AN54460

PSoC 3 および PSoC 5LP の割込み

著者: Vivek Shankar Kannan

関連プロジェクト: あり

関連部品ファミリ: すべての PSoC 3 および PSoC 5LP ファミリ

ソフトウェア バージョン: PSoC Creator™ 4.1 Update 1 以降

関連アプリケーション ノート: AN90799、AN90833、AN60630、AN89610

本アプリケーション ノートの最新バージョン、または関連プロジェクト ファイルを入手するために、http://www.cypress.com/AN54460 へアクセスしてください。

AN54460 は、PSoC® 3 および PSoC 5LP 内の割込みアーキテクチャを説明し、またサンプル プロジェクトを用いてその PSoC Creator™ IDE でのコンフィギュレーションを説明します。再入可能関数の実行、割込みコードの最適化、割込みレイテンシ、およびデバッグ技術などの高度な割込みのトピックについても説明します。

目次

1 はじめに

割込みは、どのような組込みアプリケーションにとっても重要な要素です。CPU を特定のイベントの発生に対して継続的にポーリングする負担から解放し、かわりにそのイベントが発生した時だけ CPU に通知します。PSoC などのようなシステムオンチップ (SoC) アーキテクチャでは、割込みは内蔵ペリフェラルの状態を CPU に知らせるためによく使用されます。

AN54460 は、PSoC 3 および PSoC 5LP の割込みアーキテクチャについて紹介します。また、割込みがこれらの PSoC デバイスに対応した開発ツールである PSoC Creator IDE でどのようにサポートされるかも示します。高度な割込み概念について詳細に説明します。いくつかのサンプル プロジェクトを本アプリケーション ノートに添付して、各種割込みの使用事例を示します。

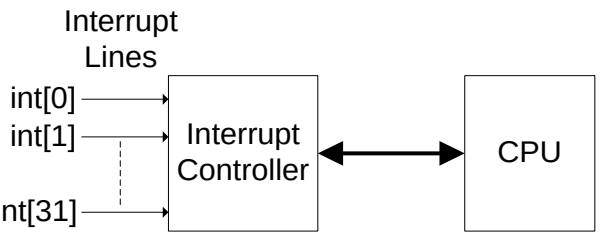
本アプリケーション ノートでは、ユーザーが少なくとも PSoC Creator IDE の基本的な使用方法を知っていることを前提としています。PSoC Creator を初めてご使用になる場合は PSoC Creator のホームページを参照してください。また、それぞれ PSoC 3 と PSoC 5LP のデバイスを説明する「AN54181 - Getting Started with PSoC 3」および「AN77759 - Getting Started with PSoC 5LP 」のアプリケーション ノートを参照し、簡単なプロジェクト使用して IDE ツールを参照してください。

その他の PSoC 製品ファミリの割込みアーキテクチャを説明するアプリケーション ノートも用意されます。PSoC 4 と PSoC 1 デバイスについては、本アプリケーション ノートに相当する「AN90799 – PSoC 4 Interrupts」および「AN90833 – PSoC 1 Interrupts」のアプリケーション ノートをご覧ください。

2 PSoC 割込みアーキテクチャ

ここでは、PSoC 3 および PSoC 5LP の割込みアーキテクチャの概要について説明します。図 1 は簡単なブロックダイヤグラムを示します。

図 1. PSoC 3 と PSoC 5LP の割込みアーキテクチャ



PSoC 3 および PSoC 5LP には、int[0]～int[31]の 32 本の割込みラインがあります。各割込みラインには、0 が最高優先順位である 8 段階の優先順位 (0～7) の内から 1 つを割り当てられます。各割込みラインには、割込みコードの開始アドレスを示す割込みベクタ アドレスが割り当てられます。CPU は、割込み要求を受け取った後、このアドレスへ分岐します。割込みコードは割込みサービス ルーチン (ISR) と呼ばれます。

割込みコントローラーは、割込みラインと CPU 間のインターフェースとして機能します。割込みコントローラーは、割込み要求信号と共に割込みラインの割込みベクタ アドレスを CPU に送信します。割込みコントローラーは、割込みの出入り状態で CPU からの確認信号も受信します。割込みコントローラーは、複数の割込みラインから要求がある場合、割込みの優先順位を決めます。

割込み動作の詳細については、PSoC 3 または PSoC 5LP のテクニカル リファレンス マニュアル (TRM) を参照してください。

2.1 PSoC 割込みの機能

PSoC 3 および PSoC 5LP は、他の従来のマイクロ コントローラーがサポートしていない以下の強化された割込み機能を提供します。

- **コンフィギュレーション可能な割込みベクタ アドレス:** PSoC により、割込みベクタ アドレスを動的に設定できます。割込みが発生したとき、CPU の実行を任意の ISR コードへ直接分岐させられます。これは、従来のマイクロコントローラーと比べて PSoC 内の割込み実行のレイテンシを低減させます。

  従来のマイクロコントローラーでは、割込みベクタ アドレスは各割込みライン用に固定されます。通常、「JUMP」命令はその固定アドレスに配置されており、実際の ISR コードへ CPU の実行を分岐させます。
- **フレキシブルな割込みソース:** 従来のマイクロコントローラーでは、割込みソースは各割込みラインに固定設定されます。PSoC では、各割込みラインの割込みソースを柔軟に選択できます。このフレキシブルなアーキテクチャにより、どのデジタル信号も割込みソースに選択できます。

PSoC デバイスは、レベル トリガーとエッジ トリガー両方の割込みに対応します。割込みをレベル トリガーあるいはエッジ トリガーのどちらへ分類するかは、割込みソースで生成された割込み信号によります。図 2 および図 3 は、レベル トリガーおよびエッジ トリガーの割込みがどのように動作するかを示します。

図 2. レベル トリガーの割込み

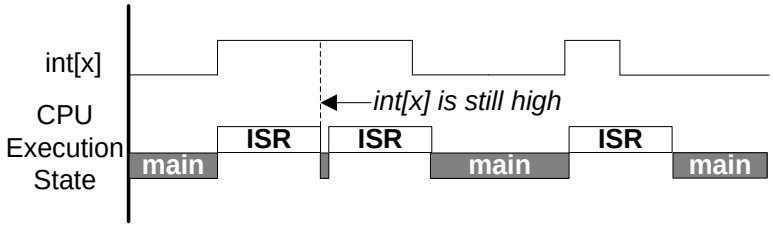
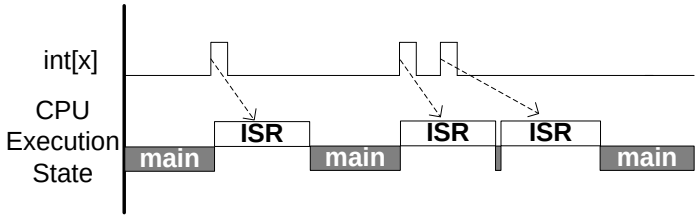


図 3. エッジ トリガーの割込み



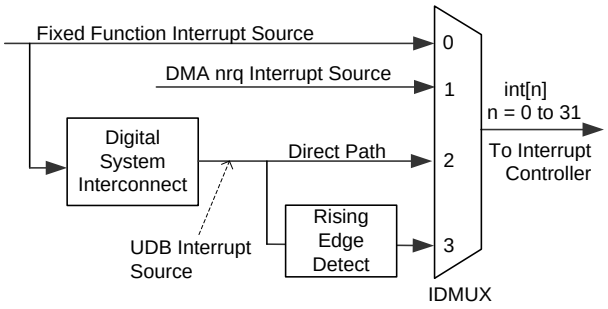
割込みラインが初期に非アクティブ (論理 LOW) であることを前提として、以下のイベント シーケンスは、レベル トリガーとエッジ トリガーの割込みがどのように処理されるかを説明します。

- 割込みラインの立ち上りエッジで、割込みコントローラーは割込み要求を送信します。すると、割込みラインは保留状態になります。つまり、その要求は CPU でまだ処理されていないということです。
- その後、割込みコントローラーは割込みベクタ アドレスを割込み要求信号とともに CPU に送信します。CPU が割込みラインの ISR を実行し始めると、割込みラインの保留状態はクリアされます。
- レベル トリガーの割込み: ISR を完了した後、割込みラインがまだ HIGH の場合、図 2 に示すように割込みラインが保留され、ISR が再実行されます。割込みラインが HIGH である限り、ISR は継続して実行されます。
- エッジ トリガーの割込み: ISR が実行されている間に、割込みラインに 1 つ以上の立ち上りエッジが発生すると、それらは 1 つの保留中の要求として記録されます。図 3 に示すように、現時点の ISR の実行が完了すると、保留中の割込みは再び処理されます。

エッジ トリガーの割込みはパルス信号であるため、パルス割込みとも呼ばれます。エッジ トリガーの割込みの最小パルス幅は 1 バス クロック サイクルです。PSoC は、割込みが立ち上りエッジのたびにトリガーされることを保証するための立ち上りエッジ検出ロジックを内蔵します。この機能については次の節で説明します。

図 4 に示すように、各割込みラインは 3 つの割込みソースのいずれかで駆動されます。マルチプレクサ ロジックは、各割込みラインのソースを選択します。

図 4. PSoC 3 および PSoC 5LP における割込みソース



割込みソースの詳細は以下のとおりです。

2.1.1 固定機能割込みソース

これらは内蔵ペリフェラルの前もって定義された割込みソースのセットです。例としては、固定機能タイマー、カウンター、ポート割込み制御ユニット (PICU) などからの割込み信号があります。PSoC 3 および PSoC 5LP デバイスにおける割込みソースの一覧については、付録 A を参照してください。

固定機能割込みソースのほとんどはレベル割込みです。これらの割込みの場合、ペリフェラルのステータス レジスタは、以下の 2 つの理由から、ISR 内で読み出されなければなりません。

1. ステータス レジスタは、何の条件が割込みを生成したか示します。例えば、PICU 割込みの場合、PICU ステータスレジスタの各ビットは 1 つのポート ピンに対応します。
2. ステータス レジスタを読み出すとステータス ビットがクリアされ、割込みラインは LOW の状態に戻ります。ステータス レジスタが ISR 内で読み出されない場合、ISR は連続して実行されます。

2.1.2 DMA nrq 割込みソース

PSoC 3 および PSoC 5LP 内の各ダイレクト メモリ アクセス (DMA) チャネルは、DMA 転送動作の終了時にパルス信号を生成します。この転送完了信号 (*nrq* 信号と呼ばれる) は、DMA のデータ転送が終わった後、割込みをトリガーするために使用できます。これはエッジ トリガーの割込みです。

2.1.3 UDB 割込みソース

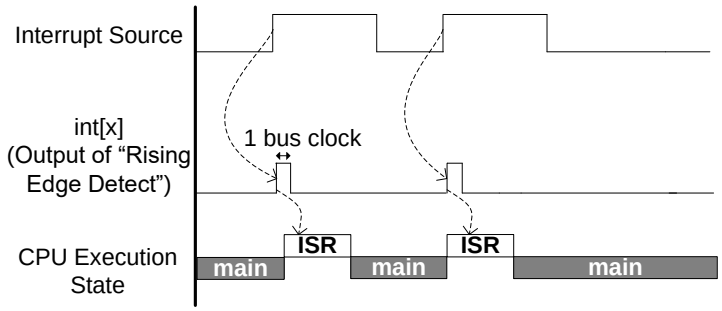
いかなるデジタル信号も、デジタル システム インターコネクト (DSI) を介して送信することで割込みソースとして設定できます。これらの割込みソースはほとんどが PsoC 内のユニバーサル デジタル ブロック (UDB) からのものであるため、UDB 割込みソースと広く呼ばれます。UDB は UART、SPI、I2C、タイマー、カウンター、PWM などの設定可能なデジタルペリフェラルの基本構成要素です。

固定機能の割込みソースは、3 ページの図 4 に示したように DSI インターフェースや専用配線を介しても送れます。

UDB 割込みソース用に 2 本のパスがあります。

1. 最初のパスは、UDB 割込みソースから割込みラインへの直接接続です (3 ページの図 4 内の直接パスを参照してください)。これはレベル トリガーの UDB 割込みに使用されます。例えば、データ バッファに読み出しデータがあることを示すために UART や SPI などの通信用ペリフェラルによって使用されます。
2. もう 1 つのオプションは、図 4 に示したように立ち上りエッジ検出ロジックを介して UDB 割込みソースを渡すことです。図 5 は、UDB 割込みソースからの立ち上りエッジ信号をパルス信号に変換する方法を示します。この機能は、エッジ トリガーされる割込みソースに使用されます。

図 5. エッジ トリガーUDB 割込みソース

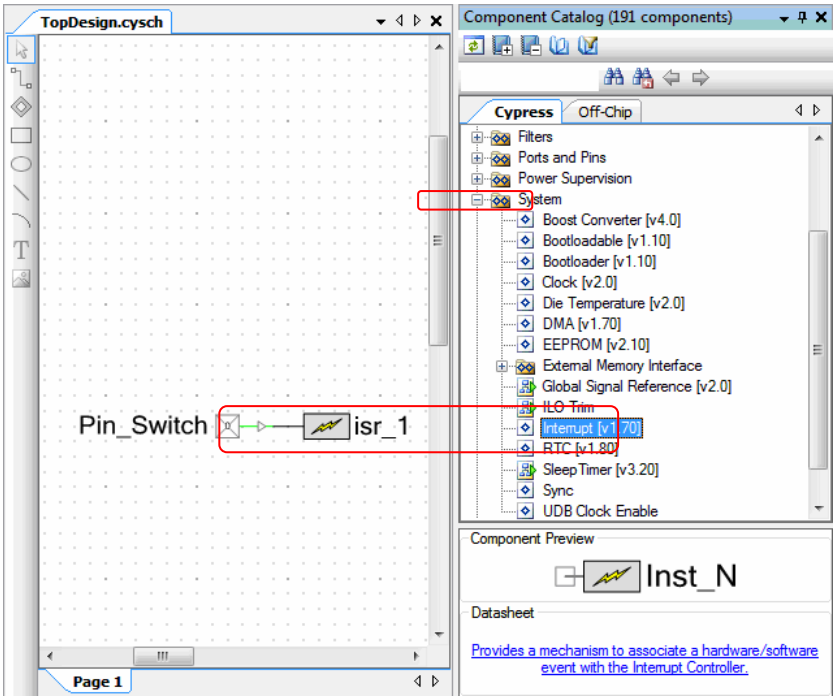


PSoC 割込み機能およびその動作を確認しました。以下の節では、PSoC CreatorIDE により PSoC 割込みの使用がいかに容易になるかを示します。

3 PSoC Creator での割込みサポート

PSoC Creator は、割込みをコンポーネントとして提供することでサポートします。割込みコンポーネントは、図 6 に示すようにコンポーネント カタログ ウィンドウ内の「**System**」タブの下に表示されます。割込みコンポーネントの各インスタンスは 1 つの割込みラインを使用します。プロジェクトの回路図では、割込みソース (この例では、ピンコンポーネント) を割込みコンポーネントに接続します。

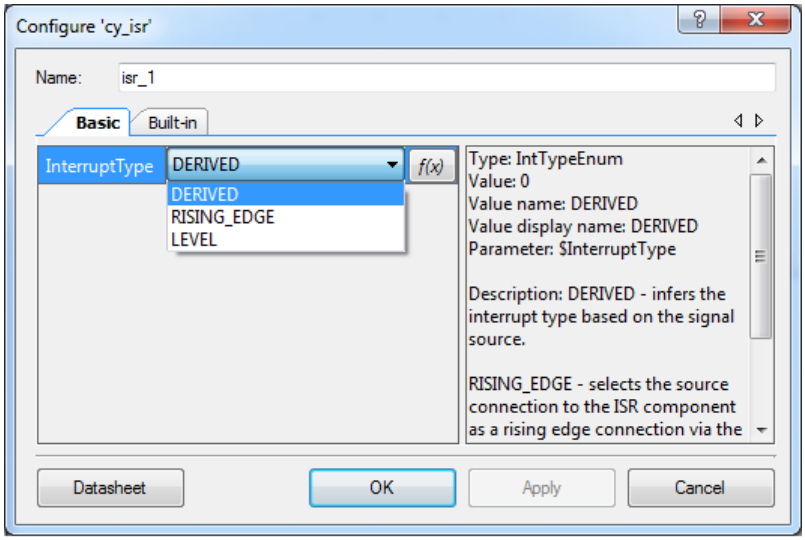
図 6. PSoC Creator での割込みコンポーネント



3.1 割込みコンポーネントのコンフィギュレーション

図 7 は、割込みコンポーネントのコンフィギュレーション ダイアログを示します。InterruptType パラメーターを使用して割込みソース用にマルチプレクサを設定します (3 ページの図 4 を参照してください)。

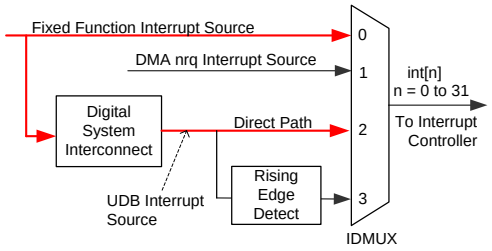
図 7. 割込みコンポーネントのコンフィギュレーション



InterruptType パラメーターの詳細は以下のとおりです。

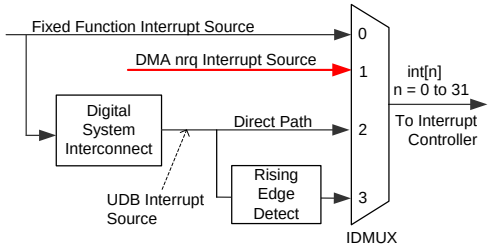
- **DERIVED:** このオプションでは、PSoC Creator は割込みソースの種類に応じてマルチプレクサを設定します。
  - 固定機能ブロック (付録 A を参照してください) からのデジタル出力信号の場合、割込みソースは割込みコントローラーに直接接続されます。図 8 に示すように、割込みソースの専用接続または DSI インターフェース経由の直接パスのどちらかを使用します (図 4 も参照してください)。

図 8. PSoC 3 および PSoC 5LP における固定機能ブロックの派生割込みの配線図



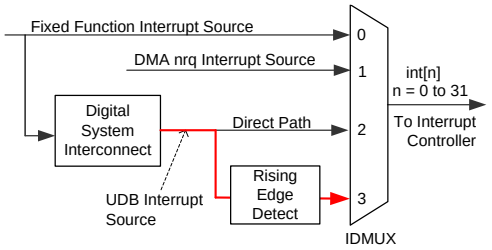
- 割込みソースが DMA nrq 信号である場合、図 9 に示すように DMA nrq 割込みソースの専用パスを選択します (図 4 も参照してください)。

図 9. PSoC 3 および PSoC 5LP における DMA の派生割込みの配線図



- UDB ベースのコンポーネントやその他のデジタル信号などの割込みソースの場合、図 10 に示すように DSI インターフェースの立ち上りエッジ オプションを選択し、割込みソースをエッジ トリガーの割込みとして扱います (図 4 および図 5 も参照してください)。

図 10. PSoC 3 および PSoC 5LP における UDB コンポーネントの派生割込みの配線図



- **RISING_EDGE:** このオプションでは、図 10 に示したように立ち上りエッジ検出ロジックを通るパスを介して割込みソースの信号を送ります。エッジでトリガーする割込みソースに対してこのオプションを選択してください。
- **LEVEL:** このオプションでは、6 ページの図 8 に示したように割込みソースを DSI インターフェース経由の直接パスを介して送ります。UART、SPI、タイマー、カウンター、PWM などの UDB ベースのコンポーネントのような、レベルでトリガーする割込みソースに対してこのオプションを選択してください。

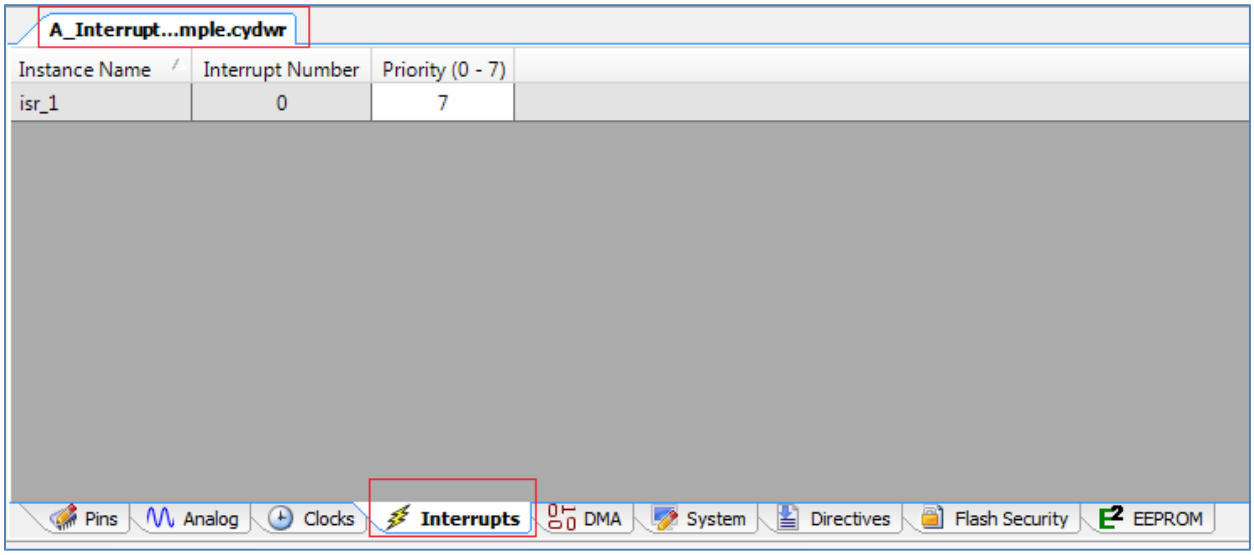
  注: UART や SPI などの UDB ベースのコンポーネントが生成する割込みに対しては LEVEL オプションのみを選択してください。これは、それらのコンポーネントは UDB FIFO バッファ ステータス信号を示すためにレベル割込みを生成するからです。深さ 4 バイトの FIFO バッファは、送信するデータ (Tx FIFO) または受信するデータ (Rx FIFO) を一時的に格納するために使用されます。

  Rx FIFO の場合、少なくとも 1 データ バイトが CPU によって読み出される限り、FIFO ステータス信号は HIGH です。DERIVED か RISING_EDGE オプションが選択された場合、バッファに 1 つ以上のバイトがあっても割込みは 1 回だけトリガーされます。割込みが発生するたびに 1 バイトだけが読み出されるように ISR が書かれている場合、バッファ内の追加データは失われる場合もあります。

3.2 割込み優先順位のコンフィギュレーション

図 11 に示すように、PSoC Creator プロジェクトの「Design Wide Resources」(設計全体のリソース) ウィンドウ (*project_name.cydwr*) 内の「**Interrupts**」タブには、すべての使用中の割込みの名前、優先順位、およびベクタ番号が表示されます。

図 11.*cydwr* ウィンドウ内の「**Interrupts**」タブ



cydwr ウィンドウを使用して割込みの優先順位を変更します。0 が最高の優先順位で、7 が最低の優先順位であることに注意してください。各割込みコンポーネントの割込みベクタ番号は、プロジェクトがビルドされたときに PSoC Creator で自動的に割り当てられます。

通常、ベクタ番号は PSoC Creator で自動的に選択されますが、手動でも変更できます。詳細については、割込みベクタ番号の割り当てを参照してください。

4 割込みプロジェクト

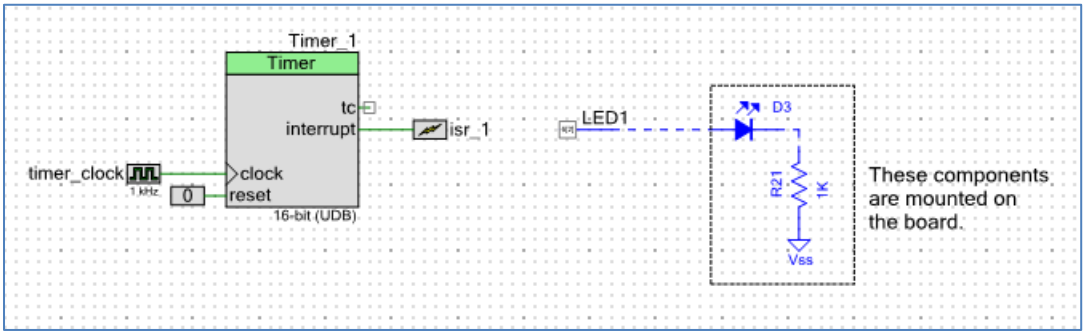
本アプリケーション ノートに添付された 4 つの PSoC Creator プロジェクトは、PSoC 3 と PSoC 5LP の基本的な割込み、ポート割込み制御ユニット (PICU)、低電圧検出 (LVD)、および SysTick 割込みを示します。SysTick は Cortex-M3 CPU の機能であり、PSoC 3 にはないことに注意してください。

4.1 簡単な割込みプロジェクト

ここでは、簡単な割込みベースの PSoC Creator プロジェクトを作成する手順を説明します。このプロジェクトの完成版は *A_InterruptExample* と名付けられ、本アプリケーション ノートに添付されています。

このプロジェクトの目的は、所望の周波数でピンをトグルする (したがって、LED を点滅させる) ために周期的な割込みを生成することです。周波数は、クロック コンポーネントとタイマー コンポーネントにより決められます。図 12 はプロジェクトの回路図を示します (サイプレス開発キット CY8CKIT-030／CY8CKIT-050 向けに開発されています)。プロジェクトは、KBA「Migrating project from CY8CKIT-001 to CY8CKIT-030 or CY8CKIT-050」で説明されているように他のキットに容易に適用できます。

図 12. プロジェクト *A_InterruptExample* の回路図

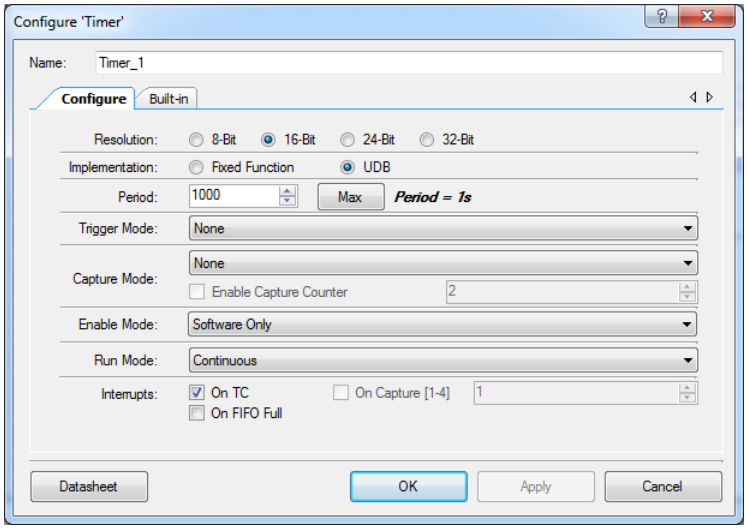


タイマー コンポーネントはダウン カウンターであり、ターミナル カウントはタイマーがその最小値 (0 値) に達した状態を示します。ターミナル カウントでは、タイマーが周期値でリロードされ、カウントを続けます。割込みは、ターミナル カウント状態が起きるたびに発生させられます。

タイマー コンポーネントからの割込み出力は割込みコンポーネントに接続されます。デジタル出力ピン LED1 は、タイマー ターミナル カウントのイベントが発生するたびに ISR コードでトグルされます。

割込み周波数は timer_clock 周波数とタイマー周期で制御されます。図 13 に示すようにタイマー コンポーネントを設定します。timer_clock 周波数が 1KHz で、1 周期が 1000 回である場合、1 秒に 1 回ターミナル カウント条件に達します。「Interrupt on TC」ボックスにチェックを入れ、「Implementation」が「UDB」であることを確認してください。他のすべてのパラメーターを初期設定のままにしてください。

図 13. タイマー コンポーネントのコンフィギュレーション



タイマー コンポーネントのデータシートから、割込み出力はタイマー ステータス レジスタが読み出されるまでは、アサートされたままです。つまり、割込みがレベル トリガーであるということです。5 ページの図 7 に示すように、InterruptType が LEVEL であるように割込みコンポーネントを設定します。

このプロジェクトには割込みが 1 つしかないため、割込みコンポーネントの優先順位をその初期値のままにできます (7 ページの図 11 を参照してください)。

4.1.1 割込みサービス ルーチンの作成

回路図が完成した後、PSoC Creator のメニュー内の Build > Generate Application に移動します。PSoC Creator はプロジェクトで使用されるコンポーネント用のソース ファイルおよびヘッダ ファイルを生成します。これにより、対応するソース ファイルに各割込みコンポーネント用のデフォルト割込みサービス ルーチン (ISR) が含まれます。この ISR 関数の名前は `CY_ISR(isr_name)` です。ISR コードを関数内のプレースホルダに書き、関連する `#include` ステートメントを追加する必要があります。

このプロジェクトでは、ISR コードは 2 つのタスクを行います。

- タイマー ステータス レジスタを読み出して割込みソースをクリアします。タイマー コンポーネントのヘッダ ファイルは ISR ソース ファイルに含まれていなければなりません。
- ピン コンポーネントをトグルして LED を点滅させます。ピン コンポーネントのヘッダ ファイルは ISR ソース ファイルに含まれていなければなりません。

コード 1 が示すように、PSoC Creator は割込みソース ファイルの先頭にヘッダ ファイルを含むための領域を用意しています。

コード 1. ISR ソース ファイル内のヘッダ コード

```
/********************************************
**
* Place your includes, defines and code here
********************************************
*/
/* `#START isr_1_intc` */
/* Timer component header file */
#include "Timer_1.h"
/* LED1 pin component header file */
#include "LED1.h"
/* `#END` */
```

ヘッダ コードは、`#START` と `#END` 行の間にのみ書いてください。ビルド プロセス中では、PSoC Creator はこれらの 2 行間に書かれたコードのみ保存します。これら 2 行の外に書かれたコードは、プロジェクトが再ビルドされるたびに削除されます。

コード 2 がこのプロジェクトの ISR コードを示します。このコードは、ISR 関数内に用意されたプレースホルダ領域に書かれます。

コード 2. ISR ソース ファイル内の ISR コード

```
CY_ISR(isr_1_Interrupt)
{
/* Place your Interrupt code here */
/* `#START isr_1_Interrupt` */
/* Read Timer status register to bring the interrupt line low
*/
Timer_1_ReadStatusRegister();
/* Toggle the LED */
LED1_Write(~LED1_Read());
/* `#END` */
}
```

ISR コードは、`#START` および `#END` 行の間にのみ書いてください。ビルド プロセス中では、PSoC Creator はこれらの 2 行間に書かれたコードのみ保存します。これら 2 行の外に書かれたコードは、プロジェクトが再ビルドされるたびに削除されます。

4.1.2 Main コードの完成

コード 3 が示すように、main コードでは、コンポーネントを初期化し、開始します。

コード 3. 簡単な割込みプロジェクト *main.c*

```
void main()
void main()
{
    /* Initialize the interrupt and timer*/
    isr_1_Start();
    Timer_1_Start();

    CyGlobalIntEnable; /* enable Global interrupts */

for(;;)
    {

       /* Do nothing in the main loop; code to do something is in
the ISR. */
    }
}
```

コード 3 では、`CYGlobalIntEnable` マクロは CPU に割込み要求信号を生成するように割込みコントローラーを設定します。また、CPU は要求信号を受け取るように設定します。このマクロは選択した PSoC 3 または PSoC 5LP デバイスファミリに応じて、自動生成された *CyLib.h* ファイル内で適切に定義されます。マクロを使用して、複数コンパイラのツールチェーンおよびデバイス ファミリにわたってコードの移植性を確保します。

`isr_1_Start()` 関数は *isr_1* 割込みコンポーネントを初期化して有効にします。特に、以下のことを行います。

- 割込みベクタ アドレスを割込みコンポーネント ソース ファイルで用意されているデフォルト ISR 関数のアドレスにセットします。
- cydwr ウィンドウで割り当てられた優先順位の値に応じて割込みの優先順位を設定します。
- 割込みを有効にします。

4.1.3 CY_ISR キーワードの重要性

割込みソース ファイルは `CY_ISR` マクロを使用して ISR 関数を定義します。各コンパイラは、関数を ISR として認識するためには特定の関数定義のフォーマットを必要とします。自動生成された *cytypes.h* ファイルで定義されたマクロは、コンパイラから独立した ISR 定義のフォーマットを提供します。

同様に、`CY_ISR_PROTO` マクロは、ISR 関数のプロトタイプを宣言します。宣言は割込みコンポーネントのヘッダ ファイルにあります。例えば、割込みコンポーネントの *isr_1* は、*isr_1.h* のヘッダ ファイル内に以下の関数プロトタイプ宣言を持ちます。

```
CY_ISR_PROTO(isr_1_Interrupt);
```

このマクロも *cytypes.h* 内で定義され、選択したコンパイラに応じて適切に定義されます。

これで、簡単な割込みベースのプロジェクトを完了します。他の割込みベースのプロジェクトでも同じ手順に従ってください。以下の節では、PSoC Creator の割込みに使用する高度な機能のいくつかを説明します。

4.2 PICU 割込みプロジェクト

これは 2 番目のサンプル プロジェクトであり、プロジェクト *B_PICU* として本アプリケーション ノートに添付されています。これは以下のトピックについて説明します。

1. ポート割込み制御ユニット (PICU) およびデジタル入力ピンに対応する PICU の割込みの使用。
2. PSoC Creator で生成されたデフォルト ISR を使用する代わりにユーザー定義関数を ISR にする方法。

このプロジェクトでは、スイッチが *押される*と、ピン (およびLED) は切り替わりますが、スイッチが解放されるときは切り替わりません。この機能を正しく実装するために、スイッチ デバウンス機能 (跳ね返り防止機能) が組み込まれます。2 つのスイッチは監視され、跳ね返りが防止されます。それぞれのスイッチは、対応する個別のLEDを持ちます。これらの要件を実装する方法がたくさんあります。このプロジェクトでは、割込みベースの実装を使用します。

このプロジェクトはPSoCにおける割込み固有のGPIOピンの使用にのみ対応します。サンプル プロジェクトによりGPIOのコンフィギュレーション機能の詳細を理解するには、アプリケーション ノート「AN72382 – Using PSoC 3 and PSoC 5LP GPIO Pins」を参照してください。

このプロジェクトはサイプレス開発キットCY8CKIT-001 向けに開発されていますが、KBA「Migrating project from CY8CKIT-001 to CY8CKIT-030 or CY8CKIT-050」で説明されているように他のキットに容易に適用できます。

4.2.1 PICU 割込み

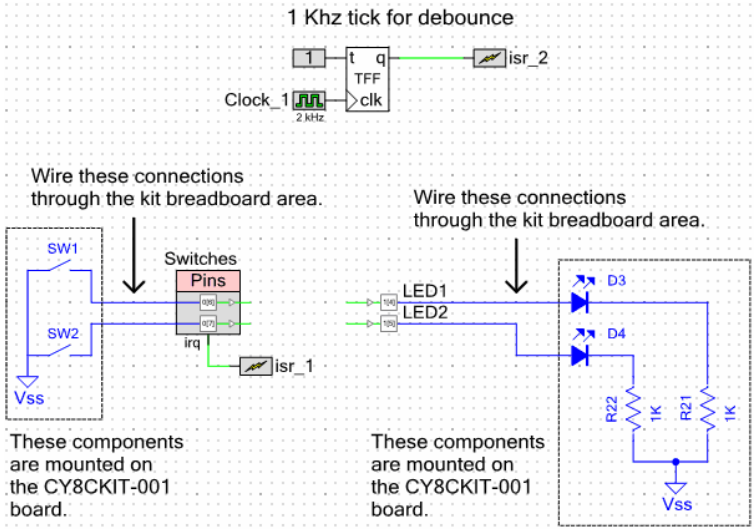
スイッチが押されたことを検出するために、ポート割込み制御ユニット (PICU) 割込みを使用してください。各ポート(8 本の I/O ピンから構成されている) は 1 つの PICU 割込みに対応します。8 本の I/O ピンはそれぞれ、立ち下りエッジ、立ち上りエッジ、あるいは両方のエッジで PICU 割込みを生成するように個別に設定できます。

設定されたイベントがピンで発生すると、8 ビット PICU ステータス レジスタの対応するビットはセットされます。PICU 割込みはステータス レジスタ ビットの論理和出力です。PICU ステータス レジスタを読み出して、どのポートが割込みを生成したかを判定できます。

4.2.2 プロジェクトの回路図

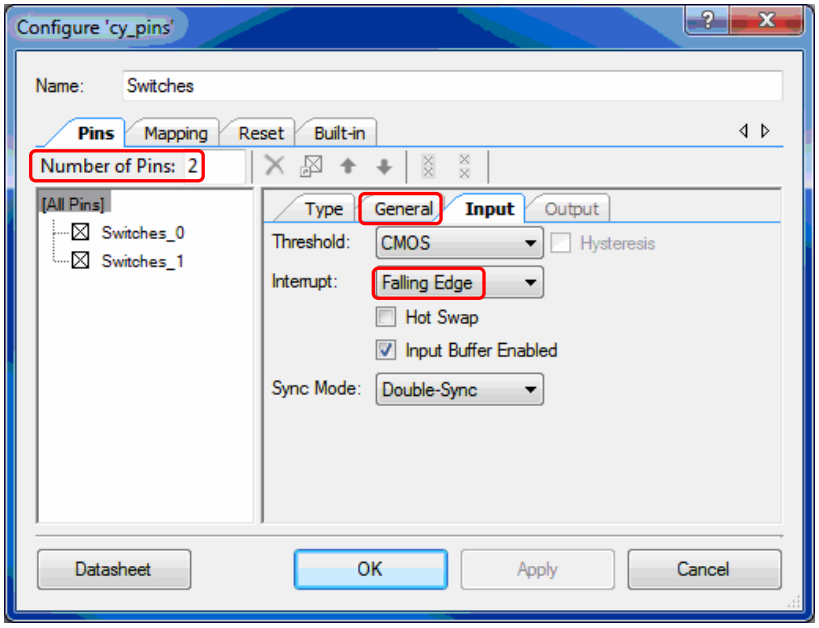
プロジェクトの回路図は図 14 に示されます。開発キット上の SW1 と SW2 スイッチは、「Switches」と名付けられた 1 つのデジタル入力ピンコンポーネントと見なされた 2 本の入力ピンに接続されます。各入力ピンからの PICU 割込み信号は割込みコンポーネント「isr_1」に接続されます。2 番目の割込みコンポーネント「isr_2」は、スイッチ デバウンスに使用される 1kHz ティック割込みを生成するために、クロック コンポーネントおよび TFF (T 型フリップフロップ) コンポーネントに接続されます。

図 14. プロジェクト *B_PICU* の回路図



各スイッチは、押されると瞬間的にグランドに接続するプッシュ ボタンです。そのため、「**General**」タブでピン コンポーネントを初期状態が論理 HIGH であるように抵抗プルアップのドライブ モードに設定されます。コンポーネントは、図 15 に示すように立ち下りエッジで PICU 割込みを生成するように設定されます。

図 15. PICU 割込みのコンフィギュレーション



4.2.3 プロジェクトのファームウェア

簡単な割込みプロジェクトと同様に、*main.c* 内のコードは非常に単純なものです。このコードは単に 2 つの割込みコンポーネントを開始し、グローバル割込みを有効にします。main ループでは何もせず、すべての動作は ISR で行われます。

このプロジェクトの違う点は、ユーザーが自動生成された割込みコンポーネントの ISR を使用しないことです。そのかわりに、次の節で示すようにユーザー独自の ISR を書きます。*InterruptRoutines.h* および *InterruptRoutines.c* ファイル内の添付プロジェクト *B_PICU* のコードを確認します。2 つの ISR は `PICU_ISR` および `Tick_ISR` と名付けられています。ティック ISR はミリ秒ごとに 1 回実行しますが、通常はアイドル状態です。`PICU ISR` は、いずれかのスイッチが押されたときに実行し、対応する変数でデバウンス タイムアウトを開始します。その後、ティック ISR はその変数をカウント ダウンし、0 に達すると、対応ピンをトグルします。

デバウンス タイマーがゼロ以外の場合、`PICU ISR` は何もしません。つまり、デバウンス期間内の後続の立ち下りエッジが無視されるということです。

スイッチのデバウンスの詳細については、アプリケーション ノート「AN60024 – Switch Debouncer and Glitch Filter with PSoC」を参照してください。

4.2.4 ユーザー独自の ISR を書く

独自の ISR を書く主な理由は、図 16 に示すように、プロジェクトが PSoC デバイス ファミリ間で移植される時、ユーザーの ISR コードの複数のコピーを割込みコンポーネントのソース ファイルに加える必要があることを避けるためです。同じコードの複数のコピーを保守するという問題が発生する可能性があるためです。

そのかわりに、図 17 に示すように、ISR を自動生成コードのかわりにユーザー独自のソース ファイル内で書けます。その結果、ISR の変更は、任意の PSoC ファミリから選択したデバイスに適用されます。

図 16. 複数の個別 ISR ソース ファイル

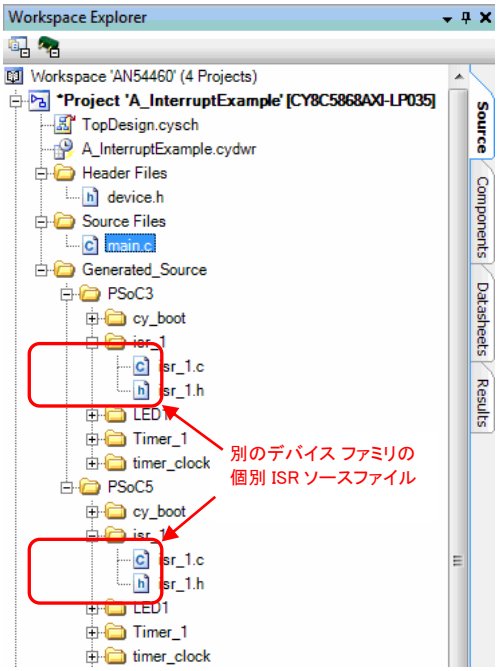
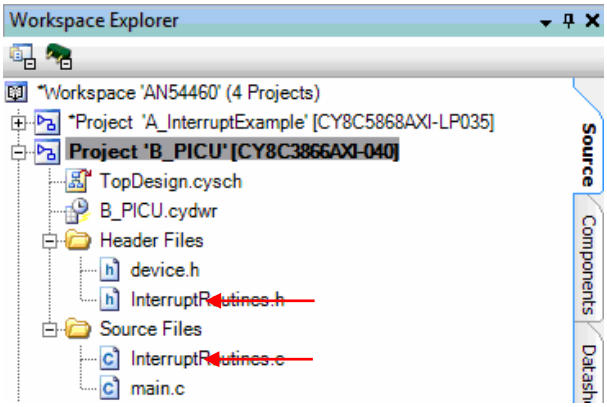


図 17. 共通の ISR ソース ファイル



ユーザー独自の関数 (例えば、`MyCustomISR`) を割込みコンポーネント *isr_1* の ISR にするためには、以下の手順を行ってください。

1. `CY_ISR_PROTO` マクロを使用して関数を宣言します。

   ```
   CY_ISR_PROTO(MyCustomISR);
   ```

2. `CY_ISR` マクロを使用して関数を定義します。

   ```
   CY_ISR(MyCustomISR)
   {
     /* ISR code goes here */
   }
   ```

3. *main.c* のスタートアップ コードでは、API 関数 `isr_1_StartEx()` に呼び出しを加えます。`isr_1_StartEx()` にはユーザーの ISR 関数用のパラメーターがあることを除いて、この関数は API 関数 `isr_1_Start()` に似ています。

   ```
   isr_1_StartEx(MyCustomISR);
   ```

   `isr_1_StartEx()` 関数は、割込みベクタ アドレスをユーザーの関数にセットします。

この手法の例については、本アプリケーション ノートに添付されているプロジェクト *B_PICU* を参照してください。以下の節では、容易に参照できるようにコードが用意されます。

プロジェクトの main() 関数は以下のとおりです。

```
#include <device.h>

/* Header file containing the custom ISR prototypes */
#include "InterruptRoutines.h"

void main()
{
    /* Initialize the two custom defined ISRs */
    isr_1_StartEx(PICU_ISR);
    isr_2_StartEx(Tick_ISR);

    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        /* Do nothing in the main loop; code to do something
          is in the ISRs */
    }
}
```

このプロジェクトの PICU_ISR は以下のとおりです。

```
CY_ISR(PICU_ISR)
{
        /* copies of PICU registers */
        uint8 CYDATA temp_stat;

        /* read the PICU interrupt status register, with a clear on read */
        temp_stat = Switches_ClearInterrupt();

        /* Process the PICU event on SW1 only if any
           ongoing debounce period has timed out */
        if(((temp_stat & SW1_MASK) != 0) && (switch_1_timeout == TIMED_OUT))
        {
                /* reset the debounce timer for this button */
                switch_1_timeout = DEBOUNCE_TIME;
        }

        /* Process the PICU event on SW2 only if any
           ongoing debounce period has timed out */
        if(((temp_stat & SW2_MASK) != 0) && (switch_2_timeout == TIMED_OUT))
        {
                /* reset the debounce timer for this button */
                switch_2_timeout = DEBOUNCE_TIME;
        }
}
```

`Switches_ClearInterrupt()` API は、PICU ステータス レジスタを読み出し、対応するピンが割込みを発生させるかどうかを確認するために使用します。ポート ピンで立ち下りエッジ変化が検出され、対応するデバウンス タイマーがタイムアウトした場合、デバウンス タイマーが再起動されます。デバウンス タイマーは、`switch_1_timeout` および `switch_2_timeout` 変数を個別に使用してソフトウェアで実装されます。デバウンス期間は、*InterruptRoutines.h* ファイル内のマクロ定義 `DEBOUNCE_TIME` を修正することで設定できます。

Tick_ISR と名付けられた 1ms の周期 ISR は、デバウンス ロジックをポート ピンに実装するために使用されます。ISR はデバウンス期間の終わりに出力ピンを切り替えます。プロジェクトの Tick_ISR コードは以下のとおりです。

```
CY_ISR(Tick_ISR)
{
        if(switch_1_timeout != 0)
        {
                /* Come here if an edge was detected by the PICU.
                 Decrement and check timeout */
                if(--switch_1_timeout == TIMED_OUT)
                {
                        /* Timed out. Toggle LED only on a switch press (= 0). */
                        if((Switches_Read() & SW1_MASK) == 0)
                        {
                                LED1_Write(~LED1_Read()); /* toggle the LED */
                        }
                }
        }
        /* repeat for switch 2 */
        if(switch_2_timeout != 0)
        {
                if(--switch_2_timeout == TIMED_OUT)
                {
                        if((Switches_Read() & SW2_MASK) == 0)
                        {
                                LED2_Write(~LED2_Read()); /* toggle the LED */
                        }
                }
        }
}
```

4.3 LVD 割込みプロジェクト

これは 3 番目のサンプル プロジェクトであり、プロジェクト *C_LVD* として本アプリケーション ノートに添付されています。以下のトピックについて説明します。

1. 割込み生成用のグローバル信号リファレンス コンポーネントの使用
2. main コードと ISR の間で通信するためのセマフォ、フラグまたは変数の使用

このプロジェクトは、デバイスの Vdd ピンに印加される電圧を監視します。これは、電圧がトリガー レベルを下回ると LED をオンにし、電圧がそのレベルを上回ると LED をオフにします。サイプレス開発キット CY8CKIT-001 は、この機能をテストするための調整可能なレギュレータを備えます。調整可能なレギュレータを使用するために、ジャンパ J2、J3、J4、J5、J6、J7 を対応する VADJ 位置に移動します。 図 18 に示すように、ポテンショメーターR11 (キットの旧バージョンでは R10) を回してレギュレータ出力電圧を調整します。電圧を監視するには、電圧計を GND と VADJ テスト ポイントに接続します。

図 18. CY8CKIT-001 上の調整可能なレギュレータ ポテンショメーターおよびテスト ポイント

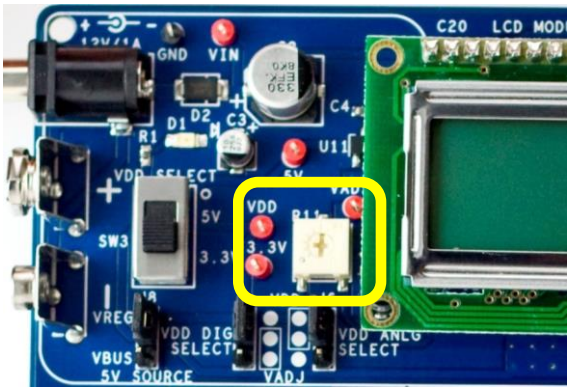


図 19 に示すように、グローバル信号リファレンス コンポーネントを使用して LVD の監視を可能にします。

図 19. プロジェクト *C_LVD* の回路図

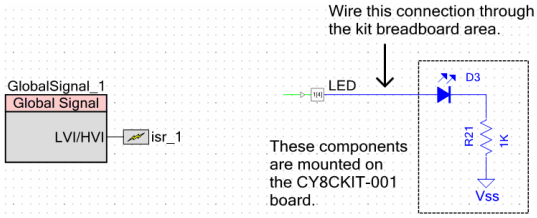
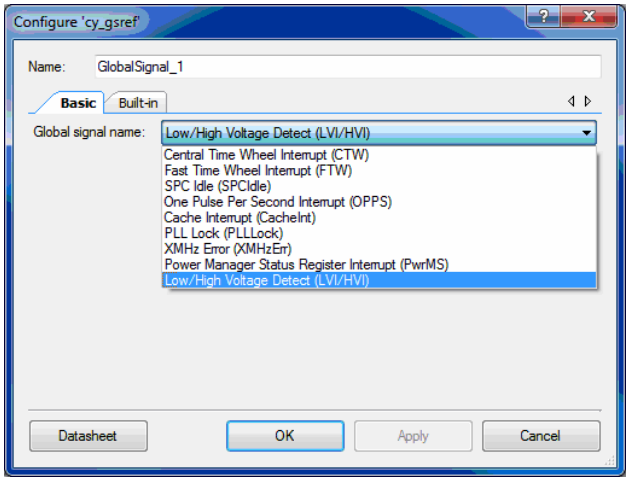


図 20 に示すように、グローバル信号リファレンス コンポーネントにより、PSoC で多くのシステム レベルの割込みを容易に使用できます。

図 20. PSoC 3 および PSoC 5LP におけるグローバル信号リファレンス コンポーネントのコンフィギュレーション



プロジェクトでは、ISR と main コード間で通信するためにセマフォ、フラグ、または変数を使用する方法も説明します。これは、組込みシステム デザインでの一般的な方法であり、タスクを ISR から main コードに渡すため、CPU が ISR に使用した時間を減少させます。main コードと ISR が同じファイルにあれば、変数はスタティック型のグローバルと定義されます[1]。

```
static volatile CYBIT isr_flag = 0; /* semaphore between ISR and main() */
```

変数は、ISR でセットされます。LVD_ISR コードは以下のとおりです。

```
CY_ISR(LVD_ISR)
{
    /* Disable the LVD interrupts. This causes this ISR to execute only once
       when the voltage is below the trigger level, otherwise the interrupt
       occurs continuously.
        */
    isr_1_Disable();

    isr_flag = 1;
}
```

V_{DD} 電圧がトリガー レベルより低い間、割込みは連続的に生成されます。効率的にするために、ISR を 1 回だけ実行したいので、ISR で割込みを無効にします[2]。main コードはフラグを監視します。フラグが ISR でセットされると、main コードは LED をオンにして、電圧がトリガー レベルを上回るのを待機します。その後、LED はオフになり、割込みは再び有効になります。

注: main コードで割込みを処理することにより、割込みの発生イベントと割込みが処理される時点との間に不確定の遅延時間が常に挿入されます。割込みの処理に時間的制約があるなら、コードは main のかわりに ISR に置く必要があり、この技術を使用するべきではありません。

[1] 変数がブール型であるため、CYBIT マクロは PSoC 3 で 8051 CPU のビット処理機能が利用できます。詳細については、「AN60630 – PSoC 3 8051 Code Optimization」および「PSoC Creator System Reference Guide」の 2 節を参照してください。PSoC 5LP では、このマクロは `uint8` 型です。

[2] 割込みコンポーネントパラメータの InterruptType は DERIVED にセットされます (「割込みコンポーネントのコンフィギュレーション」を参照してください)。これにより、レベル割込みが発生します。これは、電圧が LVD トリガー レベルより低い間にも割込みが連続的に発生するためです。電圧がトリガー レベルより低い間に 1 つだけの割込みが発生するように InterruptType を RISING_EDGE に設定できます。しかし、電圧がトリガー レベルを再び上回ると追加の割込みが発生するという理由で、この代替手段は効果的ではありません。そのため、1 つだけの LVD 割込みを発生させる最も一般的で効果的な方法は、ISR で LVD 割込みを無効にし、そして電圧がトリガー レベルを上回った後に割込みを再び有効にすることです。

main()コードは以下のとおりです。

```
void main()
{
    /* initialize LVD for both digital and analog (Vddd and Vdda) */
    CyVdLvDigitEnable(0/*interrupt, not reset*/, 5/*~2.95 V*/);
    CyVdLvAnalogEnable(0/*interrupt, not reset*/, 5/*~2.95 V*/);

    isr_1_StartEx(LVD_ISR); /* initialize the custom defined ISR */
    isr_flag = 0; /* initialize semaphore before turning on interrupts */
    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        /* monitor for LVD interrupt */
        if(isr_flag != 0)
        { /* an LVD interrupt occurred */
            isr_flag = 0;
            LED_Write(1); /* turn LED on to indicate an interrupt occurred */

            /* Wait here for the LVD condition to clear: get the interrupt
               source until the response stays off.
            */
            while(CyVdStickyStatus(3/*LVID and LVIA only*/) != 0)
            {
                /* There is minimal hysteresis in the LVD trigger circuits so
                   add a delay to compensate.
                */
                CyDelay(1/*msec*/);
            };

            /* LVD condition cleared */
            isr_1_ClearPending(); /* in case the interrupt is still pending */
            isr_1_Enable(); /* re-enable LVD interrupt */
            LED_Write(0u); /* turn the LED back off */
        }

        /* Nothing else to do in main except monitor for LVD condition. */
    }
}
```

4.4 SysTick 割込みプロジェクト

これは 4 番目 (最後) のプロジェクト例であり、プロジェクト*D_SysTick*として本アプリケーション ノートに添付されています。PSoC 5LPにおけるCortex-M3 CPUのSysTick機能について説明します。PSoC 3 内の 8051 CPUはSysTick機能を備えていませんが、「PICU割込みプロジェクト」で説明された技術を使ってユーザー独自の機能を作れます。このプロジェクトはサイプレス開発キットCY8CKIT-050 向けに開発されていますが、他のキットに容易に適用できます。

ほとんどのリアルタイム オペレーティング システム (RTOS) はシステム ティック割込みを使用してタスクの切り替えおよびタイミングを制御します。標準システム ティック レートは 1 ミリ秒です。PSoC Creator は、*CyLib.c*／*CyLib.h* ファイルの高レベル API を提供することにより、SysTick タイマーの使用を簡単にします。SysTick API'の詳細は「*System Reference Guide*」資料を参照してください (PSoC Creator の Help > Documentation メニューにも掲載されます)。

main.c で添付プロジェクト *D_SysTick* のコードを確認してください。main 関数で SysTick が開始され、Systick 割込みで呼び出されるコールバック関数がセットされます。このコールバック関数 (SysTickISRCallback) は 1 秒に 1 回 LED をトグルするミリ秒カウンターを処理します。`main()`関数のコードは以下のとおりです。

```
void main()
{
uint32 i;

/* Enable global interrupts. */
CyGlobalIntEnable;

/* Configure the SysTick timer to generate interrupt every 1 ms
    * and start its operation. Call the function before assigning the
    * callbacks as it calls CySysTickInit() during first run that
    * re-initializes the callback addresses to the NULL pointers.
    */
    CySysTickStart();

/* Find unused callback slot and assign the callback. */
for (i = 0u; i <CY_SYS_SYST_NUM_OF_CALLBACKS; ++i)
    {
if (CySysTickGetCallback(i) == NULL)
        {
/* Set callback */
            CySysTickSetCallback(i, SysTickISRCallback);
break;
        }
    }

for(;;)
    {
/* Do nothing in the main loop; code to do
               something is in the ISR callback */
    }}
```

このプロジェクトの回路図には割込みコンポーネントが存在しないことと、SysTick 割込みが *cydwr* ファイルの「**Interrupts**」タブに示されないことに注意してください。上記に述べられた機能を提供することを除いて、PSoC Creator は SysTick と Cortex M3 CPU の内部の他の割込みに対応していません。

5 高度な割込みトピック

5.1 割込みコードの最適化

割込みベースのアプリケーションでは、重要な性能要件の 1 つは ISR コードの実行時間です。一部のアプリケーションでは、ISR 内の重要なコードは割込み要求受信用の特定時間以内で実行しなければなりません。また割込み実行は、余りに多くの時間を要して、main コードの実行あるいは他の割込みを停止してはいけません。これらの要件を満たすためには、以下のガイドラインにしたがってください。

- **ISR で関数呼び出しを回避:** 関数の呼び出しは、関数の呼び出しや戻りに必要なスタック プッシュ／ポップオーバヘッドにより ISR コードの実行時間を増やします。キャラクタ LCD ディスプレイ ルーチンなどの場合では、関数の実行は不確定で非常に長い時間かかることがあります。そのような関数が優先順位の高い ISR で実行される時に、優先順位が低い割込みを処理することによってレイテンシが増加する場合もあります。

  推奨される方法は、「LVD 割込みプロジェクト」で説明したとおりに重要でない関数呼び出しを main コードに移動し、ISR で単にフラグ変数をセットすることです。main コードはフラグを周期的にチェックし、フラグがセットされている場合、それをクリアして関数を呼び出します。

- **割込みへ適切な優先順位を割当て:** 割込みが多いアプリケーションでは、よりタイム クリティカルな割込みに対して、より高い優先順位を付けます。
- **コードの最適化技術を使用:** 8 ビット 8051 CPU を内蔵した PSoC 3 は、32 ビット Arm Cortex CPU を内蔵した PSoC 5LP より長いコード実行時間を要します。そのため、コード実行時間に関する要求を満たすために、PSoC 3 の ISR コードを最適化する必要がある場合があります。

  8051 CPU アーキテクチャはコードの実行を加速させるために様々な機能を備えます。例としては、ISR ローカル変数を 8051 の内部データ空間に置いて、バイナリ値 (ゼロおよび非ゼロ) だけを持つ変数をビット変数として定義する場合があります。詳細については、アプリケーション ノート「AN60630 – PSoC 3 8051 Code Optimization」を参照してください。PSoC 5LP におけるコードの最適化の詳細については、アプリケーション ノート「AN89610 – PSoC 4 and PSoC 5LP Arm Cortex Code Optimization」を参照してください。

5.2 割込みコンポーネント API

割込みコンポーネント用に生成されたソース ファイルとヘッダ ファイルは割込みの設定にいくつかの API を提供します。今まで説明したプロジェクトでは、一般的に使用されている `isr_Start()` および `isr_StartEx()` 関数のみを使用します。その他の関数は追加のタスクを実施するために用意されています。

- `Enable()`、`Disable()`を使用して割込みを有効／無効にします。割込みを無効にしても、保留中の割込み要求をクリアできないことに注意してください。保留中の要求をクリアするために、`ClearPending()` API を使用する必要があります。
- `SetVector()`、`SetPriority()`を使用して割込みベクタ アドレスと割込みの優先順位を動的に変更します。これらの関数を呼び出す前に割込みを無効にすることは一番適切なやり方です。
- `SetPending()`を使用して割込み要求なし (即ち、ファームウェア制御の下の場合) の割込みを保留します。
- `ClearPending()`を使用して割込みが処理されないようにその保留状態を削除します。この関数は割込みソース信号に影響を与えず、ただ割込みコントローラー内の割込みラインの保留ステータス ビットのみをクリアします。ソースで割込み信号生成をクリアする手順は、該当するペリフェラルのコンポーネント データシートを参照してください。

割込みAPIの詳細な説明については、「Interrupt Component Datasheet」を参照してください。

PSoC Creator は、一般的な割込み関数セットを *CyLib.h および CyLib.c* ファイルに用意されています。未対応の固定機能割込みで説明されるように、これらの関数は PSoC Creator でサポートされない固定関数割込みを設定します。これらの関数の詳細については、「*System Reference Guide*」を参照してください (PSoC Creator メニューの Help > Documentation にも用意されています)。

処理速度が重要なタスクの実行、または割込みルーチンからもアクセスされるメイン スレッドの変数を更新するとき、グローバル割込み生成を無効にすることは一般的な方法です。PSoC Creator では、この動的なグローバル割込みを有効／無効にするために、`CyEnterCriticalSection` と `CyExitCriticalSection` の 2 つの API が使用できます。これら 2 つの API は、ファームウェア変数とハードウェアレジスタの破損を防ぐために使用されます。`CyEnterCriticalSection` API は割込みを無効にし、以前に割こみが有効化されていたか否かを示す値を返します。`CyExitCriticalSection` API は割込みを復元します。

この動作を理解するために、タイマー制御レジスタへの書き込み例を以下に示します。

```
TIMER_CONTROL_REG |= TIMER_MASK;
```

上記のステートメントの実行中に、以下の一連の動作が行われます。

1. CPU はタイマーの制御レジスタを読み出し、一時的なレジスタに保存します。
2. CPU は一時的なレジスタと MASK 値の論理和を実行します。
3. この論理和の結果を制御レジスタにロードします。

ステップ 1 の後、割込みが発生し、そのハンドラが同じ制御レジスタに新しい値をロードします。ハンドラの実行後、CPU がステップ 2 の実行を再開すると、一時的なレジスタにあった古い制御レジスタの値が使用され、データが破損します。

この問題を回避するために、以下のコードを使用してください。

```
InterruptState = CyEnterCriticalSection();
TIMER_CONTROL_REG |= TIMER_MASK;
CyExitCriticalSection(InterruptState);
```

`CyEnterCriticalSection` および `CyExitCriticalSection` API の使用は、制御レジスタに書き込むときにすべての割込みの発生を無効にするで、この問題を回避します。このシナリオでは、一時変数を使用することになるため、単純な書き込み操作でも起きる可能性があります。main()関数と割込みハンドラで変更される変数またはハードウェアレジスタがある場合は、これらの 2 つの API の使用を推奨します。

これらの API の詳細については、「*System Reference Guide*」を参照してください (PSoC Creator メニューの Help > Documentation にも用意されています)。

5.3 割込みおよびその他のコンポーネント

多くのPSoC Creatorコンポーネントは割込みコンポーネントを実装の一部として備えます。例としては、リアル タイムクロック (RTC)、UARTやSPIなどの通信コンポーネント、デルタ シグマADCがあります。

割込みコンポーネントと同様に、これらのコンポーネント内のISRにはユーザー コードを書き込むためのプレースホルダ領域が用意されています。

それぞれのコンポーネントでの割込み使用法を理解するために、PSoC Creator に用意されている該当コンポーネントデータシートおよび関連サンプル コードを参照してください。ハード フォールト例外の処理については、「Hard Fault Interrupt Handler Function – KBA86934」知識ベース記事を参照してください。

5.4 割込みレイテンシ

割込みレイテンシは、割込みのアサートから ISR で最初の命令が実行されるまでの遅延時間と定義されます。PSoC 3 の 8051 CPU は割込みレイテンシが 25 サイクルであり、PSoC 5LP の Arm Cortex M3 CPU は割込みレイテンシが 12 サイクルです。これらのレイテンシ値はそれぞれのデバイスのデータシートで指定されており、CPU 割込みアーキテクチャ固有のレイテンシ値のみが含まれます。実際に観測される割込みレイテンシは次の原因によってそれ以上になります: フラッシュメモリ内に存在する割込みコードをフェッチするためのフラッシュ メモリ待機状態によるオーバーヘッド、または実際の割込みコードを実行する前に CPU にすべての CPU レジスタをスタックにプッシュさせる ISR 内の関数呼び出し。割込みコードの最適化の節に記載される手法を使用して、割込みレイテンシを短縮できます。

実際の ISR コードの実行前に、以下のことが行われます。

1. プロセッサは現時点のプログラム カウンター (PC)、CPU コア固有のレジスタ、およびいくつかの汎用 CPU レジスタをスタックに入れます。
2. プロセッサは NVIC からのベクタ アドレスを読み出して、それを PC に更新します。
3. プロセッサは NVIC レジスタを更新します。

割込み処理プロセスを効率的にするために、PSoC 5LP の Cortex M3 プロセッサは以下の 2 つの方式を実装します。

1. *テール チェーン:* プロセッサが他の割込みハンドラを実行している間に割込みが保留状態にある場合、最初の割込みの実行が終わったらアンスタックがスキップされ、保留中の割込みハンドラがすぐに実行されます。これにより、レジスタをスタックから復元してそれを再びスタックに入れる時間を節約できます。これは、優先順位の低い割込みのレイテンシを減らす際に役立ちます。
2. *レイト アライバル:* 優先順位の低い割込みのスタック処理中に優先順位の高い割込みが発生した場合、プロセッサは優先順位の低い割込みのかわりに優先順位の高い割込みハンドラにジャンプします。プロセッサは、スタック処理の終わりに優先順位の高い割込みのベクタ アドレスを読み出します。優先順位の高い割込みハンドラの実行が終わると、保留中の優先順位の低い割込みハンドラのベクタ アドレスがフェッチされ、実行されます。これにより、優先順位の低い ISR に入ってレジスタ値をスタックに入れることに起因した遅延時間をなくすことで、優先順位の高い割込みのレイテンシが減少します。

デバイスが割込みにより復帰した場合、電源投入シーケンスおよび関連する低消費電力ウェイクアップ コード シーケンスの後に、電圧安定化で追加の遅延時間が生じます。異なる低消費電力モードからデバイスが復帰する時間の仕様については、デバイスのデータシートを参照してください。

5.5 デバッグ

PSoC 3 および PSoC 5LP デバイスは、シリアル ワイヤ デバッグ (SWD) インターフェースおよびジョイント テスト アクション グループ (JTAG) インターフェースを使用してオンチップ デバッグ機能に対応します。PSoC Programmer – MiniProg3 はホスト (PSoC Creator) からのデバッグ データおよびコマンドを SWD／JTAG インターフェースを介して通信します。これにより、ブレークポイントの追加、変数値の評価と編集、CPU レジスタの表示、書かれたファームウェアのアセンブリ命令の監視、およびメモリの読み出しが可能になります。デバッグ モードは、以下のように割込みを確認する際に役立ちます。

- 割込みが実行中であるかを確認するために、ISR のいずれかの命令にブレークポイントを追加します。
- 特定の割込みがどの時点で実行されるかを確認するために、デバッガの呼び出しスタック ウィンドウを使用します。このウィンドウは、シーケンスで実行された関数呼び出し命令を一覧表示します。これは、優先順位の高い割込みが優先順位の低い ISR の実行中に発生したかを確認するためにも使用できます。

ブレークポイント ヒット カウントを使用して割込みがトリガーされる回数を検出します。これは、割込み信号には、割込みを複数回トリガーさせるグリッチが発生したかを確認する際に特に役立ちます。

デバッガの使用方法の詳細は、PSoC Creator の Help メニュー内の「Using the Debugger」セクションを参照してください。ドキュメントにアクセスするためには、**「F1」**を押す、または PSoC Creator 内の Help > Topics メニューを使用してください。

デバッガのかわりに、ビット バング技法を使用して以下のことを行えます。
- CPU が ISR に入ったかを確認します。
- ISR の実行時間を測定します。これは、ISR の始まりにピンをセットし、ISR の終わりにピンをリセットすることで行います。

5.6 未対応の固定機能割込み

グローバル信号リファレンス コンポーネントにも関わらず、いくつかの PSoC 割込みソースは PSoC Creator でサポートされません。これらは付録 A の表に「未対応」として一覧表示されます。固定機能割込みソースはサポートされていませんが、*CyLib.c* および *CyLib.h* ファイルに用意されている一般的な割込み関数を使用して、これらのいくつかを設定できます。

固定機能割込みのサポートを手動で追加するために、以下の手順を行ってください。

1. **割込みを生成するように固定関数割込みソースを設定します。**ペリフェラルからの割込み生成を有効にするように特定のペリフェラル レジスタに書き込みます。レジスタの詳細については、PSoC 3 または PSoC 5LP のテクニカル リファレンス マニュアル (TRM) を参照してください。
2. **割込みコントローラー ロジック回路 (PSoC 3 のみ) を有効にします。** PSoC 3 では、割込みコントローラー ブロックのクロック信号を有効にし、割込みコントローラーは IRQ 信号を CPU に生成するように設定する必要があります。「0x01」の値を CYREG_INTC_CSR_EN レジスタに書き込むことにより、これができます。

   ```
   CY_SET_REG8(CYREG_INTC_CSR_EN, 0x01);
   ```

   PSoC Creator がこのレジスタを自動的に設定するのは、割込みを手動で設定するときではなく、割込みコンポーネントがサンプル回路図に配置されるときです。
3. **割込みベクタ アドレスおよび割込みの優先順位を設定します。***CyLib.h* ファイルに宣言されている CyIntSetVector() および CyIntSetPriority() 関数を使用します。

   ```
   CyIntSetVector(Interrupt_Vector_Num, MyISR);
   CyIntSetPriority(Interrupt_Vector_Num, myPriorityValue);
   ```

   Interrupt_Vector_Num は、固定機能割込みソースの割込みベクタ番号を示します (付録 A を参照してください)。MyISR は、「ユーザー独自の ISR を書く」の節で説明した手順に従って作成されたユーザー定義の ISR 関数です。myPriorityValue は 0～7 の値です。
4. **固定機能割込みおよびグローバル割込みを有効にします。** 以下のコードを使用します。

   ```
   CyIntEnable(Interrupt_Vector_Num);
   CYGlobalIntEnable;
   ```

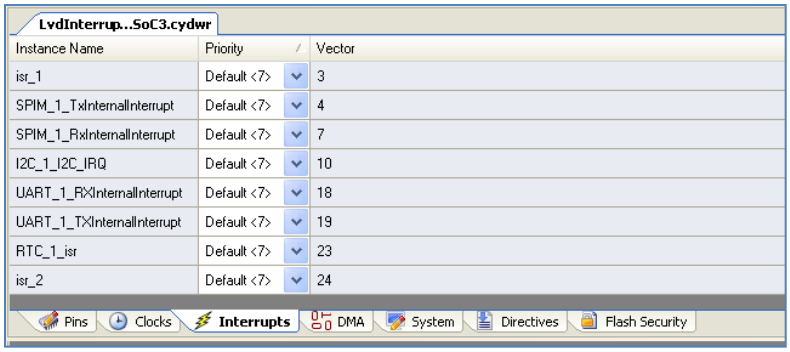
これで、PSoC Creator に固定機能割込みソースのサポートを追加する一連のステップを完了します。ほとんどの固定機能割込みソースがレベル割込みであるため、割込みラインを LOW にするために ISR 内のペリフェラル ステータスレジスタを読み出す必要があります。各種の固定機能ペリフェラルが生成する割込み信号のタイプについては、TRM ドキュメントを参照してください。

重要な注: ここで説明した手順では、PSoC Creator フレームワークを知らずに固定機能割込みソースの割込みベクタ番号を使用します。PSoC Creator は、割込みベクタ番号が既に設計で使用されていることを認識していません。そのため、PSoC Creator は回路図に配置されている割込みコンポーネントに同じベクタ番号を割り当てることもあります。この場合、次の節で説明するように手動でベクタ番号を割り当てられます。

5.7 割込みベクタ番号の割り当て

PSoC Creator は、プロジェクト内の割込みコンポーネントにベクタ番号を自動的に割り当てます。プロジェクトのビルド後、図 21 が示すように、*cydwr* ウィンドウの「**Interrupts**」タブには、割り当てられたベクタ番号が表示されます。

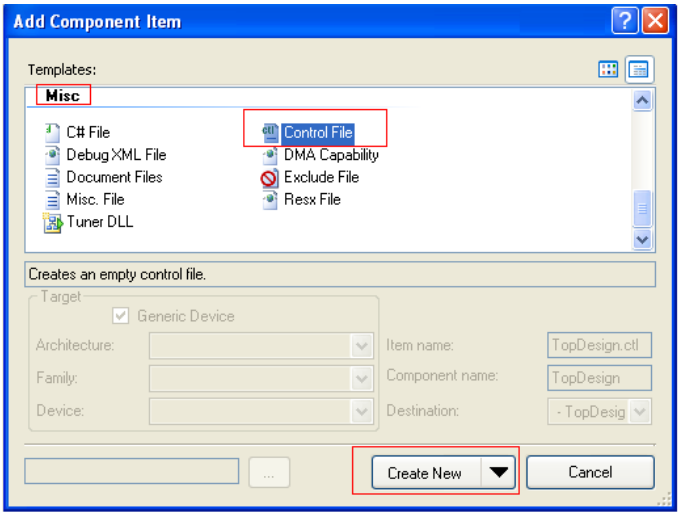
図 21.*cydwr* ウィンドウ内の割込みベクタ番号



前節で述べたように、ユーザーは手動でベクタ番号を割り当てることで、PSoC Creator で割り当てられたベクタ番号を上書きする必要があります。これを行うために、以下のように *Control File* (制御ファイル) を使用します。

1. ワークスペース エクスプローラ ウィンドウの **Components** タブをクリックします。
2. **TopDesign** コンポーネントを右クリックして、**Add Component Item…** を選択します。「Add Component Item」ダイアログが開きます。
3. 図 22 に示すように **Misc** グループにスクロールダウンし、**Control File** を選択して、**Create New** をクリックします。

図 22. Control File の追加



TopDesign.ctl ファイルが作成され、ワークスペース エクスプローラ ウィンドウに追加されます。

4. *TopDesign.ctl* ファイルをダブルクリックして開いて修正を行います。*attribute* キーワードは、各割込みコンポーネントの割込みベクタ番号を指定するために制御ファイルで使用します。割込みベクタ番号を指定する方法は、割込みコンポーネントがユーザーによってサンプル回路図に配置されるか、または割込みコンポーネントが回路図内の PSoC Creator コンポーネントの内部で使用されるかに依存します。2 つの方法は以下のとおりです。

a) ユーザーによって回路図に配置された割込みコンポーネントの場合、構文は以下のとおりです。

```
attribute placement_force of instance_name : label is "Intr(0,
DesiredVectorNumber)";
```

ここで、instance_name は回路図内の割込みコンポーネント名を示し、DesiredVectorNumber はベクタ番号 (0～31) です。例えば、ベクタ 17 を割込みコンポーネント *isr_1* に割り当てる構文は以下のとおりです。

```
attribute placement_force of isr_1 : label is "Intr(0, 17)";
```

b) RTC、UART、SPI、デルタ シグマ ADC など割込みを内部で使用するコンポーネントの場合、構文は以下のとおりです。

```
attribute placement_force of \top_instance_name : InternalInterruptName\: label is
"Intr(0, DesiredVectorNumber)";
```

ここで、top_instance_name は内部で割込みを使用するコンポーネントの名前を示します。23 ページの図 21 に基づいた例は SPIM_1、UART_1、および RTC_1 です。

InternalInterruptName はコンポーネントの内部割込みに割り当てられた名前を示します。これは、*cydwr* ウィンドウの「**Interrupts**」タブ (図 21) から参照できます (ここでは、割込み名が最上位のコンポーネントのインスタンス名に付加されます)。図 21 では、*isr* は RTC_1 という RTC コンポーネントの内部割込み名です。I2C_IRQ は I2C_1 という I2C コンポーネントの内部割込み名です。図 21 の割込みに基づいた割り当ての例は以下のとおりです。

```
attribute placement_force of \I2C_1:I2C_IRQ\ : label is "Intr(0,19)";
attribute placement_force of \RTC_1:isr\ : label is "Intr(0,3)";
attribute placement_force of \SPIM_1:RxInternalInterrupt\ : label is "Intr(0,4)";
```

5. 割込みベクタ番号を割り当てた後、**Save** をクリックして、制御ファイルへの変更を保存します。
6. 割込みベクタの新しい割り当てが発効するために、例に対して **Clean and Build** を選択します。これで、*cydwr* ウィンドウ内の「**Interrupts**」タブは、修正した割込みベクタ番号の割り当てを示します。

6 まとめ

割込みは、組込みアプリケーションで一般的に使用されます。システムオンチップ アーキテクチャ (例えば PSoC 3 や PSoC 5LP のアーキテクチャ) では、割込みはオンチップ ペリフェラルの状態を CPU に通知する重要な役割を果たします。本アプリケーション ノートでは、割込みベースの PSoC Creator プロジェクトを迅速かつ容易に作成するために必要な情報を提供しました。さらに、高度な割込み機能も説明しました。

6.1 プロジェクトの要約

AN54460.cywrk:

このワークスペースは、本アプリケーション ノートで説明された異なるトピックを実演する 4 つのサンプル コードを含んでいます。

- *A_InterruptExample*: ピンを切り替えるためにタイマー割込みを使用する簡単なアプリケーションを実演します。
- *B_PICU*: ポート割込み制御ユニット (PICU) 割込みおよび複数の割込みの使用方法を実演します。
- *C_LVD*: 低電圧検出 (LVD) 割込みを例として使用して、グローバル信号リファレンス コンポーネントの使用方法を実演します。
- *D_SysTick*: (PSoC 5LP 専用) Cortex-M3 CPU で SysTick 割込みを使用する方法を実演します。

著者について

氏名: Vivek Shankar Kannan

役職: シニア アプリケーション エンジニア

A PSoC 内の割込みソース

表 1 には、PSoC 3 および PSoC 5LP 内の 32 の割込みベクタ用の割込みソースの一覧を示します。固定機能割込み信号を生成する PSoC Creator コンポーネントも示します。PSoC Creator でまだサポートされていない固定機能割込みは「未対応」とマークされます。

表には、UDB 割込みソースまたは DMA nrq 割込みソース用の PSoC Creator コンポーネント名が無いことに注意してください。これは、PSoC Creator が、デジタル信号配線などの複雑な要因に基づいて DMA と UDB 割込みソース用の割込みベクタを動的に割り当てるためです。これらの割込みは、それらの内部信号名で示されます。これらの内部信号の割り当てが内部で PSoC Creator で制御されるため、これらに関する詳細情報を知る必要があります。

PSoC 5LP での割込みベクタ 0～31 は、Cortex-M3 での例外番号 16～47 に対応します。詳細については、PSoC 5LP Technical Reference Manual (TRM) を参照してください。

表 1. PSoC 3 および PSoC 5LP の割込みソース

割込みベクタ番号	固定機能割込みソース		DMA nrq 割込みソース	UDB 割込みソース
	割込みソース	PSoC Creator のコンポーネント		
0	低電圧検出 (LVD)	グローバル信号リファレンス	phub_termout0[0]	udb_intr[0]
1	キャッシュ	グローバル信号リファレンス	phub_termout0[1]	udb_intr[1]
2	予約済み	該当なし	phub_termout0[2]	udb_intr[2]
3	パワー マネージャ	RTC、スリープ タイマー、グローバル信号リファレンス	phub_termout0[3]	udb_intr[3]
4	PICU[0]	デジタル入力ピン、デジタル双方向ピン	phub_termout0[4]	udb_intr[4]
5	PICU[1]		phub_termout0[5]	udb_intr[5]
6	PICU[2]		phub_termout0[6]	udb_intr[6]
7	PICU[3]		phub_termout0[7]	udb_intr[7]
8	PICU[4]		phub_termout0[8]	udb_intr[8]
9	PICU[5]		phub_termout0[9]	udb_intr[9]
10	PICU[6]		phub_termout0[10]	udb_intr[10]
11	PICU[12]		phub_termout0[11]	udb_intr[11]
12	PICU[15]		phub_termout0[12]	udb_intr[12]
13	組み合わせたコンパレータ	未対応	phub_termout0[13]	udb_intr[13]
14	組み合わせたスイッチトキャパシタ	未対応	phub_termout0[14]	udb_intr[14]
15	I2C	I2C	phub_termout0[15]	udb_intr[15]
16	CAN	CAN	phub_termout1[0]	udb_intr[16]
17	タイマー／カウンター0	タイマー／カウンター／PWM	phub_termout1[1]	udb_intr[17]
18	タイマー／カウンター1	タイマー／カウンター／PWM	phub_termout1[2]	udb_intr[18]
19	タイマー／カウンター2	タイマー／カウンター／PWM	phub_termout1[3]	udb_intr[19]
20	タイマー／カウンター3	タイマー／カウンター／PWM	phub_termout1[4]	udb_intr[20]
21	USB SOF Int	USBFS	phub_termout1[5]	udb_intr[21]
22	USB Arb Int		phub_termout1[6]	udb_intr[22]
23	USB Bus Int		phub_termout1[7]	udb_intr[23]

割込みベクタ番号	固定機能割込みソース		DMA nrq 割込みソース	UDB 割込みソース
	割込みソース	PSoC Creator のコンポーネント		
24	USB エンドポイント[0]		phub_termout1[8]	udb_intr[24]
25	USB エンドポイント データ		phub_termout1[9]	udb_intr[25]
26	予約済み	該当なし	phub_termout1[10]	udb_intr[26]
27	LCD	セグメント LCD	phub_termout1[11]	udb_intr[27]
28	DFB	フィルター	phub_termout1[12]	udb_intr[28]
29	デシメータ	デルタ シグマ ADC	phub_termout1[13]	udb_intr[29]
30	PHUB エラー	未対応	phub_termout1[14]	udb_intr[30]
31	EEPROM フォールト	未対応	phub_termout1[15]	udb_intr[31]

B PSoC 3 の再入可能関数

再入可能関数は、同時に main および ISR などの複数の個別のプロセスで実行できる関数です。例えば、再入可能関数が main 関数内で実行されている時、割込みが発生し、ISR は同じ関数を呼び出せます。

関数の再入は、C プログラミング言語の標準機能で、PSoC 5LP のコンパイラによって対応します。しかし、PSoC 3 8051 CPU の制約上、関数の再入は通常、Keil C51 コンパイラによって実装されません。ここでは、Keil C51 コンパイラで再入可能関数を処理する手順について説明します。

B.1 Keil C51 コンパイラでの再入可能

PSoC 3 の 8051 CPU 内の使用可能なスタックと RAM 空間が制限されているため、Keil C51 コンパイラはデフォルトで関数を再入不可と見なします。Keil コンパイラは関数の引数とローカル変数を固定メモリ位置に格納します。関数への同時の呼び出しは、同じメモリ位置を使用します。これにより、関数の引数とローカル変数は破損される可能性があります。

関数はデフォルトで再入不可である場合、個別単位で再入可能に対応するようにできます。Keil コンパイラは、関数のローカル変数と引数を格納する個別のスタックを作成します。関数への複数の呼び出しが正しく実行されるために、このスタックへのポインタが関数の呼び出し時に調整されます。再入可能スタックは xdata メモリ空間 (PSoC 3 の SRAM メモリ空間) で作成されます。

再入可能関数がPSoC Creatorプロジェクトに存在する場合、プロジェクトの*KeilStart.a51* ファイルは、再入可能スタックを有効にし、スタック ポインタを初期化するための自動生成コードを持ちます。

```
XBPSTACK EQU 1
XBPSTACKTOP EQU CYDEV_SRAM_SIZE
```

最初の命令文により、スタックをxdataメモリ空間 (SRAMメモリ) に配置します。2 番目の命令文により、スタックの先頭アドレスをSRAMの最後のバイトに割り当てます。理由は、再入可能スタック ポインタが下向きに進むからです。スタック ポインタがSRAMの最終アドレスに割り当てられているため、SRAM内でスタックが変数を上書きするリスクは最小限にされます。

B.2 PSoC Creator での再入可能性サポート

PSoC Creator で関数を再入可能にする手順は、関数が PSoC Creator によって生成された API であるか、ユーザー定義の関数であるか、またはユーザーが作成したカスタム コンポーネントに対応する関数であるかに依存します。異なる手順は、以下の節で説明します。

B.2.1 生成された API 関数を再入可能に

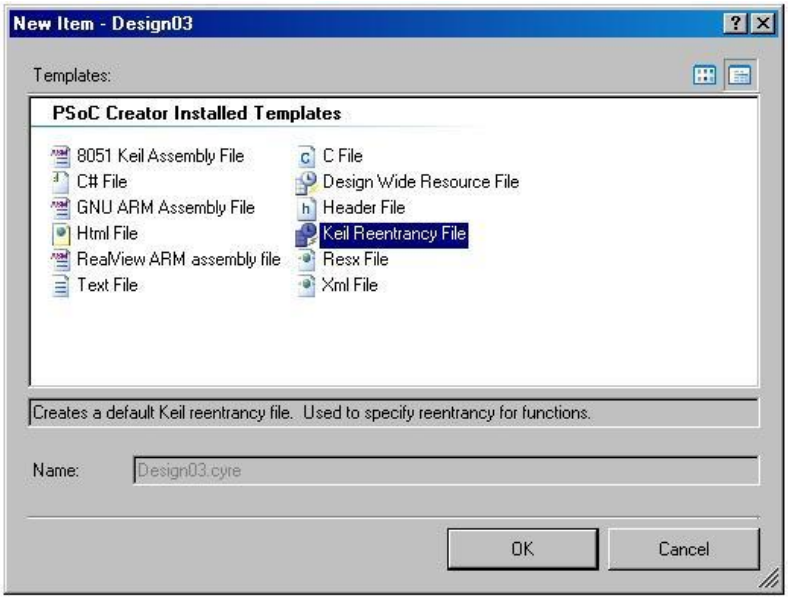
これらは、コンポーネントAPIのように、PSoC Creatorが生成したソース コードと関連付けられた関数です。PSoC Creatorにより、どの生成されたAPI関数が「再入可能ファイル」(*cyre*ファイル) で再入可能にする必要があるかを指定できます。このファイルは、どの関数を再入可能にするかを指定します。ビルド プロセス中に、再入可能ファイルで注記された関数が、再入可能に対応するように自動的にマークされます。

APIファイル内のほとんどの関数は、再入可能をサポートする候補です。再入可能にできない関数の場合、コンポーネント ソース ファイルのコメントは関数が再入可能サポートの候補ではないことを示します。理由は、関数がグローバル変数または静的変数を使用するか、または再入可能性をサポートできない他の関数を呼び出すことがあるからです。

プロジェクトに *cyre* ファイルを追加するためには:

1. ワークスペース エクスプローラでプロジェクトを右クリックし、Add > New Item を選択します。
2. 「New Item」ダイアログでは、「**Keil Re-entrancy File**」を選択して **OK** をクリックします (図 23 を参照してください)。*Project_Name.cyre* ファイルはコード エディタ内で開きます。

図 23. プロジェクトへ *cyre* ファイルを追加



再入可能関数に入るためには:

3. *cyre* ファイル内の各行に 1 つの関数名を入力します (行ごとに 1 つだけの関数名)。関数名だけは *cyre* ファイルに入力する必要があり、関数名は大文字と小文字を区別することに注意してください。関数の戻り型または関数を示すための括弧は、*cyre* ファイルに入力しないでください。例:

```
ADC_Start
PWM_Start
```

4. 完了したら *cyre* ファイルを保存します。

B.2.2 ユーザー定義の関数を再入可能に

これらは、ユーザーがアプリケーション コードの一部として定義する関数です。*cyre*ファイルは、ユーザー定義の関数を再入可能関数にするために使用できません。ユーザー定義の関数の再入可能をサポートするために、関数定義とともにcytypes.hの*CYREENTRANT*マクロを関数プロトタイプの一部として使用してください。これは、Keilツール チェーンではREENTRANTキーワードを参照しますが、他のツール チェーンでは何も参照しません。これにより、コードは、さまざまなツール チェーンやデバイス ファミリにわたって正常に機能できます。以下の例では、ユーザー定義関数Fooをどうのように再入可能にするかを示します。

```
void Foo(void) CYREENTRANT;

void Foo(void) CYREENTRANT
{
. . .
}
```

B.2.3 カスタム コンポーネント API 関数を再入可能に

これらは、ユーザーが作成するコンポーネント ソースとヘッダ ファイルの一部である関数です。PSoC Creatorにより、ユーザーは自分のコンポーネントを作成し使用できます。

カスタム コンポーネントを作成する場合、ビルド表現を使用して再入をサポートするために、コンポーネントのソース ファイルとヘッダ ファイル内の任意の関数を作成できます。

```
`=ReentrantKeil($INSTANCE_NAME . "_FunctionName")`
```

ビルド表現の FunctionName を実際の関数名で置き換えます。他のすべてのフィールドは変更されないままにしてください。*.h* ファイル内の関数宣言および*.c* ファイル内の関数定義の両方はこの表現を含める必要があります。これにより、関数は PSoC Creator と共に出荷される他のコンポーネントと同様の動作が可能になります。初期設定では、これは標準的な関数ですが、ユーザーが関数名を**.cyre* ファイルに加えることによりこれを再入可能な関数にできます (「生成された API 関数を再入可能に」の節を参照してください)。

元の関数の宣言と定義:

```
void `$INSTANCE_NAME`_Foo (void);

void `$INSTANCE_NAME`_Foo (void)
{
    . . .
}
```

修正された再入可能関数:

```
void `$INSTANCE_NAME`_Foo(void) `=ReentrantKeil($INSTANCE_NAME . "_Foo")`;

void `$INSTANCE_NAME`_Foo(void) `=ReentrantKeil($INSTANCE_NAME . "_Foo")`
{
    . . .
}
```

B.3 再入可能関数の判定

関数を同時に呼び出す場合に加えて、Keilコンパイラが関数用にRAM空間を割り当てる場合にのみ、関数を再入可能としてマークを付ける必要があります。Keilコンパイラは、再入可能としてマーク付けする関数を判定する手助けをします。最適化レベルが 2 以上に設定された場合、ビルドが実行されている時、コンパイラは、再入可能としてマーク付けされず同時に呼び出す可能性がある各関数に対して警告を出します。

```
Warning: L15 MULTIPLE CALL TO FUNCTION MYFUNCTION/MAIN ?C_C51STARTUP
ISR_1_INTERRUPT/ISR_1
```

警告メッセージでは、`L15 MULTIPLE CALL TO FUNCTION` は再入可能性の警告を示します。`MYFUNCTION/MAIN` は、*main.c* ファイルで定義された関数 `MyFunction` が同時に呼び出される関数であることを示します。`MyFunction` の呼び出し元の 1 つは main コードです。Keil コンパイラは`?C_C51STARTUP` 用語を使用して `main()` 関数から派生した実行の main フローを参照します。

`MyFunction` の第 2 呼び出し元は、`ISR_1_INTERRUPT/ISR_1` が示した *isr_1.c* ファイル内の `isr_1_Interrupt` 関数です。これは割込みサービス ルーチンです。そのため、`MyFunction` 関数は再入可能にする必要があります。Keil コンパイラの再入可能性の警告は、大文字と小文字を区別せずに常に関数名とファイル名を大文字で表すことに注意してください。

以下は警告メッセージのもう 1 つの例です。

WARNING: L15: MULTIPLE CALL TO FUNCTION DELAY/TIMING ISR_1/INTERRUPT_1 ISR_2/INTERRUPT_2

この警告メッセージでは、*Timing.c* ファイル内の `Delay()` 関数は同時に 2 つの割込みサービス ルーチン (*Interrupt_1.c* ファイル内の `isr_1()` 関数および *Interrupt_2.c* ファイル内の `isr_2()` 関数) から呼び出されます。そのため、`Delay()` 関数は再入可能にする必要があります。

警告メッセージのフォーマットは、PSoC Creator の「Notice List」(通知リスト) ウィンドウにどのように表示するかに依存します。フォーマットは、出力ウィンドウと例のマップ ファイルで少し違います。出力ウィンドウとマップ ファイルの両方で、表示された初期の警告は以下のフォーマットにです。

Warning: L15 MULTIPLE CALL TO FUNCTION NAME: MYFUNCTION/MAIN CALLER1: ?C_C51STARTUP CALLER2: ISR_1_INTERRUPT/ISR_1

再入警告は、すべての場合に適用されなくても絶対に無視してはいけません。例えば、関数が main コードから呼び出されると、割込みは無効になることもあります。そのため、関数が同時に main と ISR の両方から呼び出されることはありません。ただし、Keil リンカーはこれらのシナリオを識別できません。これは、関数への同時の呼び出しに関する再入可能性の警告を依然として出しています。適用できない場合でも再入可能性の警告を無視すると、データの上書き中にリンカーは異なった実行フロー用に同じメモリ位置を割り当ててしまうことがあります。これで、データ破損が発生して、コードの機能が影響を受ける可能性があります。

7 改訂履歴

文書名: AN54460 - PSoC 3 および PSoC 5LP の割込み

文書番号: 001-89250

版	ECN	発行日	変更内容
**	4129750	09/20/2013	これは英語版 001-54460 Rev. *F を翻訳した日本語版 001-89250 Rev. ** です。
*A	4722741	04/21/2015	これは英語版 001-54460 Rev. *G を翻訳した日本語版 001-89250 Rev. *A です。
*B	5473927	10/13/2016	新しいテンプレートに更新しました。 サンセットレビューを完了する。
*C	5716265	04/27/2017	ロゴと著作権を更新しました。
*D	6655957	01/08/2020	これは英語版 001-54460 Rev. *I を翻訳した日本語版 001-89250 Rev. *D です。
*E	6887487	05/29/2020	これは英語版 001-54460 Rev. *K を翻訳した日本語版 001-89250 Rev. *E です。

ワールドワイドな販売と設計サポート

サイプレスは、事業所、ソリューションセンター、メーカー代理店、および販売代理店の世界的なネットワークを保持しています。お客様の最寄りのオフィスについては、サイプレスのロケーション ページをご覧ください。

製品

Arm® Cortex® Microcontrollers	cypress.com/arm
車載用	cypress.com/automotive
クロック&バッファ	cypress.com/clocks
インターフェース	cypress.com/interface
IoT (モノのインターネット)	cypress.com/iot
メモリ	cypress.com/memory
マイクロコントローラ	cypress.com/mcu
PSoC	cypress.com/psoc
電源用 IC	cypress.com/pmic
タッチセンシング	cypress.com/touch
USB コントローラー	cypress.com/usb
ワイヤレス	cypress.com/wireless

PSoC®ソリューション

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

サイプレス開発者コミュニティ

コミュニティ | サンプルコード | Projects | ビデオ | ブログ | トレーニング | Components

テクニカル サポート

cypress.com/support

本書で言及するその他すべての商標または登録商標は、それぞれの所有者に帰属します。

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709