

この資料は、更新された最新の英語版が存在します。こちらの日本語版は参考用としてご利用下さい。設計の際は、必ず最新の英語版で内容をご確認下さい。

NII52006-1.2

### はじめに

この章では、Nios® II プロセッサ・アーキテクチャで例外を処理するプログラムの記述方法について説明します。ユーザ定義の割り込みサービス・ルーチン (ISR) を HAL (Hardware Abstraction Layer) に登録することによって、ハードウェア割り込み要求を処理する方法を重点的に説明します。

この章では、以下のトピックを扱います。

- Nios II 例外の概要
- HAL 実装
- ISR
  - ISR 用の HAL API (Application Programming Interface)
  - ISR の記述
  - ISR のイネーブルおよびディセーブル
  - C の例
- 高速 ISR 処理
- ISR 使用のデバッグ
- ISR の記述に関する提案の要約



Nios II アーキテクチャでの例外および割り込み処理の低レベル仕様の詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」のプログラミング・モデルの章を参照してください。

### Nios II 例外の概要

Nios II 例外処理は、従来からの RISC 形式、すなわちすべての例外タイプが 1 つの例外ハンドラで処理される形式で実装されます。したがって、すべての例外 (ハードウェアおよびソフトウェア) は、「例外アドレス」と呼ばれる 1 つの位置に存在するコードによって処理されます。

Nios II プロセッサでは、以下の例外タイプが利用できます。

- ハードウェア割り込み例外。
- ソフトウェア例外。この例外は以下のカテゴリに分類されます。
  - 未実装命令
  - ソフトウェア・トラップ
  - その他の例外

例外が生成されると、プロセッサは以下のステップを自動的に実行します。

- status レジスタ (ct10) の内容を estatus レジスタ (ct11) にコピーして、例外発生前のプロセッサの状態を保存します。
- status レジスタの PIE ビットをクリアして、以降のハードウェア割り込みをディセーブルします。
- 例外発生後の命令のアドレスを ea レジスタ (r29) に格納して、例外ハンドラが戻るリターン・アドレスを提供します。
- 例外アドレスへのベクタ・ジャンプ

## HAL 実装

このセクションでは、HAL システム・ライブラリが使用する例外ハンドラ実装について説明します。これは参考のためであり、HAL ISR サービスを利用するのに、完全に理解する必要はありません。



HAL API (Application Programming Interface) を使用して ISR をインストールする方法の詳細については、[6-6 ページの「ISR」](#)を参照してください。

HAL システム・ライブラリを持つ例外ハンドラは、例外アドレスに配置されます。このハンドラは、以下のアルゴリズムを実行して、ハードウェア割り込みとソフトウェア例外を区別します。

- estatus レジスタの EPIE ビットがイネーブルされているかどうか判断する。
  - イネーブルされていない場合、例外はソフトウェア例外となります。
  - イネーブルされている場合は、以下のステップに進みます。
- ipending がゼロ以外かどうかを判断します。
  - ipending のいずれかのビットがゼロ以外の場合、例外はハードウェア割り込みであり、ハードウェア割り込みを処理します。
  - すべてのビットがゼロの場合、例外はソフトウェア例外です。

このアルゴリズムは以下の 3 つのルーチンを使用します。

- `_irq_entry()`
- `alt_irq_handler()`
- `software_exception()`

## `_irq_entry`

Nios II システムにハードウェア割り込みが存在する場合、トップ・レベルのアセンブリ・ルーチン `_irq_entry` が例外アドレスに配置されます。このアセンブリ・ルーチンは、発生した例外のタイプを確認し、適切なルーチン呼び出しを呼び出します。例外がソフトウェア例外の場合は、ルーチン `software_exception` を呼び出し、ハードウェア割り込みの場合は、ルーチン `alt_irq_handler` を呼び出します。

ルーチンのアセンブリ・コードを確認するには、<Nios II インストール・パス >/components/altera\_nios2/HAL/inc/sys/alt\_irq\_entry.h ファイルを参照してください。あるいは、ハードウェア割り込みを使用するプロジェクトをビルドした後に、`objdump` でリンクされたアセンブリ・ルーチンを調べることもできます。

以下のコードは、`_irq_entry` ルーチンの擬似コード例です。

### 例：`_irq_entry` の擬似コード例

```
_irq_entry:
if EPIE = 0
    // ソフトウェア例外
    goto software_exception_handler_assembly.
else if ipending = 0
    // ソフトウェア例外
    goto software_exception_handler_assembly.
else
    // ハードウェア割り込み
    store pre-exception_processor_state
    // alt_irq_handler を呼び出して適切な ISR をディスパッチする。
    call the alt_irq_handler_routine
    restore the pre-exception_processor_state
    // 例外からの復帰
    issue the exception_return_instruction, eret. .
```

### `alt_irq_handler()`

関数 `alt_irq_handler()` は、割り込みの原因（つまり、割り込みを発生させたデバイスに関連付けられた割り込み番号）を特定し、HAL に登録されたその割り込みに対する関数を実行します。ループが記述された順序によって、最も高い割り込み要求（IRQ）の優先順位は `IRQ0` に、最も低い順位は `IRQ31` に与えられます。

以下のコードは、`alt_irq_handler()` の擬似コード例です。

### 例：alt\_irq\_handler() の擬似コード例

```
alt_irq_handler(void)
// 0 から 31 までのすべての IRQ に対するループ
// ipending が最初に「1」となったときに
// ユーザ定義の関数を実行します。
for i from 0 to 31:
// ipending のどのビットが「1」であるかチェック。
if ipending[i] == '1':
// ユーザ定義の関数を実行します。
// 注：alt_irq_arg[i] と i はそれぞれ、
// ユーザの関数プロトタイプにおける
// void* 型の context および id に対応します。
// alt_irq[] は、ISR への関数ポインタの配列です。
alt_irq[i]( alt_irq_arg[i], i )
// 最初のアクティブな割り込みを検出すれば、
// チェックを停止します。
break;
```

ソース・コードは、<Nios II インストール・パス >

/components/altera\_hal/HAL/src/alt\_irq\_register.c ファイルにあります。

## software\_exception

`software_exception` ルーチンは、ソフトウェア例外の原因を特定します。現在、`software_exception` ルーチンは基本的に、どの未実装命令が例外を発生したかを特定し、適切な命令エミュレーション・ルーチンを呼び出します。

Nios II システムに、ハードウェア割り込みを使用するペリフェラルが含まれていない場合、`software_exception` ルーチンは例外アドレスに直接配置されます。また、`_irq_entry` および `alt_irq_handler` はプロジェクトにリンクされません。

ソフトウェア例外の原因を特定するには、命令ワード内の `OP` フィールドと `OPX` フィールドを調べる必要があります。



`OP` フィールドおよび `OPX` フィールドの詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」の命令セットのリファレンスを参照してください。

以下のコードは、`software_exception` アセンブリ・ルーチンの擬似コード例を示します。

### 例：`software_exception` の擬似コード例

```
software_excetion:
if encoding = trap instruction
  // ソフトウェア・トラップ
  // 現在は未実装 (nop 命令のように動作)
  goto trap_handler
else
  // 命令エミュレーション
  case op / opx
    muli:goto mul_immed // 即値乗算
    mul:goto multiply // 乗算。
    mulxss:goto mulxss // 符号付き - 符号付き乗算
    mulxsu:goto mulxsu // 符号付き - 符号なし乗算
    mulxuu:goto mulxuu // 符号なし - 符号なし乗算
    div:goto divide // 符号付き除算
    divu:goto unsigned_division // 符号なし除算
例外からの復帰
```

すべてのソース・アセンブリ・コードは、<Nios II インストール・パス> /components/altera\_nios2/HAL/src/alt\_exceptions.S ファイルにあります。



上記の擬似コードは、`alt_exceptions.S` とは厳密には一致しません。厳密な実装の詳細については、アセンブリ・ソース・コードを参照してください。

### 未実装命令

`software_exception` は、未実装命令と考えられる各命令に対するエミュレーション・ルーチンを定義します。このようにして、特定の Nios II コアがハードウェアですべての命令を処理しない場合でも、すべての Nios II 命令セットが常にサポートされます。一方で、Nios II コアがハードウェアに特定の命令を実装する場合は、対応する例外は発生しません。エミュレーション・ルーチンは十分小さいため、エミュレーション・ルーチンを必要としない Nios II コアをターゲットとする場合でも、取り除くメリットはほとんどありません。



未実装命令の詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」のプロセッサ・アーキテクチャの章を参照してください。



エミュレーション・ルーチンは例外の状況で実行されるため、例外ルーチンは絶対に未実装命令を発行できません。「未実装命令」は「無効な命令」を意味するものではありません。現行の Nios II コア実装では、OP フィールドおよび OPX フィールドに有効な命令エンコーディングが含まれていない場合、結果は未定義です。したがって、`software_exception` ルーチンは、無効な命令を検出したり、それに応答することはできません。未定義の OP エンコーディングおよび OPX エンコーディングに対するプロセッサの動作は、Nios II コアによって異なります。



詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」の Nios II コア実装の詳細の章を参照してください。

### ソフトウェア・トラップの例外処理

`software_exception` は、現行ではソフトウェア・トラップの例外に対して null 処理を実行します。`alt_exceptions.S` のコードでは、ソフトウェア・トラップに対する OP エンコーディングおよび OPX エンコーディングは検出しませんが、空の `trap_handler` ルーチンに分岐します。

### その他の例外

将来の Nios II プロセッサ・コア実装では、新しい例外タイプが定義される予定です。それによって、例外の正確な原因を特定することなく、`software_exception` が実行できるようになります。HAL 実装は現在定義されていない例外タイプには対応しません。

## ISR


多くの場合、ペリフェラルとの通信は、割り込みを使用して達成されます。ペリフェラルが IRQ をアサートすると、プロセッサの通常の実行フローに例外が発生します。このような割り込みが発生すると、適切な ISR によってこの割り込みを処理し、処理が完了したらプロセッサを割り込み発生前の状態に戻すことが必要です。このセクションでは、割り込み処理用に HAL システム・ライブラリが提供するフレームワークについて説明します。

## ISR 用 HAL API

HAL システム・ライブラリは、ISR の作成とメンテナンスを容易にするために API を提供しています。また、MicroC/OS-II プログラムではすべての HAL API を利用できるため、この API も MicroC/OS-II ベースのプログラムで使用できます。HAL API は割り込みを管理するための以下の関数を定義しています。

- `alt_irq_register()`
- `alt_irq_disable_all()`
- `alt_irq_enable_all()`
- `alt_irq_interruptible()`
- `alt_irq_non_interruptible()`
- `alt_irq_enabled()`

ISR の実装に HAL API を使用するには、2 つのステップを実行します。まず、特定のデバイスに対する割り込みを処理する割り込みサービス・ルーチンを記述します。次に、プログラムで `alt_irq_register()` 関数を使用して ISR を HAL に登録する必要があります。このプログラムでは、`alt_irq_enable_all()` 関数と `alt_irq_disable_all()` 関数を使用して、実行中に割り込みをイネーブルまたはディセーブルできます。

 割り込みをディセーブルすると割り込みレイテンシが影響を受け、このためシステム・パフォーマンスも影響を受けます。

### alt\_irq\_register() を使用した ISR の登録

HAL はこの関数ポインタをルックアップ・テーブルに登録します。特定の IRQ が発生すると、HAL はルックアップ・テーブル内で IRQ を探し、登録済みの ISR をディスパッチします。

`alt_irq_register()` のプロトタイプは以下のとおりです。

```
int alt_irq_register (alt_u32 id,
                    void*   context,
                    void    (*isr)(void*, alt_u32));
```

プロトタイプには以下のパラメータがあります。

- `id` は、`system.h` で定義されたデバイスのハードウェア割り込み番号です。割り込みの優先順位は IRQ 番号と逆の関係にあります。したがって、IRQ 0 は優先順位が最高の割り込み、IRQ 31 は最低の割り込みを表します。
- `context` は、コンテキスト固有の情報を ISR に渡すために使用されるポインタです。このポインタは、どの種類の ISR 固有情報でも指すことができます。`context` の値は HAL からは読み取れません。この値は、ユーザ定義 ISR での利用のみを目的として提供されます。
- `isr` は IRQ 番号 ID に応答して呼び出される関数です。この関数には、2 つの入力引数として、`context` ポインタと `id` が提供されます。`isr` に `null` ポインタを登録すると、割り込みがディセーブルされません。

ISR が正常に登録されると、関連付けられた割り込み (`id` で定義) は、`alt_irq_register()` からの復帰時にイネーブルされます。



`alt_irq_register()` の詳細については、[10-1 ページの「HAL API リファレンス」](#)を参照してください。

## ISR の記述

記述する ISR は、`alt_irq_register()` が認識するプロトタイプに一致する必要があります。ユーザの ISR 関数のプロトタイプは、以下のプロトタイプと一致しなければなりません。

```
void isr (void* context, alt_u32 id)
```

`context` および `id` のパラメータ定義は、`alt_irq_register()` 関数のパラメータ定義と同じです。

ISR の機能は、関連付けられた割り込み条件をクリアまたはマスク・アウトして、割り込みハンドラに戻ることです。

### 制限された環境

ISR は制限された環境で動作します。HAL API 呼び出しの大部分は ISR からは利用できません。例えば、HAL ファイル・システムへのアクセスは許可されません。一般的な規則として、独自の ISR を記述する場合、割り込みの待機を妨害する可能性がある関数呼び出しは、絶対に使用しないでください。

さらに、ISR の内部で ANSI C 標準ライブラリ関数を呼び出すときにも注意が必要です。C 標準ライブラリ I/O API を呼び出すと、システム内でデッドロックが発生する（つまり、システムが ISR 内部で永久にブロックされる）可能性があるため、これらの API 関数を使用した呼び出しは避ける必要があります。特に、不用意に ISR 内部から `printf()` を呼び出すことは避けてください。`stdout` が、適切に動作するために割り込みを使用するデバイス・ドライバにマップされている場合、`printf()` 呼び出しは、割り込みがディセーブルされるため、システムは発生することのない割り込みを待ち続けデッドロックに陥る可能性があります。ISR 内部から `printf()` を安全に使用できるのは、デバイス・ドライバが割り込みを使用しない場合に限られます。

## ISR の性能

性能を向上させるために、ISR は通常、割り込みをディセーブルして実行されます。これによって、ISR はリエントリ操作が可能である必要がなくなるため、割り込み処理に関連するシステム・オーバヘッドが低減され、ISR の開発が容易になります。ただし、ISR が処理に時間を要する場合、システムの応答性に悪影響を与える可能性があります。特に、システムの他の ISR のリアルタイム動作（割り込みレイテンシ）に影響を与えます。このため、ISR は可能な限り効率的にすることが必要です。ISR は、割り込み条件をクリアして復帰するのに必要な最小限の作業を実行しなければなりません。

## 低速割り込みハンドラ

割り込みハンドラが実行に時間を要する場合、システムの性能と機能に悪影響を与えることがあります。割り込みハンドラを再編成して実行時間を短縮できない場合は、より優先順位の高い割り込みを低速割り込みハンドラに割り込ませることができます。これは、ネスト式割り込みハンドラとして知られています。

ネスト式割り込みハンドラを使用すると、優先順位が低い（割り込みを再イネーブルする割り込みハンドラよりも優先順位の低い）割り込みの割り込みレイテンシが増加するため、この方式を採用するときは検討が必要です。



ISR 経由で最も遅いパスが約 70 命令未満のときにネスト式割り込みを許可すると、優先順位の高い割り込みの割り込みレイテンシが増加します。このような割り込みハンドラでは、割り込みを再イネーブルしないでください。

ネスト式割り込みが望ましい場合、`alt_irq_interruptible()` 関数と `alt_irq_non_interruptible()` 関数を使用して、より優先順位の高い割り込みから割り込みを受ける可能性のある低速 ISR 内に、コードをまとめる必要があります。これらの関数を使用すると、優先順位の高い ISR の割り込みレイテンシを改善することができます。これらの関数はペアで使用する必要があります。一方の関数しか使用しないと、システムがロックする可能性があります。

### 低速動作の抑制

一般に、ISR はハードウェアの状態の変化に対して、レイテンシの低い迅速な応答を行います。バルク・データ転送など、低レイテンシ機能が必要としない低速動作の実行は避けるべきです。低速動作は延期して、割り込み処理外で実行する必要があります。

MicroC/OS-II スケジューラなどのリアルタイム・オペレーティング・システム (RTOS) をベースにしたシステムでは、タスクの延期は簡単です。この場合、低速処理を扱うスレッドが作成でき、ISR はイベント・フラグやメッセージ・キューなど、MicroC/OS II の通信メカニズムを使用して、このスレッドと通信できます。

シングル・スレッドの HAL ベース・システムでも同じ方法が使用できますが、処理がやや煩雑になります。低速コードは、メイン・プログラムから定期的に呼び出す必要があります。このプログラムは ISR で管理されるグローバル変数をポーリングして、低速処理ルーチンを呼び出す必要があるかどうかを判断します。

## ISR のイネーブルおよびディセーブル

HAL は関数 `alt_irq_disable_all()`、`alt_irq_enable_all()`、`alt_irq_enable()` を提供しており、これによりプログラムはコードの特定のセクションに対する割り込みをディセーブルし、後で再イネーブルすることができます。`alt_irq_disable_all()` はすべての割り込みをディセーブルし、`context` の値を返します。割り込みを再イネーブルするには、`alt_irq_enable_all()` を呼び出して、`context` パラメータに値を渡します。このようにして、割り込みは、`alt_irq_disable_all()` が呼び出される前の状態に戻されます。`alt_irq_enabled()` は、割り込みがイネーブルされている場合はゼロ以外の値を返すために、プログラムでこの関数を使用すれば割り込みの状態を確認できます。



最大割り込みレイテンシは、割り込みがディセーブルされている時間によって増加するため、割り込みのディセーブルは可能な限り短時間にする必要があります。

## C の例

以下の C コードの例は、ISR に対して HAL API を使用する場合に必要  
なプロセスを理解するのに役立ちます。

以下の例は、4 ビット PIO ペリフェラルをプッシュボタンに接続した  
Nios II システムをベースにしています。この場合、IRQ はボタンが押さ  
れたときに生成されます。ISR コードは、PIO ペリフェラルのエッジ・  
キャプチャ・レジスタを読み取り、その値をグローバル変数に格納しま  
す。グローバル変数のアドレスは、context ポインタを介して ISR に渡さ  
れます。

以下のコードは、ボタン PIO からの割り込みを処理する ISR の例を示し  
ます。

### 例：ボタン PIO IRQ を処理する ISR

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context, alt_u32 id)
{
    /* context ポインタを整数ポインタにキャストします。 */
    volatile int* edge_capture_ptr = (volatile int*) context;
    /*
     * ボタン PIO のエッジ・キャプチャ・レジスタを読み取ります。
     * 値を保存します。
     */
    *edge_capture_ptr =
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
    /* エッジ・キャプチャ・レジスタに書き込み、リセットします。 */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
    /* ボタン PIO の割り込み機能をリセットします。 */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
}
```

以下のコードは、ISR を HAL に登録するメイン・プログラムのコード例  
を示します。

### 例：HAL へのボタン PIO ISR の登録

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

...
/* エッジ・キャプチャの値を保持するグローバル変数を宣言します。 */
volatile int edge_capture;
...
/* 割り込みハンドラを登録します。 */
alt_irq_register(BUTTON_PIO_IRQ,
                (void*) &edge_capture,
                handle_button_interrupts);
```

このコードに基づいて、以下の実行フローが可能になります。

1. ボタンが押され、IRQ が生成されます。
2. HAL 例外ハンドラが起動され、`handle_button_interrupts()` ISR をディスパッチします。
3. `handle_button_interrupts()` が割り込みを処理して戻ります。
4. `edge_capture` の値を更新して、通常のプログラム動作を継続します。



その他にも、プロジェクト・テンプレート例の `count_binary` など、ISR の実装方法を示すソフトウェアの例が Nios II 開発キットとともにインストールされています。

## 高速 ISR 処理

ISR の性能を最大限に高めるには、以下のガイドラインに従ってください。

- 例外コードを最速で実行するには、例外アドレスを高速メモリ・デバイスにマップします。例えば、低速 SDRAM よりも、待ち状態がゼロのオンチップ RAM の方が適しています。例外アドレスはシステム生成時に決定されるため、この設定はソフトウェアでは選択できません。ただし、例外アドレスは、容易に変更可能な Nios II CPU ハードウェアのプロパティです。
- また、ISR 関数も高速メモリ・セクションにマップする必要があります (4-34 ページの「メモリの使用」を参照)。
- 通常、ISR 内部での長時間に及ぶ計算の実行は避けてください。

HAL ISR サービスは、ISR を登録するための使いやすい汎用フレームワークを提供しており、ほとんどのアプリケーションで利用できます。アプリケーションの性能が特に必要な場合、例えば、異なる割り込み優先順位方式を実装するために、`alt_irq_entry` または `alt_irq_handler` を置き換えて、システム性能を向上させることができます。ただし、これらのルーチンを置き換えるには、高度な専門知識と相当な労力が必要です。



システムが割り込みレイテンシに耐えるようにハードウェア・デザインを変更すれば、必要な労力も少なくて済みます。

## ISR の性能データ

このセクションでは、Nios II プロセッサでの ISR 処理に関連する性能データを示します。重要な以下の 3 つの測定基準によって、ISR の性能が決定されます。

- 割り込みレイテンシ - 割り込みが最初に生成されてから、プロセッサが例外アドレスの最初の命令を実行するまでの時間。
- 割り込み応答時間 - 割り込みが最初に生成されてから、プロセッサが ISR の最初の命令を実行するまでの時間。
- 割り込み回復時間 - ISR の最後の命令が通常の処理に復帰するまでに要する時間。

Nios II プロセッサは高度にコンフィギュレーション可能なため、各測定基準に対する単一の標準値は存在しません。このセクションでは、以下の仮定に基づいて、各 Nios II コアのデータ・ポイントを示します。

- すべてのコードとデータがオンチップ・メモリに格納されていること。
- ISR コードが命令キャッシュ内に存在しないこと。
- テストに使用するソフトウェアは、アルテラが提供する HAL システム・ライブラリの例外ハンドラ・ルーチンをベースにしていること。
- コードは、コンパイラ最適化レベル「-O3」または高度な最適化を使用してコンパイルされていること。

表 6-1 に、各 Nios II コアの割り込みレイテンシ、応答時間、および回復時間を示します。

コア	レイテンシ	応答時間	回復時間
Nios II/f	8	129	78
Nios II/s	8	146	165
Nios II/e	15	362	260

表 6-1 の注：


- (1) 数値は CPU クロック・サイクル単位で測定した時間を示します。

実際に測定した結果は、以下の主な要因によって、大幅に異なることがあります。

- 例外アドレスおよび ISR コードが存在するメモリ。表 6-1 の数値は、オンチップ・メモリの使用に基づきます。低速オフチップ・メモリを使用すると、測定結果も低速になります。
- コンパイラ最適化レベル。上記の結果は、レベル「-O3」に基づきます。レベル「-O2」を使用しても同様の結果が得られます。ただし、最適化を完全に排除すると、割り込み応答時間が大幅に増加します。

- Nios II コア。Nios II/f コア (高性能化指向設計) は、性能で Nios II/e コア (小型化指向設計) を上回ります。
- 例外ハンドラ・ルーチン。HAL システム・ライブラリが提供する汎用 IRQ ハンドラは、C で記述されており、きわめて広範囲のアプリケーションに対応するように設計されています。IRQ ハンドラをアプリケーションの正確なニーズに合わせて設計すれば、応答時間を大幅に短縮することが可能です。

割り込みレイテンシは、割り込みが CPU のパイプラインのどの部分に挿入されているかによって異なる場合があります。ISR コードが命令キャッシュに常駐していれば、ISR の性能が改善されます。

 デフォルトでは、HAL システム・ライブラリは、ISR をディスパッチするときに割り込みをディセーブルしますが、頻繁に割り込みを生成するシステムでは、これによって ISR の性能が著しく影響を受けることがあります。

## ISR 使用のデバッグ

ISR 内にブレークポイントを設定することにより、Nios II IDE で ISR をデバッグできます。ブレークポイントに到達すると、デバッガはプロセッサを完全に停止させます。ただし、その間もシステムのハードウェアは動作し続けます。したがって、プロセッサが停止している間に、他の IRQ が無視されるのは避けられません。デバッガを使用して ISR コード内をステップ移動できますが、一般にプロセッサを通常の実行に復帰させるまでの間に、その他の割り込み駆動デバイス・ドライバのステータスは無効になります。システムを既知の状態に戻すには、プロセッサをリセットする必要があります。

シングル・ステップの間、`ipending` レジスタ (`ctl14`) は、すべてゼロにマスクされます。このマスク処理により、プロセッサは、コードのシングル・ステップ中にアサートされた IRQ を処理しなくなります。その結果、`ipending` レジスタを読み込む例外ハンドラ・コードの部分 (つまり、`_irq_entry` または `alt_irq_handler()`) をシングル・ステップしても、コードがペンディング中の IRQ を検出することはありません。このブレークポイントが、ソフトウェア例外のデバッグに影響を与えることはありません。例外ハンドラがすでに `ipending` を使用して、例外を発生した IRQ を特定しているため、ユーザは ISR コード内にブレークポイントを設定する (さらに、シングル・ステップで移動する) ことが可能です。

## ISR の記述に関する提案の要約

このセクションでは、HAL フレームワーク用の ISR の記述に関する提案事項を要約します。

- HAL API が提供する `alt_irq_register()` 関数を使用して、ISR を登録します。
- ISR 関数は、プロトタイプ `void isr (void* context, alt_u32 id)` に適合するように記述します。
- ISR 内部で実行される処理量を最小にします。
- 低速処理タスクは、ISR から復帰するまで延期します。ISR はメッセージ引渡しメカニズムを使用して、低速処理タスクを実行するよう外部に通知できます。
- ISR の内部では、`printf()` などの C 標準ライブラリ I/O 関数は使用しないこと。
- `alt_irq_interruptible()` 関数および `alt_irq_non_interruptible()` 関数を使用して、ISR コードの各部分で優先順位の高い ISR をイネーブル（およびディセーブル）できます。ISR が非常に短い場合、優先順位の高い割り込みを再イネーブルしても、オーバーヘッドが発生するため効果的でないことがあります。
- 最速の実行性能を実現するには、例外ハンドラと ISR コードを高速メモリ・デバイス内のメモリ・セクションに配置します。

