に実行されます。GCC では、コマンドラインから gcc コマンドとソース・ファイル名を指定してタイプすれば、フル・コンパイルが実施されて、実行ファイルが生成されます。Nios II Software Build Tool では、図 3-2 のように nios2-elf-gcc コマンドとして提供されます。

```
[NiosII EDS]$  nios2-elf-gcc   foo.c   -v
```

図 3-2.　gcc コマンドの例①

gcc コマンドでフル・コンパイルを行うと、cc1 や as が背後で実行されていることがメッセージから確認できますが、図 3-3 のようなオプションを指定すれば、段階を踏んだコンパイルもできます。

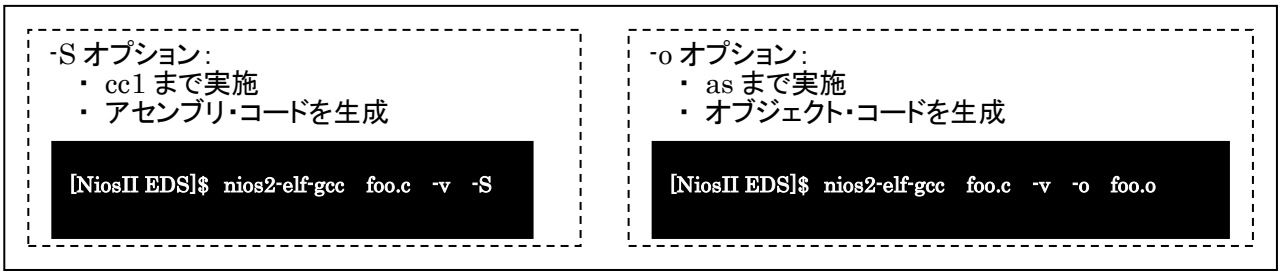

図 3-3.　gcc コマンドの例②

このように gcc コマンドは、フル・コンパイルする為の指令でもあるので、GCC は、C コンパイラ cc1 と区別して、「コンパイラ・ドライバ」と呼ばれることもあります。実際には図 3-3 のような単純なものでは無く、様々なオプションや関連した設定項目が追加されます。（例:　-g オプション、-I オプションなど）

3-2. make ユーティリティ

これらの gcc に対する複雑な設定項目は、プロジェクト毎に設定ファイルを用意して、所定フォームで記録・変更するような仕組みが利用できれば便利です。GNU では、GNU make (gmake) というユーティリティが用意されており、コンパイル・オプション、ファイル名、PATH 名など様々な設定をプロジェクト毎に管理できます。（プロジェクト毎の設定という意味では、Quartus II の QSF ファイル (Quartus II 設定ファイル) と部分的に似ています。）

【 Nios II Software Build Tool の make ファイル 】

Nios II Software Build Tool は、プロジェクト毎に make ファイルを自動生成します。図 3-4 は、make ファイルの一部を抜粋したものですが、ゼロから make ファイルを作り上げるのは現実的ではありません。（精通した方を除く。）カスタマイズする際には、自動生成ファイルをベースにモディファイすることを推奨します。

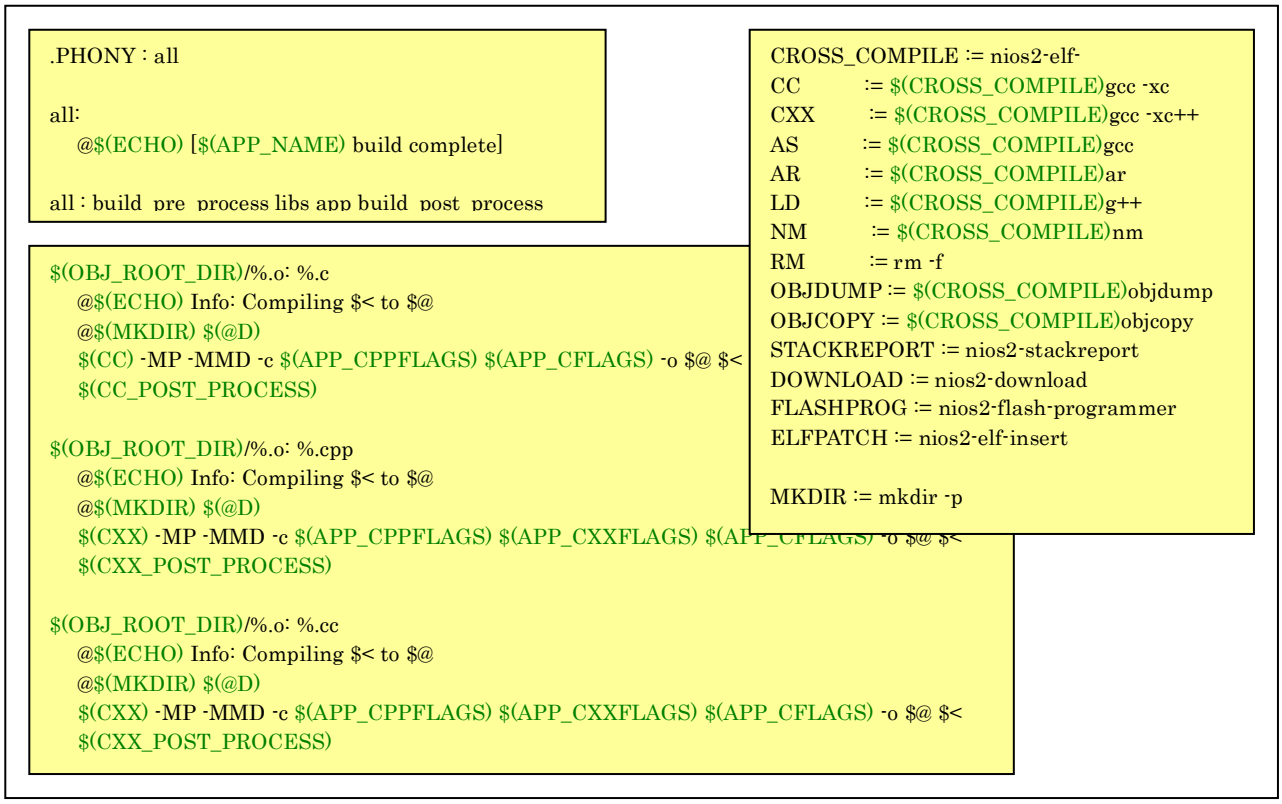

図 3-4.　Nios II Software Build Tool が自動生成した make ファイルの一部抜粋

Nios II Software Build Tool でプロジェクトを作成すると make ファイルが自動生成されます。コマンドラインのカレント・ディレクトリを、この make ファイルがあるディレクトリに移動して、図 3-5 のように make もしくは make all とタイプすると、GNU make ユーティリティが gcc コマンドを実行して、ELF ファイルを生成します。このビルド（コンパイルからリンクまでの処理）の過程は、表示されたメッセージを追うことで確認できます。

```
[NiosII EDS]$  make all
```

図 3-5.　make のタイプ例①

make ユーティリティでは、図 3-6 のようにビルド後に make clean とタイプすると、ディレクトリの状態を、ビルド前の状態に戻すことができます。（オブジェクト・ファイルが消えます。）

```
[NiosII EDS]$  make clean
```

図 3-6.　make のタイプ例②

make ユーティリティは、複数のファイルをまとめてコンパイルするような開発の場合、修正されたファイルのタイム・スタンプを見て、その修正ファイルのみを再コンパイルして、既存のオブジェクト・ファイルとのリンクを行いますので、ソース・コードとオブジェクト・コードの整合を保つ役割を持っています。この意味では、Quartus II の Smart Compilation やインクリメンタル・コンパイルに似ているかも知れません。

3-3. GCC の最適化レベル

GCC のコマンド・オプションの中の最適化オプションを以下に紹介します。Quartus II と同様に、GCC でもサイズとパフォーマンスを考慮して、複数の最適化レベルが用意されています。

◇ -O :-O1 と同じ

◇ −O0 :最適化レベル 0

最適化は行いません。GCC はデバッグしやすいコードを生成します。

◇ -O1 :最適化レベル 1

コンパイルされたコード・サイズと、実行時間の両方の削減をはかります。コンパイル時間は、-O0 の場合よりもかかります。メモリ・アクセスを減らす為に CPU 内蔵の汎用レジスタに値を保存するような最適化が行われます。

◇ -O2 :最適化レベル 2

-O1 よりも、さらに最適化を行います。ほとんど全ての最適化を行います。コンパイル時間は、より長くなり、生成されたコードは概ね高速化します。

◇ -O3 :最適化レベル 3

-O2 で指定される全ての最適化を行い、かつループ展開やインライン展開も実施します。

◇ -Os :最適化レベル s

-O2 で指定される最適化の中で、実行時間よりもコード・サイズの削減を優先した最適化が行われます。

3-4. バイナリ・ユーティリティ

オブジェクト・ファイルと実行ファイルを操作するユーティリティであり、Binutils (GNU Binary Utilities) とも呼ばれています。代表的なものを以下に紹介します。

◇ objcopy : 代表的な使用用途としては、ROM 化の際のフォーマット変換があります。

Nios II Software Build Tool では、図 3-7 のように nios2-elf-objcopy コマンドとして提供されます。

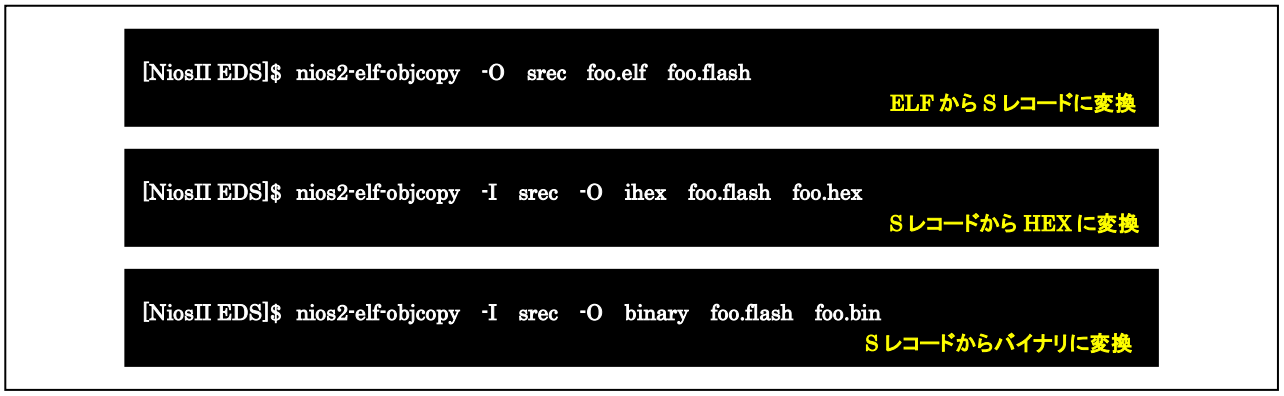

図 3-7. バイナリ・ユーティリティ objcopy のタイプ例

◇ nm : オブジェクト・ファイルのシンボル・テーブルを表示します。Nios II Software Build Tool では、図 3-8 のように nios2-elf-nm コマンドとして提供されます。出力メッセージは、リダイレクションできます。

```
[NiosII EDS]$ nios2-elf-nm foo.elf > foo.nm
```

図 3-8. バイナリ・ユーティリティ nm のタイプ例

※シンボル:

「C 言語で変数や関数を定義する。」ことは、「その実体が、利用可能なメモリ空間上のどこかのアドレスに領域が確保される。」ことを意味します。つまり、変数名や関数名は、確保したメモリ領域名であると言えます。関数や変数は、オブジェクト・ファイル上では、単なる「名前」として扱われます。この「名前」のことを「シンボル」と呼びます。

※シンボル・テーブル:

シンボルを管理する為の「シンボル情報」として、「シンボル名、シンボルのタイプ、シンボルの」実体の配置位置など」の情報が、関数や変数ごとに存在します。オブジェクト・ファイルは、これらの「シンボル情報」を管理する為の「シンボル・テーブル」を配列として持っています。

◇ objdump ： オブジェクト・ファイルの情報を表示します。以下に代表的なオプションを紹介します。

- --disassemble： 逆アセンブルを行います。
- --syms： シンボル情報を表示します。
- --all-header： ELF ファイルのヘッダ情報を表示します。
- --source： 逆アセンブルコードを元ソースと対比させて表示させます。

Nios II Software Build Tool では、図 3-9 のように nios2-elf- objdump コマンドとして提供されます。

```
[NiosII EDS]$ nios2-elf-objdump --disassemble --syms --all-header --source foo.elf > foo.objdump
```

図 3-9. バイナリ・ユーティリティ objdump のタイプ例

◇ size ： サイズ表示。セクションのサイズを表示します。Nios II Software Build Tool では、図 3-10 のように nios2-elf-size コマンドとして提供されます。

```
[NiosII EDS]$ nios2-elf-size foo.elf
   text    data     bss     dec     hex filename
  67200    7612     532   75344   12650 foo.elf
```

図 3-10. バイナリ・ユーティリティ size のタイプ例

◇ readelf ： ELF ファイルの情報（ヘッダ情報やシンボル情報）を表示します。

Nios II Software Build Tool では、図 3-11 のように nios2-elf- readelf コマンドとして提供されます。出力メッセージは、リダイレクションできます。

```
[NiosII EDS]$ nios2-elf-readelf --all foo.elf > foo.readelf.txt
```

図 3-11. バイナリ・ユーティリティ readelf のタイプ例

4. リンカの役割とリンカ・スクリプト

コンパイラやアセンブラから受け取った複数のオブジェクト・ファイルや、元々ライブラリとして用意されている複数のオブジェクト・ファイルを、リンク（結合）した上で、リンカへの指示書に従って、プログラム・コードやデータをメモリ空間上に配置します。リンカへの指示書をリンカ・スクリプトといいます。図 3-1 の GCC コンパイル・フローでも示していますが、リンカはリンカ・スクリプトを読み込んでリンク作業を行い、最終的に、オブジェクト・ファイルを結合し、未解決のアドレス情報を解決して、実行ファイルを生成します。アドレス解決については、この資料では割愛します。

4-1. データ構造とセクション

C 言語のソフトウェアはソフトウェア設計者が作成したプログラム・コードですが、プログラムを作成中には、メモリ上のどの場所にどのコードを配置するかといった問題は厳密には考えていません。しかし、最後はハードウェアであるメモリのどこかに配置されなければなりません。

C 言語は、「データ構造（データ）」＋「プログラム・コード」で構成されていますが、データ構造は個々に異なった性質を持っているので配置場所はそれぞれ別々に考えます。

4-1-1. データ構造とセクション

図 4-1 は、性質の異なるデータ構造の具体例を簡易的に説明しています。ソース・コード上で記述される変数や関数が、リンカの段階では、どのようなデータ構造に分類されていて、最終的にターゲット・ボードのメモリ・デバイス上に、どのように配置されているかを図示しています。

オブジェクト・ファイルは、用途に応じて性質の異なるデータ領域に分割されています。個々のデータ領域のことを「セクション」といいます。セクションは、“.bss” や “.text” というように、セクション名の先頭に “.”（ドット）を付けて表現します。

図 4-1 では、ソース・コードで初期値を指定した変数や配列データは、.rwdata セクションとして分類されます。変数では無く定数を初期値に指定したデータは、リード専用データの .rodata として分類されます。ソース・コード上の初期値を持たない配列データは .bss セクションに分類されます。ソース・コード上の main 関数は.text セクションとして分類されます。.text セクションはプログラム・コードそのものなので、ROM のような不揮発性メモリに常駐させます。 .rodata セクションのリード専用データも、不揮発性メモリに配置できます。

.rwdata セクションと .bss セクションは RAM のような書き換え可能な揮発性メモリ内に配置するのが一般的です。「初期値の有無」つまり「実データの有無」に着目すると、.rwdata セクションと .bss セクションは別々の性質を持つものと見なせるので、両者は RAM 上の別々の領域に配置する必要があります。

リンカ・スクリプトは、このようなデータ構造の違いを考慮して、セクションの配置場所を指定するスクリプト・ファイルです。

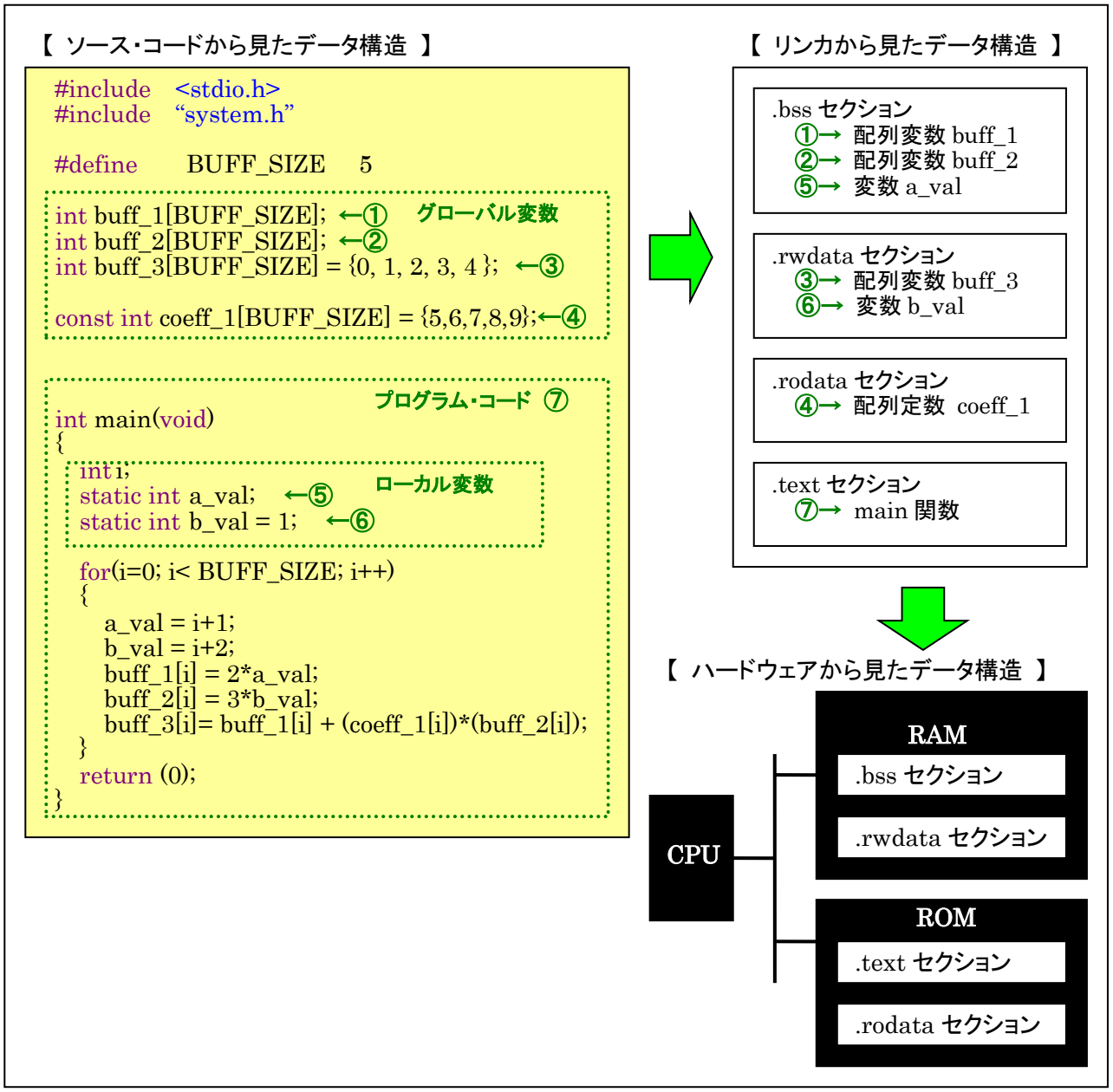

図 4-1. リンカがプログラムをメモリ上に配置している様子

【 基本となるデータ領域 】

C 言語のプログラムでは、データがどのメモリ領域に確保されるかが重要です。大雑把には以下の 3 つの領域に分類されます。

◇スタック領域: ローカル変数、auto 変数、関数の引数などを確保します。

◇静的記憶領域: グローバル変数などの static 変数を確保します。

◇動的記憶領域: 動的に確保される領域として、ヒープ領域などを確保します。

実行ファイルには ELF 以外にも様々な形式（a.out など）があります。それらに共通するベーシックな 3 つのデータ構造を以下に紹介します。

◇テキスト領域： プログラム・コードが配置されます。リード専用データなのでライト禁止です。

◇データ領域： 初期値を持つ変数の本体が配置されます。ライト可能なデータです。グローバル変数や static 変数が配置対象となり、auto 変数は対象外です。

◇BSS 領域： BSS とは、Block Started by Symbol の略語であり、初期値を持たない変数が配置されます。auto 変数は配置の対象外です。

【 ELF ファイルを出力する GNU ツールのリンカ・セクション 】

ELF ファイルを出力する GNU ツールで定義される代表的なセクションを紹介します。

◇ .text セクション： テキスト領域。通常 ROM に配置。
リンカ・スクリプトで、このセクションを外部 RAM に指定した場合、ROM 化の際に Nios II Flash Programmer がブートコピア（ALTERA 提供の専用ブートローダ）の付加を自動的に行い、リセット後（ROM 書込み後）ブートコピアが起動して、RAM 上の .text セクションにコピーします。

◇ .data セクション： データ領域。初期値を持つグローバル変数を RAM 上に配置。
初期値を持たないグローバル変数は、.data セクションもしくは .common セクションのどちらかに配置されます。（コンパイラ依存。）

◇ .bss セクション： BSS 領域。初期値を持たない変数を RAM に配置。
その他、関数内の構造体、動 的記憶領域に配置されるヒープ領域や malloc() 等が配置されます。
注： malloc() ・・・ 引数で指定されたサイズ（バイト数）の領域を動的に確保します。この領域がヒープ領域内に割り当てられるかは、コンパイラ依存。free() で領域を開放しないとメモリの空き容量が食いつぶされてしまいますので注意が必要です。

◇ .rodata セクション： リード専用のデータが配置されます。const 宣言したデータ、文字列リテラルなどが主に配置されます。

◇ .common セクション： 初期値を持たないグローバル変数が RAM に配置されます。

◇ .stub セクション、.stubstr セクション： GDB によるデバッグ情報が配置されます。プログラム・コードの情報や行番号といった情報が集約されています。GDB とは、GNU のデバッガを指します。

◇ .comment セクション： ファイル名や時刻情報が配置されます。

◇ .rwdata セクション： リード・ライト可能なデータを RAM に配置します。Nios II Software Build Tool では、値 0 は初期値と見なします。

◇ .entry セクション： Nios II Software Build Tool では、リセット・ハンドラを自動的に配置します。

◇ .exceptions セクション： Nios II Software Build Tool では、例外ハンドラを自動的に配置します。

◇ .(メモリ・デバイス名)セクション： Nios II Software Build Tool が、system.h で定義（おおもとは、SOPC Builder のコンポーネント名）されたメモリ・デバイス名に関連したセクションを自動的に定義します。

例： .ext_flash, .ext_ssram, .ddr_sdram_0

◇ ユーザ定義のカスタム・セクション： Nios II Software Build Tool のサブ・ツール BSP Editor を使用して GUI 上から任意に定義できます。

4-1-2. ツール自動生成のリンカ・スクリプト linker.x

Nios II Software Build Tool では、linker.x というファイル名のリンカ・スクリプトがデフォルトで生成されます。（後述の bsp プロジェクト・フォルダ内に生成されます。） この linker.x は、概ね、次のような記述で構成されています。

MEMORY コマンド

ターゲットのメモリブロックの位置と大きさを定義して、リンカがどのメモリ領域を使って良いか、どの領域を使ってはいけないかを記述できます。メモリブロックは必要な数だけ定義できます。 同様の設定は、Nios II Software Build Tool の BSP Editor より GUI でも行うことができます。

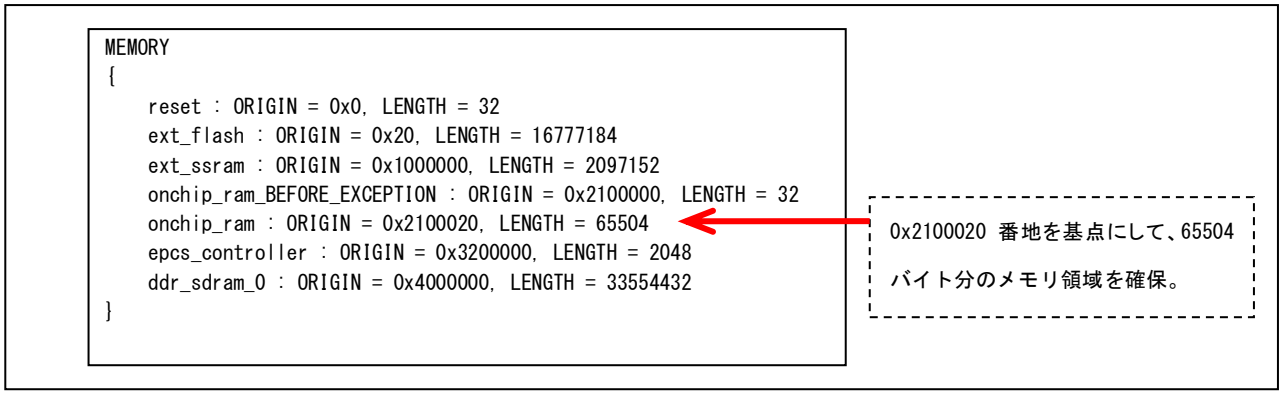

図 4-2. MEMORY コマンド

OUTPUT_FORMAT コマンド

出力フォーマットやエンディアンを指定します。

```
OUTPUT_FORMAT( "elf32-littlenios2",
               "elf32-littlenios2",
               "elf32-littlenios2" )
```

図 4-3. OUTPUT_FORMAT コマンド

OUTPUT_ARCH コマンド

ターゲット・プロセッサのアーキテクチャーを指定します。

```
OUTPUT_ARCH( nios2 )
```

図 4-4.　OUTPUT_ARCH コマンド

ENTRY コマンド

エントリポイントを設定します。エントリポイントとは、プログラムで最初に実行される命令へのポインタ（もしくは、開始アドレス）です。

```
ENTRY( _start )
```

図 4-5.　ENTRY コマンド

SECTIONS コマンド

リンカ・セクションやシンボルを、メモリ空間に配置します。

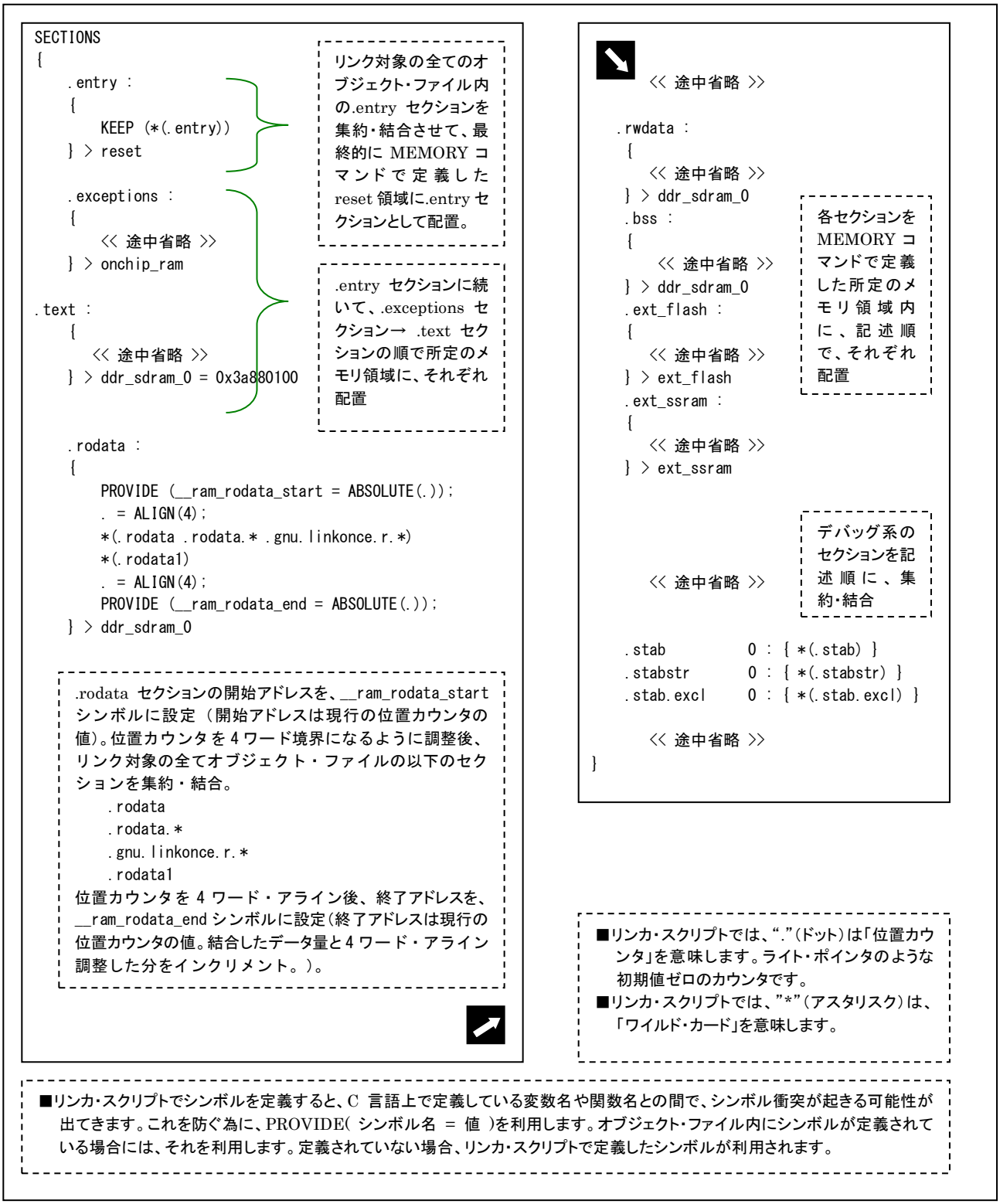

図 4-6. SECTIONS コマンド

4-2. その他

アトリビュート（attribute）

リンカ・スクリプトを使用しないで、ソース・コードから直接、セクションを指定することもできます。図 4-7 のように、GCC で提供されている関数の "__attribute__" 機能を使用して、特定の変数や関数を任意のセクションに配置することが可能です。ANSI C では #pragma がありますが、attribute 機能の使用を強く推奨します。

```
int  foo_var __attribute__ ((section (".data_tcm"))) = 0;
/* 変数foo_varを .data_tcm セクションに配置。*/

void bar_func(void* ptr) __attribute__((section (".ext_ssram")));
/* 関数bar_func()を.ext_ssram セクションに配置。*/
```

図 4-7.　attribute の使用例

※#pragma：（プラグマ）

プリプロセッサ命令のひとつ。attribute 機能と同様にデータのメモリ領域への配置をソース・コード上から明示できますが、コンパイラ固有の拡張機能もあり、ソフトウェアを他の CPU アーキテクチャから移植する場合には互換性に注意が必要です。

マップ・ファイル

リンカが何をどこに配置したのかをテキスト形式でレポートしてくれるファイルであり、リンカ ld が出力します。リンカ・スクリプトで指定通りにメモリ空間上に配置できたかどうか確認するときに利用します。Nios II Software Build Tool では、図 4-8 のように nios2-elf- ld コマンドの -M オプションとして提供されます。出力メッセージは、リダイレクションできます。

```
[NiosII EDS]$ nios2-elf-ld  -M  foo.elf  >  foo.map
```

図 4-8.　マップ・ファイルの生成例

5. BSP (Board Support Package) について

デバイス・ドライバとは、ハードウェア（固有のデバイス）とアプリケーションや OS とを結びつけるインタフェース・ソフトウェアのことを指します。

そして、BSP (Board Support Package) とは、OS メーカや CPU ボード・メーカが提供するソフトウェアパッケージであり、その実体は、CPU ボード上で特定の OS を実行させるために必要なソフトウェア・ライブラリやデバイス・ドライバの集合体です。Nios II Software Build Tool の場合、SOPC Builder システム内のコンポーネント用のデバイス・ドライバや HAL システム・ライブラリに関連するソース・コードやオブジェクトが、BSP プロジェクト内に展開されます。

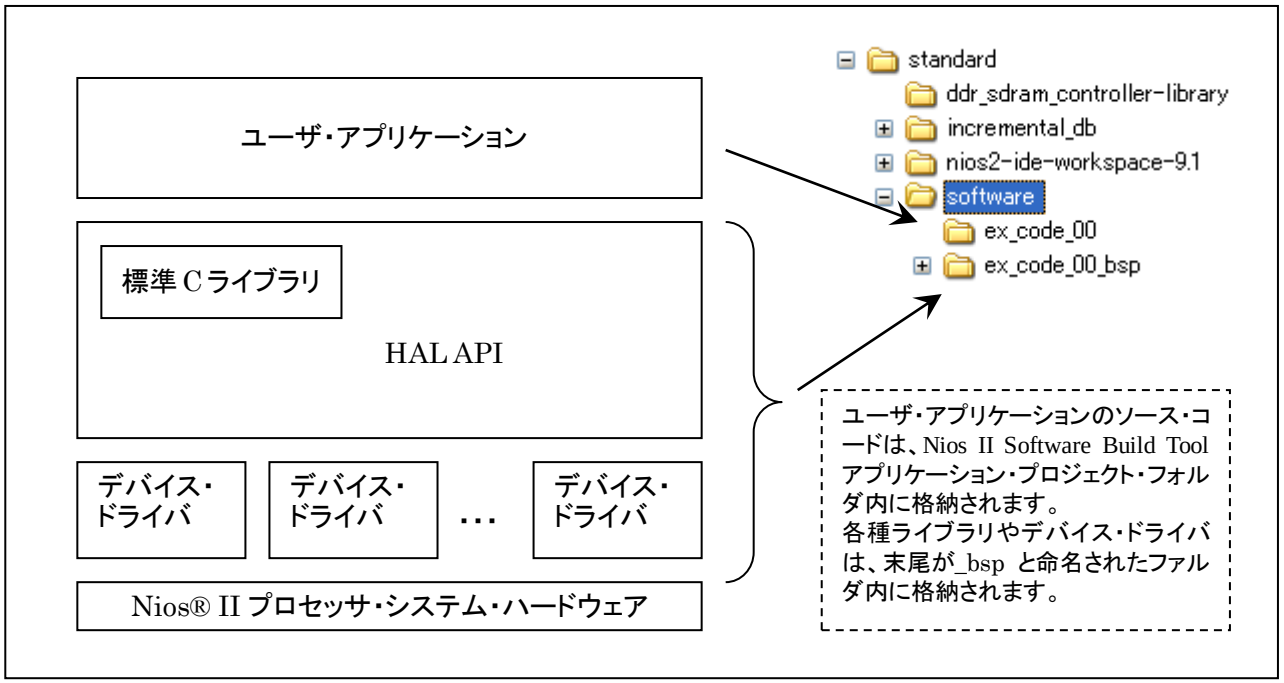

図 5-1.　ソフトウェア階層モデルと、プロジェクト・ディレクトリとの関係

Nios II Software Build Tool では、リンカ・スクリプトや make ファイルの設定の他、最適化オプション等の GCC 関連の各種設定を、Nios II Software Build Tool のサブ・ツール BSP Editor 上から GUI で設定できます。

6. おわりに

ver 9.1 から導入された Nios II Software Build Tool は、従来の Nios II IDE と比べて、GNU やリンカ・スクリプトに対する設定が柔軟になりました。精通されている設計者には有用な GUI ツールですが、GNU やリンカ・スクリプトの概念を考慮せずに使用するとリスクを伴う恐れがあります。この資料で GNU やリンカ・スクリプトの概念を持たれた後、次章で紹介する参考文献や専門サイトを通じて理解を深められます。

尚、GNU では ”AS IS（無保証）” と表記されています。フリーツールなので「不具合は自己解決」という暗黙のルールが存在しますので取り扱いには十分な注意をお願いします。

7. 参考文献

1. 『GNU ソフトウェアプログラミング』 オライリー・ジャパン

   Mike Loukides & Andy Oram 共著　引地美恵子、引地信之 共訳

2. 『実例で学ぶ GCC の本格的活用法』 CQ 出版社

   岸哲夫著

3. 『MAKE の達人』 株式会社トッパン

   C. ドント / A. ネイサンソン / E. ヤント 共著　小川晃夫訳

4. 『リアルタイム組み込み OS 基礎講座』 株式会社翔泳社

   チング・リー / キャロライン・ヤオ 共著　宇野みれ訳

5. 『リンカ・ローダ実践開発テクニック』 CQ 出版社

   坂井 弘亮著

6. 『Interface 2007 年 12 月号』 CQ 出版社

改版履歴

Revision	年月	概要
1	2012年4月	新規作成