ハードウェア割り込み解説資料

Interrupt and API PSoC Experiment Course Material 6

ハードウェア設定と対応ソースの
自動生成メカニズム割り込みのメカニズム

Interrupt and API Lab Material 6 V2.12
April 9th., 2019
int_api_6.pptx (42 Slides)
Renji Mikami

まったくのブランク状態から生成ソースをトレースしてみる

PSoCのどのリソースも使用しない状態では、どのようなファイルがテンプレートとして生成されているのかをみてみます.

以降割り込みのメカニズムと自動生成ファイルとソースの追加について解説します.

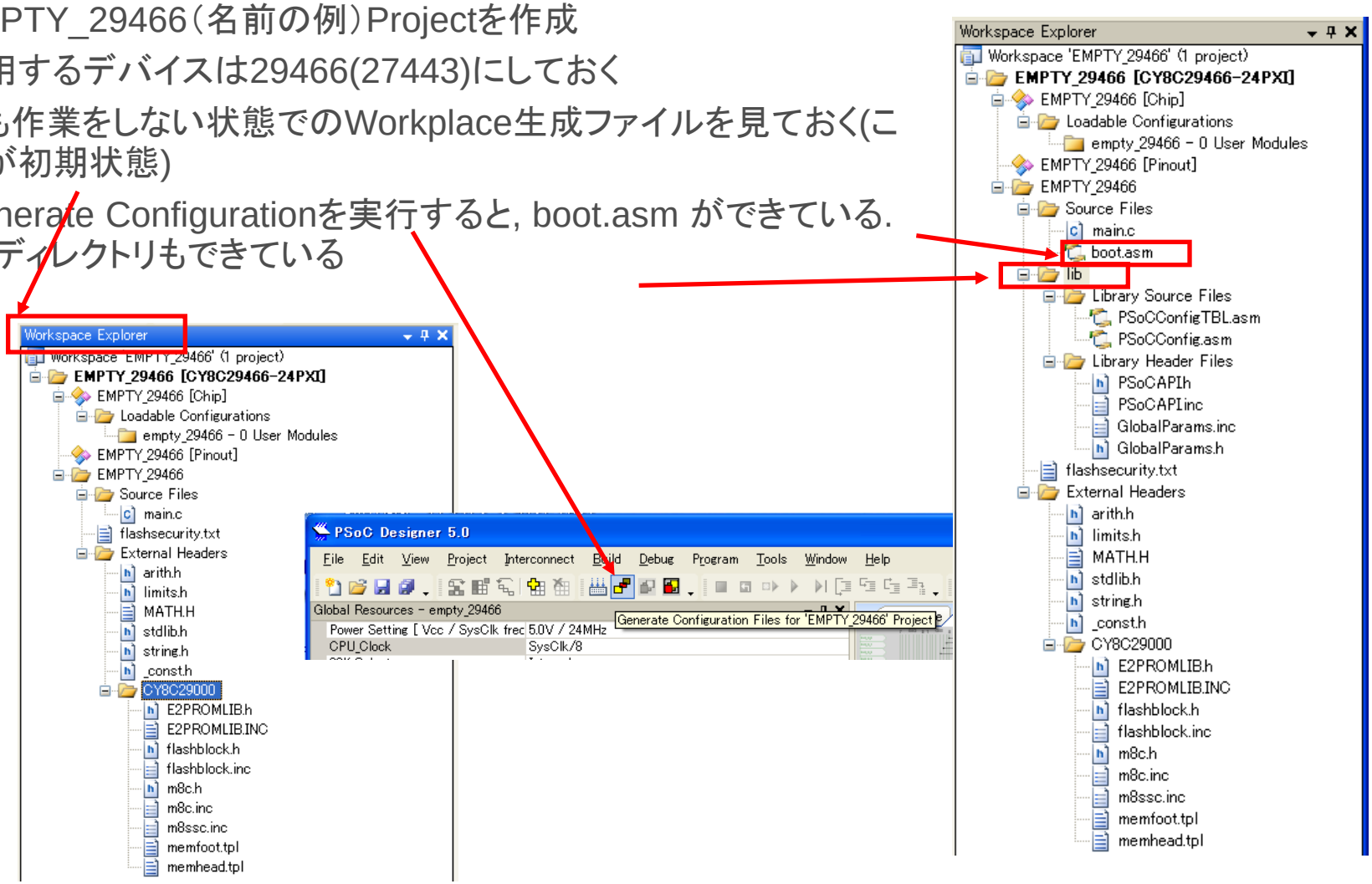
ブランクデフォルト状態

EMPTY_29466（名前の例）Projectを作成

使用するデバイスは29466(27443)にしておく

何も作業をしない状態でのWorkplace生成ファイルを見ておく(これが初期状態)

Generate Configurationを実行すると, boot.asm ができている. lib ディレクトリもできている



ブランクデフォルト時のソース

main.c の中身を見てみる 記述用のテンプレートが生成されている

boot.asm の中身を見る

lib¥Library Source Files¥の中の2つのファイルを見る

PSoCConfigTBL.asm

PSoCConfig,asm

PSoC Designer v4系では*Generate Application*と呼ばれていました.

ハードウェアの設定や変更は*generate configuration* の処理によってWorkplace内のファイルの新規生成,ファイル内容の変更として反映される

PSoCではGUIでハードウェアの設定が行われるがこれらはすべてレジスタのデータ値としてCPUがイニシャライズ時に書き込む.よってGUIによる以下の4つの設定がどのように生成ソースに反映されるかがわかれば解析できる.

グローバル・リソースの設定はどのファイルに反映されるか

ユーザー・モジュールの設定はどのファイルに反映されるか

ピンアウトの設定はどのファイルに反映されるか

配線の設定はどのファイルに反映されるか(あまり重要ではない)

ハードウェアの設定

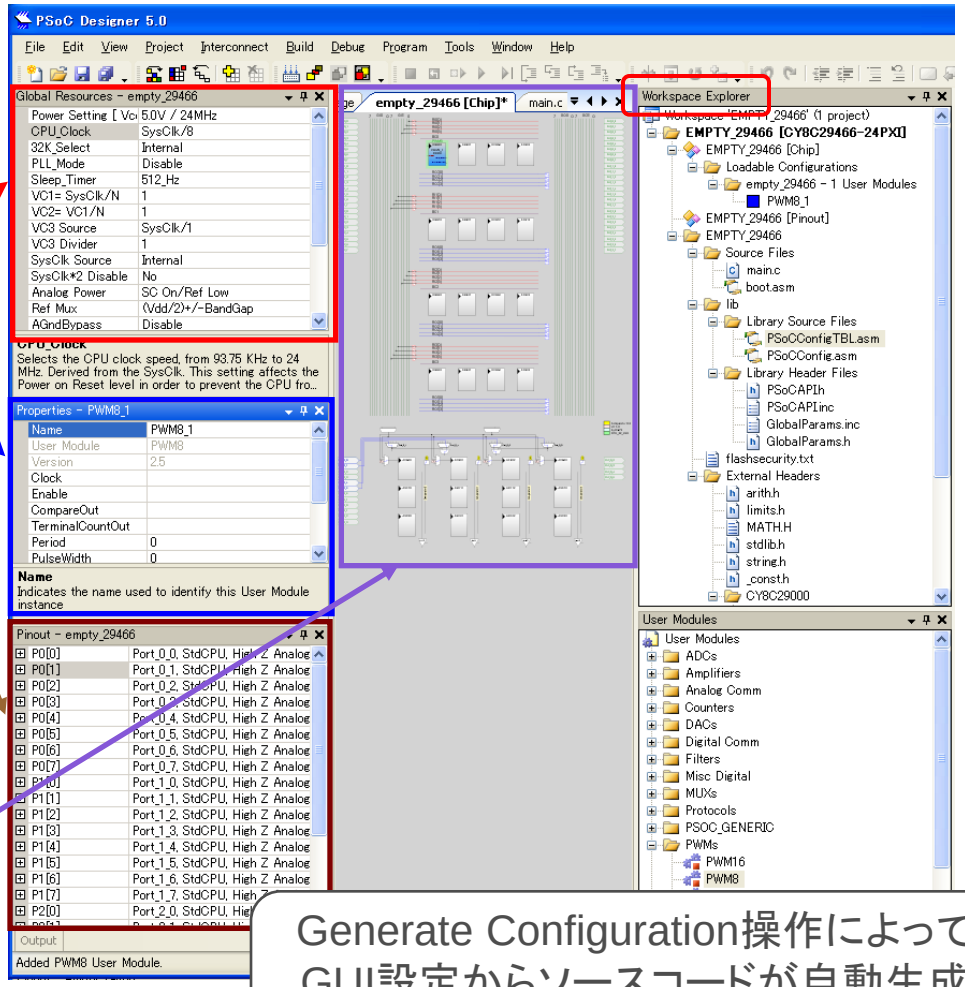
ユーザーモジュールPWM8を
追加した場合

グローバル・リソースの設定

ユーザー・モジュールの設定

ピンアウトの設定

配線の設定



Generate Configuration操作によって
GUI設定からソースコードが自動生成
コードはWorkplaceで確認できる

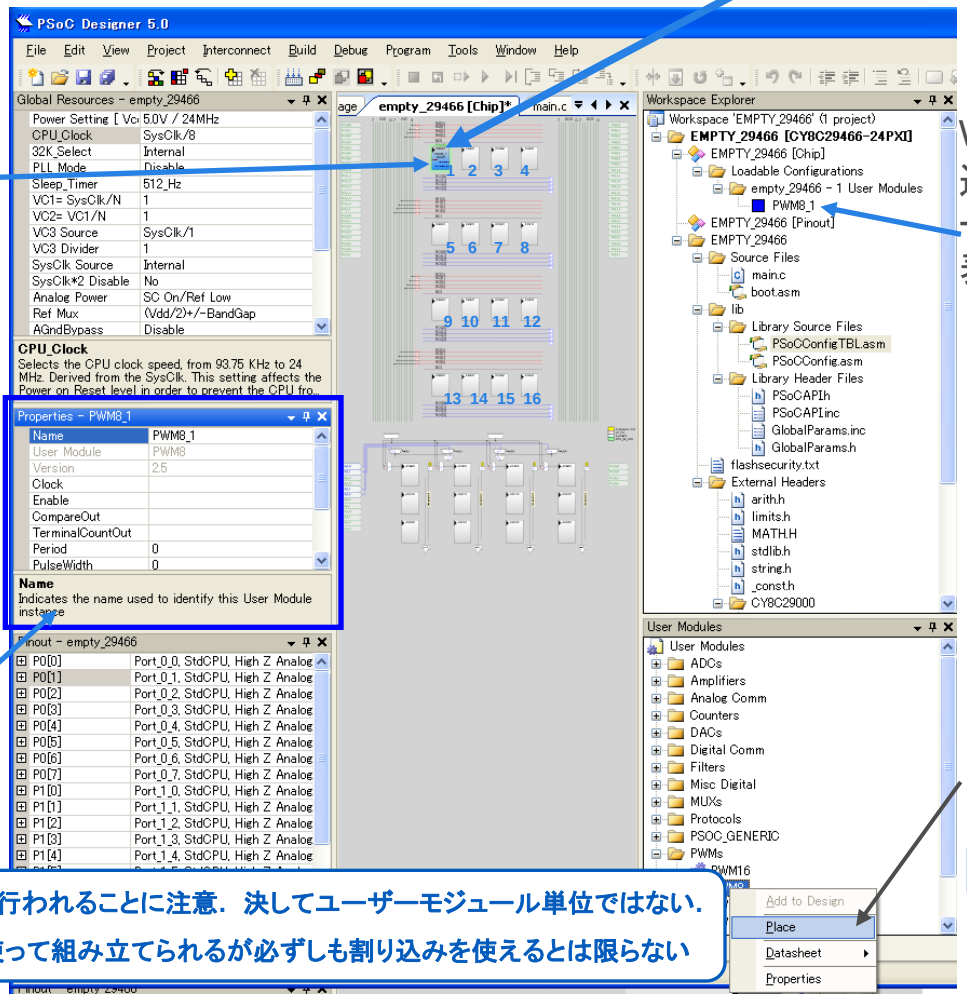
ユーザーモジュールの選択と配置

PWM8ユーザーモジュールの配置PWM8_1とインスタンス命名

ユーザーモジュールはデフォルトでは左上から順に配置される

複数のユーザーモジュールがある場合割り込み優先度は左上が最優先となり順次右のブロックに移り右端に到達したら下の段に移り同様に右へと進む（数字 1-16 で示す）
この順でベクタ・テーブルが作られる(1はDBB00)

プロパティ・ウインドウに選択したユーザーモジュールが表示され設定ができる

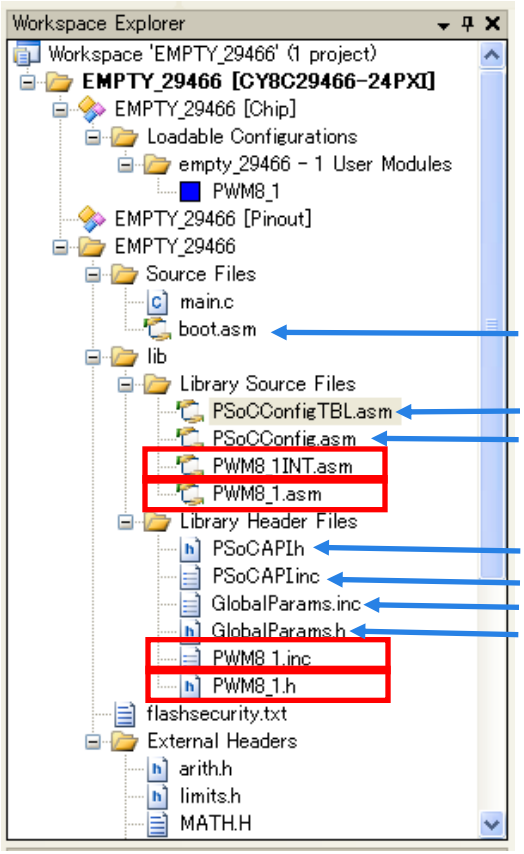


Workplaceに選択したユーザーモジュールが表示される

PWM8を選択してPlaceする

割り込みは,ハードウェアのブロックを単位として行われることに注意．決してユーザーモジュール単位ではない．
ユーザーモジュールはハードウェアブロックを使って組み立てられるが必ずしも割り込みを使えるとは限らない

生成されたファイル



ISR : Inturupt Service Routine

PWM8 を追加した状態でGenerate Configuration を実行

自動生成と修正のメカニズムを理解する

新規に自動生成されたソース

PWM8_1INT.asm ← *ISR*のあるファイル

PWM8_1.asm

PWM8_1.inc

PWM8_1.h

矢印が修正(アップデート)されたソース

同様にグローバル・リソース設定,
ピンアウト設定, 配線を変更
した後は必ずGenerate Configurationを
実行しますが, 個別ソースに修正を加えて
いる場合には上書きされてしまう場合が
ありますので注意してください
(英文ワーニング・メッセージは出ますが..)

Generate Configurationで上書き(更新)されるファイル

Name	Overwritten	Description
…/lib/PSoCConfig.asm	Yes	Configuration loaded upon system access
…/lib/PSoCCOnfigTBL.asm	Yes	Contains chip configuration
Boot.asm	Yes	Boot code and initial interrupt table
…/lib/<User module name>.asm	Yes	User module API source
…/lib/<User module name>.h	Yes	User module API C include header
…/lib/<User module name>.inc	Yes	User module API assembly include
…/lib/<User module name>INT.asm	Yes	User module interrupt .asm file (if needed)
…/lib/<Project name>API.h	Yes	Project API include header
…/lib/<Project Name>_GlobalParameters.inc	Yes	Project parameters include
main.asm	No	Main code

割り込みベクタ･テーブルは, boot.asm にあります

C ソースは更新されません

ハードウェアの設定を変更してもGenerate Configurationを実行するまでは,ソースファイルは更新されません

ソフトで設定するハードウェア

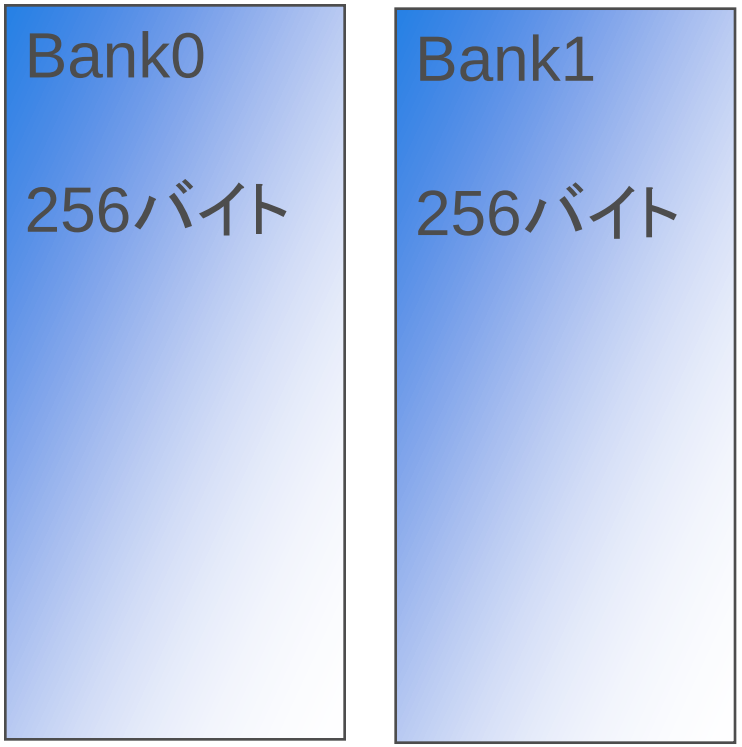
PSoCの本質はハードウェアをすべてレジスタで設定できるところにあります,このレジスタへの書き込みはすべてプロセッサが行います.

つまりソース・コードですべてのことができるということです.

ハードウェアのプロパティー設定ウインドウやchipウインドウの配線の設定は,すべてソースコード生成のためのプリプロセッサになっています.

ハードウェアの詳細やレジスタのリファレンスなどはTRM(Technical Reference Manual)を参照してください-psoc_r__technical_reference_manual__trm__14.pdf(ファイル名中の長いアンダースコアはアンダースコアが2つ続いたものです)

PSoCレジスタ



PSoCレジスタは2バンク構成で各256バイトのアドレスを持ちます．以下の機能設定を行います

M8Cレジスタ(Fレジスタ)RAMページング
割り込みコントローラ設定
デジタルブロック設定
アナログブロック設定
GPIO設定
オシレータ・PLL設定

バンクの切り替えはFレジスタで行います．

ハードウェアの使いこなし

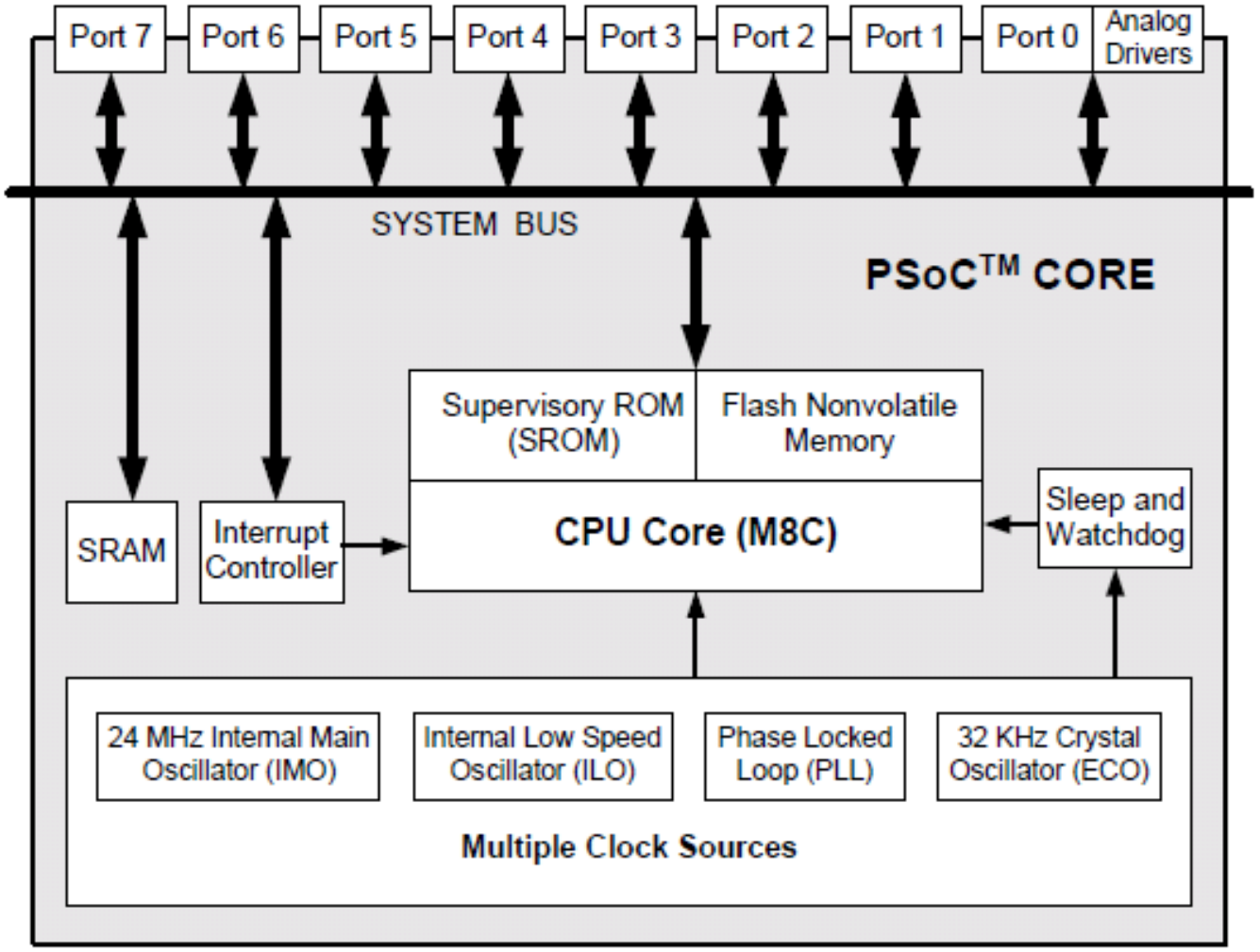
PSoCを使いこなすためには、特有ハードウェアの特徴を理解することとハードウェアとソフトウェアの接点となる割り込みを理解する点にあります.

ハードウェアとしてのユーザーモジュールをC言語などのコードから使いこなすためには,各ユーザーモジュールのポーリングでの使い方と割り込みでの使い方を理解します.

これらのサンプル・コードはユーザー・モジュール・データシートに書かれていますが,割り込みで使う機能とポーリングで使う機能が分かれているものもあります.たとえばADCINC(ADコンバータ)のPWM出力は割り込みがありますがAD変換が終了しデータが使用可能か否かはレジスタをポーリングしてチェックしてからデータを取得にいきます.

割り込みハンドラと自動生成ソースの使い方はそのメカニズムを理解すればC言語から非常に容易に使えます.

PSoCコアトップレベルブロック図



CPUコア（M8C）内部レジスタ

Aレジスタ
（アキュムレータ）

Xレジスタ
（インデックス）

PC
（プログラムカウンタ）

SP
（スタックポインタ）

Fレジスタ
（フラグレジスタ）

全てのインターナルレジスタは8ビット．ただし，PCは16ビット

A,X,PCレジスタはリセット後，0クリアされます

SPは次のスタック位置を指します．0XFFを指している状態でPUSH命令を実行すると0X00へロールします．

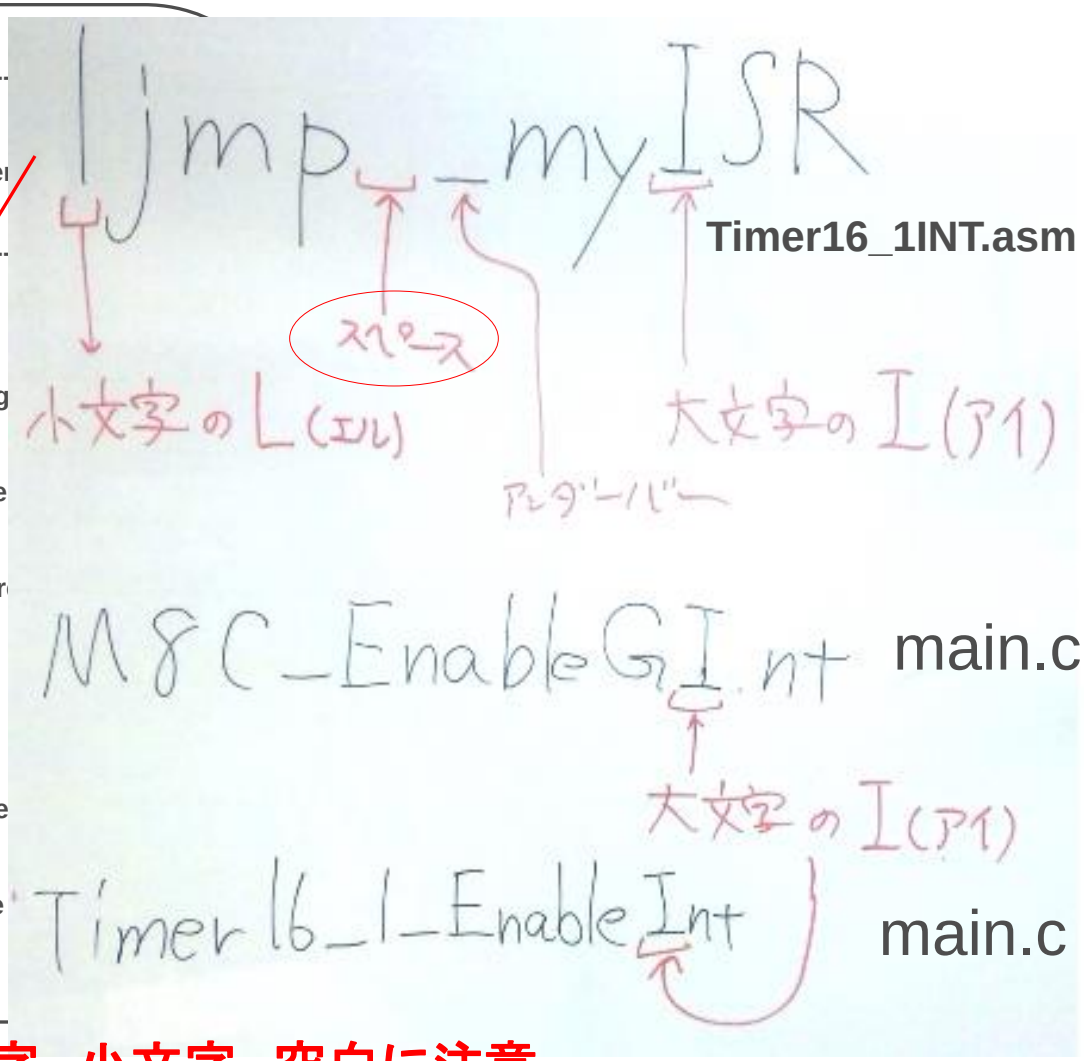
ハードウェア割り込みの使い方とプログラミング

後続スライド補足説明：文字と空白の注意事項

ロングジャンプ命令、割込み許可関数の大文字、小文字と空白について

```
;-------------------------------------------------------------------
;  FUNCTION NAME: _Timer16_1_ISR
;  DESCRIPTION: Unless modified, this impler
handler stub.
;-------------------------------------------------------------------
_Timer16_1_ISR:

   ;@PSoC_UserCode_BODY@ (Do not chang
   ;---------------------------------------------------
   ; Insert your custom code below this banne
   ;---------------------------------------------------
   ;   NOTE: interrupt service routines must pr
   ;   the values of the A and X CPU registers.
ljmp _myISR
   ;---------------------------------------------------
   ; Insert your custom code above this banne
   ;---------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change
   reti
```



大文字、小文字、空白に注意

boot.asmに割り込みベクタ･テーブルを生成

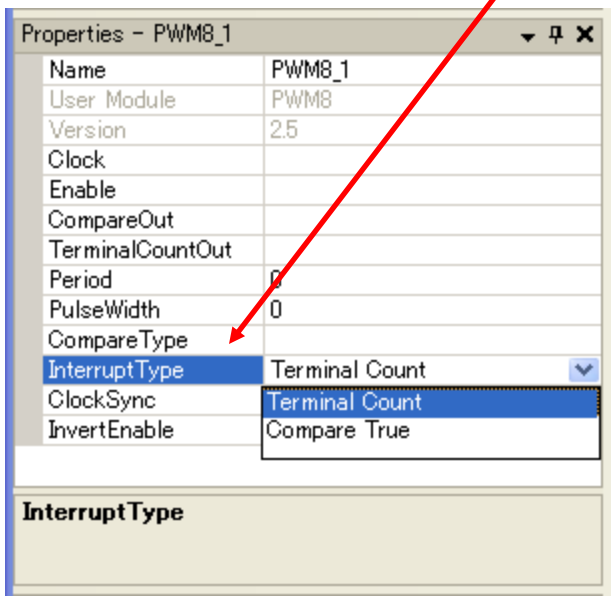
最初に割り込みベクタ･テーブルの自動生成を行う

PWM8_1のプロパティー･テーブルからInterrupt を設定

割り込み発生条件をTerminal Count とCompare Trueを選択できる(2つの機能差はユーザーモジュール･データシートで確認)

設定後にGenerate Configurationを実行

boot.asmにPWM8_1の割り込みベクタ･テーブルが追加される



優先度1の左上の場所DBB00

```
org   20h      ;PSoC Block DBB00 Interrupt Vector
// call      void_handler
reti
```

PWM8_1 を追加する前のboot.asm

Generate Config.
実行で変更

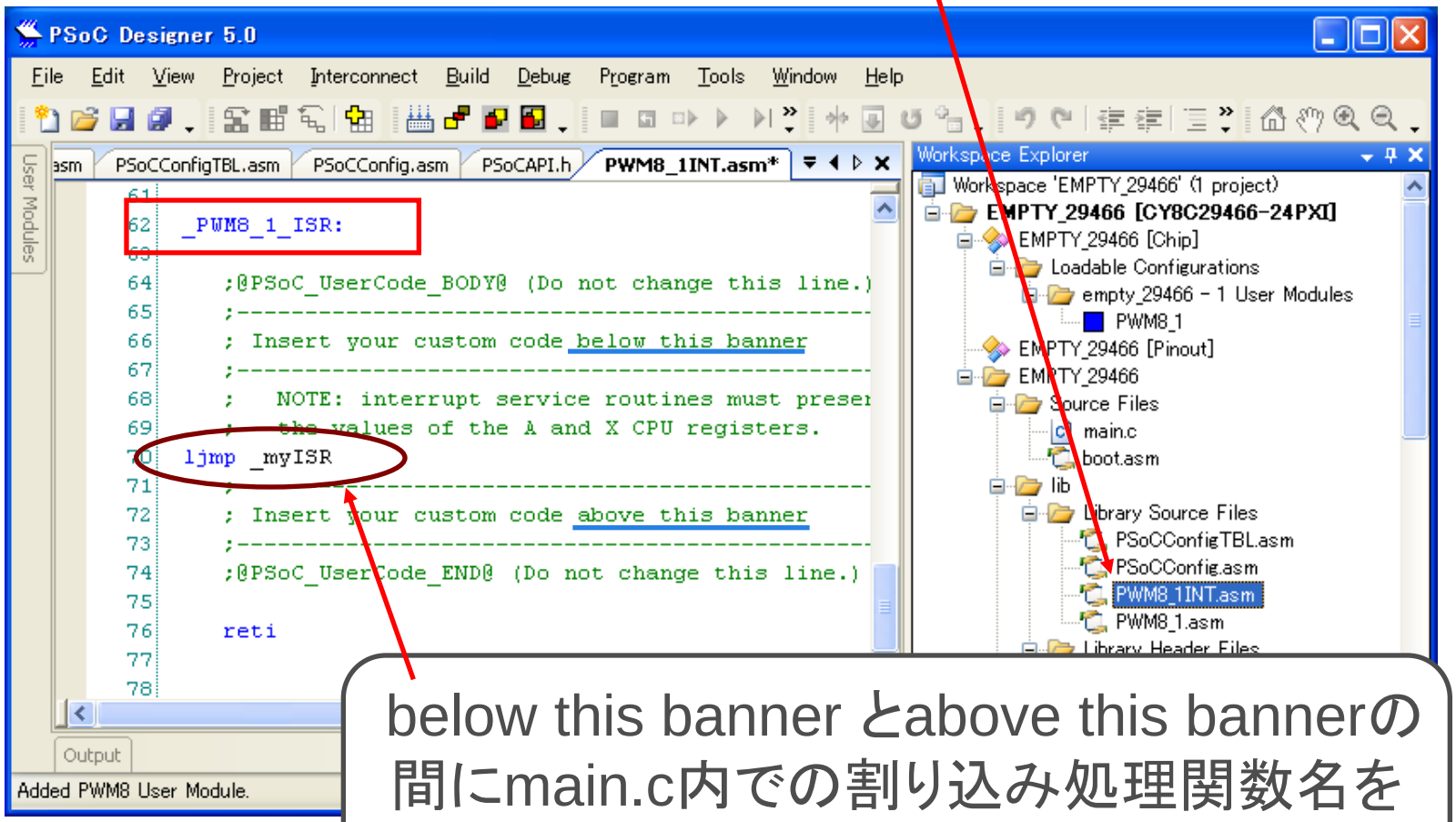
```
org   20h      ;PSoC Block DBB00 Interrupt Vector
ljmp      _PWM8_1_ISR
reti
```

PWM8_1 をDBB0に置いたのboot.asm

割込み発生時に_PWM8_1_ISRロングジャンプ

PWM8_1INT.asmにISR関数名を記述

boot.asmからのジャンプ先のISR(Intrrupt Service Routine)



below this banner とabove this bannerの間にmain.c内での割り込み処理関数名を追加しておく(ここでは_myISRと命名)

main.cへの記述

まずmain.c内で#pragmaで割込みサービスルーチンを記述

```
#pragma interrupt_handler myISR
void myISR(void)
{
        //割り込み時の処理
}
```

続いてmain()にCPU割込み許可を設定し,割込みを使用するAPIを記述

```
void main()
{
   M8C_EnableGInt;
   PWM8_1_EnableInt();
   PWM8_1_Start();

   while(1){};
}
```

M8Cの割り込みを“有効”にする

PWM8_1の割り込を“有効”にする

割り込みのハードウェア

PSoCの割り込みコントローラ回路

複数のユーザーモジュール・ハードウェアから割り込みがある場合はプライオリティー・エンコーダで優先度がつけられ,実行まちとなる割り込みは*ペンディング・インタラプト*となります

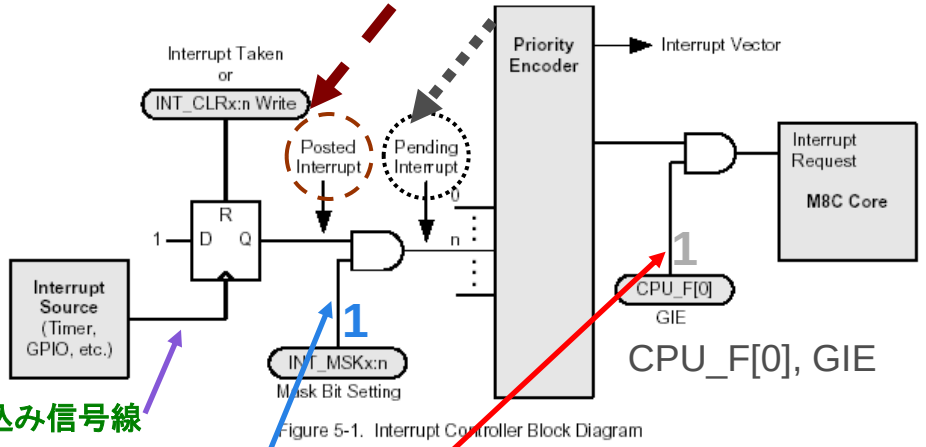


Figure 5-1. Interrupt Controller Block Diagram

PWM8_1を配置したDBB00からの割り込み信号線

ハードウェアとの関係

```
void main()
{
    M8C_EnableGInt;
    PWM8_1_EnableInt();
    PWM8_1_Start();

    while(1){};
}
```

INT_MSKx

M8Cの割り込みが1=“有効”
CPU_Fレジスタ[0]により
CPU割り込みはマスク可能=0

PWM8_1の割り込が1=“有効”
ポステッド割り込みはマスク可能=0

割り込みが処理されると*ポステッド・インタラプト*がリセットされます

割り込みの処理

割り込みが発生すると現在実行中の命令が終了し割り込みサービス・ルーチン(ISR)13サイクルが発生し次の処理を実行.

PCH,PCL,CPU_Fの順にスタックにPUSH

CPU_Fをクリア,GIEは0になる

PCHが0になりPCLに割り込みベクタ値をセット

割り込みベクタのアドレスにロングジャンプ

ISR(割り込みサービス・ルーチン)の実行

RETIでISRが終了

CPU_F,PCL,PCHの順にスタックからPOPして割り込み前に復帰

割り込みレイテンシは、以下の時間の和になります

現在の命令の終了までの時間

+M8Cがプログラムカウンタを割り込みアドレスに変更する時間

+割り込みテーブルのLJMP命令の実行時間（7サイクル）

（例:割り込みアクティブから5サイクルのJMP命令でISR開始とすると5Cycle+ISR13Cycle+LJMP7Cycleで25Cycleですから約1.042uSec(24MHz,25Cycle)かかります

複数割り込みと優先順位

多重割り込み

現在処理中の割り込みが継続され2つめ以降の割り込みは無視される

現在処理の割り込みが終了次第,次のプライオリティーの割り込み処理が実行される

ハードウェア割り込みのプライオリティー順

リセット　（00h）

電源モニタ　（04h）

4つのアナログカラム(08h,0Ch,10h,14h左の位置から順に右へ)

VC3　（18h）

GPIO　(1Ch)

16のデジタルブロック(左上DBB00-20hから右へ向かい右端DCB03-2Chまでいったら次の段の左端DBB10-30hへ,同様に右に進む)

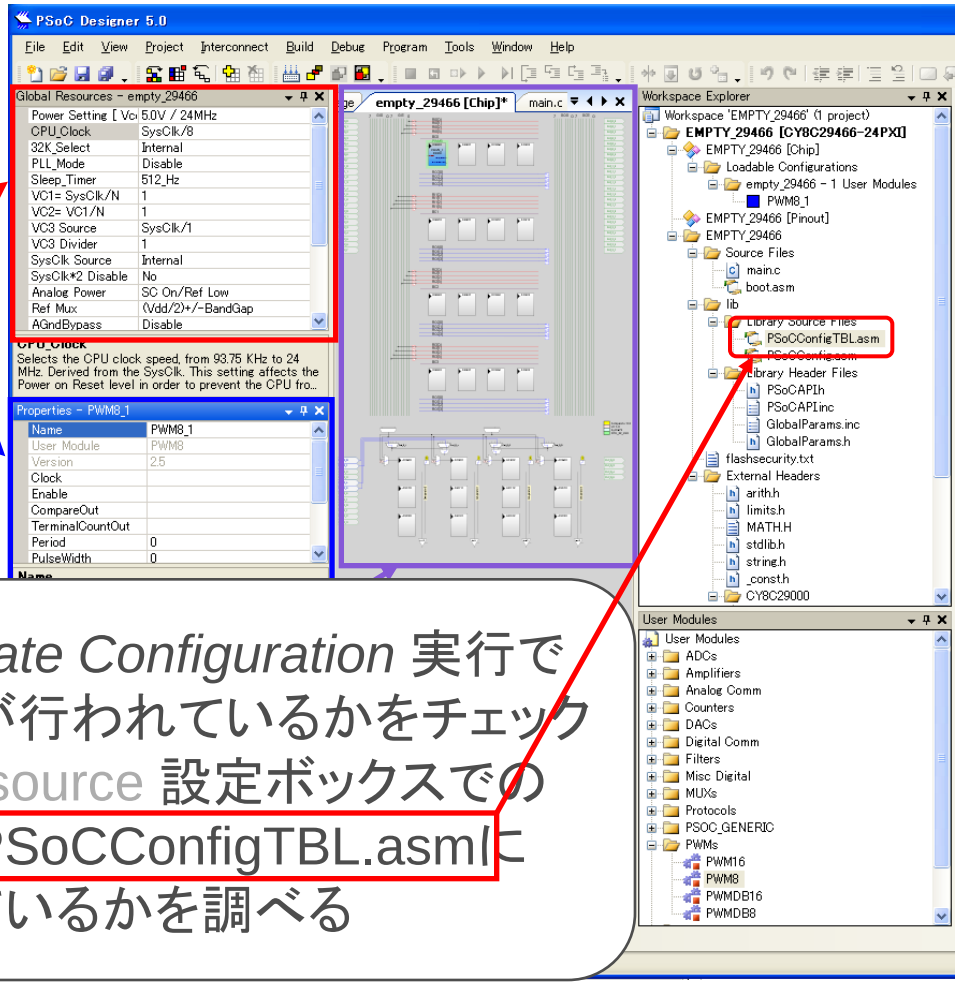
I2C　(60h)

SleepTimer (64h)

CPUへの割り込み数は
*固定6*つを含む26

グローバル・リソースの設定とソース化

ユーザーモジュールPWM8を
追加した場合



グローバル・リソースの設定

ユーザー・モジュールの設定

ピンアウトの設定

GUI設定から*Generate Configuration* 実行で
確実にレジスタ設定が行われているかをチェック
ここではGlobal Resource 設定ボックスでの
パラメータ設定がPSoCConfigTBL.asmに
反映されているかを調べる

アナログ・リファレンスの構造

PSoC オペアンプは単一電源型のためAGNDをV_{DD}の中間付近に設定します．AGND電位は各ブロック毎にバッファされるので各ブロック間で若干のオフセットが発生します．RefHi/LoはDACのコンパレータ・スパンを設定します．

これは 0,63h ARF_CRレジスタのREF[2:0]値で設定します

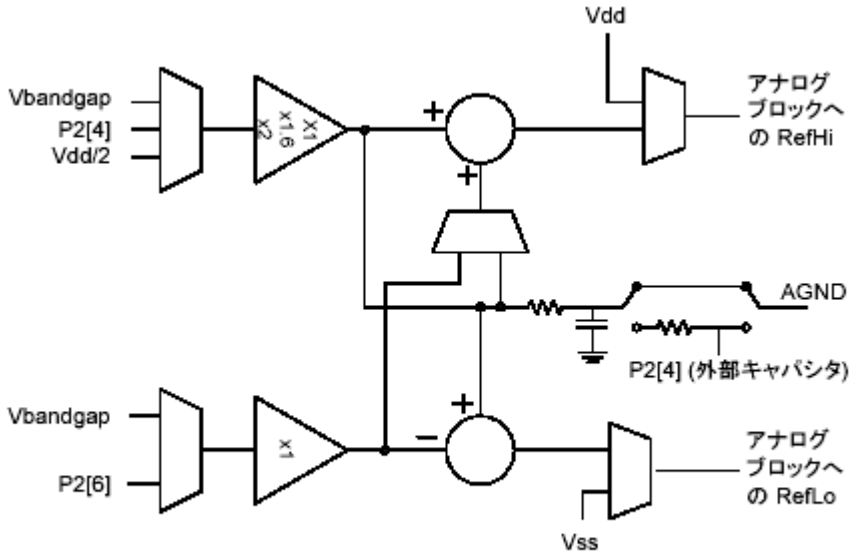


図 21-1. アナログリファレンスコントロール模型

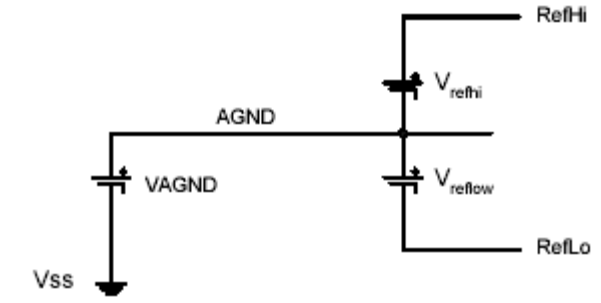
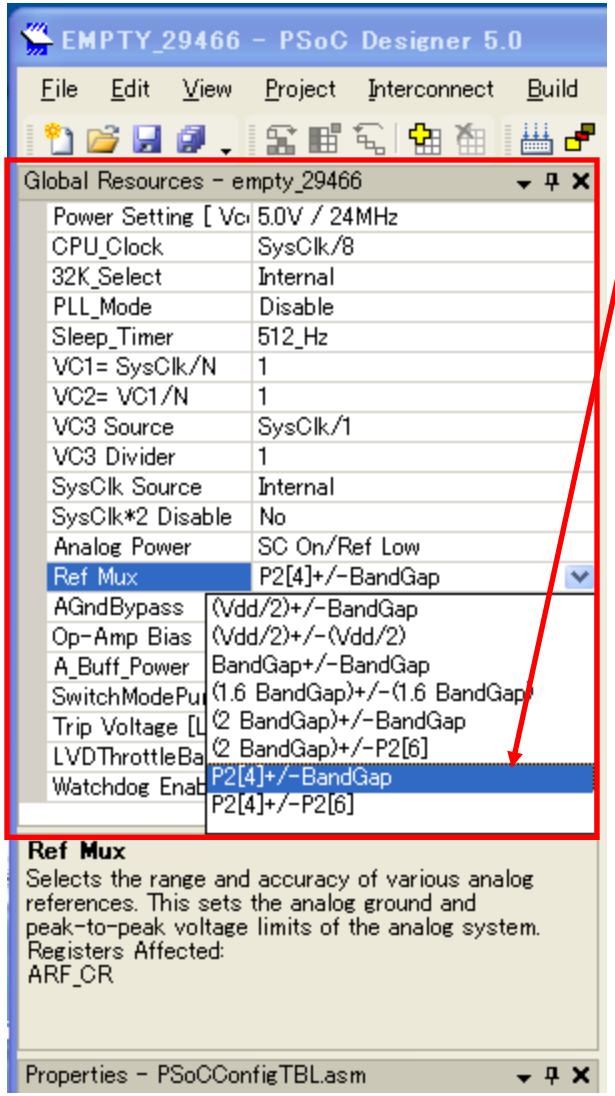


図 21-2. リファレンス構造

表 21-1. アナログリファレンスレジスタ

アドレス	名前	ビット 7	ビット 6	ビット 5	ビット 4	ビット 3	ビット 2	ビット 1	ビット 0	アクセス
0,63h	ARF_CR		HBE	REF[2:0]			PWR[2:0]			RW : 00

アナログ･リファレンスの設定

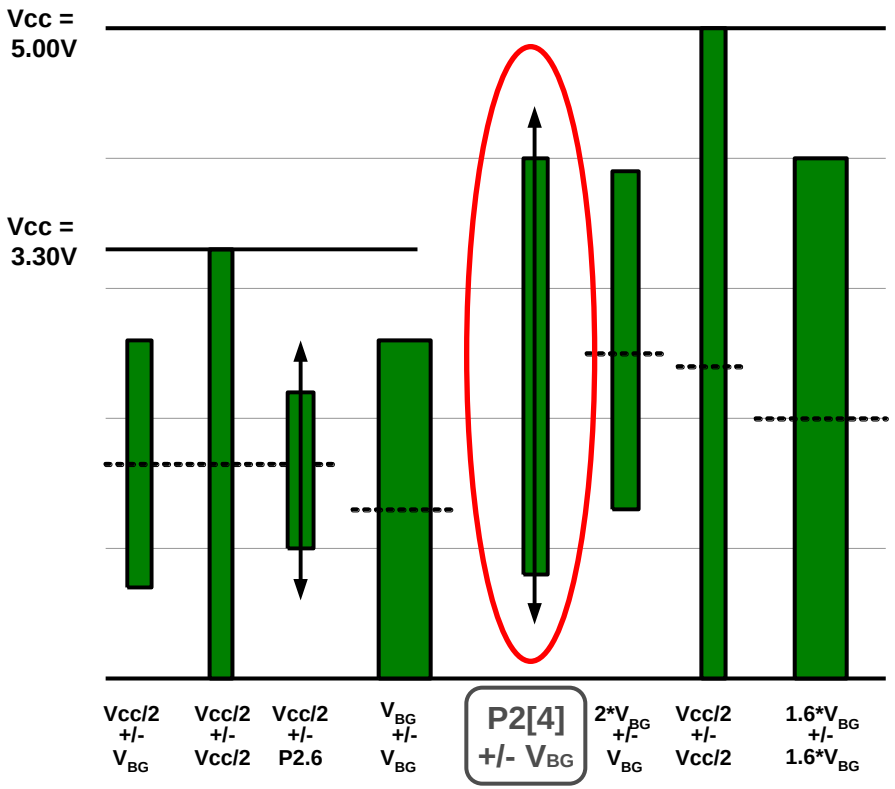


アナログのリファレンス電圧を設定する

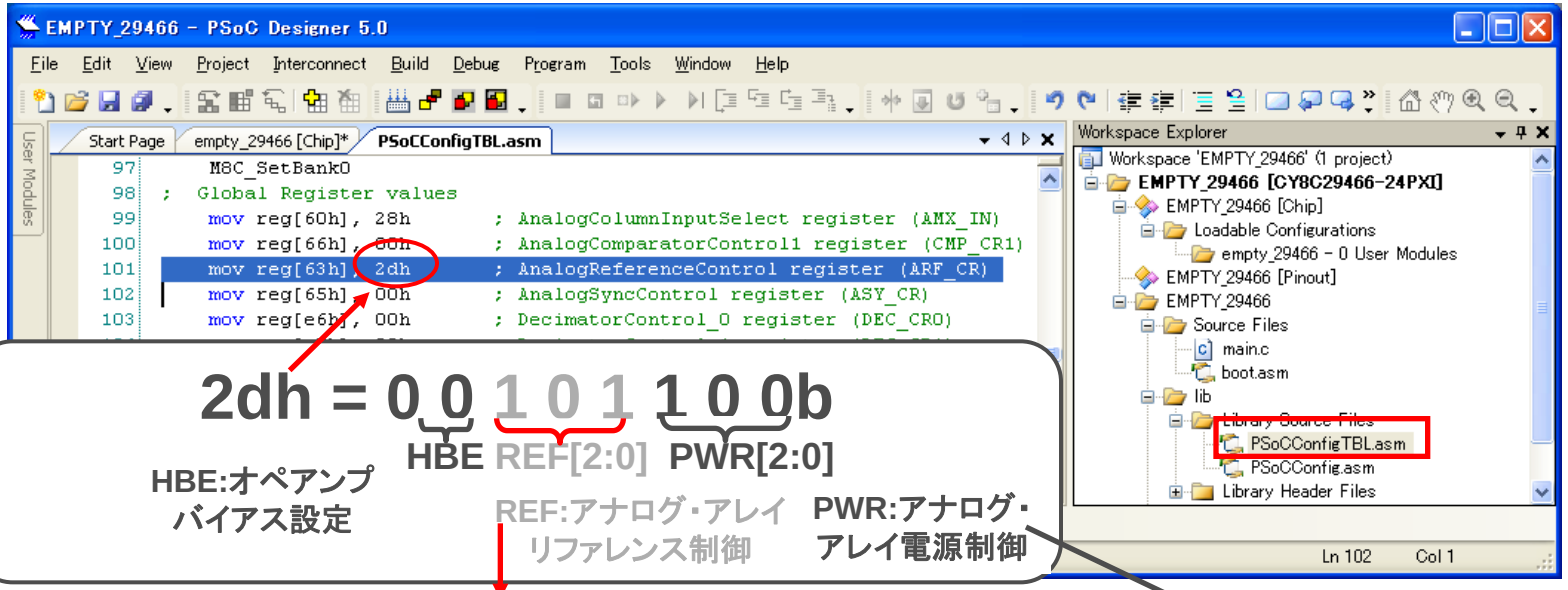
ここではP2[4]+/- BandGap としてみる

BandGap電圧は1.3V ($1.2V_{BG}+0.1V_{Gain}$)

レジスタはARF_CR: 0,63h REF[2:0]の3ビット



PSoCConfigTBL.asm内のreg[63h]の値2dh



2dh = 0 0 1 0 1 1 0 0b

HBE REF[2:0] PWR[2:0]

HBE:オペアンプ バイアス設定

REF:アナログ・アレイ リファレンス制御

PWR:アナログ・アレイ電源制御

表 21-3. REF[2:0]: AGND、RefHI、および RefLO 操作パラメータ

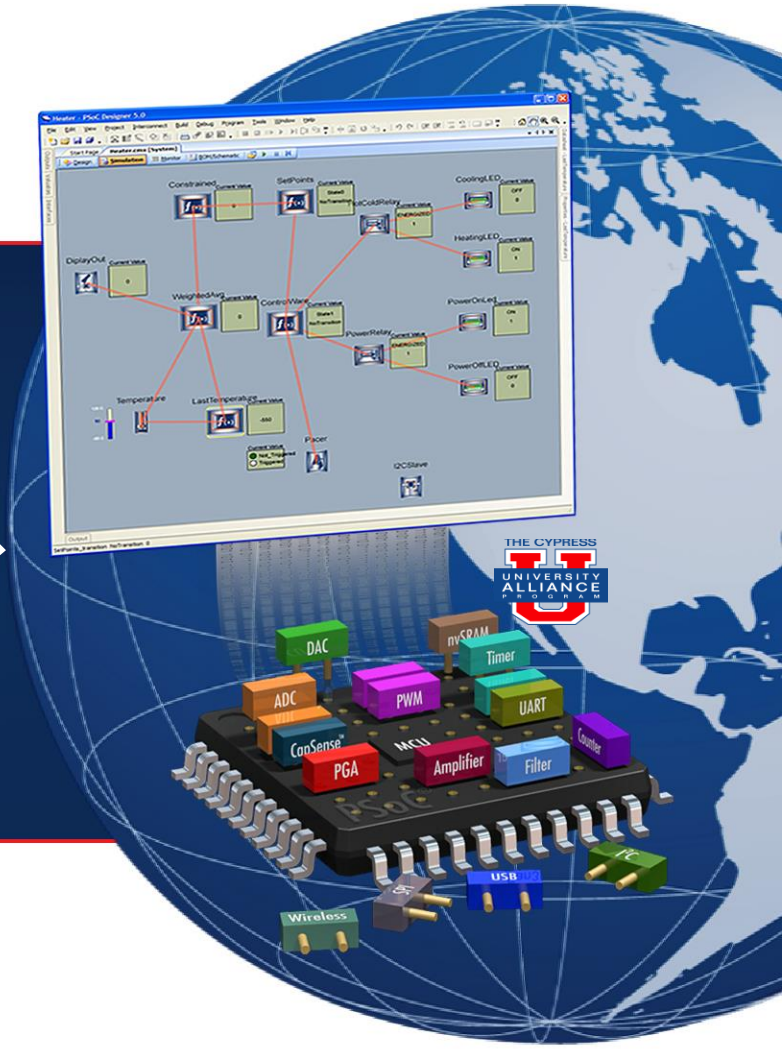
	AGND		RefHI		RefLO		注記
	ソース	電圧	ソース	電圧	ソース	電圧	
000b	Vdd/2	2.5 V 1.65 V	Vdd/2+Vbg	3.8 V 2.95 V	Vdd/2-Vbg	1.2 V 0.35 V	5.0 V システム 3.3 V システム
001b	P2[4]	2.2 V	P2[4]+P2[6]	3.2 V	P2[4]-P2[6]	1.2 V	ユーザ調整可能 例: P2[4]=2.2V および P2[6]=1.0V
010b	Vdd/2	2.5 V 1.65 V	Vdd	5.0 V 3.3 V	Vss	0.0 V 0.0 V	5.0 V システム 3.3 V システム
011b	2*Vbg	2.6 V	3*Vbg	3.9 V	1*Vbg	1.3 V	3.3 V システム用ではありません
100b	2*Vbg	2.6 V	2*Vbg+P2[6]	3.6 V	2*Vbg-P2[6]	1.6 V	P26 < Vdd - 2.6 例: P2[6]=1.0 V
101b	P2[4]	2.2 V	P2[4]+Vbg	3.5 V	P2[4]-Vbg	0.9 V	ユーザ調整可能。例: P2[4]=2.2V 1.3 < P2[4] < Vdd - 1.3
110b	Vbg	1.30 V	2*Vbg	2.6 V	Vss	0	5.0 V システム 3.3 V システム
111b	1.6*Vbg	2.08 V	3.2*Vbg	4.16 V	Vss	0	3.3 V システム用ではありません

PWR[2:0]	CT ロウ	SC ロウ	基準バイアス
000b	オフ	オフ	オフ
001b	オン	オフ	低バイアス
010b	オン	オフ	中バイアス
011b	オン	オフ	
100b	オフ	オフ	オフ
101b	オン	オン	低バイアス
110b	オン	オン	中バイアス
111b	オン	オン	

TRMで確認

割り込み演習

CY3214(24894)評価基板ベース
CY3210(27443/29466)使用可



プロジェクト内容

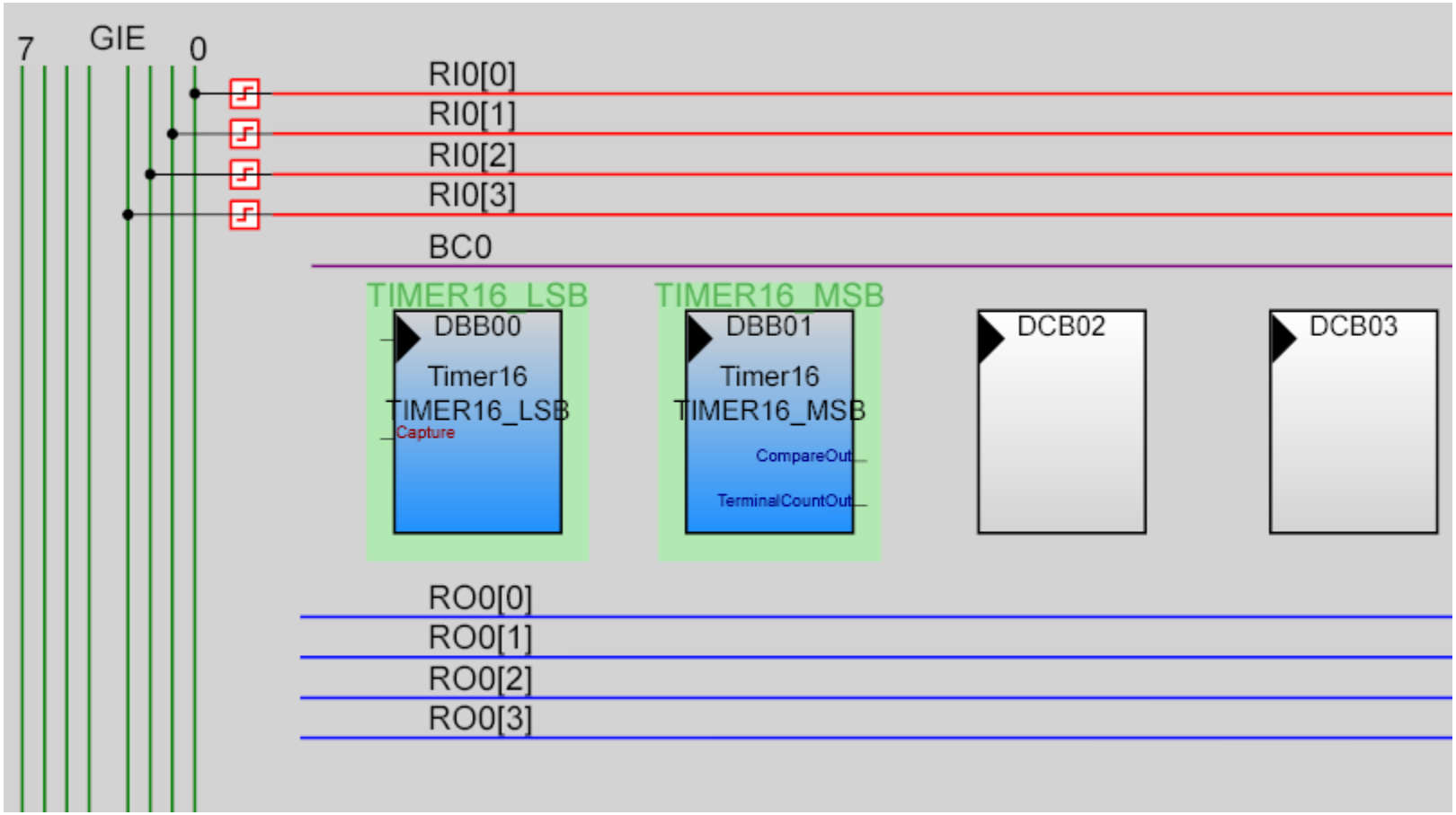
1. 新規プロジェクトを作成
   プロジェクト名はnoodle_timer
2. ターゲットデバイスをCY8C24894-24LFXI

CY3210基板を使用の場合は、
デバイス指定は、27443
または29466になりますので
ソケットのデバイスを確認して
デバイス名を選択してください

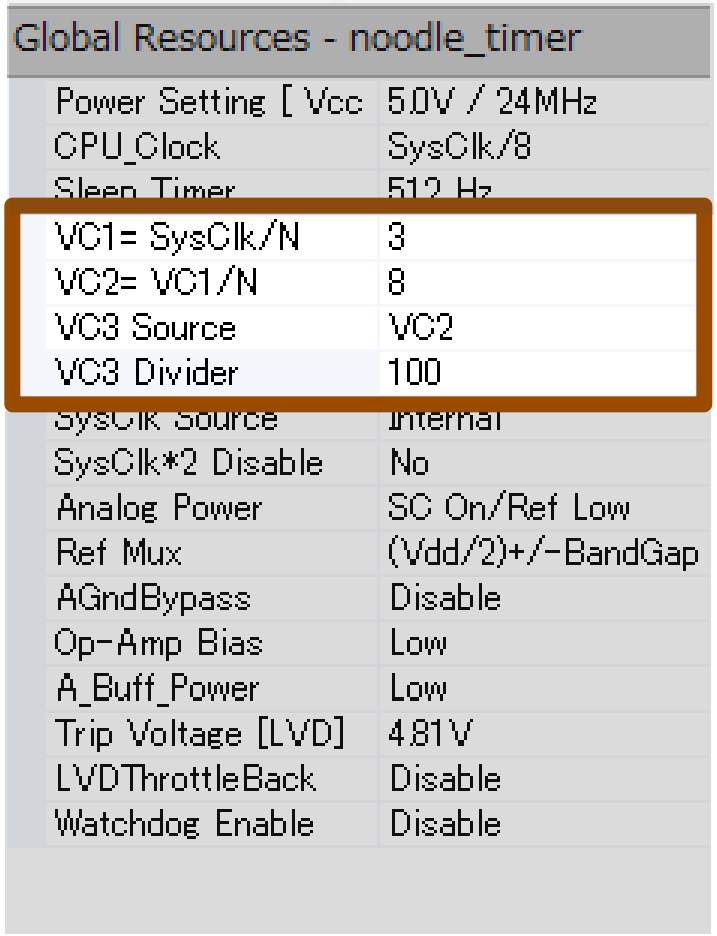
使用モジュールはTimer16，LCD
1秒ごとにイベント(割り込み)を起こす
今回はLCDへ割り込み回数を表示
3分たったら割り込みを停止（ラーメン・タイマなので）

ユーザーモジュールの追加

- Timers >>Timer16
- Misc Digital >> LCD(LCDにリネーム)をPort4へ



Global Resource



Global Resources - noodle_timer	
Power Setting [Vcc	5.0V / 24MHz
CPU_Clock	SysClk/8
Sleep Timer	512 Hz
VC1= SysClk/N	3
VC2= VC1/N	8
VC3 Source	VC2
VC3 Divider	100
SysClk Source	Internal
SysClk*2 Disable	No
Analog Power	SC On/Ref Low
Ref Mux	(Vdd/2)+/-BandGap
AGndBypass	Disable
Op-Amp Bias	Low
A_Buff_Power	Low
Trip Voltage [LVD]	4.81V
LVDThrottleBack	Disable
Watchdog Enable	Disable

本プロジェクトでは1秒を作るために、まず、IMO(24MHz)を2400分割して、10KHzを作ります。

24MHz÷3÷8÷100 = 10KHz

この10KHzをTimer16で10000カウントすることで、1秒を作ります。

User Module “Timer16” Parameter

Properties - Timer16

Name	Timer16
User Module	Timer16
Version	25
Clock	VC3
Capture	Low
TerminalCountOut	None
CompareOut	None
Period	9999
CompareValue	0
CompareType	Less Than Or Equal
InterruptType	Terminal Count
ClockSync	Sync to SysClk
TC_PulseWidth	Full Clock
InvertCapture	Normal

ClockはVC3(10kHz)

周期：10kHz/(9999+1)=1Hz
でTerminal Count(カウント終了)

割り込み条件：
Terminal Count(カウント終了)

PSoCにおける割り込み処理(C言語)

割り込み発生

割り込みコントローラが優先度順に処理

boot.asmの割り込みベクタテーブルから, ユーザーモジュール毎の割り込み処理に飛ぶ

ここからユーザーコード

ユーザーモジュールの割り込み処理記述部に,
main.cに記述した割り込み処理へ飛ばす
Long jump文を記述

main.cで割り込み処理を受け取るための#pragma文を記述

実際の割り込み処理を記述

実際の割り込み記述方法

割り込み処理をアセンブリで記述する場合
　　○○ int.asm にアセンブリを直接記述

割り込み処理をCで行う場合は○○int.asmから、C言語で記述された関数を呼び出します。

1. ○○int.asmに　“ljmp　_myISR”　を記述
   (バナーの内側に書くこと，名前は任意）
2. #pragma interrupt_handler myISR　をmain.cに記述
3. void myISR(){ } に処理をコード記述
4. 割り込みを許可するコードをmain.cに記述

＊○○：ユーザーが任意に名づけたUserModule名
　ex：Timer16→Timer16int.asm

boot.asm内の割り込みベクタテーブル

ここはGenerate Configuration をすることで、Designerによって自動設定されます。

Source Files >> boot.asm

```
122
123     org    20h
124     // call void_handler
125     reti
126
127     org    24h
128     ljmp    _Timer16_ISR
129     reti
130
131     org    28h                    ;PSoC Block DCB02 Interrupt Vector
132     // call void_handler
133     reti
134
```

ljmp _Timer16_ISR

割り込みが起きたらTimer16_ISRへ飛ぶ

Timer16INT.asmの編集

Library Source Files
>>Timer16int.asm

Timer16int.asm
ファイル全体

_Timer16_ISR以下のユーザーカスタム領域に
main.c割り込み処理を記述する 関数を定義する。
今回は　void myISR(void)　を使うので、
ljmp _myISR と記述する

```
62   _Timer16_ISR:
63
64       ;@PSoC_UserCode_BODY@ (Do not change
65       ;---------------------------------------
66       ; Insert your custom code below this
67       ;---------------------------------------
68
69       ljmp _myISR
70
71       ;---------------------------------------
72       ; Insert your custom code above this
73       ;---------
74       ;@PSoC_Us                          t
75
76       reti
```

記述

ユーザー
カスタム領域

割り込みが起きたら
myISRへ飛ぶ

main.cの編集

・main.c内に

```
#pragma interrupt_handler myISR
void myISR(void)
{
        //割り込み時の処理
}
```

と記述することで、割り込み時の処理を定義できる。

void myISRをmain関数から呼ぶことが無ければ、
記述位置はmain関数の下でも良い。

Application Editor − void main()

```
void main()
{
    LCD_Start();
    M8C_EnableGInt;
    Timer16_EnableInt();
    Timer16_Start();

    while(1){};
}
```

M8Cの割り込みを許可
Timer16の割り込みを許可

Application Editor−main.c

```
#include <m8c.h>
#include "PSoCAPI.h"

int TimeCount = 0;

#pragma interrupt_handler myISR
void myISR()
{
  TimeCount += 1;
  LCD_Position(0,0);
  LCD_PrHexInt(TimeCount);

  if(TimeCount >= 180){
    Timer16_DisableInt();
    LCD_Position(1,0);
    LCD_PrCString("RamenDekita");
  }
}
```

```
void main()
{
  LCD_Start();
  M8C_EnableGInt;
  Timer16_EnableInt();
  Timer16_Start();

  while(1){};
}
```

補足
3分が待てない方は、Debuggerを用いて、TimeCountの値を編集することで3分後の結果を見ることが出来ます。

割り込みの使い方まとめ

割り込み条件を設定する

- ユーザーが割り込み条件を設定できるUM
- 標準で割り込みが存在するUM
- GPIO

GenerateCofigurationを実行する

生成された〇〇int.asm内に、C言語で記述された割り込みサービスルーチンへのljmp文を記述する

- Timer16_1_int.asm(Timer16_1)
- Psocgpioint.asm(GPIO)

main.c に割り込みベクタを受け取る#pragma文と、割り込みサービスルーチンを記述する

main関数内に、割り込みを許可するAPIおよび、M8C_EnableGIntを記述する

割り込みコントローラ ブロック図

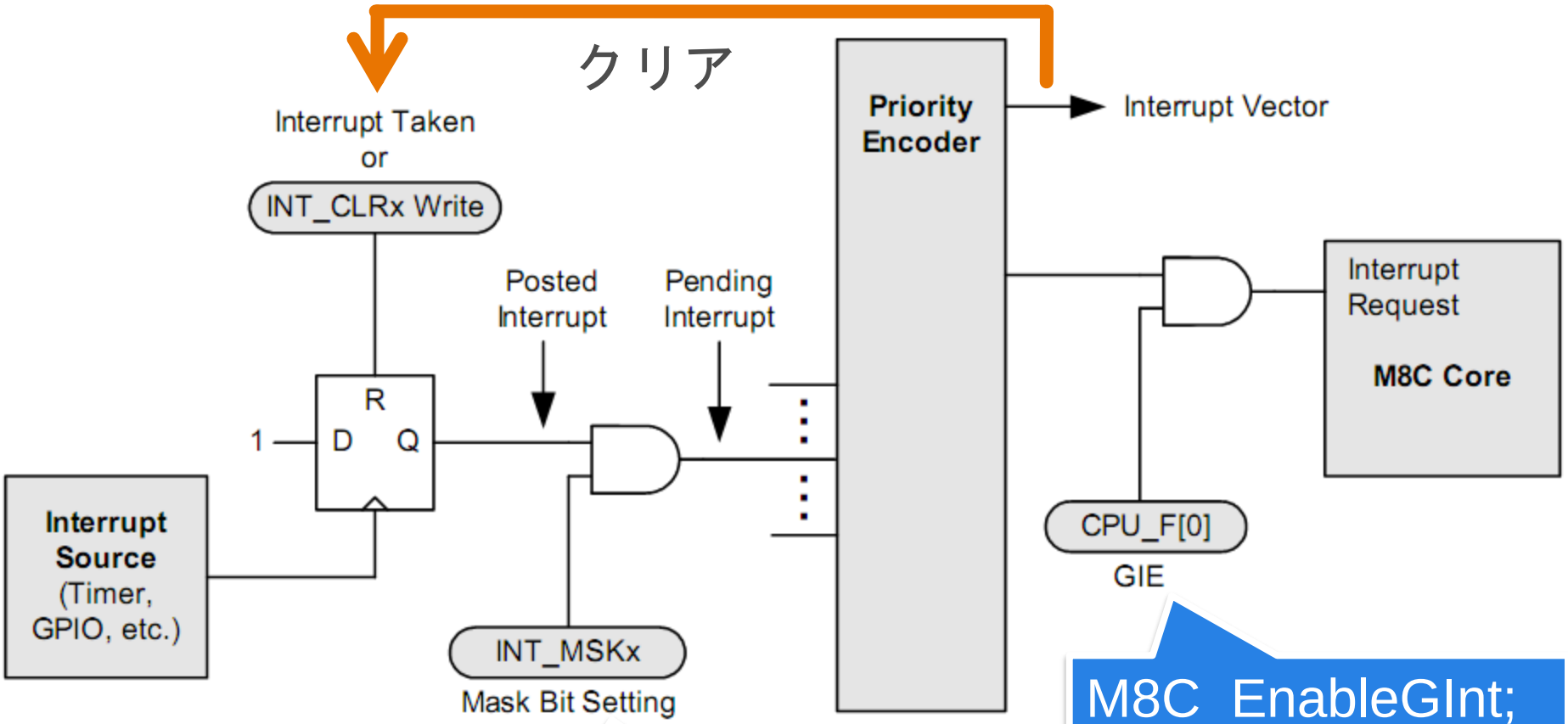


Figure 5 rupt Controller Block Diagra

Timer16_EnableInt();
Timer16_DisableInt();

M8C_EnableGInt;
M8C_DisableGInt;

TRM 89ページより

(補足その1) 多重割り込みに関して

割り込み中の割り込みはどう処理されますか?

現在の割り込みサービスルーチンが続行されます。これは,割り込み処理中は自動的にGIE(全割り込み許可)がクリアされ、二つ目以降の割り込みは無視されるからです.

ただし、割り込みがあったという情報は保持されます。

現在、実行中の割り込み処理が終了すると、GIEがセットされ、保持されていた割り込みの中から、優先度(TRM 91ページ)に応じて次の割り込みから処理が開始されます。

優先度概要(TRM 91ページ)

リセット >> 電源モニタ >> GPIO >> デジタルブロック(左上から) >> I2C >> SleepTimer

補足その2 Hi-TECHコンパイラでの変更点

割り込み記述方法がImageCraftとHi-TECHで若干異なります.（ 旧Hi-TECHでは割り込み処理関数の引数の後に, 割り込みベクターアドレスを記述.）

旧記法(Hi-TECHスタイル)

void timer16_isr(void)@**0x24**{割り込み処理}

新記法(ImageCraftスタイル)

void timer16_isr(void){割り込み処理}

ベクターアドレスは, TRM 91ページ もしくは boot.asmを見ると簡単に調べることが出来ます.

	ImageCraft	Hi-TECH
旧記法	○	△(warning)
新記法	×(build出来ない)	○

Memo

フォローアップURL

http://mikami.a.la9.jp/meiji/MEIJI.HTM

担当講師

三上廉司(みかみれんじ)

Renji_Mikami(at_mark)nifty.com (Default - Recommended)

mikami(at_mark)meiji.ac.jp (Alternative)

http://mikami.a.la9.jp/_edu.htm