Nios II IDE によるソフトウェア開発
セクション 2

ver.1.0

2010 年 8 月

Nios II IDE によるソフトウェア開発 セクション 2

目次

1. はじめに

この資料は、Nios II IDE によるソフトウェア開発について紹介しています。セクション 2 で取り上げている内容は、以下のとおりです。

- ❑ Nios® II ソフトウェア・プロジェクトが必要とするファイル
- ❑ ブート・シーケンス
- ❑ Nios II コード・サイズ
- ❑ 例外処理

2. Nios II ソフトウェア・プロジェクトが必要とする重要なファイル

以下のファイルは Nios II ソフトウェア・プロジェクトを構成する際に重要なファイルです。これらのファイルは、基本的に Nios II IDE にてプロジェクトのビルド時に自動生成されます。

※ ファイルの場所: syslib プロジェクト => Debug / Release => system_description

◆ システム・ヘッダ (“system.h”)

SOPC Builder で生成したシステム内の全ペリフェラルのメモリマップが定義されたファイルです。

◆ リンカ・スクリプト (“generated.x”)

プログラム・セグメントをメモリのどこに配置するかを指定したファイルです。ユーザは System Library Properties にてプログラム・セグメントの設定を行うことができます。

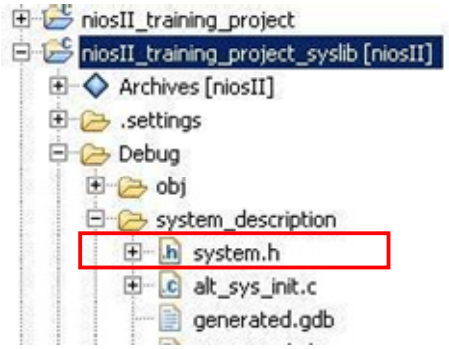
◆ 初期化ファイル (“alt_sys_init.c”)

システムが使用するデバイス・ドライバを初期化するためのソース・ファイルです。

2-1. HAL システム・ヘッダファイル

HAL システム・ライブラリを使用するにあたっての各ペリフェラルの基本情報が定義された“system.h”ファイルについて説明します。

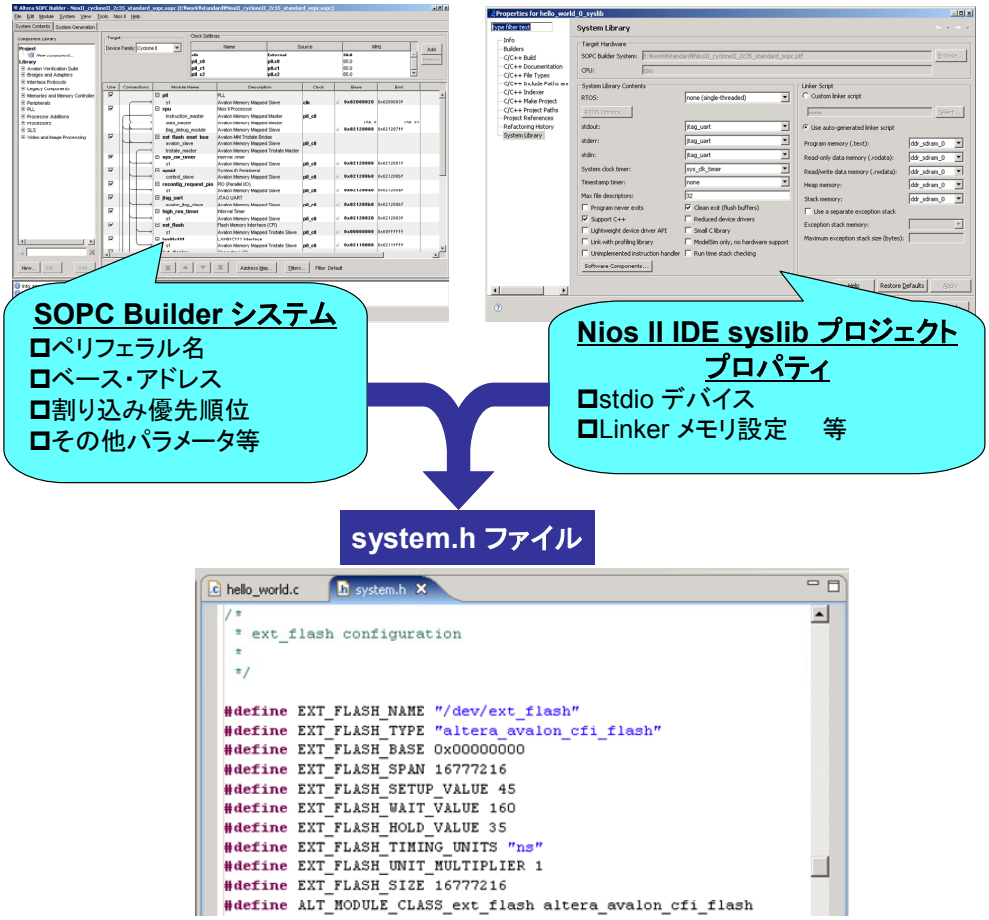
system.h ファイルには SOPC Builder で生成した Nios II や各ペリフェラルのハードウェア情報のすべてが記述されたファイルです。system.h ファイルは HAL システム・ライブラリ用に Nios II IDE によって syslib プロジェクトのビルド時に自動生成されます。



system.h にはシステム内の各ペリフェラルの設定、システム・パラメータのマクロ定義が含まれます。

- ペリフェラル・ハードウェア設定
- ベース・アドレス
- 割り込み番号
- ペリフェラルの名前

関数のプロトタイプ宣言、static 宣言、構造体定義は含みません。system.h ファイルを、アプリケーション・コードで #include し、ファイルの中で定義されているベース・アドレスや割り込み番号をペリフェラル名で表したマクロを使用してアクセスするコードを書くことができます。SOPC Builder でのアドレスマップの変更を行った際などには system.h にも変更が反映されるため、アプリケーションの変更を行うことなく、容易に対応することができます。



“system.h”, “generated.x”, “alt_sys_init.c” それぞれのファイルは SOPC Builder で Generate 時に生成される .ptf ファイルのシステムのハードウェア情報と Nios II IDE でのソフトウェア・プロジェクトの syslib プロジェクトのプロパティで設定された情報が定義されたファイルです。SOPC Builder の System Contents タブで各コンポーネントのスタート・アドレスやモジュール名等、ハードウェアに変更がある場合には下記の方法でソフトウェア・プロジェクトに反映させます。

- SOPC Builder にて Generate ボタンにより .ptf ファイルの再生成
- Nios II IDE で新しい .ptf ファイルに基づいたソフトウェア・プロジェクトを作成
  - Nios II IDE が“system.h”, “generated.x”, “alt_sys_init.c” を再生成

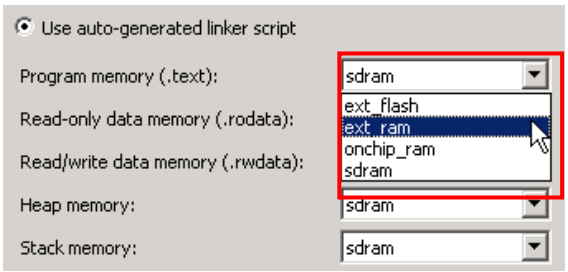
2-2. リンカ・スクリプト

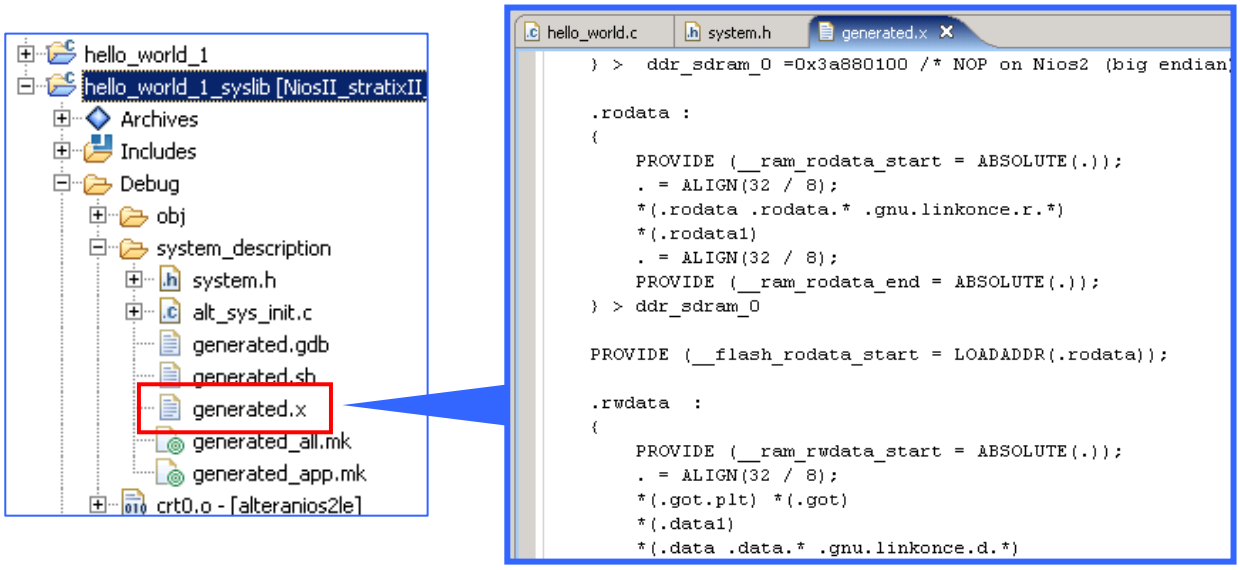
2-2-1. リンカ・スクリプト

Nios II のリンカ・スクリプト “generated.x” は Nios II IDE にてソフトウェア・プロジェクトをビルドした際に自動生成されます。syslib プロジェクトの system_description フォルダに生成されます。

このリンカ・スクリプトは、利用可能なメモリ・セクション内でコードおよびデータのマッピングを制御します。自動生成されたリンカ・スクリプトは、システム内の各物理メモリ・デバイスに対するセクション (.text、.rodata、.rwdata、.bss) を定義します。例えば、system.h ファイルで定義された on_chip_memory という名前のメモリコンポーネントが存在する場合には、 .on_chip_memory という名前のメモリ・セクションが生成されます。

コードとデータを system.h ファイルで定義された物理メモリ・デバイスに配置します。Nios II IDE の syslib ライブラリのプロパティの Linker Script 欄でプログラム・メモリ(.text)、リード・オンリー・データメモリ(.rodata)、リード/ライト・データメモリ(.rwdata) 等を配置するメモリ・デバイスを選択します。メモリ・デバイスは SOPC Builder システムに組み込まれているメモリから選択します。

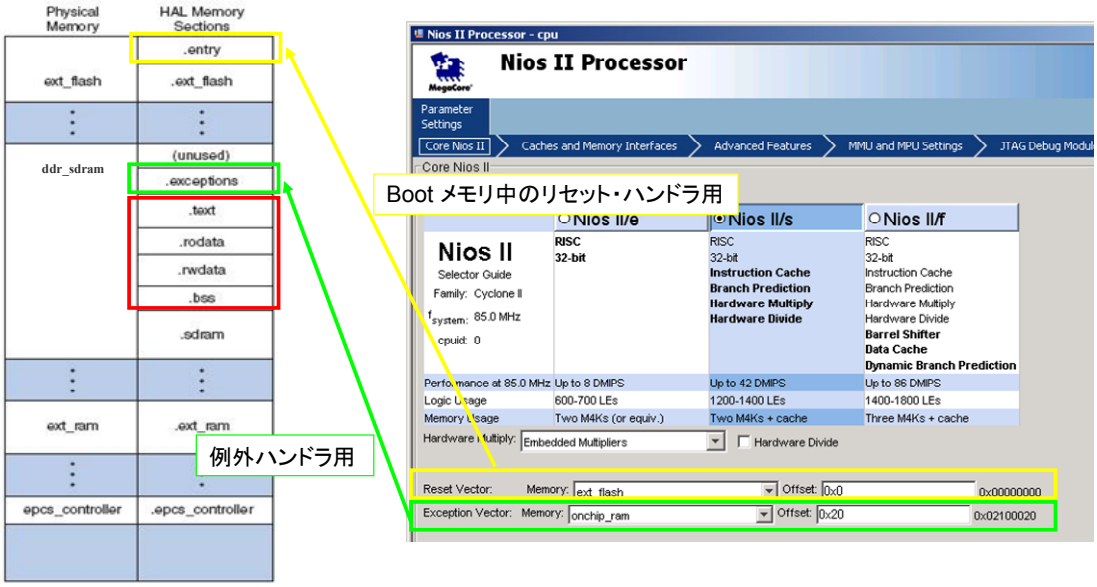




Linker Script で設定した .text 等の領域のほかにリセット・ハンドラ用の予約領域、例外ハンドラ用の予約領域も確保されます。リセット・ハンドラと例外ハンドラは Nios II の設定の Reset Vector、 Exception Vector にて設定されます。

リセット・ベクタは通常フラッシュ・メモリ等の不揮発性のメモリに配置します。

例） リンク・マップ



2-2-2. generated.x ファイル

自動生成されるリンカ・スクリプトの一部を抜粋したものです。.text、 .rodata、 .rwdata、 .bss が下記のように定義されており、それぞれのセクションが SOPC Builder で定義されている各メモリ・デバイスに割り当てられています。“SECTIONS” がメモリ・セクションを示しています。

```
SECTIONS
{
    ・・・<省略>

  .text :
  {
      ・・・<省略>
  } >  ddr_sdram_0 =0x3a880100

  .rodata :
  {
      ・・・<省略>
  } > ddr_sdram_0

  PROVIDE (__flash_rodata_start = LOADADDR(.rodata));

  .rwdata  :
  {
      ・・・<省略>
  } > ddr_sdram_0

  .bss :
  {
      ・・・<省略>
  } > ddr_sdram_0
```

プログラム・メモリ

リード・オンリー・データメモリ

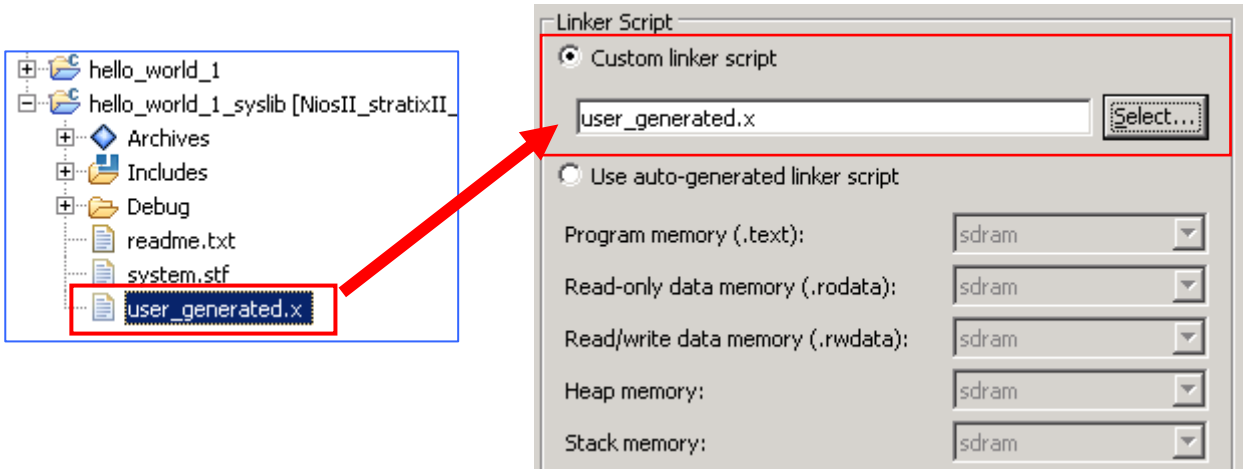
リード・ライト・データメモリ

.bss 領域

2-2-3. リンカ・スクリプトのカスタマイズ

すべてのリンク情報はリンカ・スクリプトによって定義されます。必要であればリンカ・スクリプトを編集することが可能です。その際には Nios II IDE によって自動生成されるリンカ・スクリプトの “generated.x” をカスタマイズして使用します。

まず “generated.x” を syslib プロジェクトフォルダに新しい名前（例 “generated_new.x”）でコピーし編集したものを、syslib プロジェクトのプロパティ設定のページで Custom linker script にチェックを入れて選択し、使用します。



2-2-4. 変数のメモリ配置

新たなメモリ・セクションの生成や各セクションの配置はリンカ・スクリプトで指定することが可能です。ユーザのソース・コード内で __attribute__section 記述により、特定の変数や関数を任意のメモリ領域に配置することが可能です。こちらは GCC コンパイラの機能になります。デフォルトの設定では、変数は .rwdata セクション、関数は .text セクションに配置されます。

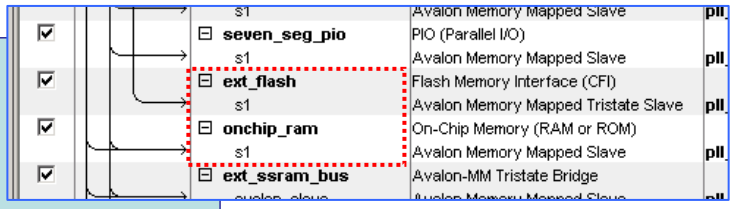
以下のコードは、.on_chip_memory という名前のメモリに変数 foo_ver、 .ext_ram という名前のメモリに関数 bar_func を配置する方法を示します。

例） 変数および関数を物理メモリ・セクション配置する

```
/* Data using the "section" attribute should be initialized */
int  foo_var __attribute__ ((section (".on_chip_memory"))) = 0;
void bar_func(void* ptr) __attribute__((section (".ext_ram")));
```

ユーザのソース・コード内で下記のようにポインタによって、変数アドレスを直接指定することも可能です。それぞれの物理アドレスは system.h 内でマクロで定義されていますので、使用することができます。

```
#include "system.h"

  ・・<省略>

  int *length_mem_ptr;
  char *type_mem_ptr;

  length_mem_ptr = (int)EXT_FLASH_BASE;
  type_mem_ptr = (char)ONCHIP_RAM_BASE;

  ...
```

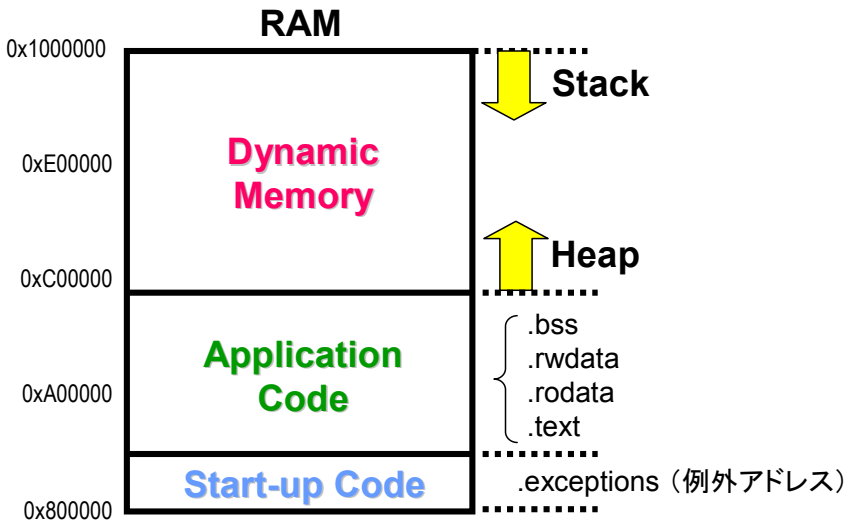


system.h ファイル

```
/*
 * ext_flash configuration
 *
 */

#define EXT_FLASH_NAME "/dev/ext_flash"
#define EXT_FLASH_TYPE "altera_avalon_cfi_flash"
#define EXT_FLASH_BASE 0x00000000
#define EXT_FLASH_SPAN 16777216
#define EXT_FLASH_SETUP_VALUE 45
```

2-2-5. スタックとヒープの配置

スタックとヒープは、デフォルトの設定ですと、.wrdata セクションと同じメモリ・パーティションに配置されます。スタックのベース・アドレスは、メモリの最終アドレスに設定され、下位方向（アドレスの若い方向）に進みます。ヒープ領域はメモリの未使用領域の先頭から上位に向かって伸びていきます。



スタックとヒープのベース・アドレスは、リンカのコマンドラインスイッチで下記コマンドを使用してオーバーライドが可能です。

◆ スタックのオーバーライドコマンド

__alt_stack_pointer

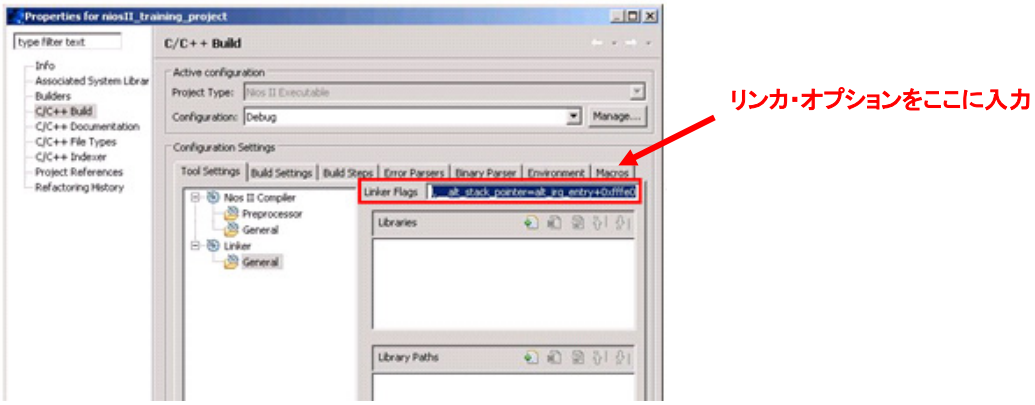
例） -Wl,-defsym –Wl,__alt_stack_pointer=alt_irq_entry+0xfffe0

◆ ヒープのオーバーライドコマンド

__alt_heap_start

これらのコマンドを使用してオーバーライドした情報は .objdump ファイルで確認できます。

（.objdump ファイルについては 4-1. 章 コード・サイズの確認 を参照）



アプリケーション・プロジェクトの C/C++ Build 設定項目の Linker Flags の欄に、リンカ・オプションを入力します。

スタックとヒープの配置をオーバーライドする場合には、プログラムの動作中に。ヒープ領域とスタック領域が使用可能なメモリ容量を超えないよう注意しなければなりません。Nios II IDE のデバッガには上記の自動チェック機能がオプションとして用意されています。（セクション 3 高度なデバッグ機能 参照）

2-3. 初期化ファイル

2-3-1. alt_sys_init.c（自動生成）

alt_sys_init.c ファイルはシステム内のサポート対象デバイスのデバイス・ドライバを初期化するためのコードが含まれています。 syslib プロジェクトの system_description フォルダに生成されます。このファイルの中では alt_sys_init () 関数が定義されています。この関数は main () の前に呼び出されデバイスを初期化し、プログラムからデバイスを使用できる状態にします。

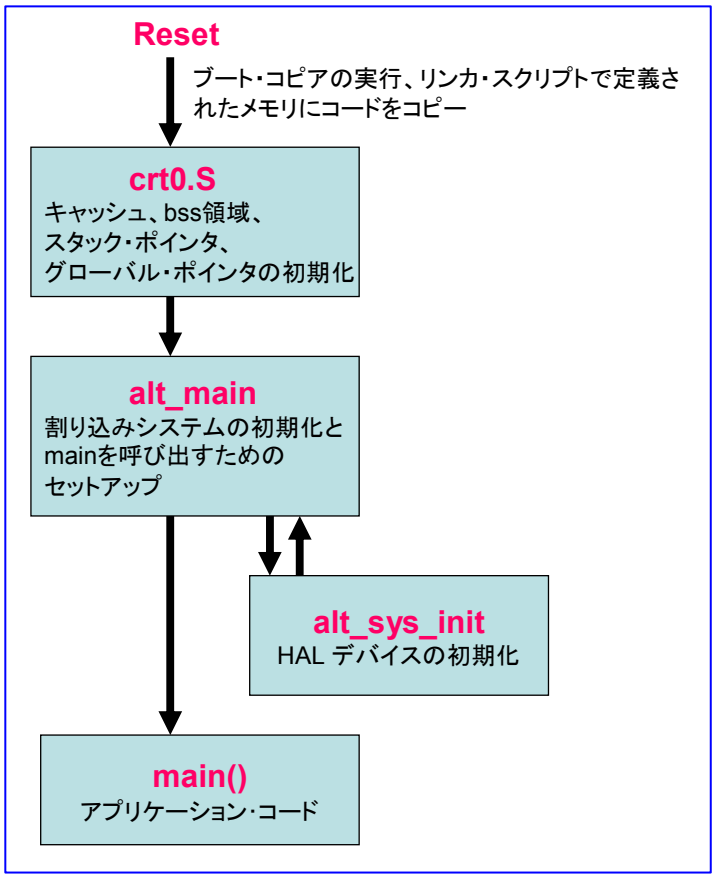
例） alt_sys_init.c 抜粋

```
#include "system.h"
#include "sys/alt_sys_init.h"

/*
 * device headers    デバイス・ドライバ ヘッダ・ファイル
 */

#include "altera_avalon_timer.h"
#include "altera_avalon_sysid.h"

/*
 * Allocate the device storage    デバイス・ドライバが定義する変数の宣言
 */

ALTERA_AVALON_TIMER_INSTANCE( SYS_CLK_TIMER, sys_clk_timer );
ALTERA_AVALON_SYSID_INSTANCE( SYSID, sysid );

/*
 * Initialise the devices    デバイスの初期化
 */

void alt_sys_init( void )
{
    ALTERA_AVALON_TIMER_INIT( SYS_CLK_TIMER, sys_clk_timer );
    ALTERA_AVALON_TIMER_INIT( HIGH_RES_TIMER, high_res_timer );
}
```

3. ブート・シーケンス

3-1. ブート・シーケンス

HAL は、以下のブート・シーケンスを実行するシステム初期化コードを提供します。



ユーザのアプリケーションで alt_main エントリ・ポイントが必要な場合には、必要に応じて alt_main.c や alt_sys_init.c のブート・コードのカスタマイズが可能です。ブート・コードを変更することによって、以下のような制御を行うことができます。

- alt_main.c　　ブート・シーケンスの制御とシステム・リソースの選択
- alt_sys_init.c　不要なデバイスの初期化コードを削除しコード・サイズを小さくする
- 自動生成されるファイルの代わりに、ローカル・ファイルを使用する

例として、以下のような方法でブート・コードのカスタマイズを行います。

- <nios2eds>¥components¥altera_hal¥HAL¥src にある alt_main.c ファイルを syslib プロジェクトにコピーしてカスタマイズを行う
- syslib プロジェクト中の alt_sys_init.c をアプリケーション・フォルダにコピーしてカスタマイズを行う

上記のような方法は free-standing development になります。

3-1-1. Hosted vs Free-Standing アプリケーション

Hosted と Free-Standing の実行環境には、以下のような違いがあります。

'Hosted' アプリケーション

- 開発するコードは main () から開始
- すべてのシステム・サービスとデバイスの初期化が行われ使用可能状態
- どのようなシステム変更でもツールが自動的に対応

'Free-Standing' アプリケーション

- カスタマイズされたブート・シーケンスを使用
- ブート・シーケンスの綿密な制御が可能
- 開発するコードは alt_main () から開始 (Altera 標準)
- 使用するデバイス、サービスの初期化はユーザが行う
  - alt_main () で、使用するキャラクタ・モード・デバイス・ドライバを初期化し、stdio をそのデバイスにリダイレクトしなければ動作しない

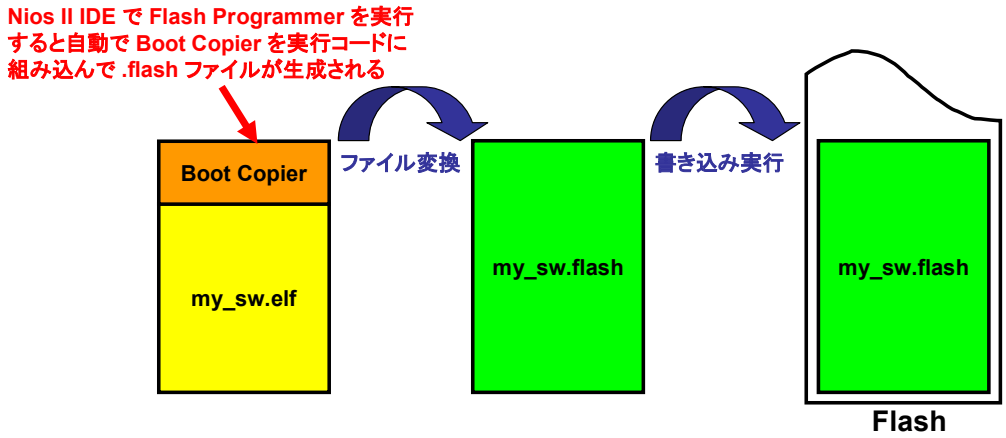
3-1-2. ブート・コピア

リセット・ベクタを持つメモリ・デバイスが Nios II プロセッサのブート・デバイスになり、プログラムが保存されます。外部フラッシュ・メモリ、または EPCS シリアル・コンフィギュレーション・デバイス、オンチップ RAM をブート・デバイスに指定できます。

システム・ライブラリ・プロパティの設定にてプログラムの実行領域(.text セクション)がブート・デバイスではなく、外部の RAM 等に配置されている場合、 Nios II Flash Programmer はすべてのコードおよびデータ・セクションをロードするブート・ローダを自動的にリセット・ベクタに配置します。ただし、.text 領域がブート・デバイス内に指定されている場合には、個別のローダは存在しません。

また、フラッシュ・メモリを .text 領域、.rodata 領域に指定することは可能ですが、フラッシュ・メモリからの実行はアクセス・スピードが遅いために、動作は低速となってしまいます。

例) リセット・ベクタをフラッシュ・メモリ、 .text を RAM に配置した場合



※ ブート・コピアのソース・コードは、Nios II EDS のインストール・ディレクトリに含まれます。

例) <nios2eds>¥components¥altera_nios2¥boot_loader_sources

3-1-3. ブート・コピアの修正

ブート時に下記のような拡張機能が必要な場合には、ブート・コピアをユーザが修正することも可能です。

- 複数のブート・イメージを切り替えて使用する
- ブート中のメッセージの表示
- ブートデータのエラーチェック
- Word-align されていないイメージデータを展開する

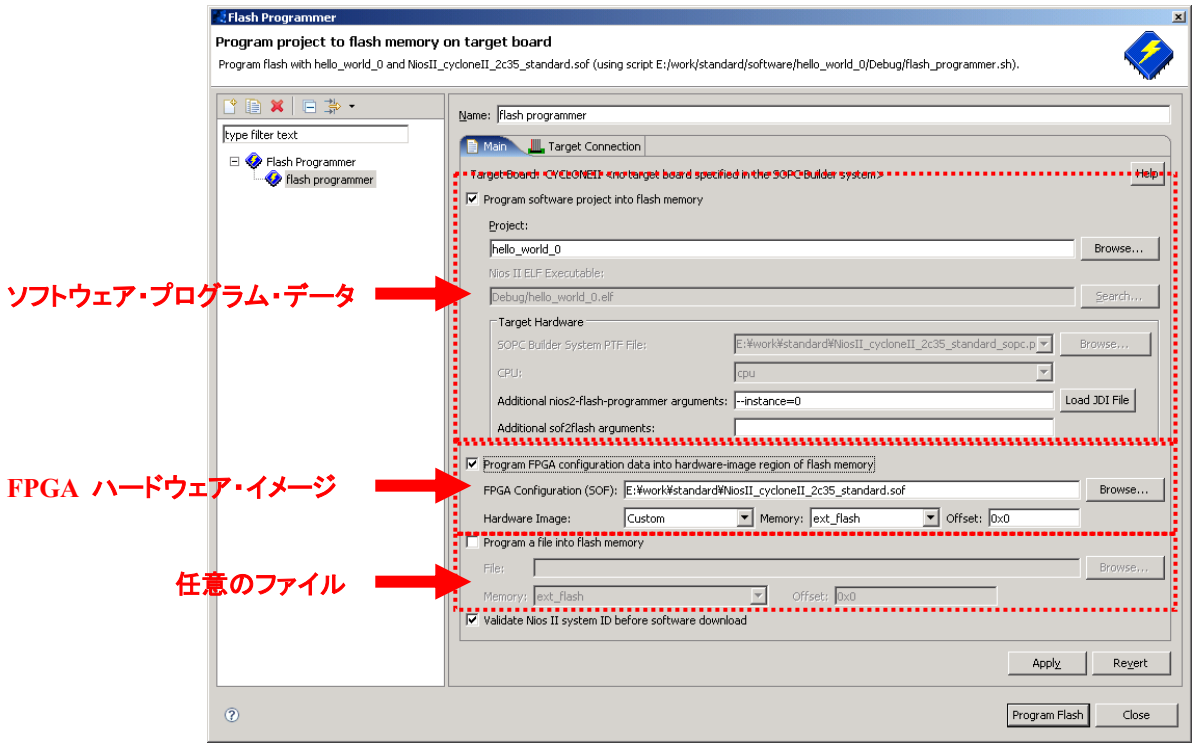
カスタマイズを行う場合には、下記のアプリケーションノートをご参照ください。

- AN458: Alternative Nios II Boot Methods （サンプル・ソース付き）
- http://www.altera.com./literature/lit-an.jsp
- http://www.altera.com/literature/an/an458.pdf

3-1-4. フラッシュ・メモリのプログラミング

CFI フラッシュと EPCS シリアル・コンフィギュレーション・デバイスは、Nios II IDE もしくはコマンド・シェルから Flash Programmer を使用してプログラミング可能です。Flash Programmer は、以下のコードやデータを簡単にフラッシュ・メモリに書き込むことができます。

- FPGA ハードウェア・イメージ
- ソフトウェア・プログラム・データ
- その他、任意のファイル
- ブート・コピアの自動組み込み



4. Nios II コード・サイズ

コード・サイズを最小サイズに縮小する必要がある場合に使用できる、各オプションについて説明します。

4-1. コード・サイズの確認

4-1-1. コード・サイズを知る方法（.objdump ファイルの生成）

以下の方法で、Nios II コマンド・シェル、もしくは Nios II IDE にてプログラムのコード・サイズを確認することができます。

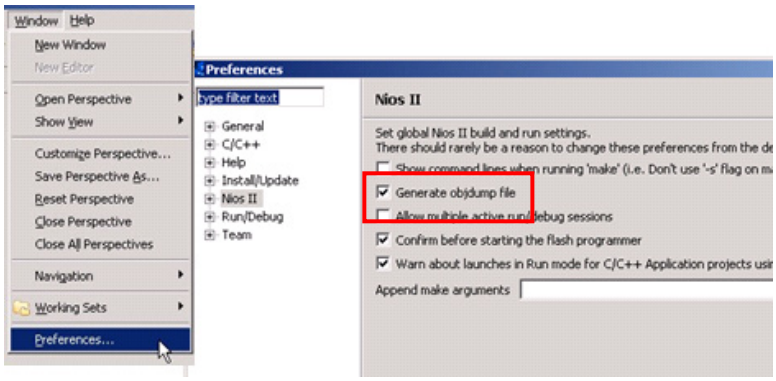
- コマンド・シェルにて、ビルド終了後に以下のコマンドを実行

  nios2-elf-size <*myproject.elf*>

  nios2-elf-readelf <*myproject.elf*>

- ビルド時に .objdump ファイルを生成

  IDE の window メニュー > Preferences > Nios II の設定で objdump ファイルの生成オプション（Generate objdump file）を On にすると、アプリケーション・プロジェクトの Debug（もしくは Release）フォルダに <Application_Project_Name>.elf.objdump の名前で生成されます。



4-1-2. .objdump ファイルからコード・サイズを読む

以下は .objdump ファイルの抜粋です。各メモリ・セクションのサイズやアドレスがレポートされています。Size の項目がそのセクションのコード・サイズになります。この例では、.text 領域に配置されたプログラム・コードは 0x228B0 byte です。

例） .objdump ファイルの抜粋

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .entry        00000020  00000000  00000000  00000094  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .exceptions   000002f8  01000020  01000020  000000b4  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         000228b0  01000318  01000318  000003ac  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00003370  01022bc8  01022bc8  00022c5c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rwdata       00001e30  01025f38  01025f38  00025fcc  2**2
                  CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA
  5 .bss          0004033c  01027d68  01027d68  00027dfc  2**2
                  ALLOC, SMALL_DATA
  6 .ext_flash    00000000  00000020  00000020  00027dfc  2**0
                  CONTENTS
  7 .ext_ram      00000000  02000000  02000000  00027dfc  2**0
```

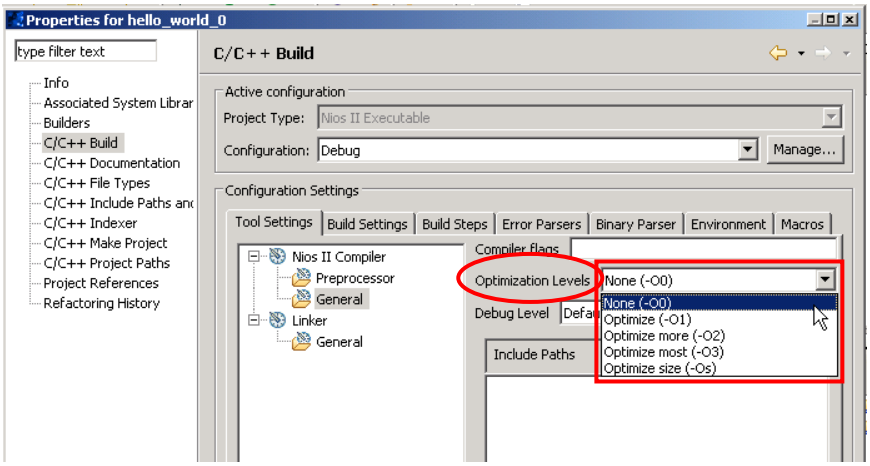
4-2. コード・サイズの縮小

4-2-1. コードのフット・プリントを小さくするオプション

下記の各オプションを使用して、コード・サイズを縮小することが可能です。

- コンパイラでの最適化

  Application / Syslib 両方のプロジェクト Properties の C/C++ Build > Optimization Level を –Os や –O3 に設定することによって、コンパイル時にコードはサイズと速度が最適化され、フット・プリントを縮小することができます。



- Reduce device drivers の選択

  いくつかのデバイスでは、フル機能の「高速」型と軽量の「スモール」型のドライバが用意されています。HAL システム・ライブラリは、常に高速型のドライバを使用します。Nios II IDE の System library properties の reduce device drivers の設定をオンにすることによって「スモール」型のドライバを使用します。このスモール・フットプリント・ドライバに対応しているペリフェラルは、UART、JTAG UART、共通フラッシュ・インターフェース・コア、LCD モジュール・コントローラ・コアです。

- Max file descriptor の値を減らす

  キャラクタ・モード・デバイスおよびファイルにアクセスするファイル・ディスクリプタは、使用可能なファイル・ディスクリプタのプールから割り当てられます。デフォルトは 32 です。例えば、プログラムで 10 のみ必要であれば、Max file descriptors: の値を小さくすることによってメモリ・フットプリントを縮小することができます。

- Small ANSI C library の選択

  Small C library を選択することによって、縮小版の newlib ANSI C 標準ライブラリを使用する設定に変更できます。縮小版のライブラリには各関数に制限事項がありますので、使用する関数に影響があるかどうかをご確認いただく必要があります。

- clean exit の非選択

  終了時に伴うオーバーヘッドを回避するために、ユーザ・プログラムでは exit () 関数の代わりに _exit () を使用することができます。Clean exit (flush buffers) の設定をオフにすることによって、_exit () も使用しない設定にすることができます。

- UNIX-style 入力関数の使用

  ANSI C ファイル I/O ではなく、UNIX 形式のファイル I/O を直接使用することによってコード・フットプリントを削減することが可能です。

- Program never exits の選択

  HAL はシステム・シャットダウン時に exit () 関数を呼び出して、プログラムからの終了を実現します。exit () 関数は main () から戻るときに使用されますが、通常組み込みシステムは終了することがないため、このコードは冗長となります。Program never exits のオプションをオンにすることによって exit () 関数を省いてコード・サイズを縮小します。

- Support C++ の非選択

  デフォルトだと、C++ プログラムをサポートしていますが、この設定をオフにすることができます。

- lightweight device driver API の選択

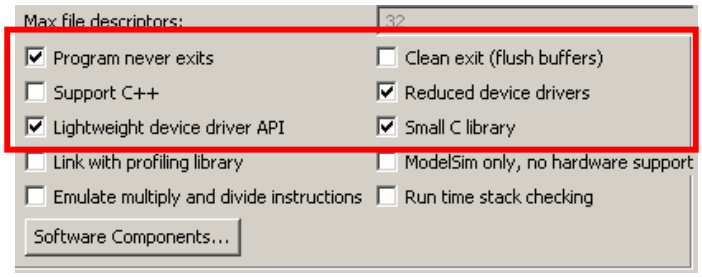
  デフォルトの設定はオフです。Lightweight device driver API の設定をオンにすることによって、いくつかの機能を省いてドライバのサイズを小さくします。この設定は、JTAG UART、UART、Optrex 16207 LCD のキャラクタ・デバイスに対して有効です。

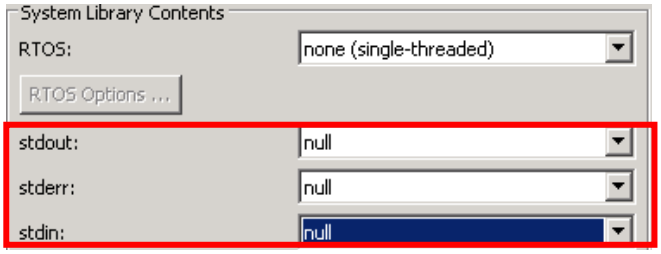
※ lightweight deriver API を使用する場合には、下記のような制限があります。

- stdin、stdout、stderr ファイル・ディスクリプタのみをサポート
- hostfs、 zipfs は使用不可
- システムに含まれるすべてのキャラクタ・モードのデバイス・ドライバが lightweight driver API をサポートしていること

- 不要な場合 stdout/stdin/stderr を null にする

  stdin、stdout、stderr ファイル・ディスクリプタはドライバがインストールされると、HAL で設定されたチャネルにリダイレクトされます。stdin、stdout、stderr の設定を null にすることによってリダイレクトのコードが削減されてフット・プリントを小さくすることができます。





4-2-2. alt_* 標準入出力ルーチンの使用

キャラクタ・モードの縮小版 API を使用することによって、通常の printf ( ) 関数 や getchar ( ) 関数を使用する場合に比べてコード・サイズを縮小することができます。この API には alt_printf ( )、alt_putchar ( )、alt_getchar ( )、alt_putstr ( ) の関数が含まれます。この API を使用するためには sys/alt_stdio.h をインクルードします。

alt_printf ( ) 関数は通常の printf ( ) 関数と似ていますが、%c、 %s、 %x のフォーマット指定子のみをサポートします。コード・サイズは、alt_printf ( ) は約 350 Byte、 small newlib の printf ( ) は約 2240 Byte です。

4-2-3. コード・サイズ例

下記は Hello World テンプレートを使用した場合に、上記のコード削減のオプションを使用してどの程度コード・サイズを縮小できるかを示しています。

Cyclone III のサンプル・デザインを使用、Nios II のコアは Standard を使用しています。

Hello World デフォルト	各オプションを使用	削減率
66 Kbytes	340Bytes	90%以上

5. Nios II 例外処理

Nios II プロセッサで例外を処理する場合のプログラムの記述方法について説明します。

5-1. Nios II 例外処理

Nios II の例外処理はすべての例外が “exception location” に置かれた例外処理ハンドラによって処理されます。この例外処理コードは HAL システム・ライブラリが提供し、アドレスは SOPC Builder の Nios II Processor の Core Nios II タブの Exception Vector で設定したアドレスに配置されます。

例外ハンドラでは例外のタイプを判定し、どのように処理するかを決定します。例外ハンドラには alt_irq_entry ()、 alt_irq_handler ()、 software_exception () のルーチンがあります。

- alt_irq_entry ()

  ハードウェア割り込みが存在する場合に、発生した割り込みのタイプを判定し適切なルーチンを呼び出します。

- alt_irq_handler ()

  ハードウェア割り込みの割り込み番号を判定し、登録されたルーチンを呼び出します。

- software_exception ()

  ソフトウェア例外の原因を特定します。

5-2. ハードウェア割り込み

5-2-1. 割り込み処理のための HAL API

下記の HAL API を使用して、プログラムの特定のセクションに対しての割り込みをディセーブルしたり、再度イネーブルしたりすることができます。

- alt_irq_register () ： ユーザ割り込み処理（ISR）関数の登録。
- alt_irq_disable_all () ： すべての割り込みを禁止します。
- alt_irq_enable_all () ： すべての割り込みを許可します。
- alt_irq_interruptible () ： ISR 関数内で使用します。

  ISR 処理中に発生した、より優先度の高い割り込みリクエストを許可します。

  デフォルトでは ISR 実行中は他の割り込みは許可されません。

- alt_irq_non_interruptible () ： ISR 処理中に発生した割り込みを許可しません。（デフォルト）

下記の HAL API を使用して、ハードウェア割り込みにマスクをかけることができます。引数は、各ハードウェア割り込みに設定した割り込み番号です。

- alt_irq_enable (alt_u32 id)
- alt_irq_disable (alt_u32 id)

5-2-2. 例外処理ルーチンの書き方

まず、呼び出される ISR をプログラム中に記述します。ISR では、引数として ISR で使用するためのデータのポインタ(*context)、SOPC Builder で割り当てられた割り込み番号(id)が渡されます。

呼び出し側では、記述した ISR を alt_irq_register () 関数を使用して登録します。alt_irq_register () には、3 つの引数、割り込み番号(id)、ISR に引き渡すデータのポインタ(*context)、呼び出し先の関数のポインタ(*isr)を渡します。

ISR を記述する際には、以下の事項を考慮してください。

- ISR 内で記述する処理は、できるだけシンプルなものにします。基本的に時間のかかる処理は、ISR 内では行わずアプリケーション内で行います。
- ISR 内での標準入出力関数や RTOS 関数の呼び出しは、正常動作しません。例えば printf () 関数はルーチン内で UART や JTAG_UART を使用するため、割り込みを使用します。割り込み処理中は、基本的には他の割り込みはディセーブルとなりますので、正常動作しません。
- ISR 内で他の割り込みを可能にするためには alt_irq_interruptible () や alt_irq_non_interruptible () を使用して制御します。

ISR を記述

```
sample_isr ( void* context, alt_u32 id)
{
   ...
}
```

id :割り込み番号(0 to 31)

context :ISR で使用する、もしくは ISR 内で作成される データへの void ポインタ

ISR の登録

プロトタイプ:

```
alt_irp_register( alt_u32 id,
                  void* context,
                  void (*isr) (void*, alt_32));
```

使用例:

```
alt_irq_register ( periph_irq, &some_data, sample_isr );
```

5-2-3. 例 1: 割り込み処理ルーチン

■ISR の記述

```
my_isr( void* context, alt_u32 id)
{
    volatile int* my_var_ptr = (volatile int*) context;

    /*IRQ を発生させたハードウェア内(例:BUTTON_PIO)のれ越したを読む*/
    *my_var_ptr = IORD_ALTERA_AVALON_PIO_DATA(BUTTON_PIO_BASE);

    /*BUTTON_PIO のレジスタリセット*/
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
}
```

※ **“context”** をローカルポインタに代入。この ISR 内、もしくは外に情報を渡すために使用可能。
“volatile” は、コンパイラによる不要な最適化を防ぐために使用。

※ ISR は、可能な限り短時間の処理のみを実行する必要があります。
必要なデータを取り込み、必要最低限のデバイス・コントロールを行い、ISR 内で割り込みをクリアし、終了します。

■IRQ の登録とデバイスの初期化

```
int main(void)
{
    /*ISR 内で参照される変数の宣言*/
    volatile int my_isr;

    /*使用するデバイス(例:BUTTON_PIO)の初期化と ISR の登録*/
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    alt_irq_register(BUTTON_PIO_IRQ, (void*) &my_var, my_isr);

    /*
     *  BUTTON_PIO の IRQ が発生するたびに
     * ISR により my_var の値が更新される
     */

      .
      .
      .
```

5-2-4. 例 2: ネストした割り込み処理

下記は、ISR 実行中に、現在実行中の ISR よりも高い優先順位の高い割り込みが入ったときに、そちらの ISR を実行するサンプルです。テスト記述として illuminate_led2 () の中で alt_irq_interruptible() を記述し、while 文を使用してより優先順位の高い割り込みが入るのを待ちます。

ボタン割り込みでエッジ・キャプチャ・レジスタを使用してエッジで割り込みを検出した場合には、必ず ISR の中でエッジ・キャプチャ・レジスタをリセットする必要があります。

呼び出し側では、button PIO の初期化と ISR（illuminate_led2）の登録を行います。

ISR

```
/*低優先度の ISR*/
static void illuminate_led2 (void* context, alt_u32 id)
{
    /*LED 2 を ON*/
    led = 0x 04;
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);

    /*BUTTON のエッジ・キャプチャ・レジスタのリセット*/
    IOWR_ALTERA_AVALON_EDGE_CAP(BUTTON_P2_BASE, 0);

    /*優先度の高い IRQ 待ち*/
    alt_irq_interruptible();

    while (1){   /*この無限ループはテスト用です*/
    }            /*実際は ISR 内で無限ループは使用しません*/
}
```

呼び出し側

```
/*BUTTON_PIO の初期化と ISR の登録*/
alt_irq_register(BUTTON_PIO_IRQ, (void*)NULL, illuminate_led2);

/*4つのボタンの割り込みを許可*/
IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);

/*エッジ・キャプチャ・レジスタのリセット*/
IOWR_ALTERA_AVALON_EDGE_CAP(BUTTON_PIO_BASE, 0);
```

5-3. 割り込みレスポンスの高速化

5-3-1. 割り込みレイテンシ 用語とその値

- 割り込みレイテンシ(Latency)

  割り込みが発生してから、例外処理コードの最初のインストラクションを実行するまでの時間(CPU サイクル)

- 割り込み応答時間(Response Time)

  割り込みが発生してから、ユーザが書いた割り込みルーチンの最初のインストラクションを実行するまでの時間(CPU サイクル)

- 割り込み復帰時間(Recovery Time)

  割り込み処理ルーチンの最後のインストラクションから通常の処理に戻るまでの時間(RTOS の場合は、割り込まれたタスクに復帰するまでの時間)

割り込み処理パフォーマンス(クロックサイクル数)			
Core	**Latency**	**Response Time**	**Recovery Time**
Nios II / f	10	105	62
Nios II / s	10	128	130
Nios II /e	12	485	222

5-3-2. 割り込みレスポンスの高速化

割り込みレスポンスを高速化するために、下記の操作が有効です。

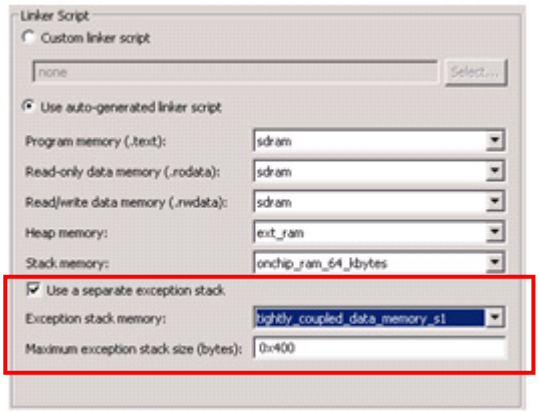
- ISR コードを高速でレイテンシの小さい tightly coupled メモリ、または on-chip メモリに配置する

  下記のように __attribute__ 記述を使用して、作成した ISR を高速メモリに配置することも可能です。

  void my_isr __attribute__ ((section (".tightly_coupled_instruction_memory ")));

- スタック(もしくは割り込み専用スタック)を高速メモリに配置する

  System ライブラリの Properties で Use a separate exception stack にチェックを入れて使用するメモリを指定します。



◆ Interrupt Vector カスタム・インストラクションを使用する

割り込み処理のディスパッチをハードウェアで実行します。

5-3-3. Interrupt Vector カスタム・インストラクション

SOPC Builder の設定にて Interrupt Vector カスタム・インストラクションがあらかじめ用意されています。これを Nios II コアに追加することによって、プライオリティ・エンコーダ（マルチプレクサ）をハードウェアでインプリメントします。

マルチプレクサの段数は、システムが持つ割り込み信号の数に比例します。使用するロジックは数 LE ですが、多くの割り込み信号が接続されていると Fmax に影響する場合があります。

Interrupt Vector カスタム・インストラクションを使用する際には、自動でカスタム・インストラクションが呼び出されるためユーザ・コードを変更する必要はありません。