参考資料

JAJA230A– 2012 年 8 月

TMS320C2000:Piccolo MCUのソフトウェア開発入門

東京第二営業部　アプリケーション技術グループ
小幡　智

アブストラクト

この資料は、Piccolo(TMS320F2806x/F2803x/F2802x) MCUシリーズのソフトウェア開発を始める際に、必要となる開発ツールやデバイスの基本的な知識を身につけて頂くための、入門書となっています。開発ツールCode Composer Studio ver5.x 及び開発リソース管理ツールのcontrolSUITE、contolSUITEに含まれるライブラリやヘッダファイルの基本的な使い方、Piccolo MCUの基本的な動作、割り込み、PWM、ADC、Flash、IQmathライブラリ、SCI(UART)、CLA、DMAの使い方をサンプルコードを交えて解説します。

目次

1 開発の準備：使用する評価ボードと、Code Composer Studio 5.x と controlSUITE

1.1 この章の目的

この章では、このドキュメントにて使用している評価ボードの解説、開発ツールCode Composer Studio 5.xのライセンスとインストールについて、C2000(Piccoloシリーズを含む、C28x/F28x製品群)開発リソース管理ツールのcontrolSUITEについて解説します。

1.2 このドキュメントに掲載されているコードと使用評価ボード

TIまたはサード・パーティから各種評価ボードがリリースされていますが、今回このドキュメントにて使用しているサンプル・コードは、以下のボードで動作確認をしております。

- F2802xシリーズ
  - F28027 Piccolo controlSTICK (TMDS28027USB)
  - F28027 Piccolo Experimenter's kit (TMDSDOCK28027)
  - C2000 Piccolo LaunchPad(LAUNCHXL-F28027)
- F2803xシリーズ
  - F28035 Piccol Experimenter's Kit (TMDSDOCK28035)
- F2806xシリーズ
  - F28069 Piccolo controlSTICK (TMDX28069USB) (TMDXのXは、量産未認定品のサンプルを使用している事を意味します。執筆している時点では、まだF2806xシリーズは量産認定前になります。量産認定後のサンプルを使用している場合は、TMDSに変更される予定です)
  - F28069 Piccolo Experimenter's Kit (TMDXDOCK28069)(TMDXのXについては、上記と同じ意味になります。)

F2802x、F2803x、F2806xの各シリーズはシリーズ内で同じパッケージであれば、完全ピン互換のデバイスになります。シリーズが違う場合(例えばF2802xとF2803x等)は、ピン互換性は全くありません。シリーズ内では、基本的には番号が大きいデバイスの方が機能は豊富になる傾向にあります。F2802xではF28027が、F2803xではF28035が、F2806xではF28069が一番機能の豊富なデバイスで、シリーズ内のデバイスは、この一番機能が豊富なデバイスから、機能が削除されていったデバイスになります。
F2802xもF2803xもF2806xも同じPiccoloシリーズを名乗っている事からわかりますが、基本的なプラットフォームは共通になっていますので、機能としては非常に似ているシリーズです。細かい仕様の違いは、各デバイスのデータシート及びリファレンス・ガイドをご参照頂きたく思いますが、非常に大雑把に話しますと、全体的な機能を比較すると、

F2802x < F2803x < F2806x

という傾向にあります。
2012年に、F2802x0(F280220/F280230/F280260/F28070)シリーズという、新しいシリーズのPiccoloがリリースされました。このシリーズは、基本的にはF2802xシリーズとほぼ同じで、F2802xから一部機能が削られていると考えて頂ければと思います。F2802x0用の評価ボードがリリースされていないため、このドキュメントでは特にこのシリーズについて記載しておりませんが、F2802xとほぼ同等ですので、F2802xの項目を参照頂ければ参考になると思います。尚、F2802x0用には、F2802x0用のHeader Filesが、F2802x用とは別に用意されています。

1.3 Code Composer Studio 5.x のダウンロードとインストール

C2000シリーズMCUは、TI製の統合開発環境、Code Composer Studio(以下CCS)を使って開発します。執筆している今日現在の最新バージョンは、5.2.1になります。これ以降は、5.2.xを使った場合で説明します。特にCLAを使うユーザーは、5.2.0以上のバージョンにして下さい。このドキュメントでは、5.2.0以上がインストールされている事を前提とします。

CCSは、従来のVer3.xからVer4.xになった時に、大幅に変更が入り、完全にＴＩオリジナルの環境から、オープンソースのEclipse環境に変更されました。そのため、従来のVer3.xとは、全く使用方法が変わりましたので、Ver3.x以下のバージョンを使用されていたユーザーはご注意下さい。Ver5.xはVer4.xと同じEclipse環境です。アップデート点は多いですが、Ver3.xからの変更程、劇的な変更ではありません。Ver4.xを使った事のあるユーザーでしたら、それ程戸惑う事はないと思います。Code Composer Studio Ver5.xは複数のライセンスが用意されています。ここでは、非常に簡単に説明しますが、必ずしも正確ではない可能性があります。また、あくまでも執筆している時点での話になりますので、アップデートされる可能性もあります。詳細は必ず最新の各License Agreementsを熟読して下さい。このドキュメントに記載されている内容とLicense Agreementsに違いがある場合は、License Agreementsに記載されている方が有効です。各ライセンスは、以下のサイトから参照できます。
http://processors.wiki.ti.com/index.php/Licensing_-_CCS

無償版	
Evaluate License	90日限定で使用できるライセンスです。機能的な制限はありません。
Free License	デバッグ環境が、以下の３つのケースに制限されます。コードサイズ制限、日数制限はありません。 1.XDS100(v1/v2)を使用する場合 2.TI製もしくは3rdパーティー製のオンボード・エミュレータが搭載されている評価モジュール及びスタータキット(MSP430のeZ430はサポートされていません) 3.シミュレータ XDS100を使う場合は、商品開発であっても、このライセンスが使えます。XDS100を購入すれば、ＣＣＳが無料で使えますので、最も安価な開発環境となります。MSP430ではこのライセンスは使用できません。
Code Size Limited License	MSP430シリーズのみで使用できます、MSP430は16KBのオブジェクトコード・コード・サイズ制限があります。注意:Ver4.xではC2000もこのライセンスが用意されていますが、Ver5.xでは無くなりました。
有償版	
Node Locked License	製品版のCCSです。Ver4.xまでは、MCU(MSP430/C2000/Stellaris)だけが使用できるMCU限定バージョンと、その他の全てのデバイスが使用できるバージョンの２種類がありましたが、Ver5.xではMCU限定バージョンは無くなりました。

表 1:CCS のライセンスの大雑把な説明(詳細は必ず各 License Agreements を確認して下さい)

有償版のライセンスをお持ちでインストールされているユーザーは、それをそのままご利用下さい。CCSのバージョンが古い場合は、アップデートして下さい。キットに付属されていたCCSは、出荷時期などにより、CCS5.2以上ではなくCCS5.1/CCS4.x/3.xの場合があります。その場合は、CCS5.2以上をダウンロード下さい。CCSの5.1がインストールされている場合、CCS5.xへのアップデートは、Help→Check for Updatesからできます（インターネットに接続されている必要があります）。

CCSをお持ちでないユーザーは、以下のサイトから最新のＣＣＳをダウンロードする事ができます。CCS4.xでは、使用するライセンスによって、ダウンロードするファイルが異なっていましたが、Ver5.xでは、全てのライセンスで共通です。インストール後にライセンスを選択します。

http://processors.wiki.ti.com/index.php/Download_CCS

今回、このドキュメントにて対象となっている評価ボードには、XDS100 JTAGエミュレータが搭載されているボードになりますので、有償版のライセンスをお持ちでないユーザーは、Free Linceseを選択してください。

1.4 controlSUITE

ソフトウェア・ライブラリ、ヘッダファイル、ドキュメント等、C2000 MCUを使ったアプリケーションを開発するための複数のリソースがTIから提供されています。従来は、それらは、個別にダウンロードする必要がありました。このため、ユーザーは、TIが提供しているリソースを全て所持しているのかが把握しにくく、また、最新のものかどうかは、それぞれ個別にチェックする必要がありました。これらを改善するために、TIでは、C28x MCU向けにcontrolSUITEというツールを用意しています。このcontrolSUITEをインストールする事で、これらの必要なリソースを一括ダウンロードし、管理してくれます。また、アップデートの自動チェックを行う事もできます。C2000の開発を始める際には、このcontrolSUITEを必ずインストール下さい。controlSUITEは、

http://www.ti.com/controlsuite

からダウンロードできます。これ以降は、controlSUITEがインストールされている事を前提とします。特に今回のドキュメントでは、このcontrolSUITEにてインストールされる、C/C++ Header Files and Peripheral Examplesを使用します。このファイルはまた後ほど説明しますが、デフォルトの場所にインストールした場合は、

C:\ti\controlSUITE\device_support

の下に、デバイス・プラットフォーム毎にディレクトリ分けされて、インストールされます。できるだけ、デフォルトの場所にインストールしてください。**このドキュメント及び、このドキュメントに付属されているサンプル・コードは、controlSUITEがデフォルトの場所にインストールされている事を前提としています。もし、デフォルト以外の場所にインストールしている場合は、ドキュメントの内容をよく読み、インストールされている場所の違いをよく考えて、作業を進めてください。また、付属されているサンプル・コードの一部は、そのままでは動作しない事に注意して下さい。**

1.5 F2802x/3x/6x シリーズと CCS5 の対応

F2802x/3x/6xシリーズはCCS5の最初のバージョンから対応しています。CLA用のCコンパイラを使う場合は、CCS5.2以上を使う事を推奨します。CCS5.1では、設定を変更する必要があります。また、CLA用のCコンパイラはコンパイラのバージョンが6.1.0以上である必要があります。

1.6 このドキュメントに付属されているサンプル・コード例について

このドキュメントにて解説されているサンプル・コードが付属されていますが、数点注意事項があります。

1. ヘッダ・ファイル(C/C++ Header Files and Peripheral Examples)が含まれていません

このドキュメントに紹介されているProjectの構成では、ヘッダ・ファイルはProjectの下のディレクトリにおかれている事が前提となっています。これは、初心者にとっては一番理解しやすい構成方法と考えたためです。一方、この方法をとるとProjectごとに同じヘッダ・ファイルをもつ事になり、ファイルのサイズが増えてしまいます。そのため、今回の付属コードでは、ヘッダ・ファイルはProjectディレクトリには含まれていません。ユーザーにてヘッダ・ファイルをダウンロードしていただき(controlSUITEです)、

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ
F2802xの場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_commonフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_headersフォルダ

	C:\ti\controlSUITE\device_support\f2802x\v200\DSP28x_Project.h C:\ti\contorlSUITE\device_support\f2802x\v200\F2802x_Device.h

を、各Projectの下にコピーして下さい。

図 1は、F2802xの場合です(F2803x/F2806xの場合はDSP28x_Project.h及びF280xx_Deice.hのコピーは必要ありません。F2803x/F2806xの場合は、この両ファイルはDSP2803x_common及びF2806x_commonディレクトリの中にあるためです)、各Projectの下にコピーしてください。コピーされた後のProjectの様子は(この例ではProject1フォルダの中)図 2のようになります。ヘッダ・ファイルにつきましては、3章に詳細が掲載されていますので、まずは、こちらを熟読してから、この作業を行ってください。

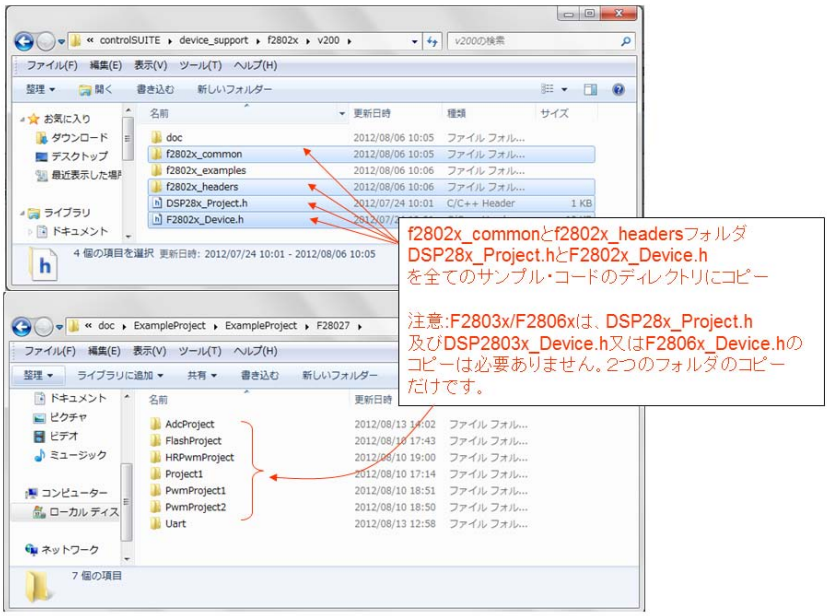


図 1:ヘッダ・ファイルのコピー(F2802x の場合)

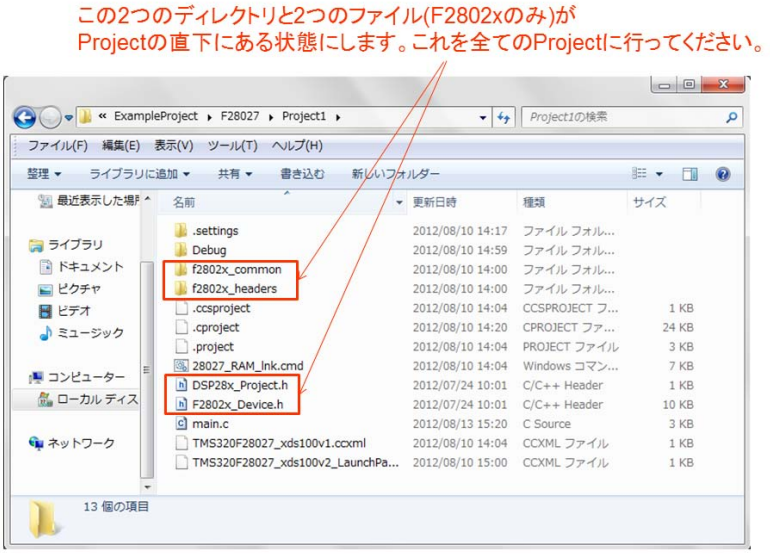


図 2:コピーされた後の Project 直下の構成(F2802x の場合)

2. 一部Projectでは、ヘッダ・ファイルの変更が必要です。

変更が必要なProjectは、F28069用の7章で紹介されている全てのProject(PwmProject1, PwmProject2, HRPwmProject)です。変更箇所は

F28069_common\include\F2806x_Exapmles.h

のファイルで、

#define DSP28_PLLCR 16 // Uncomment for 80 MHz devices [80 MHz = (10MHz * 16)/2]
をコメントアウトして、

#define DSP28_PLLCR 12
のコメントをはずして下さい。

この作業は、7章でも紹介されていますが、F28069をデフォルトの80MHzではなく、60MHz動作に設定を変更するためのものです。（変更しない場合は、PWMキャリア周波数がドキュメントの内容と変わってしまいます）。このドキュメントでは、F2802x/F2803x/F2806xで同じソースコードを使っている関係で、F28069だけ80MHz動作にしてしまうと、#ifdef-#else-#endifが多くなりコードが読みにくくなってしまうため、60MHz動作に統一しました。

3. 付属のサンプル・コードのF28035用のHRPWMサンプルコードは、controlSUITEがデフォルトのディレクトリにインストールされている事を前提としています。これは、IQmathLib.hとIQmath.libをデフォルトのインストール先から参照しているためです。デフォルトのディレクトリにインストールされていない場合は、IQmathLib.hのインクルード・パスと、IQmath.libのリンク先を変更する必要があります。

4. 付属のサンプル・コードのF28027用のTarget Configurationファイル(.ccxmlファイル)は、Explerimeter's Kit及びcontrolSTICKに搭載されているXDS100v1の設定になっています。C2000 LaunchPadを使用する場合は、この設定を変更して下さい。変更するためには、図 4のように、Project Explorerウィンドウにて、Project名を右クリックし、Propertiesを選択して下さい。図 3のように、Properties for プロジェクト名ウィンドウが表示されます。Device欄のConncectionが、Texas Instruments XDS100v1 USB Emultatorと設定されているはずです。ここをTexas Instrumetns XDS100v2 USB Emulatorに変更して下さい。

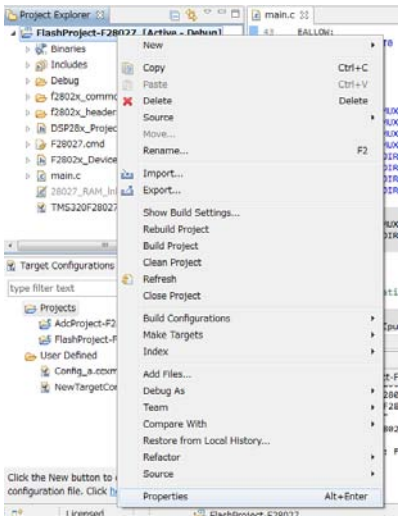


図 4：Target Configuration ファイルの変更①

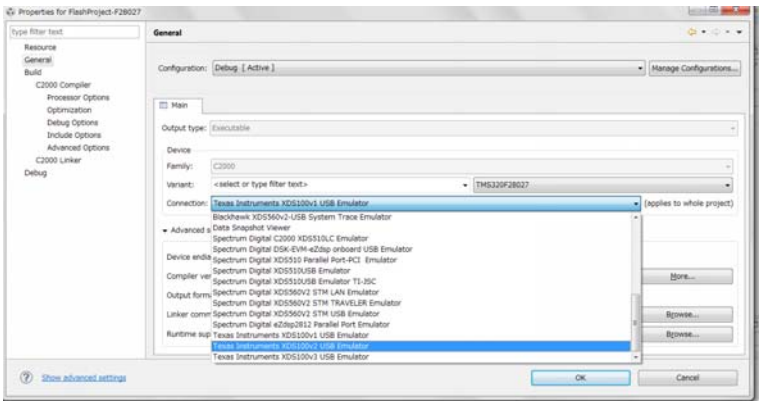


図 3：Target Configuration ファイルの変更②

2 Code Composer Studio を使って、実機を動作させる

2.1 この章の目的

前章にて、CCSのインストールおよびcontrolSUITEのインストールが完了しているとします。この章では、controlSUITEに含まれるサンプル・コードをCCSで動作させる事により、CCSの基本的な使い方を解説します。

2.2 CCS の起動と Project の追加

まず、インストールしたCCS5. x を起動してください。このドキュメント作成時のCCSのバージョンが5.2.1ですので、以降は5.2.1を使った場合について記述します。バージョンが違う場合は、若干内容が異なっている可能性があります。起動すると、図 5のような、Workspace Launcherが立ち上がります。

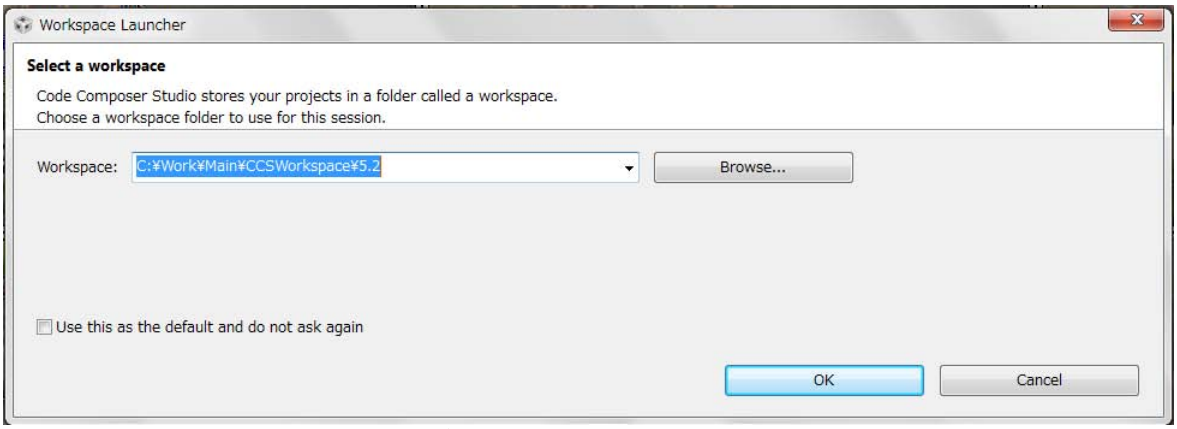


図 5:Workspace Launcher

CCS5.xでは、Workspaceという単位で、CCSの設定を管理します。このWorkspaceには前にCCSをExitした時点での設定が記録されていますので、ProjectやCCSの設定等を再現する事ができます。複数のProjectを開発する場合には、Workspaceを分けておくと便利です。Workspaceはデフォルトのままでもかまいませんし、どこか希望する適当なディレクトリを選択してもかまいません。Workspaceを選択して、OKをクリックして下さい。ここで注意事項があります。CCSを使う上で全体的に言える事ですが、ディレクトリ名は日本語を含まないディレクトリを使用してください。ディレクトリ名に日本語が入っている場合は、正しく動作しない事があります。選択したWorkspaceが初めての起動の場合は、TI Resource Explorer Window内のWelcome画面が表示されます。今回はこの画面は特に関係ありませんので、このTI Resource Explorerをcloseして下さい。それでは、まず、Projectを登録します。

File→Importを選択して下さい。

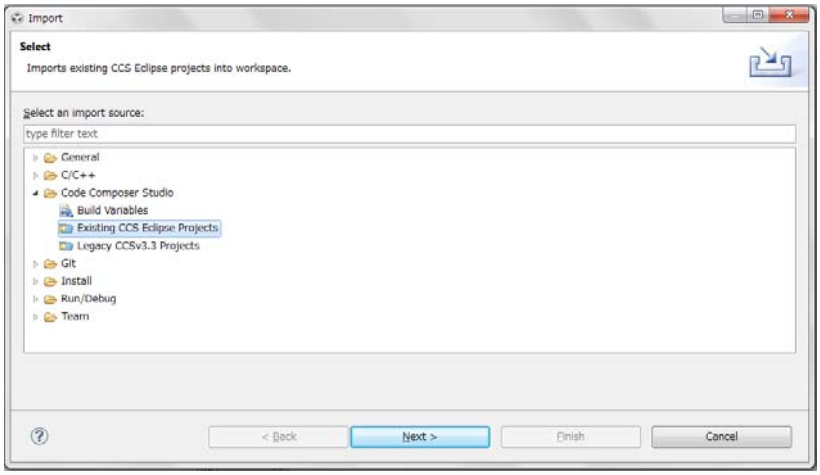


図 6:Import 画面

図 6のように、Importウィンドウが表示されますので、Code Composer Studio – Existing CCS Eclipse Projectsを選択して、Next > をクリックして下さい。

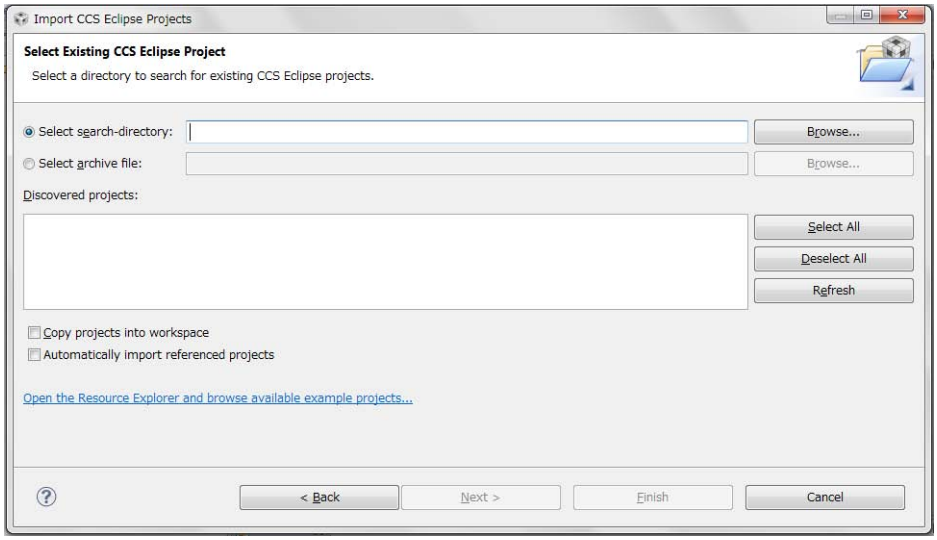


図 7:Import CCS Eclipse Projects 画面

図 7の画面が表示されますので、Select search-directory欄にて、

F28069の場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_examples
F28035の場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_examples_ccsv4
F28027の場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_examples

コラム:C2000とDSPとDSCとMCU

C28xシリーズにて最初にリリースされた製品は、F281x/C281xというシリーズです。C28xの基本コンセプトは、CPUにはTIが得意とする、DSP(Digital Signal Processing)処理が得意なDSP CPUを搭載し、従来DSPにはインテグレーションされていなかったADCやFlashを統合したデバイスです。ワンチップ・マイコンという言葉がありますが、まさにこのワンチップ・マイコンのCPUがDSP CPUに置き換わったプロセッサです。当時は、そのようなプロセッサに対して特に一般的な呼び名はなく、単にDSPと言っていました。C28xも次の世代のF280xシリーズでも、同じくDSPと呼んでいました。ある時から、DSP CPUをCPUとする、ワンチップ・マイコンのようなプロセッサに対して、DSC(Digital Signal Controller)という名称が使われるようになりました。そのため、次のF2823x/F2833x世代では、DSPからDSCと名称を変更しました。次のPiccolo世代では、今度はDSCからMCUと名称を変更しました。CPUはDSP技術がベースになっているものの、電源が単電源になったり、オシレータ、コンパレータ等インテグレーションが進み、DSPやDSCと呼ぶよりは、MCUと呼んだ方がふさわしいという判断によります。DSP→DSC→MCUと、名称は変わっていますが、CPUに変更があったわけではなく、あくまでも名称だけが変わって

いったとご理解頂ければと思います。このような歴史のために、C28xのドキュメント、ソフトウェアライブラリ等、DSPやDSCと言った言葉が今でも数多く残っています。理由は上記のとおりですので、特に気になさらないで下さい。

を選択してください。ここで、インストールされているcontrolSUITEのバージョンによっては、v130/v126/v200よりも最新のバージョンになっているかもしれません。その場合は、恐らく最新のバージョンでもかまわないと思いますが、将来のバージョンの変更は予想がつきませんので、このドキュメントの内容を試す場合はできるだけ、このバージョンをご利用頂ければと思います。F2803xの場合は、選択したフォルダ名にccs4とありますが、CCS4.xのプロジェクトは、CCS5.xでも使用できまので問題ありません(CCS5.xのプロジェクトはCCS4.xでは使用できません)。

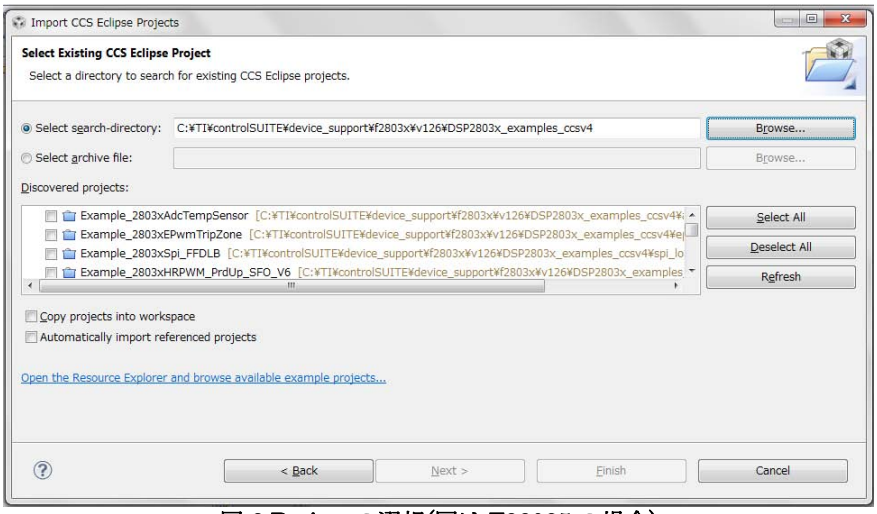


図 8:Project の選択(図は F28035 の場合)

図 8のように、Discovered projects欄に多くのProjectが表示されます。このディレクトリには、F2806x/F2803x/F2802x上で動作するたくさんのサンプルコードが用意されています。今回は、Exapmle_28069_Flash/Exapmle_28035_Flash/Example_28027_Flashを試してみたいと思いますので、

F28069の場合	Exapmle_28069_Flashを選択
F28035の場合	Example_28035_Flashを選択
F28027の場合	Example_28027_Flashを選択

して、Finishをクリックして下さい。これで、Projectに Example_28069_Flash / Example_28035_Flash / Example_28027_Flashが登録されました。

このProjectはTIが用意しているサンプルコードになりますので、特に何もコードを変更しなくても動作するべきなのですが、残念ながら、F28035用のv126(今後のバージョンでは修正されると思います)のコードには不具合があります(v126より前のバージョンではこの不具合はありません)。v126のF28035用のコードを使う場合は、Example_2803xFlash.cの152行目(以下の一行)、

```
memcpy((uint16_t *)&RamfuncsLoadStart,(uint16_t *)&RamfuncsRunStart, (unsigned long)&RamfuncsLoadSize);
```

にて、コピー元アドレスとコピー先アドレスが逆さまになっていますので、以下のように修正してください。

```
memcpy((uint16_t *)&RamfuncsRunStart, (uint16_t *)&RamfuncsLoadStart, (unsigned long)&RamfuncsLoadSize);
```

F2802x/2806x用のコードは、この不具合はありません。この不具合は既にUSにレポートされていますので、今後修正されるはずです。v126より後のバージョンを試す場合は、この不具合が残っているかどうか確認して下さい。

それでは、実機と接続してデバッグをしてみましょう。対象とするボードは、前章にて解説していますので、ご参照下さい。まず、

Experimeter's Kitの場合	キットとPCをUSBで接続し、SW1を“USB“にして下さい。これでこのボードの電力はＵＳＢから供給されます。さらにこのＵＳＢにより、PCとPiccolo MCUが、オンボードＪＴＡＧ回路(XDS100v1)を経由して接続されます。
controlSTICKの場合	キットをPCのUSBに接続してください。これで、このボードの電力はＵＳＢから供給されます。さらにこのＵＳＢにより、PCとPiccoloが、オンボードＪＴＡＧ回路(XDS100v1)を経由して接続されます。
C2000 Piccolo LaunchPadの場合	キットとPCをUSBで接続して下さい。JP1/JP2/JP3のジャンパが接続されている事を確認して下さい。JP1/JP2/JP3は、JTAG回路(XD100v2)とPiccoloの回路を絶縁するかどうかを選択するジャンパです。絶縁したい場合は、このJP1/JP2/JP3のジャンパを外します(別途、Piccolo側の電源が必要となります)。JP1/JP2/JP3のジャンパが接続されている場合は、USBから給電され、絶縁されません。C2000は主にパワー・エレクトロニクス・アプリケーションで使用されますので、絶縁はそのための配慮です。

これらキットには、XDS100v1またはv2(v1とv2の違いは主に速度で、v2では速度に劇的な改善が見られます)という低価格エミュレータが搭載されていて、別途エミュレータがなくてもJTAGデバッグを行う事ができます。controlSTICK及びExperimenter's KitはXDS100v1が搭載されています。C2000 Piccolo LaunchPadは、XDS100v2が搭載されています。これは、C2000 Piccolo LaunchPadの方が後でリリースされていますので、最新のXDS100 JTAGが搭載されているためです。CCS5.xがインストールされていれば、XDS100v1/v2のWindowsドライバは自動的にインストールされていますので、ボードを接続すると、正しいドライバが自動的に選択されるはずです。

さて、exampleに収められているサンプル・プロジェクトには、JTAG接続の設定がされていませんので、JTAG接続の設定(Target Configurationファイルの作成)を行いましょう。

Project→Properties

を選択して下さい。図 9のように、Properties for プロジェクト名 ウィンドウが表示されます。DeviceのConnection欄が現在は空欄になっていると思いますが、ここを、

C2000 LaunchPadの場合	Texas Instruments XDS100v2 USB Emulator
Experimenter's Kit及びcontroSTICKの場合	Texas Instruments XDS100v1 USB Emualtor

を選択し、OKをクリックして下さい。

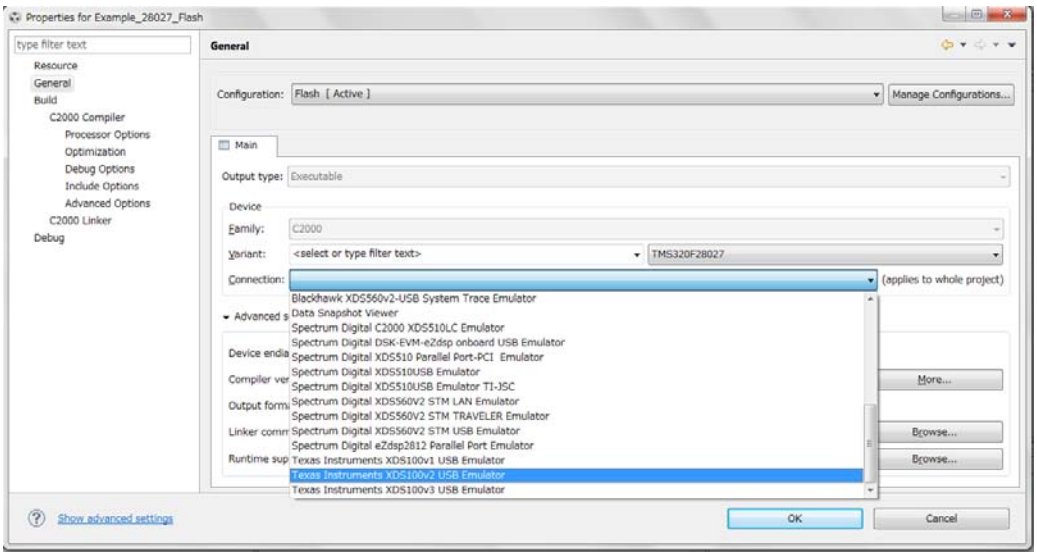


図 9：Target Configuration ファイルの作成

OKをクリックすると、図 10のように、Project Explorereウィンドウのプロジェクトに、デバイス名.ccxmlファイルが追加されているはずです（図 10はF28027の場合でTMS320F28027.ccxml[Active]が、追加されたTarget Configurationファイルです）。

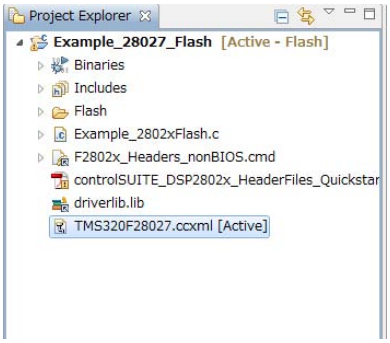


図 10：追加された Target Configuration(.ccxml)ファイル

JTAG接続の設定ができましたので、デバッグを開始しましょう。

Run→Debug

を選択して下さい。以下の図 11の虫マークをクリックしても同じ動作になります。

図 11:Debug Active Project ボタン（以降、虫マーク）

ビルドが始まり、ビルド終了後、デバッガが立ち上がり、先ほどビルドした実行ファイルがダウンロードされます。このプロジェクトはF28069/F28035/F28027のFlashにプログラムをダウンロードするため、少し時間がかかる可能性があります。まれにCCSの機嫌が悪いと、何度試しても接続に失敗する事があります。このような時は、CCSを一旦Exitし、ボードをPCから抜いて、再度ボードを指して、CCSを立ち上げてお試しください。

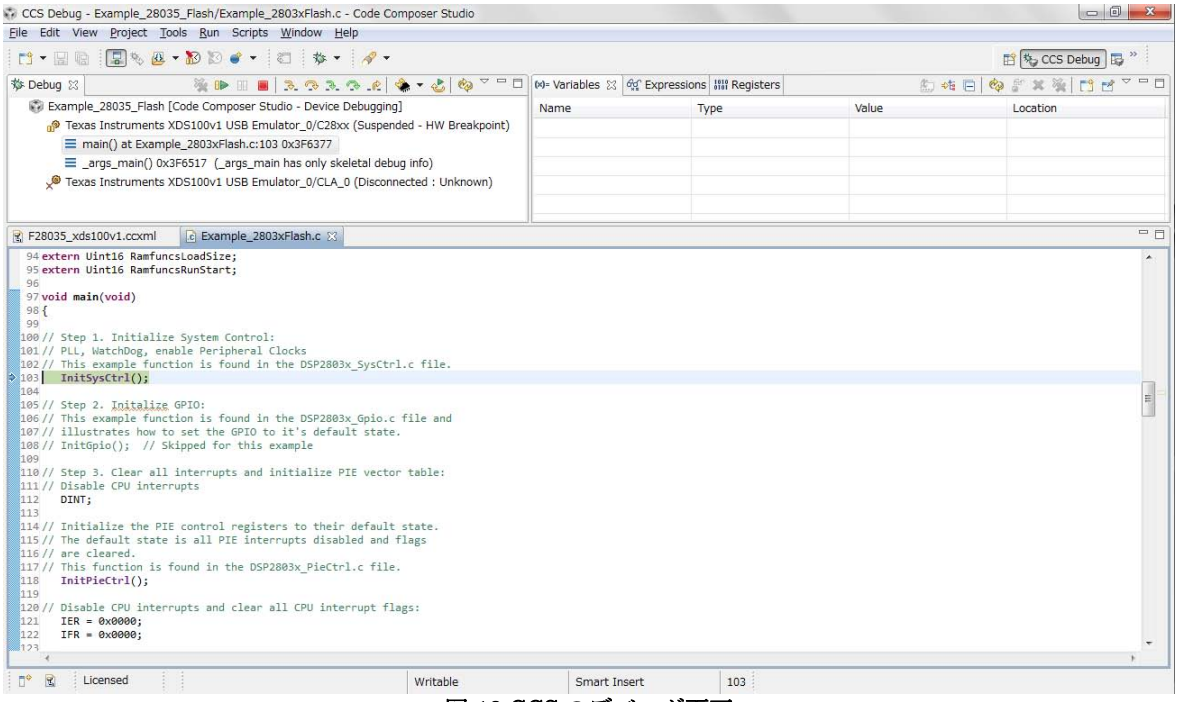


図 12:CCS のデバッグ画面

デバッガが立ち上がると、上記のような画面になると思います(F28027の場合は、main()のコードが図とは全く違うコードになっているはずです。この章の目的は、コードの中身を理解するのではなく、デバッグの方法を解説する事にありますので、コードの中身は気にしないで下さい)。

CCSでは、コード開発とデバッグ画面の２つを切り替えて使います。画面の右上に、図 13のような箇所があります(環境によっては、CCS Editが隠れている場合もあります。タブの左端をつかんで、広げてください)

図 13:デバッグ画面(CCS Debug)とコード開発画面(CCS Edit)の切り替えボタン

CCS Debugボタンをクリックすると、デバッグ用の画面が、CCS Editボタンをクリックすると、コード開発用の画面が表示されますので、作業にあわせて、切り替えて下さい。

コラム：CCSのデバッガ起動について

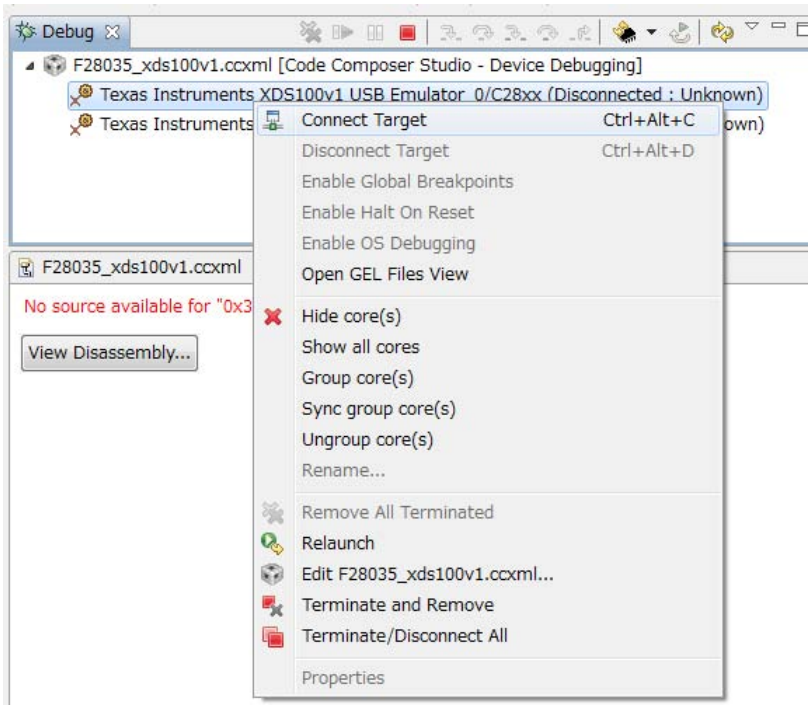
このコラムで記載する内容は、少し余談的な話になります。必ずしも読む必要はありません。

CCSでは、デバッガを起動する方法は主に２つあります。ひとつは、既に説明しました、Run→Debug(もしくは、虫マークをクリック)になります。こちらが一番簡単な方法です。この方法の意味をもう少し掘り下げます。CCSでは、デバッガの設定(どのデバイスを、どのエミュレータでの設定)は、ccxmlファイルによって定義されます。

このccxmlファイルは、複数用意する事ができます。先ほど、プロジェクトのProperties設定から自動生成する方法を示しました。この方法で作成すると(もしくは次の章で試しますが新規Project作成時でも、同様に自動生成する事ができます)、ccxmlファイルの横に自動的に[Active]マークがつきます。この、[Active]設定は、Projectを一旦Deleteし、再びImportした場合でも、再び[Active]マークがつきます。Run→Debug(もしくは、虫マークをクリック)する方法では、この[Active]に設定されているccxmlファイルを使って、デバッガが立ち上がります。もし、[Active]に設定されているccxmlファイルが無ければ、[Default]に設定されているccxmlファイルが使用されます。(実際にはDefaultはあまり利用しません)

ccxmlファイルは、Propertiesから自動生成するほかに、File→New→Target Configuration Filesを選択して、より詳細な設定をする事もできます（例えば、JTAGチェーンを組んでいる場合など）。しかし、この方法で作成されたccxmlファイルはデフォルトで[Active]マークがつきません。[Active]マークをつけるためには、Project Explorerから該当するccxmlファイルを右クリックして、Set As Active Target Configurationを選択します。しかし、このProjectをDeleteして、再びImportするとこの[Active]マークが消えてしまうため、再度[Active]にする必要があります。そのため、File→New→Target Configuration Filesで作成するよりも、Properties設定から自動生成する方が、便利なのです。

ちなみに、[Active]マークがついていないccxmlファイルからも、デバッガを立ち上げる事はできます（筆者は個人的にはこの方法を好んで使っています）。View→Target Configurationsを選択して、Target Configurationsウィンドウを立ち上げます。このウィンドウには、現在CCSに登録されているccxmlファイルが表示されます。Projectsに表示されているのが、プロジェクトに登録されているccxmlファイルです。User Definedに表示されているのが、CCSのShared Location(Shared Locationを設定するには、Window→Preferencesにて、Preferencesウィンドウを立ち上げ、Code Composer Studio－DebugのShared target configuration directory欄を変更します)に登録されている(つまりプロジェクト非依存)ccxmlファイルです。このウィンドウにて、希望するccxmlファイルを右クリックし、Launch Selected Configurationを選択すると、[Active]もしくは[Default]ではないccxmlファイルの設定でデバッガを立ち上げる事ができます。ただし、この場合は、虫マークとは違って、デバッガを立ち上げるだけで、ビルドやコードのロードは行いません。また、この操作でデバッガを立ち上げた場合は、PCとJTAGエミュレータが接続されただけで、デバイスはまだ接続されていません(設定を変更すれば、自動的に接続するよう変更もできます)。デバイスとの接続を完了するためには、下図のようにDebugウィンドウの対象デバイス(図ではTexas Instruments XDS100v1 USB Emulator_0/C28xx (Discconnected：Unknown)を右クリックし、Connect Targetを選択して下さい。これで、デバイスとのJTAG接続が完了します。



また、F2806x/F2803xの場合は注意が必要です。F2806x/F2803xの一部デバイスは、C28xメインCPUとCLAという２つのCPUが存在するために、Debugウィンドウには、2つの項目がでてきます。上がC28xメインCPUの方です。CLAをデバッグしない限り、CLAの方はそのまま未接続でかまいません。

デバイスと接続を完了しましたら、Target→Load Programにて、プログラムをロードする事ができます。

2.3 デバッグの基本操作

前節にてビルドが完了(少しWarningが出ると思いますが、今回は気にしなくて結構です)し、デバッガが立ち上がり、F28069/F28035/F28027 MCUと接続ができ、プログラムがロードされました。

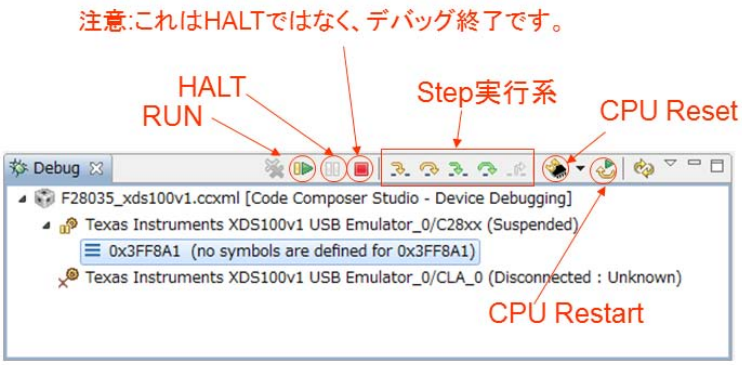


図 14:Debug 操作ウィンドウ

RUN（コードの実行）、HALT(実行の停止)、Step実行等、Debugウィンドウのボタンにて行います。図 14によく使うボタンの説明をします。赤い四角のボタンは、HALTではなく、デバッグ終了ですので、ご注意下さい。筆者は、今でもよく間違えてこのボタンを押してしまって、悲しい思いをしてしまう事が良くあります。このウィンドウが表示されていない場合には、View→Debugにて表示できます。

Step実行は、左から

Step Into	C言語レベルでのStep実行です。関数コールがあった場合は、その関数の中に入ります。
Step Over	C言語レベルでのStep実行です。関数コールがあった場合は、その関数の中に入りません。関数の処理を1 Stepと考えて実行します。
Assembly Step Into	アセンブラ・レベルのStep Intoです。
Assembly Step Over	アセンブラ・レベルのStep Overです。
Step Return	関数のreturnまで実行します。

になります。

ブレークポイントは、図 15のように、ソース・コード・エディタの左のスペースの該当行をダブル・クリックする事で、追加、削除ができます。

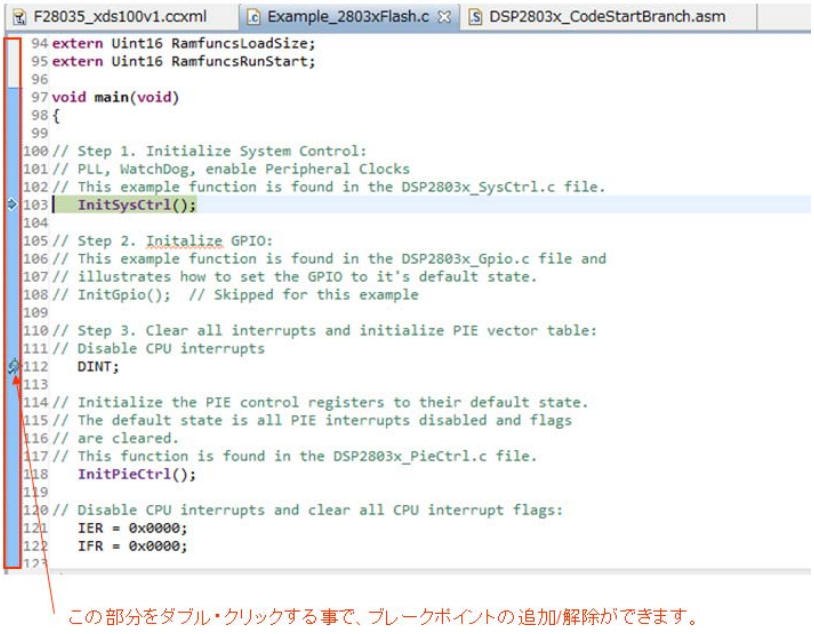


図 15:ブレーク・ポイント

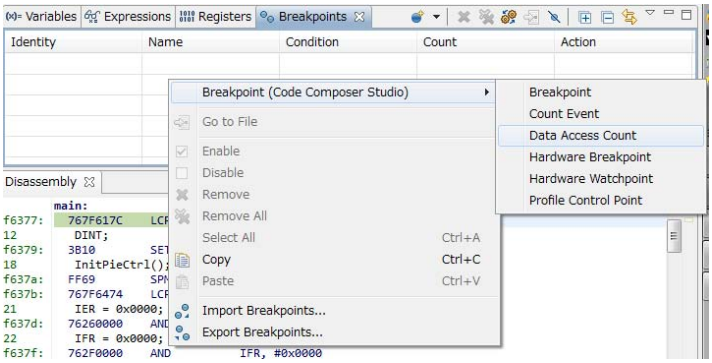
View→Breakpointsにてどこにブレークポイントを設定したか等のブレークポイントの管理ウィンドウが立ち上がりますので、用途に応じてご利用下さい。

コラム:ブレーク・ポイント

ブレーク・ポイントには、ハードウェア・ブレークとソフトウェア・ブレークの２種類があります。ハードウェア・ブレークは、デバイスの中にあるエミュレーション機能を使ってブレークを貼ります。一方、ソフトウェア・ブレークは、その対応する箇所に、ブレーク命令を埋め込みます(つまり、ユーザーからは見えませんので気づきませんが、実際には命令が置き換わっています)。このソフトウェア・ブレークは、実質的にブレークを貼る数に制限はありませんが、ハードウェア・ブレークは、デバイスのリソースを使いますので、数に制限があります。C28xの場合は、ハードウェア・ブレークは、２箇所が最大です。

RAM上のコードにブレークを貼った場合は、デフォルトでソフトウェア・ブレークが使用されます(ハードウェア・ブレークを使う事もできます)。Flash上のコードにブレークを貼った場合は、ハードウェア・ブレークが使用されます(ソフトウェア・ブレークは使用できません)。CCSはメモリマップを把握していますので、自動的に選択されます。

ハードウェア・ブレークでは、条件(例えば、ある特定アドレスに書き込みがあったときにブレーク等)ブレークなど様々な機能を使う事もできます。これらの機能は、下図のようにView→Breakpointsにて、Breakpointsウィンドウを立ち上げこのウィンドウ内にて右クリックする事で選択できます。



筆者がよく使う機能は、Data Access Countで、これは指定アドレスに書き込みがあったり、読み込みがあったりした時にブレークがかかります。よく現象はつかめないが、誰かがあるアドレスの内容を書き換えている！といった時の解析に便利です。この機能はハードウェア・ブレークのリソースを使いますので、最大2本しか使えない事にご注意下さい。

メモリの値を見たい時は、View→Memoryを選択して下さい(図 16)。メモリアドレス(グローバル変数の変数名を指定する事もできます。その場合、アドレスでの指定ですので、&変数名のようにアドレスで指定してください)、メモリ空間(Program/Data/IO(C28xはIO空間がありませんので、IOは使用しません)、希望する表示フォーマットを選択下さい。

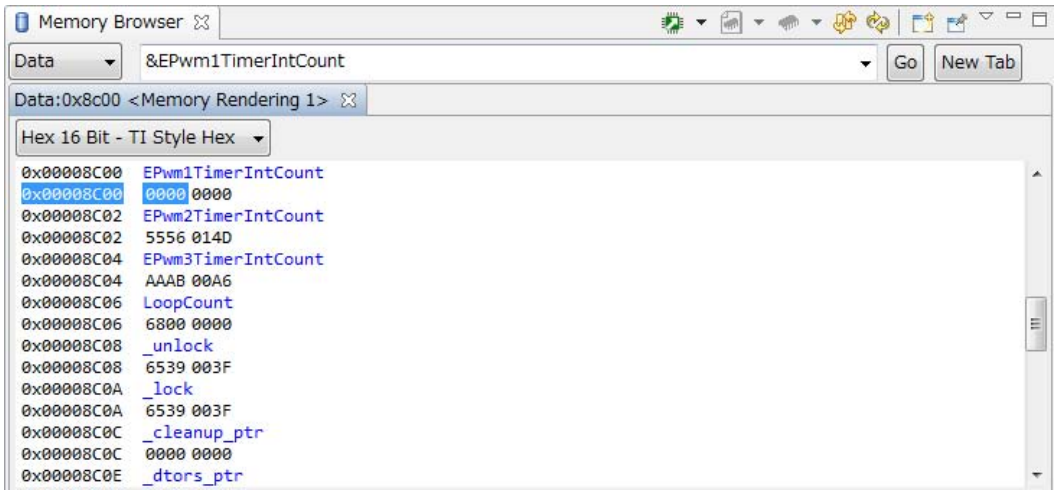


図 16:メモリ・ウィンドウ

コラム:メモリ空間について

マイコンの世界では、メモリ空間がひとつの場合がほとんどですが、TIが従来から得意としているDSPの世界では、効率よくデータ転送を行うために、Program/Data/IO(ペリフェラル)と別々のメモリ空間を持っている（つまり、それぞれのバスが分離しています）デバイスがあります。C28xは、ProgramとData（データとペリフェラル）の２つのメモリ空間が存在します。しかし、そこにマッピングされているメモリは、全く同じメモリが両方の同じアドレスにマッピングされているため、事実上、メモリ空間はひとつになります（正確にはペリフェラル・レジスタはデータ空間にのみマッピングされています）。しかし、リンク時のシンボル情報は、ProgramとData空間にわかれていますので、データを見たい時はDataを選択しないと、正しくシンボル情報を見る事ができませんので注意して下さい。C28xではIO空間はありません。

このメモリ・ウィンドウは、メモリの値をSave(ファイルに保存)/Load(ファイルからメモリにロード)/Fill(固定パターンを書き込む)する事ができます（但し、LoadとFillはRAM領域のみで、Flash領域には適用できません）。これを実行するには、

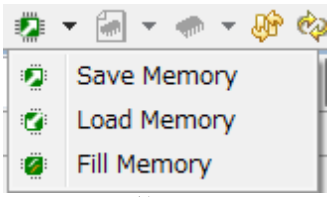


図 17:メモリ値の Save と Load

同ウィンドウの、上手の▼マーク(図 17)をクリックしてください。そうすると、Save/Load/Fillを選択する事ができます。ここで一つ注意があります。データを保存する、もしくはロードする**ディレクトリ名とファイル名には、日本語があってはなりません。日本語がある場合には、エラーも表示されずに、何も起きません**ので注意下さい。

Local/Global変数のビューワーは、デフォルトで表示されていると思いますが、表示されていない場合は、View→Variablesでローカル変数が、View→Expressionsにて、Global変数を表示できるウィンドウが立ち上がります。また、View→Registersにて、CPU/ペリフェラル・レジスタのビューワーが立ち上がります。

メモリの中身はMemory Browserウィンドウ以外にも、グラフに出力する事もできます(横軸:アドレス/縦軸:値)。どこかRAM領域にログを残しておき、それをグラフで表示するといった使い方ができます。グラフ表示はTools→Graphで数種類のグラフが選択できます。Eclipse(CCS)には様々な機能が用意されていますので、いろいろとお試し下さい。

プログラムを再度最初から実行する際には、

Run→Reset→Reset CPU

Run→Restart

を順番に実行して下さい。Main()に戻ります。

コラム：ResetとRestart

CCSにおけるResetはハードウェア・リセットとほぼ等価です（ほぼと書いたのには理由があり、一部レジスタ(例えばPLLの設定等)はリセットされないなど、多少の違いはあります）。Restartはなじみの無い機能かもしれません。CCSからコードをロードした時に、PC(プログラム・カウンタ)がある行を指しています。このロードした直後のPCの位置は、リンカの設定にて指定できます。リンカについては次の章で解説しますので、ここでは詳細を省きますが、リンカの(--entry_point, -e)オプションにてその位置を指定します。デフォルトでは、_c_int00というラベルが貼られている行が指定されます。通常Ｃ言語のコードはmainが先頭のように思われますが、mainが実行される前に、C言語コードの環境を設定する_c_int00という関数が実行されます。つまり、_c_int00が本当のCコードの先頭になります。Restartを実行すると、デバイスはリセットされませんが、PCがこの先頭に移るという事を意味します。

さて、その目的はデバッグ目的です。デバイスがリセットされてから、ユーザーのコードが実行されるまでには、ブートローダというROMに収められたTIのコードが実行され、その後、ユーザーのコードに移ります。このブートローダの使い方をきちんと理解していないと、ブートローダーからユーザーのコードに移らない可能性があります。Restartを使えば、ブートローダを通らずに、ユーザーのコードを実行できるので、非常に便利なのです。ブートローダを理解し、正しくユーザーのコードに移るように設定した後は、RestartでなくResetのみでもかまいません。ブートローダは4章で解説します。しかし、Restartだけでは、デバイスにリセットがかからないので、Resetを一度実行し、デバイスをリセットした上で、Restartを実行して、PCをユーザーコードに移していいるのです。

ＣＣＳの設定によっては、プログラムをロード、もしくはRestartした後には、_c_int00(もしくは他のエントリーポイント)ではなくmainにPCがある場合もあります(デフォルトではその設定になっていると思います)。これは、ＣＣＳがロード/Restartした後に自動的にmainまで実行するからです。Tools→Debug Options→Generic Debug Options のAuto Run Optionsの”On a program load or restart”にチェックが入っている場合は、自動的にmain(このラベルも設定変更可能です)まで実行されます。

コードを変更して、再度実行したい場合は、CCS Editボタンを押し、コード開発画面に戻り、コードを変更します。

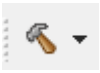


図 18:ビルド・ボタン

コードを変更し終えたら、図 18のボタンがビルドのボタンです。

ビルドが正常に終了すると、実行ファイルが更新されたので、ロードし直すかどうか聞かれますので、Yesをクリックしてください。プログラムがロードされます。もし聞かれなかったり、ここでNoをクリックした場合は、Debugボタンを押して、デバッガに画面を切り替え、Run→Load→Reload Programを選択して、ロードしてください。

コラム:CCS3.xとCCS4.x/5.xにおけるFlashへのロードの違いについて

CCS3.xでは、Load ProgramはRAMにしか行えず、C28xの内蔵Flashにプログラムを書き込む時は、Flash書き込みPlug-Inを立ち上げて、そこから書き込みを行っていました。CCS4.xでは、Load ProgramはFlash書き込みに対応しています。そのため、RAM用のコードでも、Flash用のコードでも、Load Programひとつで対応します。Flashの中身を消去するなどのFlash単体の操作をしたい場合や、設定を変更したい(例えば、Load Program時に全てのFlashを消去する設定を変更する等(時間短縮のため))は、

Tools→On-Chip Flash

を選択して下さい。以下のようなFlash操作ウィンドウが立ち上がります。ここから、Flashの消去や、Depletion Recoveryなどの操作ができます。

サイクル数の測定機能もついています。Run→Clock→Enableを選択すると、CCS画面の下の方に

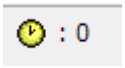


図 19:サイクル数測定

図 19のような表示がでてきます（非常にわかりにくいですが）。これがサイクル数を示します。この部分をダブルクリックすると、サイクル数表示が0にリセットされます。ここに表示されるサイクル数は、それなりに正しい値が測定される事が多いですが、あくまでも目安と考えてください。必ずしも正確ではない時があります。正確に測定したい場合は、タイマか、I/O信号を外でモニタ別の手段をご検討頂きますようお願いします。また、サイクル数を測定するためには、Step実行は使用しないで下さい。ブレーク・ポイントを貼り、RUNを実行して下さい。

一通り、デバッガの機能を試してみたら、このProjectを消去してください。CCS画面の右上のCCS Editボタンを押し、開発画面に戻り、

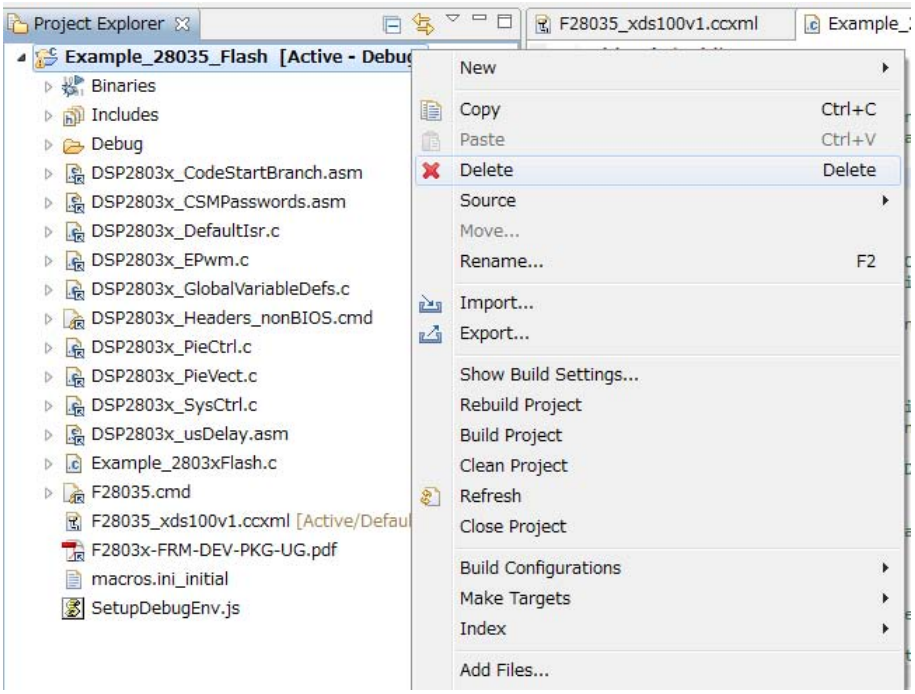


図 20:Project の Delete

図 20のように、Project Explorerウィンドウの Project名を右クリックし、Deleteを選択して下さい。図 21のように聞かれます。ここで、注意です。”Delete project contents on disk(cannot be undoe)”は絶対にチェックしないでください。**チェックした場合は、project内のファイルが、ファイル・システムから消去されます。つまり、ファイルが本当になくなります。**チェックしない場合はProjectの登録から抹消するだけで、ファイル自体はなくなりません。特に、CCS3.xユーザーは、CCS3.xと同じ感覚で操作しないように注意下さい。Eclipseのプロジェクトは、ファイル・システムそのものです。

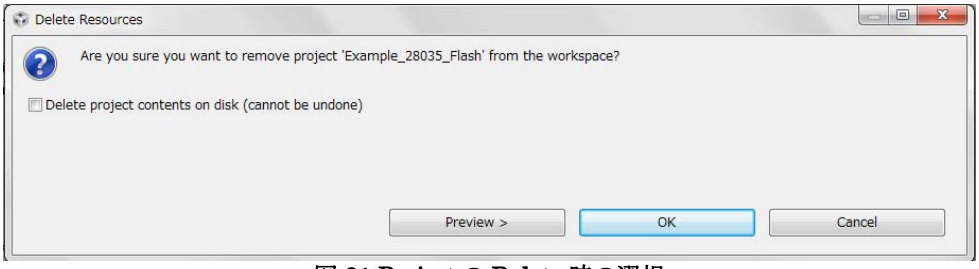


図 21:Project の Delete 時の選択

3 新規 Project の作成:GPIO、CPU タイマ、割り込み

3.1 この章の目的

前章では、あらかじめ用意されているProjectを使って、CCSの基本的な使い方を解説しました。それでは、次は新規のProjectを作成して、プログラムを書く方法を解説します。プログラムとしては、タイマ(CPUタイマ)割り込みを用いてGPIO信号をトグルするというものです。プログラムを書く前に、

1. ペリフェラル・ヘッダ・ファイル
2. コード生成の基礎
3. 割り込みの仕組み
4. GPIOの仕組み
5. CPUタイマの仕組み

を解説します。

3.2 ペリフェラル・ヘッダ・ファイル(C/C++ Header Files and Peripheral Examples)

注意：F2802x用のv200について
F2802x用のペリフェラル・ヘッダ・ファイルは、v200にてメジャー・アップデートがかかりました。 v1xxからの、大きな変更点としては、commonフォルダに、ペリフェラル・ドライバ・ライブラリが追加されました。それにともない、examplesフォルダに収められているサンプル・コードの大部分がこのペリフェラル・ドライバ・ライブラリを使ったものに変更されています。v200でも、従来(v1xx)からの記述方法もサポートされていますが、今後、F2803x/F2806xも同様の構造に変わっていくと思います。このドキュメントでは、F2802x/F2803x/F2806x共通のドキュメントのため、v1xxの記述スタイルで解説します(つまり、v200で追加されたペリフェラル・ドライバ・ライブラリは使用しません)。

新規Projectを作成する前に、ペリフェラル・ヘッダ・ファイル(正式名称は、C/C++ Hedaer Files and Peripheral Examplesといいます、以下単にヘッダ・ファイル)について解説します。一般的にMCUを設計販売するＩＣメーカーは、そのデバイス毎に、ヘッダ・ファイル等と呼ばれる、ペリフェラル・レジスタ・アクセス用のファイルを用意しています。C28xデバイスでも、シリーズ毎にヘッダ・ファイルが用意されていて、controlSUITEをインストールする事で自動的にインストールされます。インストールされる場所は、デフォルトのインストール場所であれば、

C:\ti\controlSUITE\device_support

になります。このディレクトリの下に、デバイス・シリーズ毎に個別のディレクトリがあり、その中にシリーズ毎のヘッダ・ファイルが収められています。ヘッダ・ファイルは、主に次の4つのディレクトリから構成されています(F2806x:v130/F2803x:v126/F2802x:v200の場合)。

ディレクトリ	内容
\doc	ドキュメントがありますので、必ず一読下さい。 F2802xのv200の場合は、ペリフェラル・ドライバ・ライブラリのドキュメントが収めれていますが、ヘッダ・ファイル自体の解説は大幅に少なくなりました。従来記述をする場合は、v1xxに収められているドキュメントを一読した方がよいかもしません。
\F2806x_headers \DSP2803x_headers \F2802x_headers	ペリフェラル・アクセスを行うための、ペリフェラル・ヘッダ・ファイル本体です。ペリフェラル・レジスタ名、ビット・フィールド名、レジスタのアドレスが定義されたファイル群です。アセンブラでしか表現できない命令(割り込み許可など)のマクロ定義もあります。
\F2806x_examples	ヘッダ・ファイルを使った、各ペリフェラル用のサンプル・コードです。各ペリフェラル設定

\DSP2803x_examples_ccsv4 \F2802x_examples	の参考になりますので、是非参考にしてください。また、このサンプル・コードをベースとして、修正追加することで、コードを作っていく事もできます。 F2802xは、v200から、このフォルダの中身が大幅に変更になりました。大部分はv200からサポートされているペリフェラル・ドライバ・ライブラリを使ったサンプル・コードになります。そのため、F2803x/F2806xのコード記述スタイルと大幅に異なっています。従来の記述スタイル(つまり、F2803x/F2806xの現状のスタイル)のサンプル・コードを見たい場合は、v1xxのフォルダを参照下さい。おそらく、F2803x/F2806xもv2xxにアップデートされた際は、同様にドライバ・ライブラリを使ったものになると思います。 F2803xのフォルダ名にccsv4と書いてありますように、基本的にはCCSv4用に作られたプロジェクトです。CCSv5.xは、CCSv4.x用のプロジェクトを使用する事ができますので、CCSv5.xで使う事は何の問題もありません。１点注意としましては、CCSv5.xで一度でもimportしたCCSv4.x用のプロジェクトは、CCSv4.xでは使えなくなります。何等かの理由により、CCSv4でも使う可能性がある場合は、オリジナルのコピーを取っておいて下さい。
\ F2806x_examples_cla \DSP2803x_examples_cla	CLA用のサンプル・コードです。F2802xはCLA搭載デバイスがありませんので、このフォルダはありません。
\F2806x_common \DSP2803x_common \F2802x_common	PLLやFlashの初期化など、よく用いられる便利関数が収められています。ライブラリ形式ではなく、ソースコード形式ですので、用途に応じて変更して使用下さい。F2802xは、これに加え、ペリフェラル・ドライバ・ライブラリが収められています。

F2802x/F2806は、F2802x_xxxx/F2806x_xxxxといった名称になっている一方、F2803xはDSP2803x_xxxxとなっています。これは、C28xシリーズ MCUが従来はDSPと呼ばれていた名残です。おそらく、将来のアップデートにて、F2803x用もF2803x_xxxxに変更されると思います。この違いは、さほど深い意味はありませんので、気にしないでください。

このC28x用のペリフェラル・ヘッダ・ファイルの使用方法は、一般的によく用いられる手法の一つですが、簡単に解説しておきます。ペリフェラル・レジスタには、それぞれアドレスが割り当てられています。仮にヘッダ・ファイルが提供されていなければ、特定アドレスにアクセスするためには、ポインタを使う必要があります。通常は#defineを使って、各レジスタのアドレスをポインタ定義して使います。また、レジスタは1～数ビット毎のビット・フィールドに分割され、それぞれに機能が割り振られていますので、ビット・フィールド単位でのアクセスも必要です。ビット・フィールド単位でアクセスするためには、ORやANDなどの論理演算子が用いられます。例えば、0x7100番地にあるADCCTL1のTEMPCONVビット(bit 0)に1を書き込みたい場合は、

```
#define ADCCTL1_Regis   *(unsigned int *)(0x7100)
ADCCTL1_Regis |= 0x0001;
```

といった記述をするのが一般的です。しかし、これには以下の２つの問題があります。

1. **膨大なレジスタのアドレスをユーザーが定義するのは手間である。**
2. **ビット・フィールドのアクセスがわかりにくい**

ヘッダ・ファイルを使うと、

```
AdcRegs.ADCCTL1.bit.TEMPCONV = 1;
```

と非常に簡潔にかつ、読みやすい記述ができます。ヘッダ・ファイルを使った記述には、以下の規則があります。

1	レジスタのビット・フィールドにアクセスする場合	ペリフェラル名.レジスタ名.bit.ビット・フィールド名
2	16bit/32bitレジスタに16bit/32bitアクセスする場合	ペリフェラル名.レジスタ名.all
3	ビットフィールドの無い16bit/32bitレジスタにアクセスする場合(タイマカウンタ等)	ペリフェラル名.レジスタ名
4	2つの16bitフィールドで構成される32bitレジスタの、上位もしくは下位16bitで分割されているビット・フィールド(16bit単位)	ペリフェラル名.レジスタ名.half.ビットフィールド名

	にアクセスするとき	

ほとんどのレジスタは1～3になり、ePWMのCMPAなど一部レジスタが4になります。以下に1～4の使用例を記します。

1	AdcRegs.ADCCTL1.bit.TEMPCONV = 1;	ADCのADCCTL1レジスタのTEMPCONVビット・フィールドに１を書き込む
2	AdcRegs.ADCCTL1.all = 0x1234;	ADCのADCCTL1レジスタに0x1234を書き込む
3	EPwm1Regs.TBCTR = 0x2000;	EPWM1のTBCTRレジスタに0x2000を書き込む。このレジスタはビットフィールドに分割されていないので、.bitや.allは必要ありません。
4	EPwm1Regs.CMPA.half.CMPA=0x1234;	EPWM1の32bit CMPAレジスタの上位16bitのCMPAフィールドに0x1234を書き込む。

ペリフェラル名は、\docに収められているドキュメントに、どのペリフェラルがどのような記述になっているかの対応表が掲載されています。

AdcRegs、EPwm1Regs、GpioDataRegs

のように、大文字ではじまるペリフェラル名Regsというの記述になっていますのでイメージしやすいかと思います。レジスタ名及びビット・フィールド名は、およそ、リファレンス・ガイドに記載されているとおりです。CCSのエディタには、補完機能(Content Assist機能:変数名等、途中まで入力してCTRL+スペースキーにより、補完候補が表示されます。struct変数の場合は、"."を入力すると、メンバ候補が表示されます。CCSv4ではこの機能を使うと、あまりに遅すぎて作業が進まないため、わざわざこの機能を無効に設定するという事がありましたが、CCSv5では大幅に改善され、高速に補完ができますので、是非利用下さい。デフォルトで有効になっています)がありますので、それと組み合わせると、レジスタ名やビット名などを、正確に覚えていなくても選択肢が出てきますので、非常に便利です。

3.3 コード生成の基礎とリンカ・コマンド・ファイル

この節は、今、熟読して、その内容を覚えておく必要は必ずしもありません。これは、今回作成するProjectでは、あらかじめ用意されているリンカ・コマンド・ファイルをそのまま使うためです。しかしながら、リンクの仕組みを理解しておく事は、様々な場面で役に立ちますし、将来、このリンカ・コマンド・ファイルを変更したくなる事もあるかと思います。また、このドキュメントでも後程、リンカ・コマンド・ファイルを修正する場面がありますので、その時は、この節を読み直して頂ければと思います。特にC28x MCUは、もともとDSPアーキテクチャのため、ハーバード・バス・アーキテクチャ(プログラムとデータのバスが分離されているアーキテクチャです)を採用しています。このため、同じサイクルで、複数のバスがメモリのアクセスを行います。メモリは複数のブロックに分かれているため、違うメモリ・ブロックに同時アクセスする事はできますが、同じメモリ・ブロックに複数のバスがアクセスした場合、waitがかかる事になります。これを避けるために、リンカ・コマンド・ファイルを駆使する事があります（プログラムとデータは、できるだけ別のメモリ・ブロックに配置します）。その時に、役に立てばと思いますので、ざっと目を通して頂ければ幸いです。

コード生成は、様々なステージを通りますが、慣れていないユーザーにとってなかなか理解しにくいのがリンクであり、さらに理解しにくいのが、リンカ・コマンド・ファイルと呼ばれるファイルです。最近は世の中がGUI中心になっていくなか、このファイルは、未だにテキスト・ベースなのが、なかなか理解しにくい要因なのかもしれません。TIでも何度か、このリンカ・コマンド・ファイルをGUIツールで設定できるように試みましたが、どうも良いツールができあがりません。結局、このリンカ・コマンド・ファイルは、テキスト・ベースが一番使いやすいと、筆者は個人的には思います。現在は、GUIツールはありません。

このリンカ・コマンド・ファイルとは、プログラムやデータ等をC28xの実デバイス上にて、何をどこに(どのメモリ・アドレスに)配置するかを指定するファイルです。WindowsにてC言語でプログラムを開発した場合、Cコンパイルをかけるだけ（実際にはいろいろなステージを経由しますがユーザーからは意識されません。）で、実行ファイルができあがります。しかし、C28xのような組み込み用プログラムを開発する場合は、実際に、どのメモリ・アドレスにプログラムやデータを配置するといった情報が必要になります。Windows等のOSが組み込まれている場合、メモリ配置はOSが自動的に管理してくれますが、OSが組み込まれていない場合は、メモリ配置は、ユーザーが指定する必要があります。

それでは、そのリンカ・コマンド・ファイルの説明をする前に、C28xのCプログラムがどのような過程を経て実行ファイルになるかを見てみましょう。

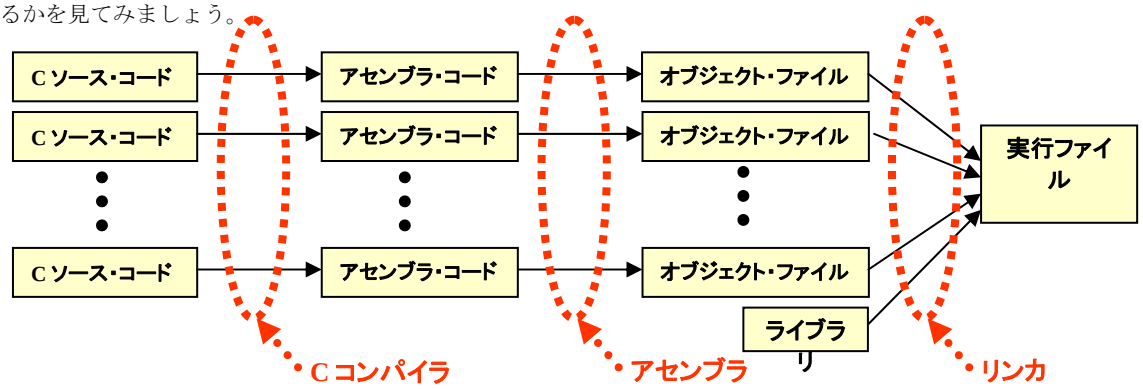


図 22:コード生成の流れ

まず、Cのソース・コード(.cファイル)がCコンパイラによって、アセンブラのコード(.asmファイル)に変換されます。次にこのアセンブラ・コードがアセンブラによって、オブジェクト・ファイル(.objファイル)に変換されます。このオブジェクト・ファイルは、ほぼC28x CPUが理解できる言語になっていますが、まだ、メモリ配置(例えば、プログラムをどのメモリ・アドレスに配置するか等)が決定されていません。このメモリ配置を決定し、複数のファイルを 1 つの実行ファイルにまとめあげるのがリンカです。このリンカのステージにて、ライブラリも一緒にリンクされます。このリンカのステージにおいて、メモリ配置を指定するのが、リンカ・コマンド・ファイルです。一般的なMCUは、メモリ配置も複雑ではなく、このリンク作業がユーザーからはあまり意識しなくても良いように作られるようになってきましたが、C28x MCUは、DSP技術をベースとしたプロセッサになりますので、メモリ配置がパフォーマンスに大きく影響する事があります。そのため、このリンカ・コマンド・ファイルを理解しておく必要があるのです。

それでは、メモリ配置を指定するとはいったいどのような事なのでしょうか？例えば、あるデバイスが以下の図 23のようなメモリマップを持っているとしましょう。

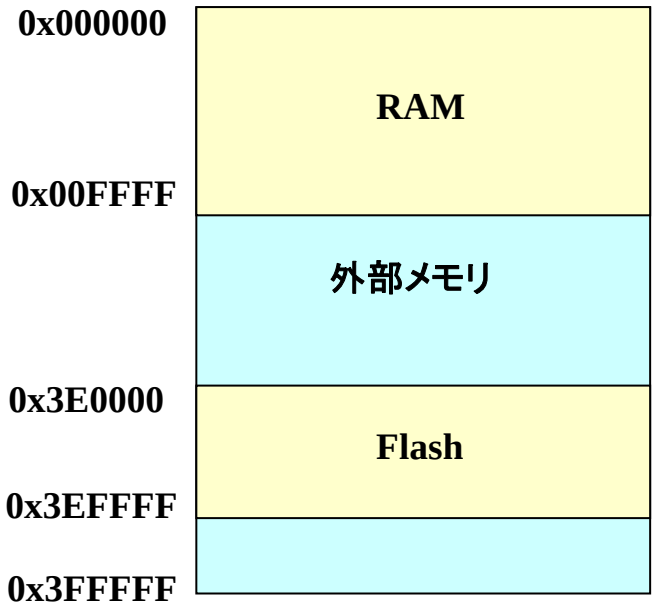


図 23:メモリ・マップ例

このようなメモリマップに対して、プログラムやデータをどこに配置するか指定するのですが、プログラムやデータ・テーブル(初期値やconstなど)はFlashに、データ関係はRAMに置く事が多いと思います。例えば、以下図 24のように配置してみましょう。

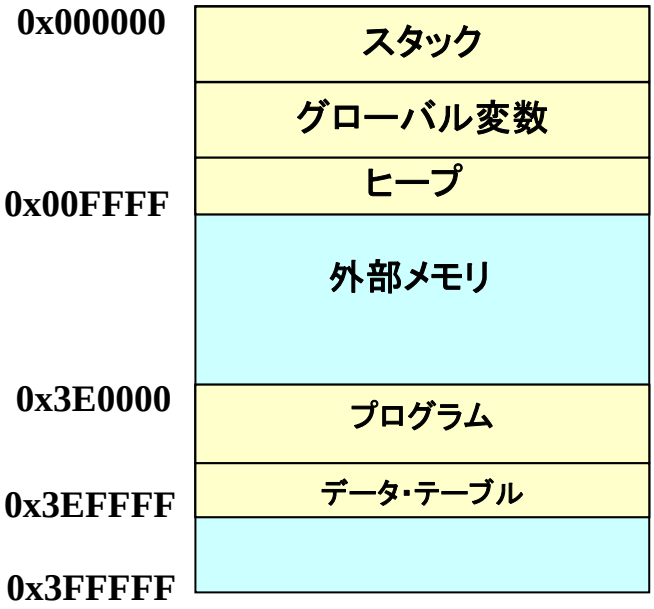


図 24:メモリのセクション配置例

このように、プログラムやデータを具体的に何番地に（もしくはどのエリアに）配置するのかを記述するのがリンカ・コマンド・ファイルです。リンカ・コマンド・ファイルは、豊富な機能がありますので、このドキュメントでは全てを紹介しきれません。より詳細を知りたい場合は、Assembly Language Tools User's Guide[SPRU513] (コード生成ツールのリビジョンにあったドキュメントを参照下さい)のLinker Description章のLinker Command Files節を参照下さい。

下のファイルがリンカ・コマンド・ファイルの一例です（このファイルはPiccolo用ではなく、前世代のF2808用ですが、考え方は全く同じです）。

```
/*=====================================================
  Linker Command File (Sample) for F2808 eZdsp
  =====================================================*/

MEMORY
{
 PAGE 0:    /* Program Memory */
   OTP:              origin = 0x3D7800,  length = 0x000400
   FLASH:            origin = 0x3E8000,  length = 0x00FF80
   ALLZERO:          origin = 0x3F7F80,  length = 0x000076
   BOOTSTART:  origin = 0x3F7FF6,  length = 0x000002
   H0SARAM:      origin = 0x3FA000,  length = 0x002000

 PAGE 1:    /* Data Memory */
   M0VECT:       origin = 0x000000,  length = 0x000040
   M0SARAM:    origin = 0x000040,  length = 0x0003C0
   M1SARAM:    origin = 0x000400,  length = 0x000400
   L0SARAM:     origin = 0x008000,  length = 0x001000
   L1SARAM:     origin = 0x009000,  length = 0x001000
}
```

```
SECTIONS
{
 .text:      load = H0SARAM, PAGE = 0
 .cinit:     load = H0SARAM, PAGE = 0
 .pinit:     load = H0SARAM, PAGE = 0
 .switch:    load = H0SARAM, PAGE = 0
 .const:     load = L1SARAM, PAGE = 1
 .econst:    load = L1SARAM, PAGE = 1
 .bss:       load = L1SARAM, PAGE = 1
 .ebss:      load = L1SARAM, PAGE = 1
 .stack:     load = M1SARAM, PAGE = 1
 .sysmem:    load = L1SARAM, PAGE = 1
 .esysmem:   load = L1SARAM, PAGE = 1
 .reset:     load = H0SARAM, PAGE = 0
}


/******************* end of file ************************/
```

リンカ・コマンドファイルは、大きく分けて２つの設定が必要になります。ひとつが、MEMORY{}です。このMEMORYは、使用するデバイスのメモリ・マップを記述します。Memory{}の中は、”PAGE 0:”から始まっています。この”PAGE 0:”や、”PAGE 1:”というのは、これから記述する領域が、データ領域かプログラム領域かを定義しています。”PAGE 0:”がプログラム領域を指し、”PAGE 1:”がデータ領域を指します。

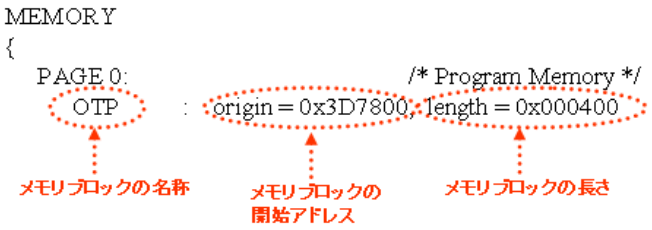


図 25:MEMORY{}内の記述内容

次のOTPは、これは、メモリ・ブロックの名称を定義します。これはユーザーが自由(禁止文字を除いて)に定義できます。次のorigin = 0x3D7800は、その名前をつけたメモリ・ブロックの開始アドレスを記述します。最後のlength = 0x000400は、そのメモリ・ブロックの長さを指定します。つまり、この行は、

“0x3D7800から始まり、0x000400の長さのOTPという名前のメモリブロック”

という記述になります。また、この行は、PAGE 0:の下にありますので、”このメモリブロックはプログラム領域です”という意味になります。リンカはこれからユーザーが使うデバイスのメモリ・マップ情報を全く知らないので、ユーザーがこのMEMORY{}に記述する事で、リンカにデバイスのメモリ・マップ情報を与えるのです。

ちなみに、リンカ・コマンド・ファイルには、記述の省略形があったり、同じ意味なのに複数の記述方法があったりします。上記の、origin = 0x3D7800は、org=0x3D7800又はo=0x3D7800と省略して記述する事ができます。また、length=0x000400もlen=0x000400又はl=0x000400と省略する事ができます。

例えば、MEMORYにて定義されたメモリ・ブロック内にて、空いている領域(後述する、SECTIONS{}にて定義されたセクションを配置しても余った領域)に、ある値を埋め込むという事もできます。その場合は、

```
OTP ： origin = 0x3D7800, length = 0x000400, fill = 0x1234
```

のように、lengthの後に、fillを追加します。この例の場合は、OTPメモリ・ブロックの余った領域は、全て0x1234を埋め込みなさいという指示になります。

さて、話を戻して、このOTPという名前ですが、この名前は、次のSECTIONS{}で使用します。それでは、次のSECTIONS{}について説明します。SECTIONS{}では、各セクションを上で定義した各メモリ・ブロックのどこに配置するかを記述します。ここで、セクションとは、プログラムやグローバル変数領域などの、まとまった領域を指します。このセクションは、ユーザーが独自に定義したセクションもありますし、コンパイラが自動的に作成するセクションもあります。Cコンパイラが自動的に作成するセクション、そして、ユーザーが定義するセクションにつきましては、次の２つのコラムを参照下さい。

このMEMORY{}にて定義する、メモリ情報は、必ずしもデータシートに記載されているメモリ・マッピングのメモリ・ブロック毎に定義しなければならないわけではありません。

例えば、M0SARAMとM1SARAMですが、
M0SARAM：0x00_0000～0x00_03FF
M1SARAM：0x00_0400～0x00_07FF
と、連続したアドレスですが、メモリ・ブロックがわかれています。この時、

M0SARAM：origin = 0x000000, length = 0x400
M1SARAM：origin = 0x000400, length = 0x400

と分けて定義してもかまいませんし、

M0M1SARAM：origin = 0x000000, length = 0x800

と一緒にしてもかまいません。ユーザーが使いやすいように定義して良いのです。

コラム:C言語が自動的に生成するセクションについて

C言語は、以下のセクションを自動的に生成します。詳細は、TMS320C28x Optimizing C/C++ Compiler User's Guide [SPRU514]のChapter 7を参照下さい。)。Page 0もしくは1は、一般的なPage配置です。状況によっては、変更する場合もあります。

セクション名	Page	初期値を持つセクション?	内容
.text	0	YES	プログラム領域
.cinitと.pinit	1	YES	global変数やstatic変数等の初期値を含む初期化テーブル領域
.const	1	YES	Small Memory Modelにてfarキーワードの無いconst定数領域。Large Memorly Modelでは、このセクションは生成されますが、確保領域0 Wordとなっていますので、実際には全く使用されません。
.econst	1	YES	Small Memory Modelにてfarキーワードがついているconst定数、もしくはLarge Memory Modelのconst定数領域
.switch	0	YES	Cプログラムのswitch文用テーブル領域
.bss	1	NO	Small Memoly Modelにてfarキーワードの無いグローバル及びstatic変数領域 Large Memorly Modelでは、このセクションは生成されますが、確保領域0 Wordとなっていますので、実際には全く使用されません。
.ebss	1	NO	Small Memory Modelにてfarキーワードが付いているグローバル及びstatic変数領域、又はLarge Memory Modelのグローバル及びstatic変数領域
.stack	1	NO	スタック領域
.sysmem	1	NO	Small Memory Modelにてfarキーワードの無いmalloc等によるダイナミック変

			数領域(ヒープ領域)。Large Memorly Modelでは、このセクションは生成されますが、確保領域0 Wordとなっていますので、実際には全く使用されません。
.esysmem	1	NO	Small Memory Modelにてfarキーワードが付いているmalloc等によるダイナミック変数領域(ヒープ領域)、又はLarge Memory Modelにてmalloc等によるダイナミック変数領域(ヒープ領域)
.reset	1	YES	_c_int00へのジャンプテーブル(実際にはこのセクションを使う事はほとんどありません)
以下のセクションは、CLA搭載のデバイスで、CLAを使う時(Cコンパイラを使う時のみ)のみ、必要になります。CLAを使用しない場合は、以下のセクションは定義する必要はありません。			
.bss_cla	1	NO	CLAを使う時のみ生成されるセクションです。CLAのグローバル変数領域です。このセクションは、必ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。
.const_cla	1	YES	CLAを使う時のみ生成されるセクションです。CLAのconst変数領域です。このセクションは、ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。
CLAscratch	1	NO	CLAのスクラッチ・パッド領域(ソフトウェア・スタック等)です。このセクションは少し特殊な記述方法が必要で、また、必ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。
Cla1Prog	0	YES	CLAのプログラム領域です。必ずCLA Program RAM領域(CLAの章にて解説します)に配置する必要があります。FlashからCLA Program RAMにコピーするが必須なため、少し特殊な記述方法が必要です。

また、C言語では、.dataセクションは使用しませんが、リンカが自動的にサイズ0にてこのセクションの割り当てを行います。サイズ０ですので、どこにマッピングされても影響ありませんが、アセンブラ・プログラムではほぼ絶対使用するセクションですので、ユーザーがきちんとセクション指定する事をお奨めいたします。

コラム：ユーザー定義のセクション

Cコンパイラが自動生成するセクションを利用するだけでは、問題があるケースがあります。例えば、あるグローバル変数(何も指定しないと.ebssセクションになります)は、DMAがアクセスできるメモリに配置しなければならないといったケースです。このような時は、ユーザーがセクションを定義する事ができます。ソース・コードのファイルにて、

```
unsigned int abc;
```

という領域があったとします。このセクションを.ebssとは別の0x9000～0x9FFF番地のどこかに配置しなければならないといったケースを考えます。この場合、#pragma DATA_SECTION(変数名, “セクション名”)という記述をします。

```
#pragma DATA_SECTION(abc, “abc_section”)
unsigned int abc;
```

と記述すると、abcという変数は、.ebssではなく、abc_sectionという名前のセクションに配置されます。後は、リンカ・コマンド・ファイルにて、abc_sectionを0x9000～0x9FFF番地のメモリに配置すれば良いのです。コードの場合は、

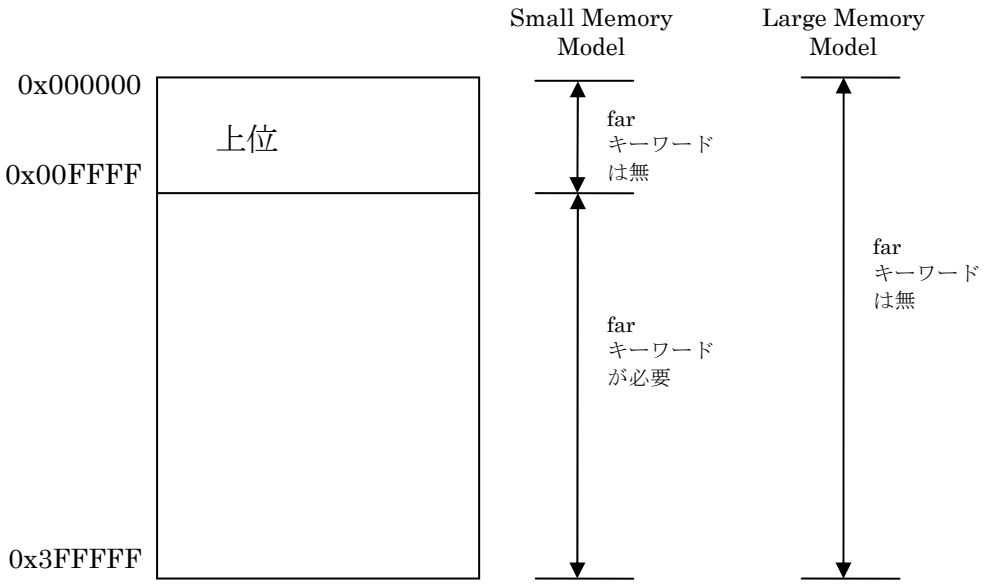
```
#pragma CODE_SECTION(関数名, “セクション名”)
```

を使うと、同様に、ユーザー定義のセクションに、その関数が配置されます。この#pragma DATA_SECTIONと#pragma CODE_SECTIONは、このドキュメントにもよく登場しますので、覚えておいてください。

コラム：メモリモデル(Large Memory ModelとSmall Memory Model)

C28xのC言語には２つのメモリ・モデルがあります。これは、メモリをどのように使用するかにより異なります(詳細は、

TMS320C28x Optimizing C/C++ Compiler User's Guide [SPRU514]のChapter 7を参照下さい。)。



C28x CPUは、メモリ中のデータにアクセスする際、0x000000～0x00FFFFまでの上位64KWordのデータ・アクセスという制限を加える事で動作が高速になる可能性があります。そのため、データを基本的には上位64KWordにしか置かないメモリモデルをSmall Memory Model、どのアドレスにもデータがある場合は、Large Memory Modelといいます。但しSmall Memory Modelの場合でも、64KWordを超えるアドレスに対するデータ・アクセスは可能で、その場合、データ宣言の時に、"far"キーワードを使用する必要があります。Small Memory Modelにおいて、farキーワードをつけたデータへのアクセスは遅くなりますので、最小限にとどめた方がよいでしょう。

[far宣言の例]

```
far unsigned int far_global = 10;   /* far宣言する事で上位64KWordを超える場所に配置されているデータで在ることをコンパイラに通知します。*/
int near_global = 1;              /*この変数は上位64KWord内に配置されているのでfarキーワードは必要ありません*/

void func(void){
  int a;

  a = far_global + near_global;

}
```

[.ebss/.econst/.esysmemと.bss/.const/.sysmemについて]

C言語が自動的に生成するセクションに、.ebss/.econst/.esysmem及び.bss/.const/.sysmemというセクションがあります。.bss/.ebssはグローバル/static変数のセクション、.const/.econstは定数、.sysmem/.esysmemはヒープ領域です。この先頭に"e"がついているのとついていないのでは、どのような違いがあるのでしょうか？

Small Memory Modelの場合

Small Memory Modelの場合は、データは上位64KWord以内のアドレスにあるものと、それ以外にあるものと２種類あります。.bss/.const/.sysmemは、前者の上位64KWord以内のアドレスにあるデータ用のセクションになります。一方.ebss/.econst/.esysmemは後者のそれ以外のデータ(つまりfar宣言したもの)用のセクションになります。

Large Memory Modelの場合

Large Memory Modelの場合は変数/定数/ヒープにアドレスによる差はありません。そのため、全てのデータは.ebss/.econst/.esysmemが使用され、.bss/.const/.sysmemは使用されません。

現在、ほとんどのユーザーにて利用されているのは、Large Memory Modelです。CCS5のデフォルトProjectでもLarge Memory Modelのオプションがチェックされていますし、Large Memory Modelしかサポートしていないライブラリ等もありますので、特に理由が無ければLarge Memory Modelを選択した方が良いでしょう。また、コンパイラのVer6以降は、ランタイムライブラリがLarge Memory Modelのものだけが用意されていますので、Small Memory Modelを使うユーザーはおそらく皆無だと思います。

図 26:SECTIONS{}内の記述説明

さて、SECITIONS{}の記述ですが、上の図 26のように、最初にこれから配置を決定したいセクション名を記述します。この例では、まずプログラムのセクション".text"の配置を決定します。次の"load = H0SARAM"というのは、この".text"というセクションをどのメモリ・ブロックに配置するかを定義しています。ここで、"H0SARAM"というのは、MEMORY{}内にて定義されたメモリ・ブロックの名称です。また、次の"PAGE=0"も、MEMORY{}内にて定義されたそのメモリ・ブロックのPAGE番号を記述します。つまり、".text"というセクションをPGAE 0のH0SARAM(0x3FA000～0x3F8FFF)内に配置するようにリンカに指示している事になります。

.text： load = H0SARAM, PAGE = 0
の変わりに
.text：> H0SARAM, PAGE = 0
という記述方法もあります。意味は同じになります。.

セクションで、全く使わないセクション(例えばCコンパイラが自動生成する.resetセクション)は、そのままリンクしてもかまいませんが、メモリ容量がもったいないですので、

.reset ：> RESET, PAGE = 0, TYPE = DSECT

のように、最後にTYPE=DSECTを追加します。DSECTの本当の意味は少し複雑ですが、単にこのセクションを無視と考えてもよいでしょう。また、ROM領域内の関数やテーブルをコードが使用している時(例えば、後程出てくるIQmathライブラリは、ROM内のデータ・テーブルを使用します)は、ROM内のデータや関数を呼び出すためにシンボル情報(つまりアドレス情報)が必要になります。そんな時は、
IQmathTables3 ：> IQTABLES3, PAGE = 0, TYPE = NOLOAD
このように、TYPE=NOLOADを最後につけます。この場合、このセクションのシンボルはリンクされますが、コードをロードする時には、このセクションはロードされません。

これら、DSECTとNOLOADは、ヘッダ・ファイルに収録されているリンカ・コマンド・ファイルにて使われていますので(ユーザーが意識する事はあまりないかもしれません)、ここで紹介しておきました。

領域が大きいセクション(例えば、.textはそうなる事がよくあります)を、配置する時に、１つのメモリ・ブロック内に入らない時があります。そんな時に便利なのが、以下のような記述です。

.text ：>> FLASHA | FLASHB

このような記述をすると、.textは、FLAHAとFLASHBのどれかに、配置しなさい、もし１つのメモリ・ブロックに入りきらない場合は、.textセクションを分割(objファイル毎の分割です)して、どちらかに配置しなさいという指示になります。FLASHA

とFLASHBが連続したメモリであれば、MEMORY{}の定義を変更すればすむ事ですが、FLASHAとFLASHBのアドレスが離れている時等に便利です。ここで、>>であることに注意して下さい。>>ではなく>を使った場合、例えば、

```
.text ：> FLASHA | FLASHB
```

と記述した場合は、.textは分割されず、FLASHAかFLASHBかどちらかに、まとめて配置しなさい、という指示になります。この２つとも、ヘッダ・ファイル内のF28027のリンカ・コマンド・ファイルで利用されていますので、記載しました。

コラム：ランタイム・ライブラリ

Ｃ言語コードをビルドするときは、ランタイム・ライブラリという、ライブラリを必ず1つリンクします。今回のこのドキュメントで使用するProjectに関しては、自動でリンクするランタイム・ライブラリを選択するように設定しますので、あまりこのランタイム・ライブラリを意識することはないと思いますが、簡単に解説しておきます。
ランタイムライブラリは、コード生成ツールがインストールされている下記のようなディレクトリ（バージョンにより異なります。）、

C:\TI\ccsv5\tools\compiler\c2000_x.x.x
の下の
\lib
にあり、以下のようなライブラリが用意されています。必要なのはどれかひとつです。

rts2700.lib, rts2700_xx.lib	C27x用(汎用品ではありません)のライブラリですので、C28xでは関係ありません。
rts2800.lib	Small Memory Model, FPU無し(ver6から未サポート)
rts2800_eh.lib	Small Memory Model, FPU無し, C++例外サポート (ver6から未サポート)
rts2800_fpu32.lib	Large Memory Model, FPUサポート
rts2800_fpu32_eh.lib	Large Memory Model, FPUサポート, C++例外サポート(ver6から未サポート)
rts2800_ml.lib	Large Memory Model, FPU無し
rts2800_ml_eh.lb	Large Memory Model, FPU無し, C++例外サポート(ver6から未サポート)

※上記のver6はCCSのバージョンではなく、コード生成ツールのバージョンです。

Smallメモリモデル、及びC++例外はほとんど利用されていませんので、ほとんどのケースで、rts2800_ml.libかrts2800_fpu32.libのどちらかではないかと思います。

ランタイム・ライブラリには大きく分けて、次の3つの事をしています。
１．標準関数の提供
２．浮動小数点乗算などの演算関数の提供
３．初期化
１の標準関数とはお馴染みのmemcpy()やmalloc()等のANSI C準拠の標準関数です。memcpy()等の標準関数がコールされた場合、ランタイム・ライブラリ内の同関数がコールされます。
２については、浮動小数点の乗算やlongの割り算等は、ユーザーが明示的に関数コールをしない場合でも、関数コールされる場合があります。その場合は、ランタイム・ライブラリ中のその機能を実行する関数がコールされます。
最後に３の初期化ですが、C言語は必ずmain()から始まります。しかし、このmain()が実行される前には、実はいろいろな処理が実行されています。例えば、スタックの初期化、グローバル変数やstatic変数の初期化、CPUレジスタの初期化等です。これらの機能を受け持つのが、_c_int00という名前の関数です。C言語のコードではmain()の前に、この_c_int00という関数が実行され、その後にmain()にブランチするようなコードが生成されます。この_c_int00も、ランタイム・ライブラリに含まれています。もし、興味があれば、このランタイム・ライブラリのソース・コードを読んでみてください。ソースコードは、ランタイム・ライブラリがあるディレクトリ内のrtssrc.zipを解凍して下さい。

3.4 割り込みの仕組み : PIE(Peripheral Interrupt Expansion)

次にC28xの割り込みの仕組みについて解説します。今回作成するプログラムはタイマ割り込みを使いますので、割り込みの仕組みを理解しておく必要があります。C28x MCUの割り込みは少しユニークな構造をとっており、一見非常に複雑に見えます。その詳細は、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章
F2803x用	*TMS320F2803x Piccolo System Control and Interrupts Reference Guide[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts Refernce Guide[SPRUFN3]*

と、*TMS320C28x CPU and Instruction Set Reference Guide[SPRU430]*を参照下さい。ここでは全体像と最低限知っている必要がある事のみを解説します。

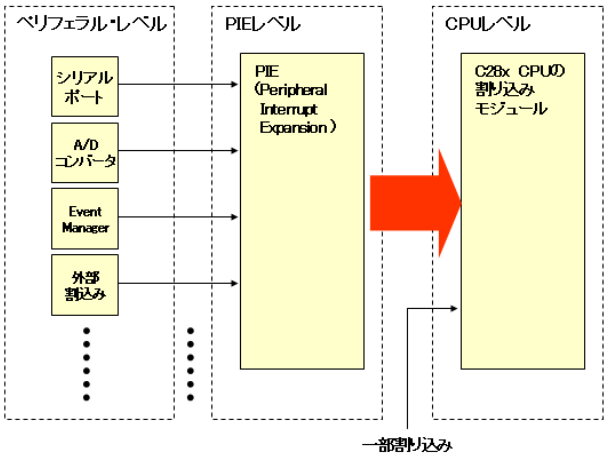


図 27:割り込みの全体像

図 27にC28x MCUの割り込み全体像を示します。基本的な仕組みは、全てのC28x MCUで共通です。C28x CPUモジュールは汎用的に作られているため、それ程多くの割り込みを扱えるようには設計されていません。しかしながら、C28x MCUの代表的なアプリケーションである制御アプリケーションにおいては、非常に多くの割り込みを扱う事となります。そのため、C28x MCUでは、CPU割り込みの足りない分を補うために、PIE(Peripheral Interrupt Expansion:パイと発音するようです)という割り込みを拡張するペリフェラルを搭載しています。また、シリアルポート、ePWM等の各ペリフェラルはそのペリフェラル毎にさらに割り込みを管理する機能を備えています（備えていないペリフェラルもあります）。つまり、C28x MCUの割り込みは以下の３つの階層構造を持っています。

1. ペリフェラル・レベル（最下位層）
2. PIEレベル(中間層)
3. CPUレベル(最上位層)

1のペリフェラル・レベルはペリフェラルによってその仕組みが異なりますので、ここでは解説せず、2と3のみ解説します。

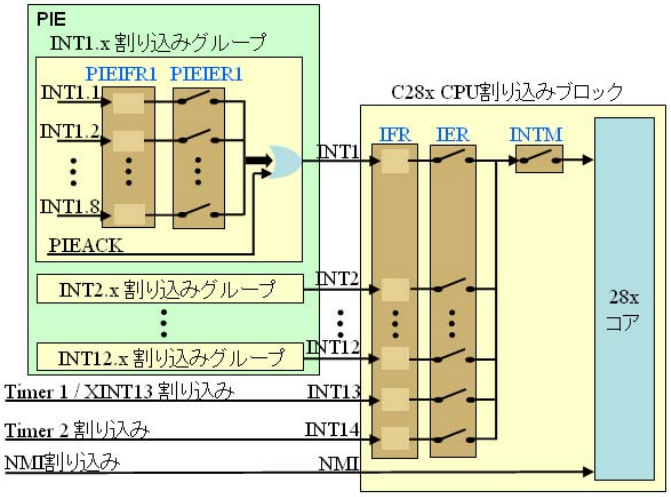


図 28:CPU レベルと PIE レベルの割り込み

図 28にPIEレベルの割り込みとCPUレベルの割り込みの全体像を示します。CPUレベルでは、INT1～14とNMI及び、図には書いてありませんがResetの計16本の割り込みを扱う事ができます。INT1～14に対しては、CPUレベルには割り込み要求が来たことを示すフラグ・レジスタIFRがあります。IFRレジスタは、図 29のように

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

図 29:IFR レジスタ(R:リード可能, W:ライト可能, －後の 0 又は 1 はリセット直後の値)

各ビットに対し割り込み番号が割り振られています(RTOSINTとDLOGINTは特殊な割り込みで、今回はこれらには触れません)。例えば、INT1割り込みリクエストが来た場合、0bit目に”1”が立ちます。そしてその割り込みリクエストが受理された場合、割り込み処理(ISR:Interrupt Service Routine)が始まる前に自動的にクリアされます。つまりこのbitが”1”の場合は割り込みリクエストが入り、まだ処理されていない事(保留中)を意味します。bitが”0”の場合は逆に割り込みリクエストが入っていないか、処理中である事を示します。

同じくINT1～14に対してはIERというその割り込みをユーザーが許可するか不許可にするかを設定するレジスタがあります。IERレジスタは、図 30のように、

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

図 30:IER レジスタ(R:リード可能, W:ライト可能, －後の 0 又は 1 はリセット直後の値)

IFRと同様で、各ビットに対し割り込み番号が割り振られています。番号の割り振り方はIFRと同じです。この各ビットを”1”に設定すると、その番号の割り込みを許可する事を意味し、”0”に設定すると、その番号の割り込みを不許可にする事を意味します。このビットにより許可されていない割り込みリクエストが来た場合、IFRのビットは立ちますがCPUに認識される事がなく、待ち状態になります。

最後にINTMというビットがあります。このビットはCPUのST1レジスタ内にあるビットでNMI/Resetを除く全ての割り込みを許可するか許可しないかを設定する、グローバル割り込みスイッチになります。このINTMにて許可にしない限り、NMI/Resetを除く全ての割り込みは許可になりません。このビットが”0”の場合は許可、”1”の場合は不許可になります。IERと

は"1"と"0"の意味が反対ですので注意して下さい。通常、このINTMビットを操作するためにはアセンブラ命令を実行する必要がありますが、ヘッダ・ファイルにはこのINTMを設定するマクロ関数(表 2)が用意されています。

ヘッダ・ファイルのマクロ	意味
EINT	INTM = 0にする(グローバル割り込み許可)
DINT	INTM = 1にする(グローバル割り込み不許可)

表 2:INTM に関するヘッダ・ファイルのマクロ関数

さて、INT1～12の割り込みはそのままペリフェラルに接続されずPIEに接続されています。既に書きましたが、CPUでは割り込みの本数が足りないため、PIEを使う事で扱える割り込みの本数を増やしています。PIEは、割り込みをPIEグループ1～12までの12のグループに分けて管理しています。この1つのグループにはINTx.1～INTx.8までの8本の割り込みが存在します。そして、これらひとつのグループにまとめられた8本の割り込みがCPUのINTxに接続されています。例えばPIEグループ1のINT1.1～INT1.8はCPUのINT1に接続されています。同様にPIEグループ3のINT3.1～INT3.8はCPUのINT3に接続されています。このINT1.1やINT3.1のような各割り込み番号に、それぞれペリフェラルの割り込みが割り当てられています。例として図31にF28035 MCUのPIE割り当て表を示します。F28027やF28069はデータシートを参照して下さい。(設計の際は必ず最新のデータシートを参照して下さい)

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1.y	WAKEINT (LPM/WD) 0xD4E	TINT0 (TIMER 0) 0xD4C	ADCINT9 (ADC) 0xD4A	XINT2 Ext. int. 2 0xD48	XINT1 Ext. int. 1 0xD46	Reserved – 0xD44	ADCINT2 (ADC) 0xD42	ADCINT1 (ADC) 0xD40
INT2.y	Reserved – 0xD5E	EPWM7_TZINT (ePWM7) 0xD5C	EPWM6_TZINT (ePWM6) 0xD5A	EPWM5_TZINT (ePWM5) 0xD58	EPWM4_TZINT (ePWM4) 0xD56	EPWM3_TZINT (ePWM3) 0xD54	EPWM2_TZINT (ePWM2) 0xD52	EPWM1_TZINT (ePWM1) 0xD50
INT3.y	Reserved – 0xD6E	EPWM7_INT (ePWM7) 0xD6C	EPWM6_INT (ePWM6) 0xD6A	EPWM5_INT (ePWM5) 0xD68	EPWM4_INT (ePWM4) 0xD66	EPWM3_INT (ePWM3) 0xD64	EPWM2_INT (ePWM2) 0xD62	EPWM1_INT (ePWM1) 0xD60
INT4.y	Reserved – 0xD7E	Reserved – 0xD7C	Reserved – 0xD7A	Reserved – 0xD78	Reserved – 0xD76	Reserved – 0xD74	Reserved – 0xD72	ECAP1_INT (eCAP1) 0xD70
INT5.y	Reserved – 0xD8E	Reserved – 0xD8C	Reserved – 0xD8A	Reserved – 0xD88	Reserved – 0xD86	Reserved – 0xD84	Reserved – 0xD82	EQEP1_INT (eQEP1) 0xD80
INT6.y	Reserved – 0xD9E	Reserved – 0xD9C	Reserved – 0xD9A	Reserved – 0xD98	SPITXINTB (SPI-B) 0xD96	SPIRXINTB (SPI-B) 0xD94	SPITXINTA (SPI-A) 0xD92	SPIRXINTA (SPI-A) 0xD90
INT7.y	Reserved – 0xDAE	Reserved – 0xDAC	Reserved – 0xDAA	Reserved – 0xDA8	Reserved – 0xDA6	Reserved – 0xDA4	Reserved – 0xDA2	Reserved – 0xDA0
INT8.y	Reserved – 0xDBE	Reserved – 0xDBC	Reserved – 0xDBA	Reserved – 0xDB8	Reserved – 0xDB6	Reserved – 0xDB4	I2CINT2A (I2C-A) 0xDB2	I2CINT1A (I2C-A) 0xDB0
INT9.y	Reserved – 0xDCE	Reserved – 0xDCC	ECAN1_INTA (CAN-A) 0xDCA	ECAN0_INTA (CAN-A) 0xDC8	LIN1_INTA (LIN-A) 0xDC6	LIN0_INTA (LIN-A) 0xDC4	SCITXINTA (SCI-A) 0xDC2	SCIRXINTA (SCI-A) 0xDC0
INT10.y	ADCINT8 (ADC) 0xDDE	ADCINT7 (ADC) 0xDDC	ADCINT6 (ADC) 0xDDA	ADCINT5 (ADC) 0xDD8	ADCINT4 (ADC) 0xDD6	ADCINT3 (ADC) 0xDD4	ADCINT2 (ADC) 0xDD2	ADCINT1 (ADC) 0xDD0
INT11.y	CLA1_INT8 (CLA) 0xDEE	CLA1_INT7 (CLA) 0xDEC	CLA1_INT6 (CLA) 0xDEA	CLA1_INT5 (CLA) 0xDE8	CLA1_INT4 (CLA) 0xDE6	CLA1_INT3 (CLA) 0xDE4	CLA1_INT2 (CLA) 0xDE2	CLA1_INT1 (CLA) 0xDE0
INT12.y	LUF (CLA) 0xDFE	LVF (CLA) 0xDFC	Reserved – 0xDFA	Reserved – 0xDF8	Reserved – 0xDF6	Reserved – 0xDF4	Reserved – 0xDF2	XINT3 Ext. Int. 3 0xDF0

図 31:F28035 MCU の PIE 割り当て表

この表の見方ですが、同行に書かれているのは同じグループ(例えばINT1.yの8個の欄は全てPIEグループ1の割り込みです)である事を意味します。また列はそのグループ内の何番目の割り込みかを示します。例えば、EPWM5_INTはINT3.5に割り振られており PIE グループ 3 の 5 番目の割り込みです。優先順位は、INT1.1 が一番高く、次に INT1.2、INT1.3.....INT2.1,INT2.2.....INT12.7,INT12.8という順番に低くなっていきます。

ここで一点注意して頂きたい事は、この"優先順位"という言葉です。C28x MCUはハードウェアにて多重割り込み（プリエンプション）には対応していません。そのため、例えば優先順位の低いINT9.1の処理中に順位の高いINT1.1がリクエストをしても、INT1.1はINT9.1が終わるまでは待ち状態になります。この優先順位とは、たとえば、INT9.1が処理中にINT1.1とINT1.2がリクエストをあげている時、INT9.1処理が終了した時には、INT1.2よりも順位の高いINT1.1の処理が優先され実行されるという意味になります。優先順位付の多重割り込みを実現したい場合はソフトウェアにて処理する必要があります。

さて、このPIEにもそれぞれIFR/IERに相当するレジスタがあり、それぞれPIEIFR/PIEIERというレジスタが各グループに対して用意されています。PIEIFR/PIEIERレジスタを図 32/図 33に示します。

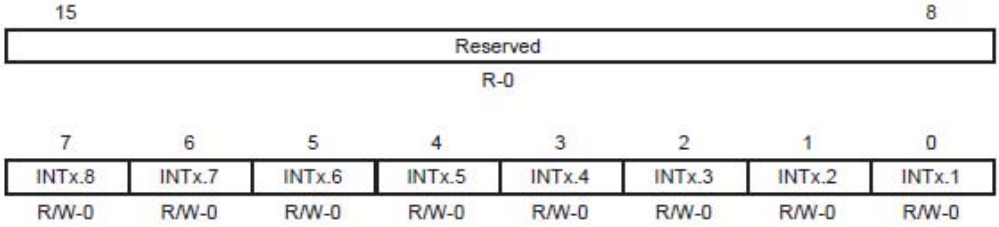


図 32: PIEIFRx レジスタ(x は 1～12, R:リード可能, W:ライト可能, －後の 0 又は 1 はリセット直後の値)

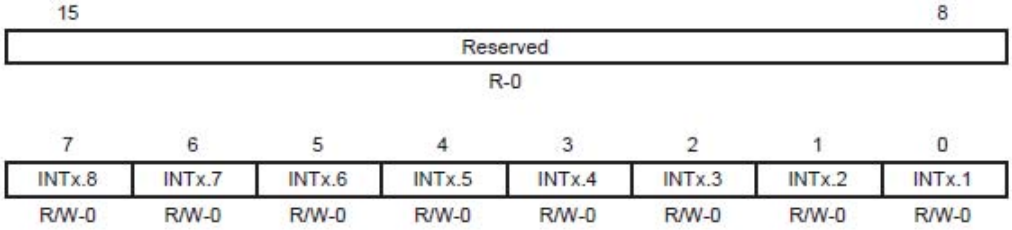


図 33:PIEIERx レジスタ(x は 1～12, R:リード可能, W:ライト可能, －後の 0 又は 1 はリセット直後の値)

役割はIER/IFRと同じで、PIEに割り振られている割り込みもPIEIERの設定によって、全て独立して許可/不許可にできます。例えば、INT1.7のTINT0というタイマ割り込みのリクエストが入った時を想定してみましょう。TINT0はINT1.7ですので、PIEIFR1レジスタの6bit目のフラグが立ちます。この時、ユーザーがPIEIER1レジスタの6bit目を許可(=”1”)に設定されていれば、PIEはCPUのINT1に対して割り込みリクエストを行います。不許可になっていればCPUへのリクエストは行われず、待ち状態になります。

次にPIEACKレジスタというレジスタについて解説します。例えばINT1.7割り込みがPIEIERによって許可されているとします。PIEIFRの対応するビットが”1”になり、PIEIERによって許可されていますので、CPUレベルに通知され、CPUレベルのIFRの対応するビットが”1”になります。この割り込みが直ぐに実行されれば、問題はおきませんが、この割り込みがIERによって許可されていなかったり、他の割り込みを実行中だったりする場合は、保留されます。この時、同じINT1グループの他の割り込みリクエストが来たとします。INT1.xの割り込みは全て、CPUレベルのINT1につながっていますので、同じグループの割り込みリクエストが複数CPUレベルに入ってしまうと、混乱してしまいます(正確には、PIEからCPUレベルへの信号パルスが１個ロストしてしまいます)。そこで、PIEでは、あるグループの割り込みがCPUレベルに通知された時に、後続の同じグループの割り込みがCPUレベルに通知されないように、CPUレベルへの通知パスを一旦遮断します。この遮断する動作を担うのがPIEACKレジスタです。PIEACKレジスタを図 34に示します。

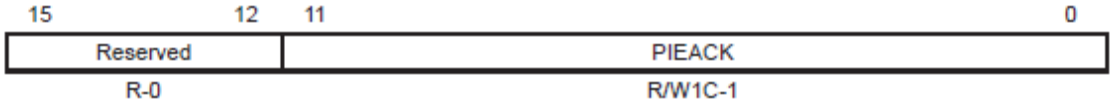


図 34: :PIEACK レジスタ(R=リード可能, W1C=1 をライトする事でクリア, －後の 0 又は 1 はリセット直後の値, bit0 が PIE グループ 1 用、bit1 がグループ 2 用…bit11 がグループ 12 用)

このPIEACKレジスタの対応ビット(0bit目がINT1.x用、1bit目がINT2.x用.....)が”1”の時はそのグループの後続の割り込みはCPUレベルに通知されません。PIEはCPUに割り込みリクエストをあげる時、自動的にこのPIEACKの対応ビットを1にします。ここで一点注意なのですが、このPIEACKレジスタは例えそのリクエストした割り込みの割り込み処理が受け付けられても、処理が終わっても、自動的にクリアされません。ユーザーが割り込み処理の中で適切(たいがいは割り込み処理終了間際でしょう)な時点でクリアする必要があります。クリアを忘れるとそのグループの割り込みが永遠に入らないことになりますので、注意して下さい。また、このビットをクリアする時は”0”をライトするのではなく、”1”をライトします。また、このビットをクリアする時のコード記述には注意が必要です。ヘッダ・ファイルに慣れますと、つい、

```
PieCtrlRegs.PIEACK.bit.ACK1 = 1;
```

と記述してしまいますが、これは明らかに間違いです。この記述はOR命令が使われます。また、このレジスタは”1”に”1”を書いてクリアですので、ORをとるという事は、”1”になっているビットには、”1”を書いてしまう事を意味します。つまり、不用意に意図していないACKビットがクリアされてしまう可能性があります。必ず、

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
```

のように、.allでアクセスする必要があります。(PIEACK_GROUP1は、ヘッダ・ファイルにて0x0001が#defineされたものです)。

さて、ある割り込みのリクエストが入り、PIE及びCPUレベルにて許可されていて、CPUが他の割り込み処理を実行していない時は、CPUは割り込み処理(ISR:Interrupt Service Routine)を実行します。このISRは各PIEのグループ毎ではなく、PIE内のそれぞれの割り込みINTy.xに対して準備、設定が行えます。各割り込みが入った時にどのISRを実行するかを記述したテーブルがPIEベクタ・テーブルです。図 35にその概念図を示します。

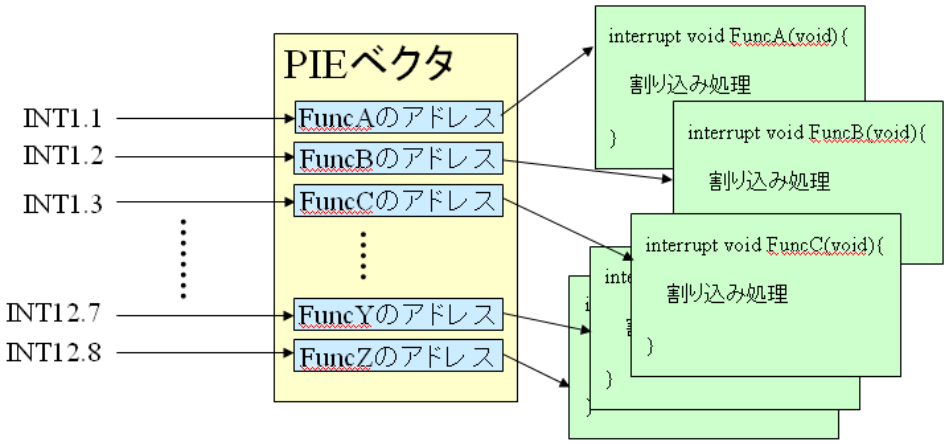


図 35:PIE ベクタ

PIEベクタには、各PIE割り込みに対するISRのアドレスをある決まった順番に書いていきます。CPUは割り込みが発生したとき、このベクタ・テーブルを見てどのISRに飛ぶべきかを判断します。このPIEベクタをおくアドレスは決まっていますので、詳細はデータシートを参照して下さい。このPIEベクタ・テーブルはヘッダ・ファイルにテンプレートが用意されていますので、ユーザーはそれを少し書き換えるだけすみますので、とても簡単です。また、ISRの記述方法ですが、3点程注意事項があります。

1. interruptキーワードを使う。

ISRの場合は、通常の関数と違ってコンテキスト・セーブ等の作業が必要です。C言語でISRを記述する場合は、Cコンパイラがこの辺は自動的にそれら処理を行うコードを生成しますので、Cコンパイラに対してこの関数はISRだという事を通知します。そのキーワードがinterruptです。例えば、通常の関数funcA()があったとしましょう

```
void funcA(void){
 …
 …
}
```

この関数をISRにしたければ、単純に

```
interrupt void funcA(void){
  …
  …
}
```

のように、関数宣言の最初にinterruptを付け加えて下さい。これだけでこの関数はISR扱いになります。

2. PIEACKレジスタをクリアする

PIEACKレジスタは自動的にクリアされませんので、必ずクリアして下さい。このACKのクリアを忘れるとそのPIEグループの割り込みが一切受け付けられなくなりますので注意して下さい。

3. ペリフェラル・レベルのフラグをクリアする

これはペリフェラルにもよりますが、各ペリフェラルも独自にIFRに相当するレジスタをもっているペリフェラルがあります。その場合そのIFRに相当するフラグ・レジスタは自動的にクリアされませんので、ユーザーがクリアする必要があります。

3.5 クロック系

次に、Piccoloのクロック系の説明をしておきます。タイマを扱いますので、クロックの設定が必要になります。図 36はクロック系の概要図です。詳細は

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章
F2803x用	*TMS320F2803x Piccolo System Control and Interrupts Reference Guide[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts Refernce Guide[SPRUFN3]*

を参照下さい。

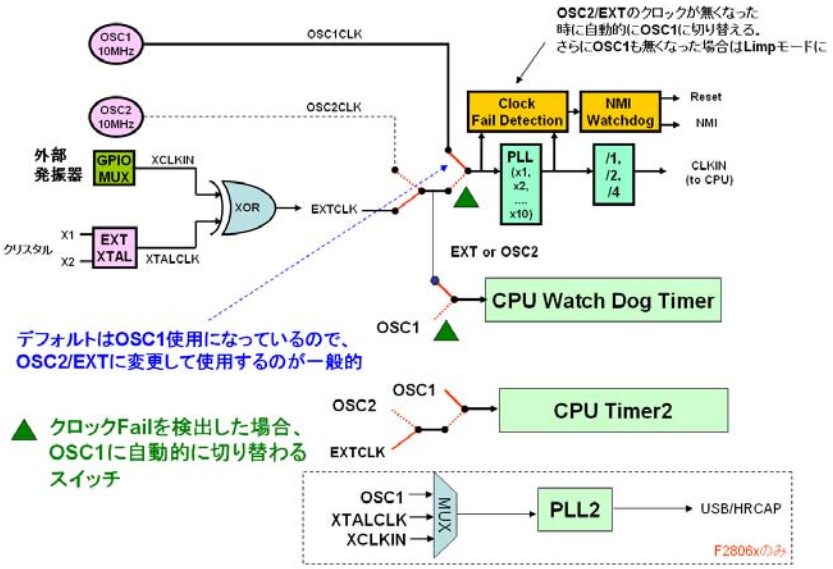


図 36:F2806x/F2803x/F2802x MCU のクロック選択

全てのF2802x/F2803x/F2806x MCUは、10MHzの発振器を２つ(OSC1とOSC2)搭載しています。２つ搭載している理由はフェール・セーフのためです。その他には、外部発振子（クリスタルやセラミック発振子等）用の発振回路も内蔵されています。また、外部発振器からのクロックを入力する事もできます。入力クロックは、常にモニタされており、もし入力クロックがフェールした場合は、自動的にOSC1に切り替わります。つまりOSC2もしくは、外部(EXTCLK)を使っている場合は、これらがフェールした時は、自動的にOSC1に切り替わります。リセット直後はOSC1が使用されますので、ユーザーは、OSC2かEXTCLKに切り替えて使うケースがほとんどでしょう。入力クロックは、PLLによりx(1～10)てい倍され、/1, /2, /4分周され、CPUにクロック供給(CLKIN)されます。F2806xでは、USBとHRCAPモジュールにクロックを供給するPLL2という別PLLがさらに搭載されています。

この図の続き(CLKINの続き)を図 37に示します。搭載されているペリフェラルが各シリーズ、デバイスによって異なるため、全てのデバイスを表す図を示すのは難しい関係で、この図はF28035の場合のみです。他のデバイスにつきましては、リファレンス・ガイドをご確認頂き、どのペリフェラルがどのクロックからクロック供給を受けているのかを確認して下さい。

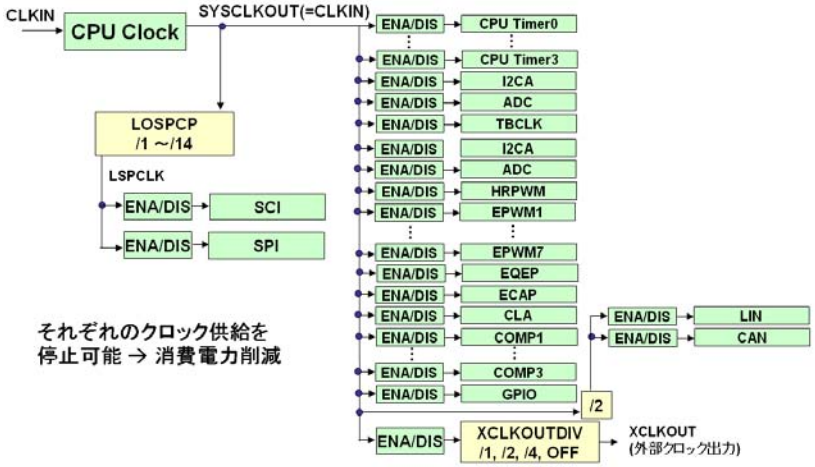


図 37:ペリフェラルのクロック系(F28035 の場合)

このCLKINと全く同じ周波数のクロックがCPUモジュールから出力されます。これがSYSCLKOUTになります。Piccoloの各リファレンス・ガイドには、このSYSCLKOUTという言葉が良く出てきます。SYSCLKOUTはつまり、CPUの動作クロックになります。ペリフェラルは大きく２系統のクロックにわかれます。ひとつは、そのままSYSCLKOUTを使うペリフェラル群。もうひとつは、SYSCLKOUTを/1～/14分周したLSPCLKを使うペリフェラル。F28035の場合は、SCIとSPIは後者で、そのほかは全て前者です。また、それぞれのペリフェラル毎にクロックを停止する事ができます。使わないクロックは、クロック供給を停止することで、消費電力を低減できます。

3.6 CPU タイマ

Piccolo MCUには何種類かのタイマがあります。ePWMに高機能タイマが搭載されていますが、このePWMタイマは単なるタイマとして使うには機能が豊富で設定がやや面倒ですので、もうひとつのタイマである単純なCPUタイマを使用しましょう。こちらは単なるタイマですので、設定が非常に楽です。CPUタイマは３本搭載されていますが、ここではCPUタイマ0を使用しましょう。CPUタイマの詳細は

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章
F2803x用	*TMS320F2803x Piccolo System Control and Interrupts Reference Guide[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts Refernce Guide[SPRUFN3]*

を参照下さい。この節ではその概要と簡単な使い方を説明します。

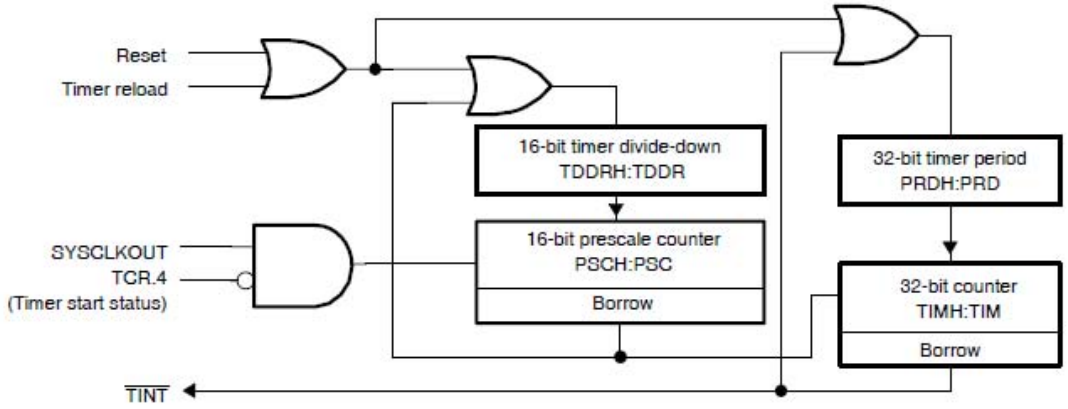


図 38:CPU タイマのブロック図

CPUタイマはSYSCLKOUT(=CPU動作クロック)をクロック源とした16bitプリスケール付32bitタイマになっています。ユーザーはTDDRH:TDDR(TDDRHが上位8bit, TDDRが下位8bit)に32bitメイン・タイマ・カウンタ,TIHM:TIM(TIMHが上位16bit, TIMが下位16bit)に入力するクロックをSYSCLKOUTの何分周（プリスケール値）にするかを設定します。このTDDRH:TDDRはいわばシャドウ・レジスタのような働きをして、実際に動作する時は、この値がPSCH:PSC(PSCHが上位8bit, PSCが下位8bit)にコピーされ、SYSCLKOUTによって、カウントダウンされます。PSCH:PSCが0になった時、

TIMH:TIMに対してカウント・ダウンするように信号を送ります。また、この時、TDDRH:TDDRの値がPSCH:PSCにコピーされます。

タイマの周期は、PRDH:PRD(PRDHが上位16bit, PRDが下位16bit)にて決定します。TIMH:TIMが0になった時、PRDH:PRDの値がTIMH:TIMにコピーされます。また、TIMH:TIMが0になった時は割り込みを発生する事ができます。ヘッダファイルには、CPUタイマを簡単に設定できる関数が用意されていますので、今回はそれを使おうと思います。そのため、それ程きちんと理解していなくても、今回は大丈夫です。

3.7 GPIO(汎用IO)

次にPiccoloのGPIO(汎用IO)について解説します。詳細は同じく

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章
F2803x用	*TMS320F2803x Piccolo System Control and Interrupts Reference Guide[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts Refernce Guide[SPRUFN3]*

を参照下さい。

さて、まず図 39見てください。一つのGPIOxピンに対して、IO入力/出力機能と３つのペリフェラル機能がMUXされています。また、Low Powerモードの解除用機能や、外部割込みピンとして設定する事もできます。また、Enable/Disableが可能な内部Pull-upも搭載しています。各GPIOピンのリセット直後のデフォルト状態は、GPIO入力モードになります。この時、内部Pull-upは、リセット直後は、GPIO0～11を除いて、Enableになっています。GPIO0～11のリセット設定にてPull-Upディセーブルになっている理由は、これらピンはPWM出力ピンとして使用できるピンになりますので、内部Pull-upがEnableになっていると、都合が悪いケースがでてしまうからです。(EPWM7A/7B/8A/8BはPull-up Enableになっている事に注意下さい)。また、今回は詳しく解説しませんが、ADC入力もIOピン(AIO)として使う事もできます。

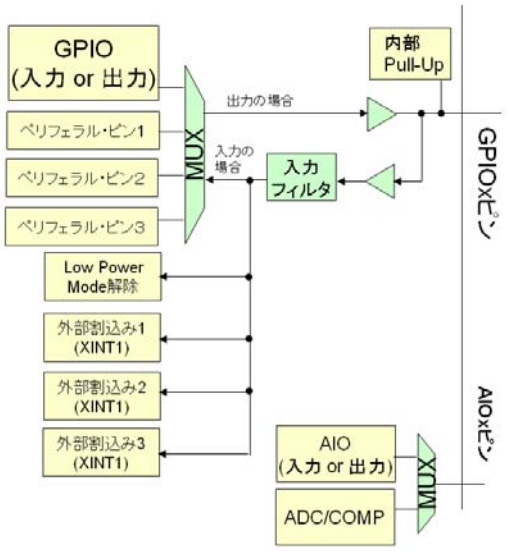


図 39:GPIO MUX

それでは、GPIOの主要なレジスタについて解説します。ピンを汎用I/Oとして使用するか、ペリフェラル機能として使用するかを決定しているのが、

GPAMUX1：GPIO0～15用

GPAMUX2：GPIO16～31用

GPBMUX1：GPIO32～47用

GPBMUX2：GPIO48～58用

の各レジスタです。図 40にF28035の場合のそのレジスタを示します。GPIO番号と機能のアサインは、シリーズにより異なりますので、F28035以外のアサインは各リファレンス・ガイド及びデータシートをご確認下さい。

	Default at Reset Primary I/O Function	Peripheral Selection	Peripheral Selection 2	Peripheral Selection 3
GPAMUX1 Register Bits	(GPAMUX1 bits = 00)	(GPAMUX1 bits = 01)	(GPAMUX1 bits = 10)	(GPAMUX1 bits = 11)
1-0	GPIO0	EPWM1A (O)	Reserved	Reserved
3-2	GPIO1	EPWM1B (O)	Reserved	COMP1OUT (O)
5-4	GPIO2	EPWM2A (O)	Reserved[1]	Reserved[1]
7-6	GPIO3	EPWM2B (O)	SPISOMIA (I/O)	COMP2OUT (O)
9-8	GPIO4	EPWM3A (O)	Reserved[1]	Reserved[1]
11-10	GPIO5	EPWM3B (O)	SPISIMOA (I/O)	ECAP1 (I/O)
13-12	GPIO6	EPWM4A (O)	EPWMSYNCI (I)	EPWMSYNCO (O)
15-14	GPIO7	EPWM4B (O)	SCIRXDA (I)	Reserved
17-16	GPIO8	EPWM5A (O)	Reserved	ADCSOCAO (O)
19-18	GPIO9	EPWM5B (O)	LINTXA (O)	Reserved
21-20	GPIO10	EPWM6A (O)	Reserved	ADCSOCBO (O)
23-22	GPIO11	EPWM6B (O)	LINRXA (I)	Reserved
25-24	GPIO12	TZ1 (I)	SCITXDA (O)	SPISIMOB (I/O)
27-26	GPIO13[2]	TZ2(I)	Reserved	SPISOMIB (I/O)
29-28	GPIO14[2]	TZ3 (I)	LINTXA (O)	SPICLKB (I/O)
31-30	GPIO15[2]	TZ1 (I)	LINRXA (I)	SPISTEB (I/O)
GPAMUX2 Register Bits	(GPAMUX2 bits = 00)	(GPAMUX2 bits = 01)	(GPAMUX2 bits = 10)	(GPAMUX2 bits = 11)
1-0	GPIO16	SPISIMOA (I/O)	Reserved	TZ2 (I)
3-2	GPIO17	SPISOMIA (I/O)	Reserved	TZ3 (I)
5-4	GPIO18	SPICLKA (I/O)	LINTXA (O)	XCLKOUT (O)
7-6	GPIO19/XCLKIN	SPISTEA (I/O)	LINRXA (I)	ECAP1 (I/O)
9-8	GPIO20	EQEP1A (I)	Reserved	COMP1OUT (O)
11-10	GPIO21	EQEP1B (I)	Reserved	COMP2OUT (O)
13-12	GPIO22	EQEP1S (I/O)	Reserved	LINTXA (O)
15-14	GPIO23	EQEP1I (I/O)	Reserved	LINRXA (I)
17-16	GPIO24	ECAP1 (I/O)	Reserved	SPISIMOB (I/O)
19-18	GPIO25[2]	Reserved	Reserved	SPISOMIB (I/O)
21-20	GPIO26[2]	Reserved	Reserved	SPICLKB (I/O)
23-22	GPIO27[2]	Reserved	Reserved	SPISTEB (I/O)
25-24	GPIO28	SCIRXDA (I)	SDAA (I/OC)	TZ2 (I)
27-26	GPIO29	SCITXDA (O)	SCLA (I/OC)	TZ3 (I)
29-28	GPIO30	CANRXA (I)	Reserved	Reserved
31-30	GPIO31	CANTXA (O)	Reserved	Reserved
GPBMUX1 Register Bits	(GPBMUX1 bits = 00)	(GPBMUX1 bits = 01)	(GPBMUX1 bits = 10)	(GPBMUX1 bits = 11)
1-0	GPIO32	SDAA (I/OC)	EPWMSYNCI (I)	ADCSOCAO (O)
3-2	GPIO33	SCLA (I/OC)	EPWMSYNCO (O)	ADCSOCBO (O)
5-4	GPIO34	COMP2OUT (O)	Reserved	COMP3OUT (O)
7-6	GPIO35 (TDI)	Reserved	Reserved	Reserved
9-8	GPIO36 (TMS)	Reserved	Reserved	Reserved
11-10	GPIO37 (TDO)	Reserved	Reserved	Reserved
13-12	GPIO38/XCLKIN (TCK)	Reserved	Reserved	Reserved
15-14	GPIO39	Reserved	Reserved	Reserved
17-16	GPIO40	EPWM7A (O)	Reserved	Reserved
19-18	GPIO41	EPWM7B (O)	Reserved	Reserved
21-20	GPIO42	Reserved	Reserved	COMP1OUT (O)
23-22	GPIO43	Reserved	Reserved	COMP2OUT (O)
25-24	GPIO44	Reserved	Reserved	Reserved
27-26	Reserved	Reserved	Reserved	Reserved
29-28	Reserved	Reserved	Reserved	Reserved
31-30	Reserved	Reserved	Reserved	Reserved

図 40:GPxMUXx レジスタ(F28035 の場合)

このF28035の場合では、例えば、GPAMUX1レジスタの10,11ビット目の各設定は、

[bit11:10] = [0:0]=0 → GPIO5

= [0:1]=1 → EPWM3B

= [1:0]=2 → SPISIMOA

= [1:1]=3 → ECAP1

のようになります。

次にGPIOとして使用する時、入力/出力のどちらに設定するかですが、これを決定するのが、GPxDIR(AはGPIO0～31用, BはGPIO32～用)レジスタです（ペリフェラルとして使用する時はこのレジスタの設定は不要です）。このGPxDIRの各ピンに対応するビットを
“0”に設定すれば、入力として(reset値です)、
“1”に設定すれば、出力として
使用できます。ビットの割り当ては、以下の図 41と図 42のとおりです。図はF28035の場合ですが、各シリーズによりGPIOの本数が違いますので、正確にはそれぞれ異なっていますが、ビットのアサイン方法は同じです。

31	30	29	28	27	26	25	24
GPIO31	GPIO30	GPIO29	GPIO28	GPIO27	GPIO26	GPIO25	GPIO24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
23	22	21	20	19	18	17	16
GPIO23	GPIO22	GPIO21	GPIO20	GPIO19	GPIO18	GPIO17	GPIO16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8
GPIO15	GPIO14	GPIO13	GPIO12	GPIO11	GPIO10	GPIO9	GPIO8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
GPIO7	GPIO6	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

図 41:GPADIR レジスタ

31	13	12	11	10	9	8
Reserved		GPIO44	GPIO43	GPIO42	GPIO41	GPIO40
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
GPIO39	GPIO38	GPIO37	GPIO36	GPIO35	GPIO34	GPIO33	GPIO32
R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

図 42:GPBDIR レジスタ(F28035 の場合)

例えば、GPIO0を出力として設定したい場合は、GPADIRのビット0を”1”に設定します。
次にGPIO出力として設定した場合、High/Lowのどちらを出力するかを設定するレジスタを紹介します。出力を制御するレジスタは、次の４つのレジスタが用意されています。

- GPxDAT　：各ピンに対応するビットを
  - ”0”に設定すればLowを出力
  - “1”に設定すればHighを出力

(GPIOを入力設定にしている場合はここを読めばHigh/Lowのどちらが入力されているかがわかります。)
注意:一般的には、GPxDATは出力制御には使いません。理由は次のページのコラムを参照下さい。このレジスタは、入力状態を読む時に使います。

- GPxSET　：各ピンに対応するビットを
  - “1”に設定すればHighを出力
  - “0”は無視

- GPxCLEAR ：各ピンに対応するビットを
  - “1”に設定すればLowを出力
  - “0”は無視

- GPxTOGGLE：各ピンに対応するビットを
  - “1”に設定すれば出力をトグル(Highの場合はLowに、Lowの場合はHighに)
  - “0”は無視

各用途に合わせてご利用下さい。下のコラムに記載しましたが、GPIO出力としては、GPxDATレジスタは一般的に使用されない事に注意下さい。

コラム:GPIO出力はどのレジスタを使う？

GPxDATレジスタを使って0を書けばLow出力に、1を書けばHigh出力にと実行するのが、一番イメージしやすい方法かと思います。しかしながら、この方法には大きな罠があります。詳細は、リファレンス・ガイドに掲載されていますが、重要な点は、GPxDATの読み込み値は、ユーザーがGPxDATに書き込んだ値を読んでいるのではなく、現在のピンの状態を示しているという点です。GPIOの状態を変更するために、GPxDATレジスタに書き込みをしても、GPxDATレジスタは、実際にIOの状態が変化するまでは反映されません。そして、GPxDATレジスタに書き込みをしてから実際にピンの状態が変わるまでには、若干のディレイがあります。つまり、次のように、連続してGPxDATを書き換える場合に問題がでます。

```
GpioDataRegs.GPADAT.bit.GPIO1 = 1;
GpioDataRegs.GPADAT.bit.GPIO2 = 1;
```

この２行は、アセンブラレベルでは、OR命令にて処理されます(32bitレジスタの特定のビットだけ1を立てるため)。一行目が実行されても、すぐにGPIO1がHighになるわけではありませんので、GPADATのGPIO1ビットは、前の状態のままになります。その時、次の命令によって、GPADATのORが実行されるわけですが、この時参照されるGPADATは古い状態のままなのです。そのため、一行目が無視されるような挙動を示してしまいます。

これを回避するために、GPxSET/CLEAR/TOGGLEレジスタは存在します。GPIOの出力の場合には、GPxDATレジスタではなく、GPxSET/CLEAR/TOGGLEレジスタを使用すべきです。

3.8 Project の新規作成と基本設定

さて、前節までで、今回のタイマ割り込みを使ってGPIOをトグルするコードを書くために必要な機能についての解説は終わりです。いよいよ、新規のProjectを作成して、コードを書いてみましょう。まず、作成するProject用の新規ディレクトリを作成しましょう。ここでは、Project1とします。この作成したディレクトリの下に、ヘッダ・ファイルのcommonとheadersディレクトリごと全てコピーして下さい。F2802xの場合は、DSP28x_Project.hとF2802x_Device.hもコピーして下さい。

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ
F2802xの場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_commonフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_headersフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\DSP28x_Project.h C:\ti\contorlSUITE\device_support\f2802x\v200\F2802x_Device.h

それでは、ＣＣＳにてProjectを作成します。CCS(CCS Edit画面の方です)にて、

File→New→CCS Project

を選択して下さい。図 43ようなウィンドウが表示されます。

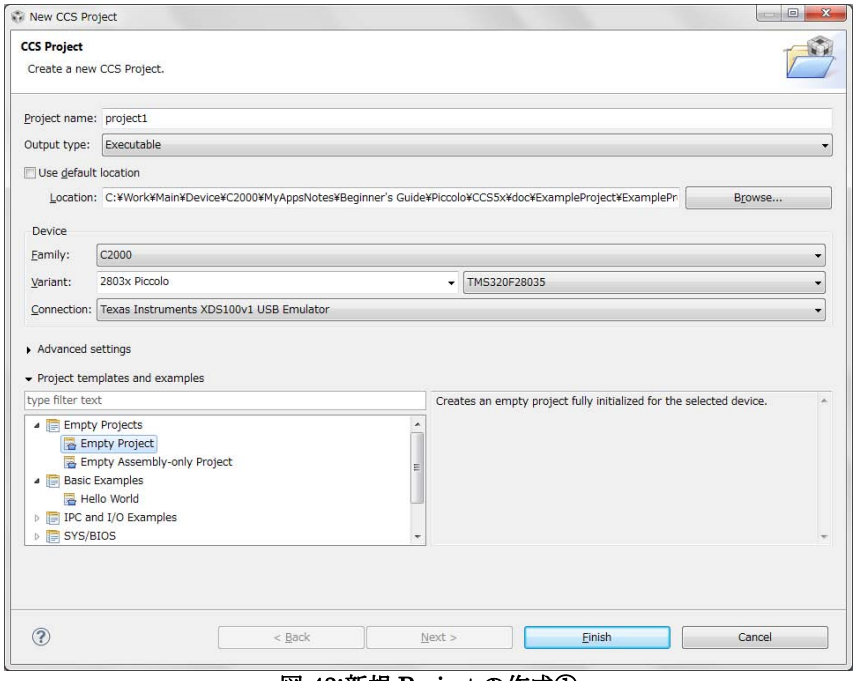


図 43:新規 Project の作成①

Use default locationをアンチェックし、Locationに先ほど作成したディレクトリを選択して下さい。また、適当な名前をProject name欄に設定してください（この例ではProject1）。Device欄に

Family	C2000
Variant	この例では2803x Piccolo, TMS320F28035 使用されているデバイスを選択下さい
Connection	この例ではTexas Instruments XDS100v1 USB Emulatorです。controlStick, Experimenter's kitを使用されている場合、

	もしくはXDS100v1エミュレータを使用されている場合は、これになります。C2000 LaunchPadを使用されている場合は、XDS100v2を選択して下さい。違うエミュレータを使用されている場合は、使用されているエミュレータ名を選択してください。
Project templates and examples	Empty projects – Empty Project

と設定したら、Nextをクリックしてください。

F2806x及びF2803xの場合は、Project Explorerウィンドウが以下の図 44 のようになっていれば(F2806xの場合は、DSP2803x_xxxxがF2806x_xxxxに、28035が28069になっているはずです)、正しく設定できています。F28027は、図 45のように、DSP28x_Project.hとF2802x_Device.hも見えているはずです。

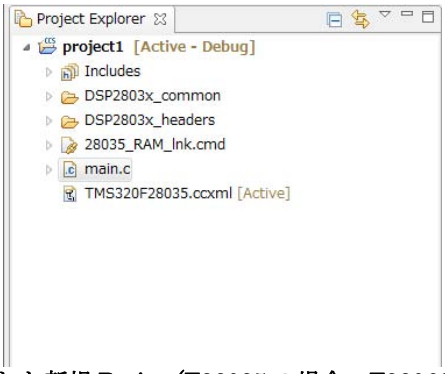


図 44:作成された新規 Project(F28035 の場合。F28069 も同様です)

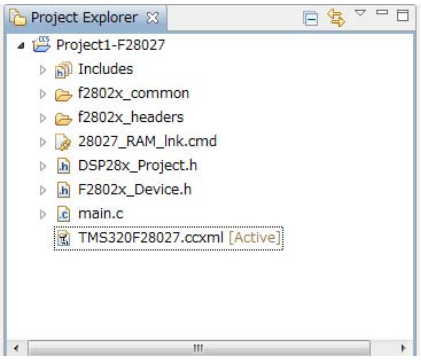


図 45：作成された新規 Project(F28027 の場合)

Projectのディレクトリに既にmain.cファイルがある場合は、その通知のメッセージがでます。main.cが無い場合は、自動的に空のmain.cを作成してくれます。

注意:このドキュメントに付属するサンプルコードでは、F28027用は、ccxmlファイルはcontrolSTICK/Experimeter's Kit用(つまり、XDS100v1)に設定されています。C2000 LaunchPadを使用されている場合は、XDS100v2になりますので、設定の変更が必要です。"1.6このドキュメントに付属されているサンプル・コード例について"章に、変更方法が記載されていますので、参照の上、設定を変更して下さい。設定を変更しないでv1のままでも、動作はしますが、せっかくv2が搭載されているにも関わらず、v1の速度になります。

コラム: CCS3.xとCCS4.x/5.xのプロジェクトの概念の違い

CCS3.xに使い慣れたユーザーであれば、CCS4.x/5.xのProjectの考え方がよく理解できないかもしれません。3.xと4.x/5.xでは、Projectの考え方が全く異なっています。CCS3.xは、空のProjectを作成して、そのProjectに希望するファイルを、"追加する"でした。そのため、ファイルは、ある意味、どのディレクトリにあってもかまわない傾向がありました。
一方、CCS4.x/5.xのProjectとは、ファイル・システムのディレクトリそのものです。つまり、Projectがあるディレクトリの下

にあるファイル/ディレクトリは、全てデフォルトでProjectに登録されているファイル/ディレクトリなのです。そのため、ディレクトリの中に、ビルドして欲しくないファイルがある場合は、わざわざ、ビルドしない設定(Exclude)をする必要があります。また、CCS3.xでは、プロジェクトのファイルをCCS上から消去しても、Projectから消去されるだけで、ファイル自体は消去されませんでした。一方、CCS4.x/5.xでは、CCSにてProjectからファイルを消去すると、本当にファイル・システムから削除されます(Linkを除く)。CCS4.x/5.xのProjectは、ファイル・システムそのものであると考えて下さい。Projectのディレクトリの下に無いファイルをProjectに追加するには、２つの方法があります。

Link Files：CCS3.xのProjectに追加に相当します。リンクが貼られるだけで、このリンクを消去しても、ファイルは消去されません。
Add Files：Projectの下にファイルをコピーします。

用途に合わせて、選択して下さい。

CCS4.x/5.xで、サードパーティ製やTI製のライブラリ、ヘッダ・ファイルを使用する時に、これらファイルをどこに置こうか悩んでしまう場合があるかもしれません。一番簡単な方法は、Projectがあるフォルダの下のディレクトリに置き、不要なファイルはExcludeする方法です。このドキュメントでも、基本的にはこの方法をとっています(一番簡単で、勘違いによる間違いが少ない(と筆者が思っている)からです。)。一方、複数の人数で開発を進めていたり、バックアップを取る事などを考えると、必ずしも共通であまり変更をしないファイルをProjectの下においておく事がよいかは、ケースバイケースかと思います。CCS4.x/5.xでは、使うファイルがProjectよりも上のディレクトリにある場合は少し悩ましくなります。Projectよりも上のディレクトリにある場合はAddではなく、Linkをする事になります。しかし、何も考えずにLinkをすると、フルパスになってしまいます。Projectの移動や共有を考えると、フルパスはあまり好ましいものではない事が多いです。そんな時に便利な機能が、Linked Resource機能です。これらの機能は、
http://processors.wiki.ti.com/index.php/Portable_Projects
に解説されていますので、ご興味がある方はご一読下さい(英語で申し訳ありません)。

次にヘッダファイル(commonとheaders)の中から今回不必要なファイルをビルドの対象からはずします。F2806x_common/DSP2803x_common/F2802x_commonを展開し、cmdを展開してください。

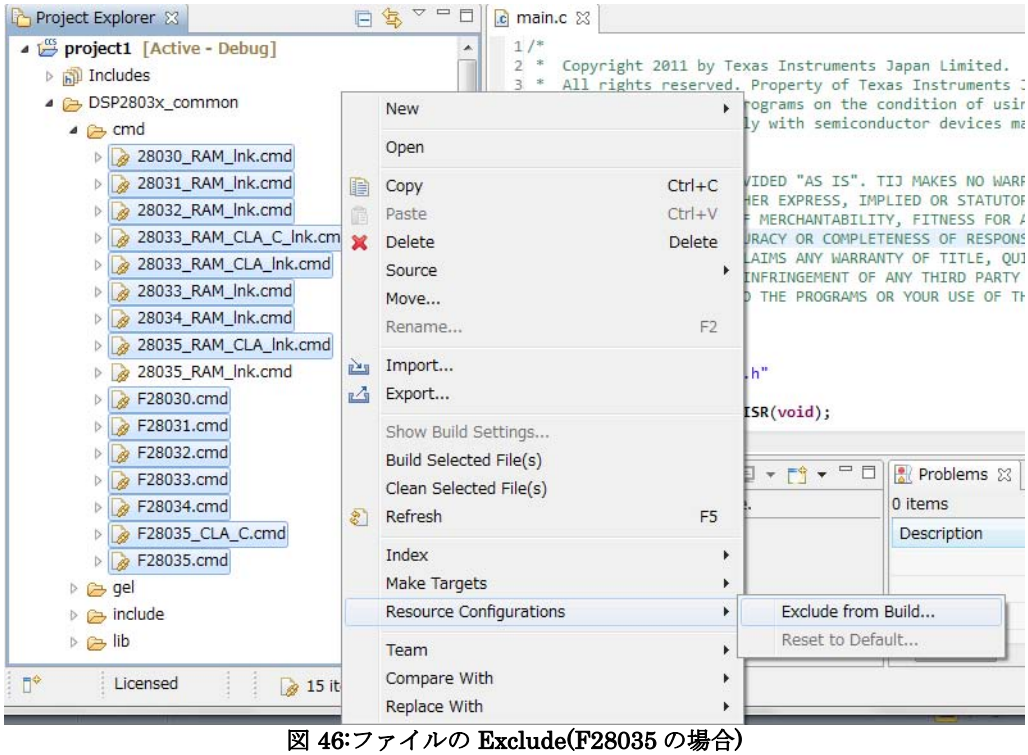


図 46:ファイルの Exclude(F28035 の場合)

このcmdの下から、図 46のように、

F28069の場合	28069_RAM_lnk.cmd
F28035の場合	28035_RAM_lnk.cmd
F28027の場合	28027_RAM_lnk.cmd

以外を全て選択(Ctrl+左クリック)し、右クリックしてResource Configurations → Exclude from Buildを選択して下さい。

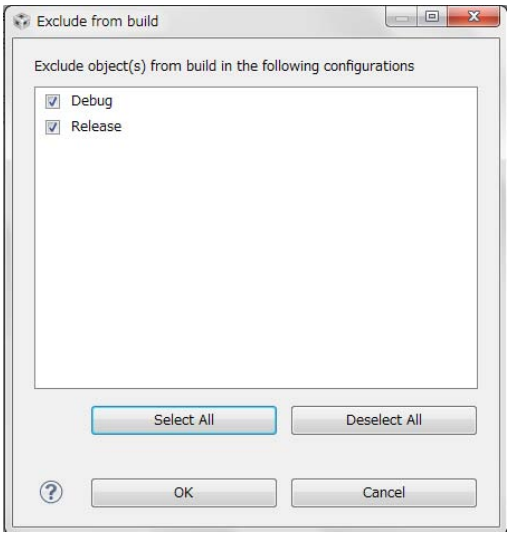


図 47：ファイルの Exclude 設定

図 47のようにExclude from Buildウィンドウが表示されますので、Select Allをクリックして下さい。Debug及びReleaseの両方にチェックがはいります。チェックが入った場合、このファイルはExclude（つまりビルドの対象からはずれます）されます。間違えてExcludeしてしまった場合は、このチェックをはずすか、Deselect Allをクリックすれば、もとに戻ります(つまりビルドの対象になります)。Excludeしたファイルは、Project Explorerでは、アイコンに斜線が入り、グレーの文字にかわったはずです。

続いて、

○F2806x_common/DSP2803x_common/F2802x_common – libの下の全てのファイル
driverlib.lib(F28027のみ)
IQmath.lib(F28027のみ)
SFO_TI_Build_V6.lib (F28027/F28035のみ)
SFO_TI_Build_V6b.lib
SFO_TI_Build_V6b_FPU.lib (F28069のみ)
をExcludeして下さい。

○F2806x_common/DSP2803x_common/F2802x_common －sourceの下の以下のファイル

F28069の場合	F2806x_CodeStartBranch.asm F2806x_CpuTimers.c F2806x_DefaultIsr.c F2806x_PieCtrl.c F2806x_PieVect.c F2806x_SysCtrl.c F2806x_usDelay.asm
F28035の場合	DSP2803x_CodeStartBranch.asm DSP2803x_CpuTimers.c DSP2803x_DefaultIsr.c

	DSP2803x_PieCtrl.c DSP2803x_PieVect.c DSP2803x_SysCtrl.c DSP2803x_usDelay.asm
F28027の場合	DSP2802x_CodeStartBranch.asm DSP2802x_CpuTimers.c DSP2802x_DefaultIsr.c DSP2802x_PieCtrl.c DSP2802x_PieVect.c DSP2802x_SysCtrl.c DSP2802x_usDelay.asm

を残して、それ以外の全てをExcludeして下さい。

○F2806x_headers/DSP2803x_headers/DSP2802x_headers – cmdの下の以下のファイル

F28069の場合	F2806x_Headers_BIOS.cmd
F28035の場合	DSP2803x_headers_BIOS.cmd
F28027の場合	DSP2802x_headers_BIOS.cmd

をExcludeして下さい。最後にプロジェクト作成時に自動生成されたcmdファイル(プロジェクト直下に作成されているはずです。)

F28069の場合	28069_RAM_lnk.cmd
F28035の場合	28035_RAM_lnk.cmd
F28027の場合	28027_RAM_lnk.cmd

をExcludeして下さい。今回のプロジェクトでは、Header Files内のリンカ・コマンド・ファイルを使用しているため、この自動生成されたリンカ・コマンド・ファイル(実は、Header Files内のリンカ・コマンド・ファイルと全く同じものですが)は使用しません。Excludeではなく、Delete（ファイル・システムから削除）してもかまいません。

次にビルド・オプションの設定にて、インクルード・パスの設定を行います。

Project→Properties

を選択して下さい。使用しているコンパイラのバージョンにより、表示が異なる可能性があります。ここで、ビルドに関する様々なオプション(最適化オプション等)を設定できます。

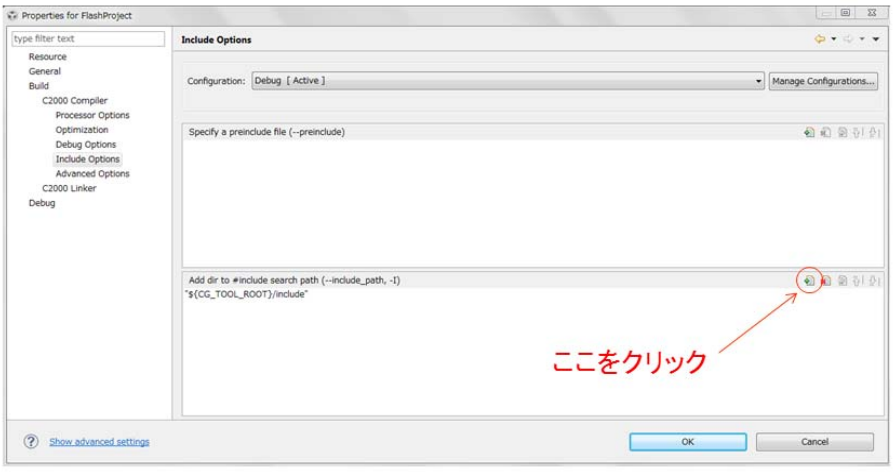


図 48:Include Path の設定①

左の欄は、Buid – C2000 Compiler – Incude Optionsを選択して下さい。
右の欄の
“Add dir to #include search path (--include_path, -I)”欄のAddボタン(上の図 48に表記)をクリックして下さい。

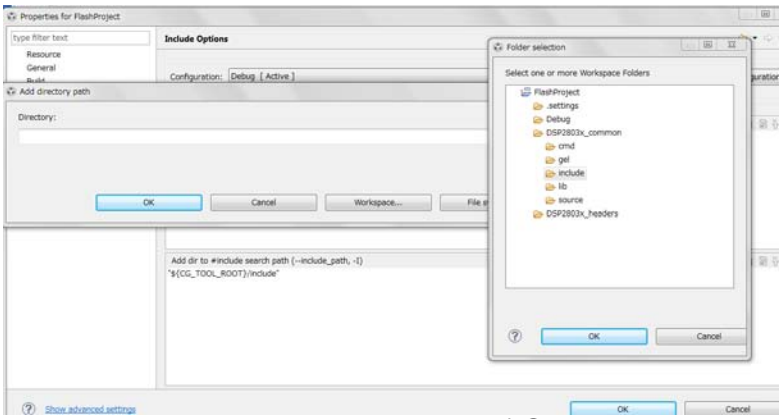


図 49:Include Path の設定②

図 49の左側のAdd directory pathウィンドウが表示されますので、Workspaceをクリックして下さい。図 49の右側のFolder selectionウィンドウが表示されますので、

F28069の場合	F2806x_common ― include F2806x_headers ― include
F28035の場合	DSP2803x_common ― include DSP2803x_headers ― include
F28027の場合	F2802x_common ― include F2802x_headers ― include 一番上のプロジェクト名(今回の例では、project1、つまり、プロジェクト・フォルダ直下のディレクトリです)。

の２つ(F28027の場合は3つ)を選択して(Ctrlキーを押しながら選択すると、同時に選択できます)、OKをクリックして下さい。これで、Include Pathに選択したディレクトリが追加されます。F2802xのみ、Projectのルート・ディレクトリ（一番上のプロジェクト名）を追加する必要がある事に注意下さい。これは、DSP28x_Project.hとF2802x_Device.hがルート・ディレクトリにあるためです。

3.9 main.c の作成

それではコード(main.c)を作成しましょう。プロジェクト作成時にmain.cが自動的に作成されProject Explorerに登録されているはずです。もし、main.cが作成されていなければ、File→New→Source Fileを実行してmain.cを作成してください。
以下のようなコードを記述して下さい。
注意:
コードの２行目の#define LAUNCH_PADは、C2000 LaunchPadを使用する時のみ、コメントを外し、それ以外は必ずコメント扱いにして下さい。
コードの３行目の#define USE_F28069は、F28069を使用する時のみ、コメントを外し、F28027/F28035を使う場合は、必ずコメント扱いにして下さい。

Main.cコード

```
#include "DSP28x_Project.h"
#define LAUNCH_PAD          //When using C2000 LaunchPad, Uncomment this line.
#define USE_F28069          //When using F28069, Uncomment this line.
```

```
interrupt void CpuTimer0ISR(void);

void main(void){

   //System Init
   InitSysCtrl();

   //Interrupt Init
   DINT;
   InitPieCtrl();
   IER=0x0000;
   IFR=0x0000;
   InitPieVectTable();
   EALLOW;
   PieVectTable.TINT0 = CpuTimer0ISR;
   EDIS;

   //GPIO Init
   EALLOW;
#ifdef  LAUNCH_PAD
   GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;  // GPIO0 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;  // GPIO1 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;  // GPIO2 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;  // GPIO3 = GPIO mode
   GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;   // GPIO0 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;   // GPIO1 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;   // GPIO2 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;   // GPIO3 = Output
#else
   GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;  // GPIO34 = GPIO mode
   GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;   // GPIO34 = Output
#endif
   EDIS;

   //Timer Configuration
   InitCpuTimers();
#ifdef   USE_F28069
   ConfigCpuTimer(&CpuTimer0, 80, 1000000);
#else
   ConfigCpuTimer(&CpuTimer0, 60, 1000000);
#endif
   IER |= M_INT1;
   PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

   //Timer Start
   CpuTimer0Regs.TCR.all = 0x4000;

   //Enable Interrupt
   EINT;
```

```
   while(1);
}

interrupt void CpuTimer0ISR(void){

#ifdef  LAUNCH_PAD
   GpioDataRegs.GPATOGGLE.bit.GPIO0 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO1 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO2 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO3 = 1;
#else
   GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
#endif

   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}
```

3.10 main.c の解説

それではmain.cコードの中身を解説していきます。
最初のインクルード

```
#include "DSP28x_Project.h"
```

ですが、このペリフェラル・ヘッダ・ファイルを使う時は、このDSP28x_Peoject.hをincludeする事が必要です。このファイルをincludeするだけで、ペリフェラル・ヘッダファイルにて必要なファイル群が自動的にincludeされます。

次の、この２行は、使用している環境に合わせて、コメントして下さい。

```
#define LAUNCH_PAD          //When using C2000 LaunchPad, Uncomment this line.
#define USE_F28069          //When using F28069, Uncomment this line.
```

1行目は、C2000 LaunchPadを使用する時のみ、コメントを外して下さい。そのExperimenter's kit及びcontroSTICKでは、この行は必ずコメントして下さい。
2行目は、F28069を使用する時のみ、コメントを外して下さい。F28027/F28035を使用する時は、必ずコメントして下さい。

main()関数の最初の

```
InitSysCtrl();
```

ですが、これは、

F28069の場合	F2806x_common\source\ F2806x_SysCtrl.c
F28035の場合	DSP2803x_common\source\DSP2803x_SysCtrl.c
F28027の場合	F2802x_common\source\F2802x_SysCtrl.c

の中にある関数で、主に、

1. Watchdogタイマ(リセット直後はEnable)のDisable
2. オシレータの選択（内部のOSC1-10MHzを選択）
3. PLLの設定

F28069の場合	８てい倍の設定になっています。10MHz x 8 = 80MHz動作です。

F28035の場合	６てい倍の設定になっています。10MHz x 6 = 60MHz動作です。
F28027の場合	６てい倍の設定になっています。10MHz x 6 = 60MHz動作です。

てい倍率を変えたい場合は、

F28069の場合	F2806x_common\include\F2806x_Examples.h
F28035の場合	DSP2803x_common\include\DSP2803x_Examples.h
F28027の場合	F2802x_common\include\F2802x_Examples.h

にて、DSP28_PLLCRとDSP28_DIVSELのdefine設定を変更する事で、変更できます。

4. 各ペリフェラルへのクロック設定(全てをEnableにしています)

を行っています（詳細は、x280x_SysCtrl.cコードを一度ご確認下さい）。

次の、

```
DINT;
InitPieCtrl();
IER=0x0000;
IFR=0x0000;
InitPieVectTable();
EALLOW;
PieVectTable.TINT0 = CpuTimer0ISR;
EDIS;
```

は、割り込みの設定です。今回のコードでは、タイマ割り込みを使用します。

```
DINT;
```

は、グローバル割り込みの禁止(INTMビット=1。1が禁止、0が許可です。)をしています。DINT命令はヘッダ・ファイルの以下のファイルにて定義されているマクロ関数です。

F28069の場合	F2806x_headers\include\ F2806x_Device.h
F28035の場合	DSP2803x_headers\include\ DSP2803x_Device.h
F28027の場合	DSP2802x_Device.h

一度、中身をご確認下さい。インライン・アセンブラ(asm("アセンブラ命令");)を使用したマクロ関数になっています。

リセット直後は、グローバル割り込みは禁止されていますので、本来であれば、このDINTは必要ありません。念のために入れている（また、デバッグ時はリセットせずに再実行する時もあるためです）だけです。

次の

```
InitPieCtrl();
```

は、PIEの初期化(PIEをディセーブルにして、PIEIERとPIEIFRのクリア)を行っています。この関数は、以下のファイルに入っています。この関数は、PIEをディセーブルにしていますので、後で必ずInitPieVectTable()関数をコールする必要があります（この関数にてPIEをイネーブルにしています）。

F28069の場合	F2806x_common\source\ F2806x_PieCtrl.c
F28035の場合	DSP2803x_common\source\DSP2803x_PieCtrl.c
F28027の場合	F2802x_common\source\F2802x_PieCtrl.c

```
IER=0x0000;
IFR=0x0000;
```

にて、CPUレベルの割り込み許可レジスタ(IER)とフラグレジスタ(IFR)のクリアを行っています。IERとIFRの定義は、先ほども見た、x280x_Device.hで定義されています。

次の

InitPieVectTable();

は、全てのPIEベクタ・テーブル（つまり、割り込みベクタテーブル）にデフォルトのダミー割り込みサービス関数を登録しています。そして、PIEをイネーブルにします。この関数と、ベクタ・テーブル自体は、

F28069の場合	F2806x_common\source\ F2806x_PieVect.c
F28035の場合	DSP2803x_common\source\DSP2803x_PieVect.c
F28027の場合	F2802x_common\source\F2802x_PieVect.c

にあり、ダミー割り込み関数は、

F28069の場合	F2806x_common\source\ F2806x_DefaultIsr.c
F28035の場合	DSP2803x_common\source\DSP2803x_DefaultIsr.c
F28027の場合	F2802x_common\source\F2802x_DefaultIsr.c

にあります。このダミー割り込み関数には、

asm(“ ESTOP0 “) ;

と書いてあります。asm(“ “);は、インライン・アセンブラ命令で、アセンブラ命令を埋め込む込むことができます。ESTOP0命令は、通常はデバッガがソフトウェア・ブレークに使う命令で、要はソフトウェア・ブレーク・ポイントとして働きます。この命令は、JTAGが接続されていない状態では、NOP（No OPeration:何もしない）扱いになります。通常は、デフォルトのダミー割り込み関数に飛ぶ事はありません(ユーザーがそのようなコードを意図して書く事は一般的には無いという意味です)。つまり、ここに飛んできた場合は、何かおかしな挙動を示した事になりますので、ブレーク・ポイントを記述しているのです。

割り込みベクタ・テーブルは、本来はユーザーが使用する割り込みについてだけ設定すれば理論的には十分ですが、誤動作、設定ミス等で、想定していない割り込みが起きてしまった場合に備え、何らかの割り込みサービス関数を登録しておくべきです。このInitPieVectTable()では、まずは、全てのベクタ・テーブルを、ダミーISRで埋めてしまう動作をします。ユーザーは、使う割り込みのみ、後でベクタ・テーブルを書き換えてしまえば良いのです。

```
EALLOW;
PieVectTable.TINT0 = CpuTimer0ISR;
EDIS;
```

次のこの3行ですが、はじめに、EALLOW/EDISというこの２行を解説します。レジスタの中には、書込み保護されているレジスタがあります。この保護されているレジスタに書き込みをする場合は、まず、EALLOW命令にて保護を解除してから、書き込みを行い、書き込みが終了したら、EDIS命令にて、再度保護をかけるという手続きを行います。EALLOWもEDISもヘッダ・ファイルにて定義されているアセンブラ・マクロです。

これは、プログラムが間違っていたり、予期せぬ誤動作(もちろん本来あってはなりませんが)等により、ユーザーが書きかえる意図がないのにも関わらず書きかわってしまう可能性を低くするために、設けられている仕組みです。

どのレジスタがEALLOW保護がかかっているかは、データシート及びリファレンス・ガイドを参照して下さい。仮にEALLOW保護がかかっているレジスタに、EALLOW命令を発行せずに書き込みを行った場合は、何も起きません（イリーガル割り込み等の割り込みもかかりません。本当に何もおきません）。つまり、EALLOW命令を発行し忘れた場合は、なかなか気づきにくいのです。プログラムにて、レジスタに値を書いたつもりなのに、それが反映されていない場合は、EALLOW保護がかかっているレジスタかどうかを、まず確認してください。

次に

```
PieVectTable.TINT0 = CpuTimer0ISR;
```

の行を説明しましょう。前にInitPieVectTable()関数にて、PIEベクタテーブルの全てにダミー割り込みサービス関数を登録しました。今回のプロジェクトはタイマ割り込み（TINT0割り込み）を使いますので、その部分のベクタ登録を、ダミー関数から、使用する割り込み関数に登録しなおしています。。

次の

```
EALLOW;
#ifdef   LAUNCH_PAD
  GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;  // GPIO0 = GPIO mode
  GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;  // GPIO1 = GPIO mode
  GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;  // GPIO2 = GPIO mode
  GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;  // GPIO3 = GPIO mode
  GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;   // GPIO0 = Output
  GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;   // GPIO1 = Output
  GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;   // GPIO2 = Output
  GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;   // GPIO3 = Output
#else
  GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;  // GPIO34 = GPIO mode
  GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;   // GPIO34 = Output
#endif
  EDIS;
```

は、GPIOの設定をしています。ボードによって、LEDに接続されているGPIO番号が違いますので、#ifdef-#else-#endifで制御しています。controlSTICKとExperimeter's KitはGPIO34番がLED(controlSTICKはLD2に、Experimenter's Kitはカードの LD3に)に接続されているため、GPIO34の設定をしています。C2000 LaunchPadの場合は、GPIO0/1/2/3がLED(D2/4/3/5)に接続されていますので、GPIO0/1/2/3の設定を行っています。

GPxMUXyとGPxDIRレジスタはEALLOW保護されているレジスタになりますので、EALLOWにて保護を解除します。

```
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;  // GPIO0 = GPIO mode
```

は、GPIO0ピンをGPIOとして使うよう設定しています(これはデフォルト設定ですので、本来はなくても良い行ですが、明示したかったので、記述しました)。GPIO1/2/3/34も同様です。

次の

```
GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;   // GPIO0 = Output
```

は、このGPIO0をGPIO出力として使うよう設定しています。GPIO1/2/3/34も同様です。最後にEDISマクロ関数にて、レジスタの再保護をかけています。

次に

```
InitCpuTimers();
#ifdef   USE_F28069
  ConfigCpuTimer(&CpuTimer0, 80, 1000000);
#else
  ConfigCpuTimer(&CpuTimer0, 60, 1000000);
#endif
IER |= M_INT1;
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

//Timer Start
CpuTimer0Regs.TCR.all = 0x4000;
```

の部分を解説します。まず、以下の行です。

```
InitCpuTimers();
#ifdef   USE_F28069
```

```
   ConfigCpuTimer(&CpuTimer0, 80, 1000000);
#else
   ConfigCpuTimer(&CpuTimer0, 60, 1000000);
#endif
```

これらの行は、CPUタイマ０の設定を行っています。F28027/F28035とF28069のCPU動作周波数設定が異なりますので、#ifdef-#else-#endifで分けています。このConfigCpuTimer()関数は、

F28069の場合	F2806x_common\source\ F2806x_CpuTimers.c
F28035の場合	DSP2803x_common\source\DSP2803x_CpuTimers.c
F28027の場合	F2802x_common\source\F2802x_CpuTimers.c

に入っています。非常に簡単にCPUタイマの設定がこの関数でできますので、今回はこれを使いました。InitCpuTimers()は、CPUタイマの初期化関数で、ConfigCpuTimer(タイマ番号(正確にはタイマ番号へのstruct変数アドレス), CPUクロック周波数(MHz), タイマ周期(usec))にて、タイマの設定をしています。この設定では、CPU Timer0を使って、CPUはF2802x/F2803xは60MHz動作で、F2806xは80MHz動作で、1秒周期のタイマに設定しています。この関数は、CPUタイマ0(TINT0)割り込みもイネーブルにしますので、1秒周期で割り込みが入ります。

CPUタイマ０割り込みは、F2802x/F2803x/F2806x全てでINT1.7にマッピングされていますので、

```
IER |= M_INT1;
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
```

にて、CPUレベル(IER)及びPIEレベル(PIEIER)のINT1/INT1.7の割り込み許可ビットをセットします。M_INT1は、

F28069の場合	F2806x_headers\include\ F2806x_Device.h
F28035の場合	DSP2803x_headers\inlucde\DSP2803x_Device.h
F28027の場合	F2802x_headers\inlucde\F2802x_Device.h

にて定義されていて、M_INTxは割り込み番号に対応したビットが1になっている定数です。

ヘッダ・ファイルには、タイマ・スタート関数が入っていませんので、ここはビット操作をします。下記の行でタイマをスタートさせます。

```
CpuTimer0Regs.TCR.all = 0x4000;
```

最後に、

```
EINT;
```

にて、グローバル割り込みを許可します。

3.11 CPU タイマ０割り込み(TINT0)の ISR(割り込みサービス・ルーチン)

タイマ割り込みのISRは非常にシンプルです。

```
interrupt void CpuTimer0ISR(void){

#ifdef  LAUNCH_PAD
    GpioDataRegs.GPATOGGLE.bit.GPIO0 = 1;
    GpioDataRegs.GPATOGGLE.bit.GPIO1 = 1;
    GpioDataRegs.GPATOGGLE.bit.GPIO2 = 1;
    GpioDataRegs.GPATOGGLE.bit.GPIO3 = 1;
#else
```

```
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
#endif

     PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}
```

まず、非常に重要な点は、割り込みサービス・ルーチンには、必ずinterruptキーワードをつける点です。これで、この関数はISRとして動くことができます。interruptをつけないと、暴走するコードになりますので、ご注意下さい。

```
GpioDataRegs.GPATOGGLE.bit.GPIO0 = 1;
```

にて、GPIO0をトグルします。GPIO1/2/3/34の記述も同様です。このISRは、CPUタイマ0によって、1秒毎に呼び出されますので、結果として 1 秒毎にGPIO0/1/2/3又はGPIO34に接続されているLEDがトグルします。

最後に、PIEの説明の時に解説しましたが、ISRが実行された場合は、必ずPIEACKの対応するPIEグループのビットをクリアする必要があります。このTINT0は、INT1.7ですので、PIEグループは1です。

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
```

この行は、PIEACKレジスタのPIEグループ 1 に対応するビットに1を書いています。このレジスタは”1”に”1”を書いてクリアするレジスタですので、.bitでアクセスしないように注意下さい。通常ペリフェラル割り込みにはペリフェラル・レベルの割り込みフラグのクリアをする必要がありますが、CPUタイマにはペリフェラル・レベル割り込みフラグはありますが、CPUタイマの場合は、クリアしなくても問題ありません（クリアしてももちろんかまいません）ので、今回は省略しています。

これで、コードの解説は終わりです。
虫マークをクリックして、デバッガを立ち上げ、コードを走らせてみて下さい。うまく動作すれば、LEDが 1 秒毎に点滅するはずです。このコードは、CCSからRAM上にコードをロードして実行していますので、CCSをつながないスタンドアローン状態では動作しません。スタンド・アローンで動作させるためには、Flashで動作するように変更する必要があります。これは、次の章で解説します。

4 内蔵 Flash を使った Project

4.1 この章の目的

前章では、CCSからプログラムをRAMにロードして動作させるProjectを試してみました。Piccolo MCUには、Flashが内蔵されていますので、そのFlashの使い方を解説します。Flashを使うためには、単にリンカ・コマンド・ファイルにて、Flash上にコードを配置するだけなく、いくつか使い方に注意が必要です。Flashの詳細につきましては、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章
F2803x用	*TMS320F2803x Piccolo System Control and Interrupts Reference Guide[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts Refernce Guide[SPRUFN3]*

をご参照下さい。また、リセット直後のブートローダの動きについても解説します。

4.2 Flash の解説

F2802x/F2803x MCUは最大60MHzで、F2806x MCUは最大90MHz(量産認定前のサンプル品(TMX型番)は80MHzになります。量産認定品はTMS型番です)動作するMCUですが、この最大動作周波数で動作している場合は、Flashメモリには0waitでアクセスする事ができません。参考に、このドキュメントを書いている時点でのF28035 MCUのデータシート(SPRS584H,設計の際は必ず最新のデータシートを参照下さい)によりますと、以下の図 50がFlashアクセスにかかるwait数になります。

SYSCLKOUT (MHz)	SYSCLKOUT (ns)	PAGE WAIT-STATE(1)	RANDOM WAIT-STATE(1)	OTP WAIT-STATE
60	16.67	2	2	3
55	18.18	2	2	3
50	20	1	1	2
45	22.22	1	1	2
40	25	1	1	2
35	28.57	1	1	2
30	33.33	1	1	1
25	40	0	1	1

(1) Random wait-state must be ≥ 1.

図 50: F28035 の SYSCLOUT と Flash アクセスの wait 数(SPRS854E-REVISED MARCH 2011 から抜粋)

F28035では、CPUクロック(=SYSCLKOUT)が最高の60MHzの場合は、PAGE(同じページ内をアクセス)/RANDOM(ページをまたぐアクセス)ともに、2waitになっています。つまり一回のアクセスに3サイクルかかる事になります。このままでは、アクセスが遅くなってしまいますので、もちろん、それを改善する仕組みが搭載されています。それが、下の図 51に示す、Flashパイプライン・モードです。

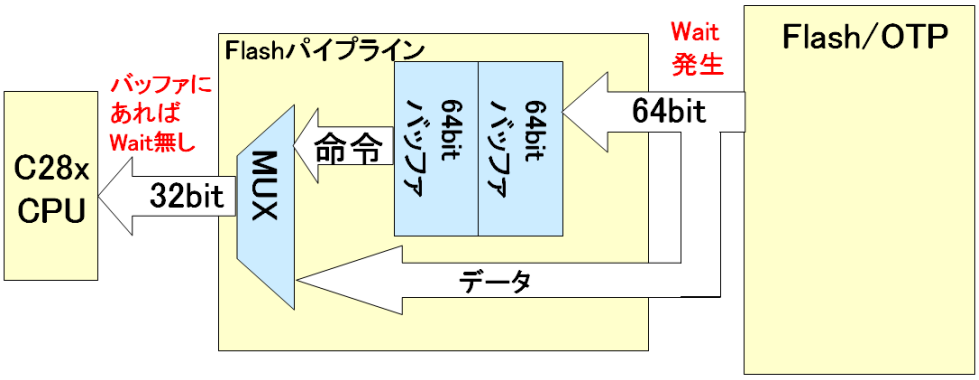


図 51:Flash パイプライン・モード

C28x CPUのプログラム・リード・バスは32bitです。一方、Flashインターフェイスは64bitのバスが用意されています。C28x CPUとFlashメモリ・モジュールの間には、64bitのバッファが2段用意されています。Flashモジュールからは、1アクセスに64bitの命令をリードでき、それを２つためておけるわけです。CPUが60MHz動作の時は、1アクセスは3サイクルになりますので、64bit/3Cycleになります。C28x CPUの命令長は16bitと32bitが混在しています。そのため、仮に全ての命令が16bit

長だった場合は、64bit=4命令になります。つまり、このケースでは、４命令/3cycleになり、実質、待ちが発生しない事になります。もちろん、これはベスト・ケースであって、命令が連続している場合の話です。もし分岐命令があれば、再度アクセスし直しますので、その場合は、waitが発生します。また、全てが32bit命令長であれば、64bit=2命令になりますので、2命令/3サイクルになります。この事から、実際のパフォーマンスは、コードに深く依存します。このため、C28xでは、割り込みルーチンなどの、高速に実行したいコードは、Flash上で走らせるのではなく、RAMで走らせる事を推奨しています。また、Flashパイプライン・モードが働くのは、プログラムに対してであって、データに対してではありません。データ・アクセスに対しては、waitがそのまま見える形になりますので、高速にアクセスする必要があるデータも、RAMにコピーして使うべきでしょう(こちらは、const扱いではなく、初期値をもつ変数にしてしまえば、すむことです)。

その他に、Flash関連のレジスタ(例えば、waitの設定など)を操作するコードは、Flash上で走らせてはならないという規則があります。このように速度は気にしなくても、必ずRAM上で走らせなくてはいけないコードが一部あります。プログラムは、最初はFlash上にあります。一部コードをRAM上で走らせるためには、そのコードをFlashからRAMにコピーして、実行させる仕組みが必要です。この章では、その方法を解説します。前章で作成したProjectをFlashで動作するコードに変更します。

4.3 ブートの仕組み

Flashにコードを書けば、JTAG接続せずに、スタンド・アローンで動作する事が可能です。スタンド・アローンで動かすためにはブートの仕組みを理解する必要があります。ここでは、Piccolo MCUのブート動作について解説します。ブートの仕組みの詳細は、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のBoot ROM章
F2803x用	*TMS320F2803x Piccolo Boot ROM Reference Guide [SPRUGO0]*
F2802x用	*TMS320F2802x Piccolo Boot ROM Reference Guide [SPRUFN6]*

を参照下さい。ここでは、簡単に解説します。

Piccolo MCUは、リセットが入ると、0x3FFFC0番地に飛びます（つまりここがリセット・ベクタです。リセット直後はPIEはディセーブルになっていて、ベクタはPIEベクタとは別のC28x CPUのベクタが使われます。通常このベクタはリセット以外では使用されません）。この0x3FFFC0は、ROM(ユーザーが使える領域ではなく、ＴＩのコードがマスクＲＯＭ化されています)領域になっています。このROM領域内のリセット・ベクタは、同じROM領域内にあるブートローダと呼ばれるコードに飛ぶように設定されています。このブートローダが、特定のGPIO２本(GPIO34/37)とJTAG信号(/TRST信号)、そしてOTP領域のあるアドレスの値により、どのようにスタートするかを判断します。まず、GPIO34、GPIO37、JTAGの/TRST信号により、一段目の選択が行われます。その一段目の選択を図 52に示します。

	GPIO37 TDO	GPIO34 CMP2OUT	TRST	
Mode EMU	x	x	1	Emulation Boot
Mode 0	0	0	0	Parallel I/O
Mode 1	0	1	0	SCI
Mode 2	1	0	0	Wait
Mode 3	1	1	0	GetMode

図 52:Piccolo の Boot ROM によるブート・モード選択

/TRSTはJTAGの信号ですが、この/TRST = 1という状態は、CCSによるデバッグ時を意味します。そのため、CCSにてデバッグをしている時は、このEmulatioin Bootが選択されます。Emulation Bootの中身は、後ほど解説します。/TRSTピンは、外部Pull-downしなければならないピンですので、JTAG接続していない時は、必ず0になります。そのため、JTAGが接続されていない時は、リセット解除時（正確にはリセット解除されてブートローダが走りブートローダがこれらGPIOの値を読みに行く時）のGPIO37とGPIO34の状態によってブート・モードが選択されます。GPIO34もGPIO37も内部Pull-Upされていますので、外で何も処理されてなく、外から信号が入っていなければ、Mode 3が選択される事になります。(注:ブート・モードの選択は、MCUの動作において、非常に重要な動作です。量産設計の際には、内部Pull-upにたよるのでなく、外部でもピン処理を行う事を推奨します。)。

Mode0とMode1は、外部からプログラムをダウンロードして実行する事を意味します(Mode0はI/Oから、Mode1はSCIから)。Piccolo MCUは、Flashを内蔵していますので、通常は外部からプログラムをダウンロードする事はあまりありません。これらのモードは、例えば、最初にFlashに書き込んだり、出荷後にFlashを書き換えたりする場合等に利用されます。Piccolo MCUは、Flashを書き換えるソフトウェアはROMには入っていませんので、外からもらう必要があります。これらのブートモード

を利用して、Flashを書き換えるソフトウェアをまずダウンロードして、それを実行する事で、Flashの中身を書き換えるのです。

Mode2のWaitは、少し理解しにくいかもしれません。このドキュメントではほとんど解説していませんが、Piccolo MCUにはFlash及び一部のRAMの中身をパスワード保護する仕組み(CSM:Code Security Module)があります。パスワード保護がかかっている場合は、PC(プログラム・カウンタ、つまり今実行しているプログラムの場所です)の値が、パスワード保護がかかっている領域を指している場合は、JTAGアクセスは切断させるように設計されています。パスワード保護は、特定のレジスタにパスワードを書く事で解除できます。しかしながら、JTAG接続をしないと、パスワード領域にパスワードを書く事ができないので、このままですと、最初のJTAG接続ができません。このような時にWaitモードを使用します。このwaitモードは、ブートローダにて永久ループを行うモードです。ブートローダがあるROMは、パスワード保護がかかっていませんので、JTAG接続を試みても、切断される事がありません。この状態でJTAG接続を行い、パスワードをCCSから入力すると、パスワード解除ができます。CSMを利用しようとされているユーザーは、このWaitモードに入れるようにボードを設計すべきでしょう。

次のMode3は、OTP(One Time Programmable ROM：1回だけ書き込みができるROMです)内の2つのアドレス(0x3D7BFEと0x3D7BFF)に書かれている値を読み、ブート・モードを判断するモードです。

アドレス	名前	値	
0x3D_7BFE	OTP_KEY	もし、このレジスタが0x55AAであればOTP_BMODEを参照。0x55AA以外であれば、Flashブートを実行	
0x3D_7BFF	OTP_BMODE	0x0001	SCIブート
		0x0004	SPIブート
		0x0005	I2Cブート(パッケージによっては使用不可)
		0x0006	OTPブート
		0x0007	CANブート(F2802xはサポート無し)
		上記以外	Flashブート

表 3:JTAG 未接続時の GetMode 動作

表 3にGetMode選択の表を示します。まず、OTP_KEY(0x3D_7BFE番地)を読み、この値が0x55AAであれば、OTP_BMODE(0x3D_7BFF番地)を参照します。OTPは、ユーザーが一回だけ書き込みができるメモリです。何も書かなければOTPの中身は全て0xFFFFになっていますので、つまり、OTPに何もかかなければ、Flashブートが実行される事になります。もし0x55AAが書いてあれば、OTP_BMODEの中身によって、さらに分岐します。

ここで、もっとも良く使うFlashブートについて、説明しておきます。Flashブート・モードは非常に単純な動作で、0x3F_7FF6番地(Flash領域です)に分岐するだけです。ユーザーは、0x3F_7FF6にプログラムの先頭アドレスに分岐するような命令を書いておけばよいのです。ヘッダ・ファイルを使うと、このFlashブートへの対応が簡単にできますので、後ほど解説します。OTPブートはFlashブートと同様に、OTPの特定番地に分岐します。SCI/SPI/I2C/CANブートは、これらのペリフェラル経由でコードをダウンロードするソフトウェアが実行されます。詳細はリファレンス・ガイドを参照下さい。

次に、JTAG接続時のEmulation Bootモードについて解説します。JTAG接続時に使うモードですので、デバッグ目的が基本です。

アドレス	名前	値	内容
0x00_0D00	EMU_KEY	もし0x55AAであれば、EMU_BMODEを参照。それ以外であれば、Wait。	
0x00_0D01	EMU_BMODE	0x0000	パラレル・ブート
		0x0001	SCIブート
		0x0002	Wait
		0x0003	Get Mode
		0x0004	SPIブート
		0x0005	I2Cブート(パッケージによっては使用不可)
		0x0006	OTPブート
		0x0007	CANブート(F2802xはサポート無し)
		0x000A	RAMブート(0x00_0000番地に分岐)
		0x000B	Flashブート

		上記以外	Wait

表 4:Emulation Boot モード選択

表 4にEmulation Bootモード選択の表を示します。まず、0x00_0D00番地のEMU_KEYを読みに行きます。ここが0x55AAであれば、0x00_0D01番地のEMU_BMODEを見に行きその値によって、各モードに分岐します。もし、EMU_KEYが0x55AAでなければ、waitになります。

さて、このEMU_KEYとEMU_BMODEは、リセット値は不定です(RAMですので)。しかし、JTAG未接続時にブートローダがEMU_KEYに0x55AAを、EMU_BMODEにはGPIO37/34の値を入れるようにプログラムが組まれています。CCSにてデバッグする時は、まずデバイスの電源を入れ、その後CCSを接続しにいきます。そのため、CCS接続した時は、一旦ブートローダがJTAG未接続モードで走っているのです。これにより、EMU_KEY=0x55AAがすでに入っているのです(書き換えてなければ)。また、EMU_BMODEにはGPIO37/34の値がはいっています。GPIO37/34にて特にピン処理をしていない時はHigh/Highですので、EMU_BMODE=3になります。従って、Get Modeになり、Get ModeにてOTPに何もかかれていなければ、結果としてFlashブートになるのです。そのため、何も設定していない場合は、ほとんどがFlashブートになるのです。

controlSTICKもExperimenter's KitもC2000 LaunchPadも、ユーザーが特に何か接続していなく、0x000D00番地の値を書きかえていなければ、Flashブートで立ち上がるようになっています。

4.4 Project の作成

さて、Flashとブートの仕組みを解説しましたので、いよいよ、Projectを作っていきましょう。今回のProjectは前章で作成したProjectを、Flash上で動作するように変更します。

前章のProjectは残しておくために、前回のProjectをディレクトリごとコピーして、それを改造しましょう。FlashProjectというディレクトリを適当な箇所に作成し、前章のProject1フォルダ以下を全てこの新しいフォルダにコピーしましょう。

コピーが終わったら、CCSを立ち上げます。前のProject1プロジェクトがProject Explorerに残っている場合は、まず、このProject1プロジェクトのDeleteを行ってください（Project Explorerのプロジェクト名を右クリックしてDeleteを選択します）。

コピーしたFlashProjectフォルダ内のProject1プロジェクトをImport(File→Import, CCS-Existing CCS Eclipse Projectsです)して下さい。尚、インポートしようとしているプロジェクト名と同じ名前のプロジェクト名が既にC/C++ Projetsウィンドウ内に登録されている場合は、その同じ名前の新たなプロジェクトは選択できない事に注意下さい（このため、前のプロジェクトをDeleteしていない場合は、Deleteしてくださいと記述しました）。

単純にコピーしましたので、Projectネームが同じProject1のままになっています。これでは、前章と同じ名前になってしまいますので、Projectの名前をFlashProjectという名前に変更しましょう。下の図 53のように、Project Explorerウィンドウにて、Project1(プロジェクト名)を右クリックし、Renameを選択して、FlashProjectという名前に変更して下さい。これで、Projectネームの変更ができます。尚、Renameを実行する場合、out of syncのため、Renameを実行できないとエラーを返してくる事があります。その場合、何かのソフトウェア(たとえば、WindowsのExplorer等)で、このディレクトリにアクセスしていないか確認して下さい。アクセスしているソフトウェアをExitした後、少し待つと、解消されます。まれに、それでも解消されない時がありますので、一旦、ProjectをDeleteして、再びImportしてみて下さい。何度か試すと、Renameが出来るようになります。

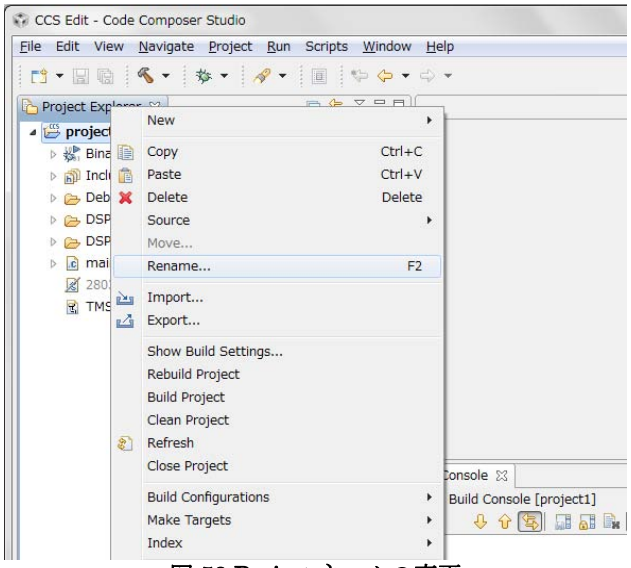


図 53:Project ネームの変更

それでは、次に、Excludeするファイルの設定を変更しましょう。以下の指示に従って、Excludeの設定を変更してください。

F28069の場合	F2806x_common ― cmd 追加でExcludeするファイル: 28069_RAM_lnk.cmd Excludeを解除するファイル: F28069.cmd
F28035の場合	DSP2803x_common ― cmd 追加でExcludeするファイル: 28035_RAM_lnk.cmd Excludeを解除するファイル: F28035.cmd
F28027の場合	F2802x_common ― cmd 追加でExcludeするファイル: 28027_RAM_lnk.cmd Excludeを解除するファイル: F28027.cmd このF28027.cmdに一点、記述ミスがあります。 SECTIONS{}欄内の **.text** : >> FLASHA | FLASHC | FLASHD. **PAGE** = 0 の行にて、FLASHDの次の文字がドットになっていますが、コンマが正しいです。 **.text** : >> FLASHA | FLASHC | FLASHD, **PAGE** = 0 ここを修正しないと、Warningがでますので、修正して下さい。 このドキュメントに付属するサンプル・コードでは、修正したF2802.cmdを用意して、それを使用しています。

この操作は、リンカ・コマンド・ファイルをRAM用からFlash用のリンカ・コマンド・ファイルを選択するよう変更しています。Excludeを解除したF28069.cmd/F28035.cmd/F28027.cmdをダブルクリックして、中身を見てみてください。

F28069.cmdの場合	ramfuncs : LOAD = FLASHD, RUN = RAML0, LOAD_START(_RamfuncsLoadStart), LOAD_END(_RamfuncsLoadEnd),

	RUN_START(_RamfuncsRunStart), LOAD_SIZE(_RamfuncsLoadSize), PAGE = 0
F28035.cmdの場合	ramfuncs : LOAD = FLASHD, RUN = RAML0, LOAD_START(_RamfuncsLoadStart), LOAD_SIZE(_RamfuncsLoadSize), RUN_START(_RamfuncsRunStart), PAGE = 0
F28027.cmdの場合	ramfuncs : LOAD = FLASHA, RUN = PRAML0, LOAD_START(_RamfuncsLoadStart), LOAD_SIZE(_RamfuncsLoadSize), RUN_START(_RamfuncsRunStart), PAGE = 0

という行があります。F28027/28035/28069共に、以前は、ここの記述がほぼ同じだったのですが、最新バージョンでは、デバイス毎に微妙に記述が異なっています。これは、従来のバージョンとの互換性や、アップデートされている、されていないによって、違いが出ています（このファイルを記述した人が共通にする認識が無かっただけかもしれません）。おそらく、この部分は将来のバージョン・アップによって、再びデバイス毎の違いはなくなっていくと思います。

RAM用のリンカ・コマンド・ファイルとの一番大きな違いは、上記の部分(SECTIONSの中にあります)です。この部分を簡単に説明しますと、”LOAD =”という部分が、CCSからロードする時の場所、”RUN=”という部分が、コードが走っている時の場所という設定で、それぞれ、ロード時がFALSHD（もしくはFLASHA）、ラン時がRAML0(もしくはPRAML0)と設定してあります。つまり、このramfuncsというセクションは、ロード時とラン時の２つのアドレスを持つセクションになります。ここで一つ注意ですが、このリンカの設定は、あくまでもアドレスを割り振るだけで、実際のコピーを自動的に行うわけではありません。ロード時のアドレスからラン時のアドレスへのプログラムのコピーは、ユーザーがコードにて記述する必要があります。プログラムをコピーするためには、そのアドレス情報が必要です。それを取得するのが、LOAD_START()、LOAD_END()、RUN_START(),LOAD_SIZE()になります。

LOAD_START(LABEL)	ロード時の先頭アドレスを”LABEL”とする。
LOAD_END(LABEL)	ロード時の最終アドレスを”LABEL”とする。F28069でのみ、このラベルを取得していますが、実際には、このラベルは使用しません。おそらく過去のバージョンの名残かと思います。あっても、特に問題はありません。
LOAD_SIZE(LABEL)	ロードするサイズを”LABEL”とする。
RUN_START(LABEL)	ラン時の先頭アドレスを”LABEL”とする。

ラベルの先頭に”_”(アンダースコア)が実際にはありますが、これは、LABELはアセンブラ表記のラベルだからです。C言語でのラベルが例えば、ABCだとしたら、アセンブラでは、必ず”_”をつけて、_ABCとなります。このラベルですが、変数のシンボルと同じと考えてください。例えば、LOAD_START(LABEL)は、ロード先頭アドレスに、LABELという名前のグローバル変数があるという事と同じシンボル扱いです。そのため、memcpy標準関数の引数としてアドレスを渡す場合は、&LABELになります。LOAD_SIZE(LABEL)は、同じように考えると少々変ではありますが、全く同じ考え方です。そのためmemcpy標準関数にコピーする変数の数の引数に渡す場合は、同じように&LABELになります。

さて、上記のリンカ・コマンド・ファイル記述により、C言語コードから、ラベルが参照できるようになります。それでは、main.cを改造していきましょう。下にそのコードをしめします。赤字の部分が、追加した部分になります（追加だけで修正/削除はありません）。

注意:
FlashからRAMへのコピーは、従来のバージョンのヘッダ・ファイルでは、MemCopy()というヘッダ・ファイル内に用意されている関数を使用していました。最新バージョンでは、MemCopy()を使うのをやめて、memcpy()標準関数を使用するスタイルに変更されています。最新バージョンでは、MemCopy()関数自体が、なくなっていますので、前のバージョンを使用されている場合は、ヘッダ・ファイルのバージョンを上げる時には、この変更に注意して下さい。

改造したmain.c

```
#include "DSP28x_Project.h"
#include <string.h>
//#define LAUNCH_PAD                //When using C2000 LaunchPad, Uncomment this line.
//#define USE_F28069                //When using F28069, Uncomment this line.

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadSize;
extern Uint16 RamfuncsRunStart;

#pragma CODE_SECTION(CpuTimer0ISR, "ramfuncs");
interrupt void CpuTimer0ISR(void);

void main(void){

   //System Init
   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
   InitFlash();

   //Interrupt Init
   DINT;
   InitPieCtrl();
   IER=0x0000;
   IFR=0x0000;
   InitPieVectTable();
   EALLOW;
   PieVectTable.TINT0 = CpuTimer0ISR;
   EDIS;

   //GPIO Init
   EALLOW;
#ifdef    LAUNCH_PAD
   GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 0;  // GPIO0 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 0;  // GPIO1 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0;  // GPIO2 = GPIO mode
   GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0;  // GPIO3 = GPIO mode
   GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;   // GPIO0 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;   // GPIO1 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;   // GPIO2 = Output
   GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;   // GPIO3 = Output
#else
   GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;  // GPIO34 = GPIO mode
```

```
   GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;   // GPIO34 = Output
#endif
   EDIS;

   //Timer Configuration
   InitCpuTimers();
#ifdef   USE_F28069
   ConfigCpuTimer(&CpuTimer0, 80, 1000000);
#else
   ConfigCpuTimer(&CpuTimer0, 60, 1000000);
#endif
   IER |= M_INT1;
   PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

   //Timer Start
   CpuTimer0Regs.TCR.all = 0x4000;

   //Enable Interrupt
   EINT;

   while(1);
}

interrupt void CpuTimer0ISR(void){

#ifdef  LAUNCH_PAD
   GpioDataRegs.GPATOGGLE.bit.GPIO0 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO1 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO2 = 1;
   GpioDataRegs.GPATOGGLE.bit.GPIO3 = 1;
#else
   GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
#endif
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}
```

まず、FlashからRAMへのコピーのために、memcpy標準関数を使いますので、string.hをインクルードしています。

```
#include <string.h>
```

次に以下の部分を解説します。

```
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadSize;
extern Uint16 RamfuncsRunStart;
#pragma CODE_SECTION(CpuTimer0ISR, "ramfuncs");
```

RamfuncsLoadStart、RamfuncsLoadSize、RamfuncsRunStartの3つは、リンカ・コマンド・ファイルにて定義された、ramfuncsセクションのロード開始アドレス、ロード・サイズ、ラン開始アドレスのラベルです。extern宣言する事により、このファイルにて、これらラベルを参照できるようにしています。次の#pragma CODE_SECTION()にて、CpuTimer0ISRという関数が、リンカ・コマンド・ファイルでは、ramfuncsというセクションにしますという宣言になります。CpuTimer0ISRはコードになりますので、何も指示がない場合は、リンカでは.textにマッピングされます。つまり、Flashにロードされます。今回は、このCpuTimer0ISRをロード時はFlashで、ラン時はRAM上で実行する事を行います（一般的にISRは高速に実行する必要がある事が多いので、RAM上で実行される事が多いのです）ので、このpragmaを使った宣言により、ramfuncsというセクション名に変更しています。

他に追記されたのは、以下の２行です。

```
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
InitFlash();
```

memcpy標準関数を使用して、ロード時のアドレスから、ラン時のアドレスにramfuncsセクションにある全てのコード/データをコピーしています。既に解説しましたが、各ラベルに&をつけるのを忘れないようにして下さい。

InitFlash()は、Flashの初期化をしています。このInitFlash()関数は、ramfuncsセクションになっています。InitFlash()は、以下のファイルにあります。

F28069の場合	F2806x_common\source\F2806x_SysCtrl.c
F28035の場合	DSP2803x_common\source\DSP2803x_SysCtrl.c
F28027の場合	F2802x_common\source\F2802x_SysCtrl.c

F2806x_SysCtrl.c/DSP2803x_SysCtrl.c/F2802x_SysCtrl.cを一度見てみてください。最初の方に、

```
#pragma CODE_SECTION(InitFlash, "ramfuncs");
```

という宣言があるのがわかると思います。これにより、InitFlash()もramfuncsセクションになっているのです。このため、memcpy関数にて、CpuTimer0Isr関数と一緒に、FlashからRAMへコピーされます。ここで、非常に重要な事があります。InitFlash()関数は、memcpy関数にてFlashからRAMへコピーされます。InitFlash()関数は、RAM上で実行されるように、アドレスが割り振られていますので、必ず、memcpy関数以降に、実行して下さい。memcpy関数前に、実行すると、一発で暴走するコートになりますので、注意下さい。

さて、InitFlash()の中身を一度みてみましょう。この関数では、Flashの設定をおこなっています。C28x搭載のFlashでは、Flash上のコードからFlash関連レジスタを操作する事は禁止です。必ずRAM上で走らせる必要があるのです。この関数では、Flashのウェイト数の設定を行っています。この章の最初の方で述べましたが、Flashは、CPUの動作周波数にあわせてwaitの調整を行う必要があります。データシートを参照の上、正しい値を設定してください。これを書いている時点でのデータシートと、InitFlash()を見ますと、F28035/F28027の場合は、60MHz動作時の設定になっています。(量産開発時は必ず最新のデータシートとInitFlash()コードを参照下さい)。F28069は、80MHz動作時のwait数が入っています。90MHz動作の場合は、wait数が増えますので、その際はご注意ください(F28069用のバージョンが新しくなれば、90MHz動作時のwait数になっているかもしれません。必ずソースコードをご確認ください)。量産設計の際は、必ず、量産認定後のデータシートを参照の上、この設定をご確認下さい。

次にFlashブートした時の流れを追ってみましょう。Flashブートした場合は、ブートローダから0x3F_7FF6に分岐されます。今回のコードでは、この番地はどうなっているのでしょうか？まず、以下のファイルを見てください。

F28069の場合	F2806x_common\soruce\F2806x_CodeStartBranch.asm
F28035の場合	DSP2803x_common\source\DSP2803x_CodeStartBranch.asm
F28027の場合	F2802x_common\source\F2802x_CodeStartBranch.asm

このファイル中、以下の行があると思います。

```
.sect "codestart"
```

```
code_start:
  .if   WD_DISABLE == 1
    LB      wd_disable      ;Branch to watchdog disable code
  .else
    LB      _c_int00        ;Branch to start of boot.asm in RTS library
  .endif
```

アセンブラで申し訳ありませんが、.sect ”codestart”は、リンカで使うセクション名の定義で、この下はcodestartというセクション名であると宣言しています（先ほどの#pragma CODE_SECTIONのアセンブラ版と考えて下さい）。そして、リンカ・コマンド・ファイルにて、このcodestartセクションが、Flashブートの先頭アドレスである、0x3F_7FF6番地に配置されています。code_startはラベルで、次の.if・.else・.endifはいわゆるプリプロセッサで、C言語の#if・#else・#endifに相当します。WD_DISABLEは、このファイル中の

```
WD_DISABLE    .set          1                    ;set to 1 to disable WD, else set to 0
```

で定義されていて、1になっています。そのため、WD_DISABLE ==1が成立しますので、

```
    LB      wd_disable      ;Branch to watchdog disable code
```

がアセンブルされます。この命令は単純です。単にwd_disableに分岐しなさいという命令です。Flashブートでは、これが最初に実行されます。wd_disableラベルは、同じファイルの後ろにあり、Watchdogをディセーブルにして_c_int00に分岐するコードになっています。さて、C言語のコードで、最初に実行するのは、main()ではありません。main()の前には、C言語環境の初期化関数が実行されます。これはランタイム・ライブラリと呼ばれるライブラリに含まれている関数で、その先頭は_c_int00というラベルになります。つまり、_c_int00とは、C言語コードの本当の最初になります。

さて、次にリンカ・コマンド・ファイル

F28069の場合	F2806x_common\cmd\F28069.cmd
F28035の場合	DSP2803x_common\cmd\F28035.cmd
F28027の場合	F2802x_common\cmd\F28027.cmd

を見てみましょう。

注目すべき箇所は、まずSECTIONS{}の

```
  codestart       : > BEGIN      PAGE = 0
```

です、codestartというセクションがBEGINという場所にアサインされています。BEGINという場所は、同ファイルの上のMEMORY{}には、

```
  BEGIN     : origin = 0x3F7FF6, length = 0x000002
```

とあります。つまり、このcodestartというセクションは、0x3F_7FF6番地に配置されるのです。この0x3F_7FF6番地とは、ブートローダのFlashブートの先頭番地です。つまり、Flashブートでは、ここに分岐してくるのです。このcodestartセクションは、先ほど確認した

```
  LB      wd_disable      ;Branch to watchdog disable code
```

の命令が配置される事になりますので、wd_disableに分岐して、WatchDogをDisableにして、_c_int00に分岐します。_c_int00は必要とされる処理を行い最後にmain()に分岐します。これで、ユーザーコードが実行される事になるのです。

さて、main.cを変更したら、動作確認をしてみてください。CCSでダウンロードしたら、CCS接続を切って、一旦USBを抜いて、再度USBを指してください。LEDが1秒間隔で動作していれば、無事Flash上でコードが動作した証です。

コラム：ロード時とラン時が違うアドレスを持つコードとブレークポイント

今回解説している、ロード時とラン時が違うアドレスを持つコードにて、ブレーク・ポイントを使ったデバッグを行う際に

は、注意が必要です。ブレーク・ポイントをはると、ラン時のアドレスにブレーク・ポイントが貼られます。前にもコラムで解説しましたが、ブレーク・ポイントにはハードウェア・ブレークとソフトウェア・ブレークがあります。RAM上では、デフォルトでソフトウェア・ブレークが使用されます。

さて、ここで、ソフトウェア・ブレークについて、もう一度おさらいしておきますが、ソフトウェア・ブレークは、ブレーク命令(ESTOP0命令)を埋め込む事で実現しています。つまり、ユーザーがブレーク・ポイントをはったとき、その場所の命令がこっそりESTOP0命令に置き換わるのです。ESTOP0命令でHaltした時、CCSは自動的に元の命令に戻します。

このような仕組みのため、ロード時とラン時のアドレスが違う場合は注意が必要です。ロードした直後にブレークをはったとしましょう。プログラムを実行すると、当然ユーザーのコードが、ロード時のアドレスからラン時のアドレスにコードをコピーします。この時、ブレークがはられていた場所には、上書きをしてしまいますので、ブレーク・ポイントがいつのまにか消えてしまうのです！

これを回避するには、ブレーク・ポイントは、プログラムのコピーを行う関数を実行した後にはるという事です。また、再度プログラムを実行し直した場合にもブレークが消えますので(再度コピーしてしまうため)注意下さい。

5 固定小数点と IQmath ライブラリ(主に F2802x と F2803x を対象)

5.1 この章の目的

組み込みプログラミングにて最も悩ましい事のひとつが小数点演算です。小数点演算を行う時、floatやdouble型の浮動小数点演算で記述するのが最も簡単です。しかしながら、FPU(浮動小数点演算ユニット:浮動小数点演算をハードウェアで実行できる)を持つプロセッサは、同じ動作速度の固定小数点演算(整数演算)用プロセッサよりも、一般的には高価になる傾向にあります。C28x MCUシリーズは固定小数点型と浮動小数点型の２種類が用意されています。この資料の対象となっているPiccoloシリーズのなかでは、F2806xシリーズは浮動小数点デバイスですが、F2803xとF2802xシリーズは固定小数点デバイスになります(CLAというモジュールは浮動小数点を使えます)。この固定小数点デバイスの場合、基本的にはＣＰＵにて浮動小数点演算を行わせる事はほとんどなく(ソフトウェアで処理するため、サイクル数がかかります)、固定小数点演算で記述します。このように書きますと、固定小数点はコストを下げるために、しかたなく使用しているように聞こえますが、固定小数点は固定小数点の利点もあります。

また、C28x MCUのようなDSP性能をもつＣＰＵには、固定小数点演算を効率よく実行するための特別の命令を持っています。しかしながら、C言語の仕様では残念ながら固定小数点を効率よく記述するための記述方法がありません。そこで、C28x MCUには、C言語にて効率よく固定小数点演算を実行させるためのライブラリが提供されています。それがIQmathライブラリです。このIQmathを使用する事で、効率よい固定小数点演算を記述できるだけではなく、浮動小数点ライクな（少し大げさかもしれませんが）記述ができます。

この章では、まず浮動小数点と固定小数点とは、そもそも何なのか、またそれぞれの利点と欠点を解説します。次に、IQ mathについて解説します。

5.2 固定小数点フォーマット(IQ フォーマット)

小数点の乗算を行いたい場合、一番簡単な記述方法は浮動小数点型を使用して、

```
float a, b, c

c = a * b;
```

と書けば、cにa * bの計算結果が浮動小数点演算され代入されます。非常に簡単です。しかし、FPUを持たないプロセッサでは、この演算はソフトウェアで処理する必要があるため、非常に多くのサイクル数がかかってしまいます。一方、固定小数点演算を使った場合、記述方法はいろいろありますが、最も一般的な方法で記述しますと、

```
i32 a, b, c;

c = (i32) ((i64)a * (i64)b )>>Q)
```

となります(i32は、32bit整数型, i64は64bit整数型,Qは定数)。固定小数点演算に慣れていないユーザーにとっては、ちょっと難解な記述かもしれません。

まず固定小数点とは、いったい何なのかを解説していきます。結論から先に話してしまいますと、実は単なる整数です。しかし、もちろんそれだけでは、小数点演算はできません。ユーザーは、“整数型の中のどこかに小数点があると仮定“してプロセッサに演算を実行させるのです。ここで、この変数フォーマットは、固定小数点フォーマット、Qフォーマット、IQフォーマットなどと呼ばれています。C28x MCUの関連マニュアルでは、IQフォーマットという呼び方が多いですので、この本でも今後はIQフォーマットと呼ぶ事にします。

C28x MCUは基本的には32bit MCUですので、32bit IQフォーマットを使用するのが最も一般的です。しかし、はじめから32bitで考えますと少し難しくなってしまいますので、話を簡単にするために8bitのIQフォーマットをまず考えてみたいと思います。32bitの場合も単にビット数が増えただけです。8bitのIQフォーマットの例を図 54に示します。

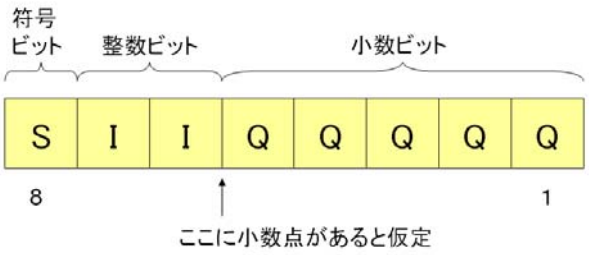


図 54:8bit IQ フォーマット

IQフォーマットは2の補数表現で表され、8bit目の最上位ビットは、符号ビットで±を表します。この例では、5bit目と6bit目に小数点があると仮定しています。そのため、7～6bit目は整数を表し、5～1bit目は小数点以下を表します。小数点以下のビット数が５bitですので、このフォーマットをIQ5フォーマットと呼びます。例えば、7bit目と6bit目の間に小数点があれば、IQ6フォーマットになります。

さて、それでは図 54のIQ5フォーマットでは、どんな数が表せるのでしょうか？正の最大値と、負の最小値はそれぞれ

正の最大値：01111111b = $2^{1}+2^{0}+2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}$ = 3.96875

負の最小値：10000000b = $-1 * 2^{2}$ = -4

分解能 ：2^{-5} = 0.03125

となります。従って、この8bitのIQ5フォーマットでは、3.96875～-4までの範囲で、0.03125単位の数を扱う事ができます。同様に、例えばIQ4フォーマットでは、

正の最大値：01111111b = $2^{2}+2^{1}+2^{0}+2^{-1}+2^{-2}+2^{-3}+2^{-4}$ = 7.9375

負の最小値：10000000b = $-1 * 2^{3}$ = -8

分解能 ：2^{-4} = 0.0625

となります。小数点が最下位ビットに近い程、扱える数の範囲は大きいですが、分解能は荒くなります。ここで、固定小数点フォーマットでは、範囲と分解能はトレードオフの関係にある事がわかると思います。つまり、ユーザーは使用するアプリケーションに応じて最適な小数点の位置を決定する必要があります。

5.3 IQ フォーマットの乗算

IQフォーマットの概要を理解したところで、乗算の話に戻ります。先ほどの固定小数点乗算の式をもう一度見てみましょう。

```
i32 a, b, c;

c = (i32) ((i64)a * (i64)b )>>Q)
```

不思議な箇所が何点かあるかと思います。

- **何故シフトをしているのでしょうか？**
- **何故キャストをしているのでしょうか？**

この疑問を解決していきます。先ほどと同様に、8bitのIQフォーマットを考えてみましょう。8bitの2の補数表現の数を掛け算を筆算形式で、図 55のように1bitずつ掛け算をしていきましょう。

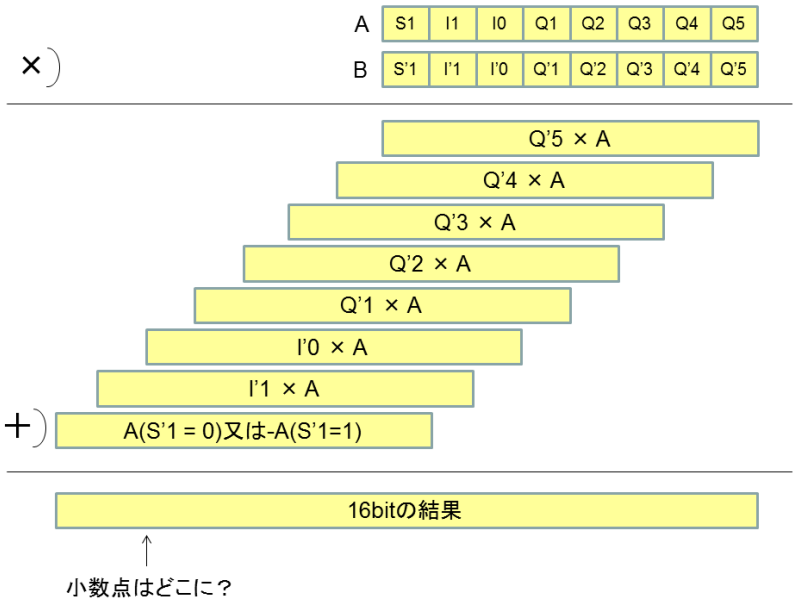


図 55: 8bit IQ5 フォーマットの乗算

まず、8bit * 8bitの乗算の結果は、倍の16bitになる事に注目して下さい。C言語の場合は、16bit 整数型 * 16bit整数型の乗算を行っても答えは16bit 整数型になってしまいます。これは、上位の16bitを切り捨てている事を意味します。Xbit * Xbitの乗算を行う場合、どんな値であっても確実に答えを得るためには、2Xbit必要となる事を示しています。先ほどの32bitIQフォーマットの乗算式をもう一度見てみますと、

c = (i32) ((i64)a * (i64)b)>>Q)

aとbをわざわざ64bit整数型にキャストしている理由は、32bit * 32bit =64bitの結果が欲しいため、わざわざ64bitにキャストして、64bit * 64bit = 64bitと記述しているのです。

さて、図 55の8bitIQ5フォーマット乗算に話を戻します。8bit * 8bit = 16bitの結果がでました。ここで、今扱っているのは仮想的ではありますが、小数です。という事は16bitの結果も小数になります。それでは、16bitの結果の小数点の位置はどこにあるのでしょうか？結論から話してしまいますと、この場合は図 56のように9bit目と10bit目の間に小数点がはいります。

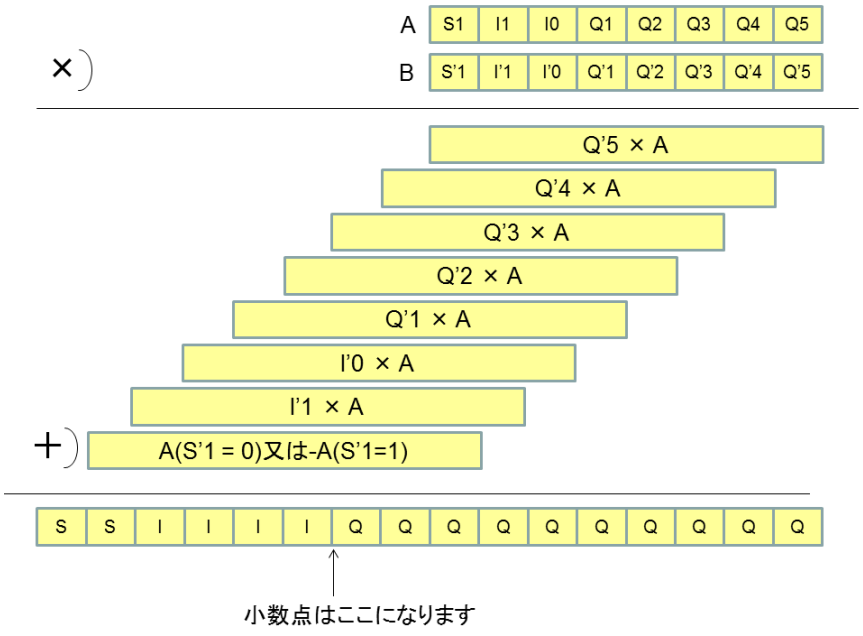


図 56:結果の小数点の位置

この原理は、通常の10進数の小数点演算を行う場合と同じです。さて、8bit * 8bitで乗算を行い、16bitのIQフォーマットの解を得ました。これをそのまま16bitのままにしておく方法もありますが、一般的にはどこかで元の8bitに戻す必要があります。例えばさらにこの値を乗算する等の場合は、bitがどんどん増えていってしまうからです。それでは、この16bitの解の中からどこの8bitを抜き出せばよいのでしょうか？小数点の位置を考えれば、抜き出す部分は明確です。図 57に示すように、

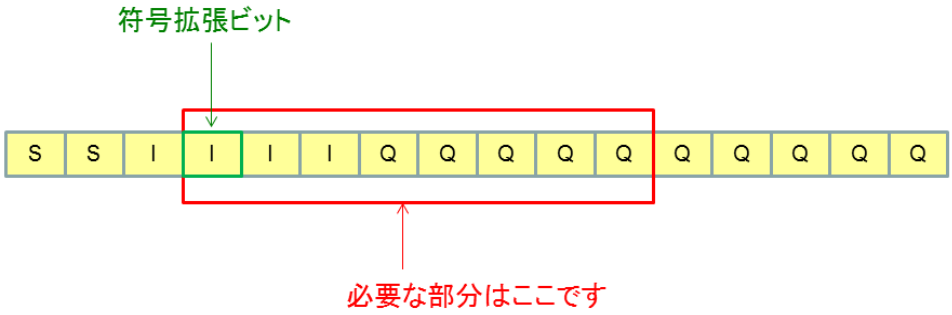


図 57: 8bit IQ5 フォーマット乗算結果の必要な部分

6bit目～13bit目の8bitを抜き出せば同じフォーマットを維持できるので都合が良くなります。この時、抜き出した8bit中、最上位bitは符号bit（符号拡張されているので）に変わります（もし、このビットが符号ビットではなく、値を持つビットの場合、オーバーフローしている事を意味します）。この抜き出す作業をC言語で記述すると、シフトになります。この場合ですと、右に4bitシフトして下位8bitを抜き出せばいいのです。ここで、また32bit乗算の式をもう一度見てみましょう。

c = (i32) ((i64)a * (i64)b)>>Q)

まず、乗算の結果を右にQだけシフトし、32bit整数型にキャストする事で、下位の32bitだけを取り出しています。つまり、このシフト及びキャストは、演算結果の小数点の桁あわせを行っています。

これで、固定小数点乗算についておよそ理解できたかと思います。ここまできて、なんて面倒なんだ！と思われると思います。日頃固定小数点に関わる仕事をしている筆者でさえ、この固定小数点演算は結構面倒です。この章の主題である、C28x MCUのIQmathライブラリは、この面倒な記述を簡単に解決します。その仕組みはまた、後ほどご説明します。

5.4 固定小数点と浮動小数点の利点と欠点

前節では固定小数点の面倒な部分ばかりを解説しました。しかしながら、固定小数点は同じbit幅の浮動小数点に比べて、非常に優れた点があります。ここで、ちょっと意地悪な質問をしてみたいと思います。

10.0 + 0.00000024

の答えは何でしょうか？もちろん、

10.0 + 0.00000024 = 10.00000024

です。それでは、実際に32bit単精度浮動小数点でこの演算をしてみますと、

10.0 + 0.00000024 = 10.000000000??

何故か10になってしまいます。実は、10.00000024は32bit単精度浮動小数点では表せないのです。32bit単精度浮動小数点フォーマットは、図 58のようなフォーマットになっています。

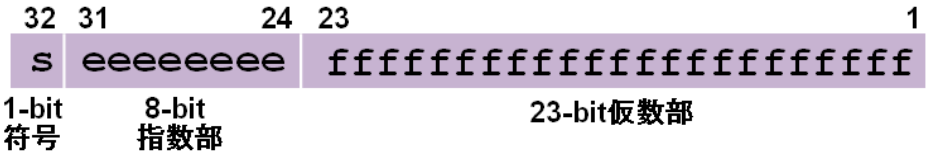


図 58:32bit 単精度浮動小数点フォーマット

最上位ビットが符号ビットであり、その次に8bitの指数部と23bitの仮数部が続きます。このフォーマットで、10.0は、0x41200000です。この数字より1bit大きい値は0x41200001ですが、これは10.00000095を意味します。つまりこの間の値である10.00000024は表せない事を意味します。

それでは、32bit固定小数点フォーマットではどうでしょうか？32bit固定小数点フォーマットは乱暴に言えば1bitの符号ビットと31bitの仮数部（に相当するもの）から成り立っています。

図 59:32bit IQ27 フォーマット例

結論から先に話しますと、10.00000024は、例えばIQ27フォーマットを使用すれば0x50000020で表すことができます。しかもこのフォーマットでの分解能は2^{-27}=約0.00000000745もあります。 なんとこのケースでは浮動小数点よりも固定小数点の方が精度が高いのです。

32bit浮動小数点フォーマットでは、23bitの仮数部に対して、32bit固定小数点は31bitの仮数部を持っています。したがって、固定小数点の方が、分解能が高いといえます(正確には、レンジによって異なります)。しかしながら、非常に幅広いダイナミック・レンジもっていますので、0に近い値では非常に高い精度を持つ事ができます。逆に０から離れるにつれて、精度が低くなり、レンジによっては固定小数点の方が精度が高いのです。少々乱暴な表現ではありますが、ダイナミック・レンジが必要であるならば浮動小数点が、精度が必要であるならば固定小数点が向いています。

5.5 IQmath ライブラリの基本コンセプト

前節にて、浮動小数点と固定小数点の特徴と演算記述方法について解説しました。一概に浮動小数点の方が優れているわけでもない事を理解して頂けたと思います。しかしながら、どちらが扱いやすいか？と聞かれれば、圧倒的に浮動小数点の方が扱いやすいのは事実です。研究段階では浮動小数点が使われ、それを実際に製品にする時に固定小数点に実装しなおすという工程を経る事も多いと思います。

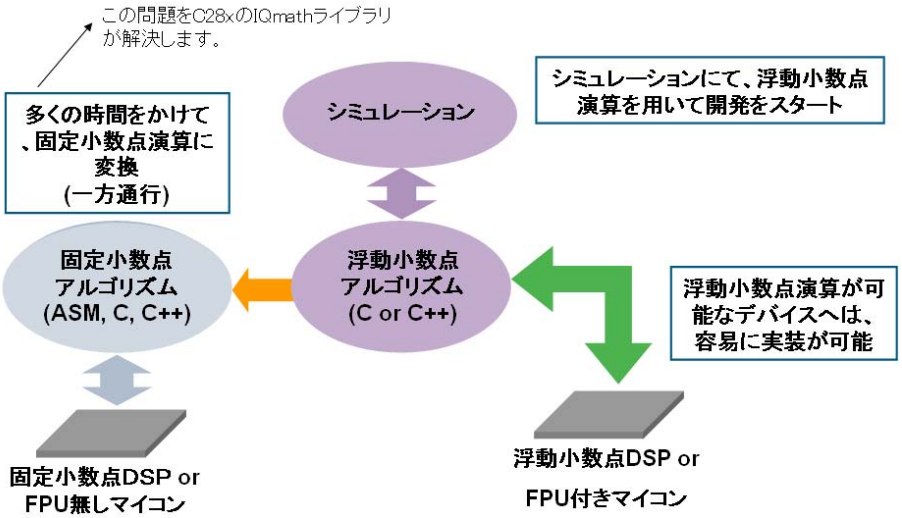


図 60:開発における浮動小数点と固定小数点

しかしながら、前節でも解説しましたように、固定小数点記述は大変面倒である等、浮動小数点アルゴリズムを固定小数点アルゴリズムに変換するには、多くの時間と労力が必要となってしまいます。これを解決するのが、IQmathライブラリです。このIQmathライブラリの基本コンセプトは以下の通りです。

- 固定小数点へ算術演算を実装する時の開発及びデバッグの時間を大幅に短縮します。
- 算術演算におけるスケーリングに関する問題を簡単に解決します。
- 広いダイナミック・レンジを必要としないアプリケーションに対して、固定小数点の精度と浮動小数点の使いやすさを同時に実現します。

5.6 IQmath ライブラリによる数値演算

まずIQmathライブラリを使った場合の固定小数点演算の記述方法から紹介していきます。通常のC言語記述では、32bit固定小数点乗算の記述は以下のようでした。

```
i32 a, b, c;
c = (i32) ((i64)a * (i64)b )>>Q)
```

これがIQmathライブラリを使用して記述すると以下のようになります。

```
 _iq a, b, c;

c = _IQmpy(a, b);
```

非常にすっきりします。ここで、_iqは、固定小数点フォーマット型の宣言型（実際にはlong(32bit)のtypedefです）で、_IQmpy()は、固定小数点乗算を行う関数です。両方とも、IQmathライブラリが提供する環境です（しかし、この記述では、小数点をどこにしているのか全くわかりません。これについては、後程ご説明致します）。

IQmathライブラリが提供する最も重要な要素のひとつが、乗算、除算、三角関数などの固定小数点演算の関数を用意している事です。主要な演算関数を以下に紹介します。

_iq _IQmpy(_iq, _iq)	IQフォーマットの乗算
_iq _IQsmpy(_iq, _iq)	IQフォーマットの丸め飽和付乗算
_iq _IQdiv(_iq, _iq)	IQフォーマットの除算
_iq _IQsin(_iq)	IQフォーマットのsin演算
_iq _IQcos(_iq)	IQフォーマットのcos演算
_iq _IQsqrt(_iq)	IQフォーマットの平方根

表 5:IQmath の主要関数例

これ以外にもいろいろ用意されていますので、詳しくはIQmathライブラリに付属されているドキュメントをご参照下さい。ここで、ひとつ疑問がわいてきます。確かに記述しやすくはなりました。それではパフォーマンスはどうなのでしょうか？なにより、sinやcosならまだしも、たかが乗算をする度に関数コールが発生しては、多くサイクル数を消費してしまいます。しかし、ご安心下さい。_IQmpy等の単純なIQmath関数の場合は、関数コールを行わずにインライン展開されます。また、もちろん、逆にcosやsinなどをインライン展開されては、コードサイズが増えてしまいますので、サイクルがかかる演算は通常の関数コールによって実装されます。どの関数がインライン展開されるのかは、付属のドキュメントを参照して下さい。

それでは、最もよく使用される積和演算のパフォーマンスを見てみたいと思います。積和演算は、

```
_iq a,b, c, d;

d = _IQmpy(a, b) + c;
```

で記述する事ができます。これをコンパイルすると、図 61のようなコードが生成されます。ここでは、詳しく説明はしませんが、コメントを追加しておきましたので、興味のある読者は解読してみて下さい。このコードは、C28x MCUにはこれ以上高速にしようがない最適なコードが生成されています。

```
MOVL    XT,@M
IMPYL   P,XT,@X         ; P   = 下位32bit((32bit)M*(32bit)X)
QMPYL   ACC,XT,@X       ; ACC = 上位32bit((32bit)M*(32bit)X)
LSL64   ACC:P,#(32-Q)   ; ACC:P = ACC:P << Q
ADDL    ACC,@B          ; ACC += B
MOVL    @Y,ACC          ; 結果は = Y = _IQmpy(M*X) + B
```

図 61: IQmath ライブラリを使用した積和演算コード(生成されたコード)

その他の関数も基本的には最適なアセンブリ・コードで記述されています。
ここまでをまとめますと、まず、IQmathには、次の２つの機能が備わっている事がわかります。

- 記述し易さ、見易さ
- 最適な固定小数点演算

5.7 IQmath ライブラリによる小数点の位置

IQmathライブラリから提供されている関数を使えば、簡単に固定小数点演算が実現できる事はわかりました。固定小数点はあくまでも、ユーザーがどこかに小数点があると仮定しているにすぎず、演算においては、どこに小数点があるかによって、乗算後のシフト量が異なってきます。先ほどの、乗算をもう一度見てみましょう。

_iq a,b, c, d;

d = _IQmpy(a, b) + c;

この記述だけでは、a, b, c, dは小数点がどこにあるのかはわかりません。これら_iq型の変数においてどこに小数点を置いて演算をするかを指定する必要があります。これを指定するためにはGLOBAL_Qという値を定義する必要があります。具体的には、以下のように記述します。

#define GLOBAL_Q 27

_iq a, b, c, d;

d = _IQmpy(a, b) + c;

上の例では、_iqはIQ27フォーマットで扱われ、_IQmpyもIQ27フォーマットの固定小数点乗算を行います。GLOBAL_Qはファイル毎に個別に設定する事ができます。
通常はIQ27フォーマットの計算で問題ないが、ある変数だけはどうしてもIQ30フォーマットを用いたいといった場合には、その方法も提供されています。その場合は、

_iq30 a, b, c, d;

d = _IQ30mpy(a, b,) + c;

という記述をします。このように_iqの代わりに_iqN(NはIQNフォーマットを意味)を、_IQmpyの代わりに_IQNmpyをといったように、指定したIQフォーマットでの定義及び演算を記述します。
しかし、一点注意してください。異なったIQフォーマットの演算は実行する事ができません。
例えば、

```
#define GLOBAL_Q 27
_iq a, c, d;
_iq30, b;

d = _IQmpy(a, b) + c;
```

といった計算はできません。また、この場合Cコンパイラはエラーを検出しませんので、ここは十分注意して下さい。これを解決するためには、演算する前にフォーマットを変更する必要があります。これは次節で解説します。

5.8 IQmath ライブラリによるフォーマット変換

算術演算の関数を用意するだけでは、まだまだ快適な固定小数点環境は整える事ができません。例えば、円周率πを考えて見ましょう。πは3.1415.....です。例えば、浮動小数点の環境では、

```
#define PI 3.14159265358979
float a, b;

a = b * PI;
```

のように、そのまま値を記述すれば良いのですが、固定小数点で3.1415.....はどのように表すのでしょうか？ IQmathにはfloatingフォーマット⇔IQフォーマット変換を行う変換マクロ(_IQ( ))を用意しています。例えば、上記のπを定義したい場合は、

```
#define PI _IQ(3.14159265358979)
```

と記述するだけで、PIはIQフォーマットに変換されます。このフォーマット変換マクロも数種類用意されていますが、主要なものを以下に記します。

_iq _IQ(float)	floatフォーマット→IQフォーマットへの変換
float _IQtoF(_iq)	IQフォーマット→floatフォーマットへの変換
_iq _IQNtoIQ	IQNフォーマット→(GLOBAL_Qで指定された)IQフォーマットへの変換
_iqN _IQtoIQN	(GLOBAL_Qで指定された)IQフォーマット→IQNフォーマット

表 6:IQmath の主要なフォーマット変換関数(マクロ)

前節にて違うIQフォーマット同士の演算はできない事を説明しました。しかし、上記のフォーマット変換を利用すれば、簡単に記述できます。

```
#define GLOBAL_Q 27
_iq a, c, d;
_iq30, b;

d = _IQmpy(a, _IQ30toIQ(b) ) + c;
```

5.9 浮動小数点コードをIQmath を使ってC28x に実装する例

それでは、IQmathを使った実例として、C2000シリーズMCUのターゲット・アプリケーションであるモータ制御でよく用いられるコード（三相モータのベクトル制御で用いられる座標変換のコード）にて解説していきます。以下のコードがfloatを使用した浮動小数点コードです(このコードの意味を理解する必要はありません。単純にこの浮動小数点コードをどのように固定小数点コードに置き換えるかという観点で見てください)。

三相モータのベクトル制御における座標変換コード例(floating記述による元コード)

```
#include “math.h”

#define  TWO_PI   6.28318530717959
void park_calc(PARK *v)
{
    float cos_ang , sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

それでは、このコードをIQmathを使った、固定小数点コードに変更していきます。まず、これから使用するIQフォーマットのGLOBAL_Q（ここではとりあえず25としておきましょう）を設定し、次にIQmathライブラリのヘッダファイルをインクルードします。

```
#define GLOBAL_Q  25
#include  “IQmathLib.h”
```

ここで、この２つの順番に注意してください。IQmathLib.hの中で、GLOBAL_Qが既に定義されていない場合は、デフォルト値としてGLOBAL_Qを設定してしまいます。必ず、GLOBAL_Qの設定を先に記述して下さい。

次に、

```
#define  TWO_PI   6.28318530717959
```

の部分は、浮動小数点で書かれていますので、IQフォーマットに変換する必要があります。浮動小数点からIQフォーマットへの変換は、_IQ( )です。この部分を

```
# define TWO_PI _IQ(6.28318530717959)
```

に変更します。

次に以下の変数が

```
float cos_ang , sin_ang;
```

floatで宣言されていますので、こちらをIQフォーマット型に変更します。IQフォーマットの宣言型は、_iqです。以下のように、変更します。

```
_iq cos_ang, sin_ang;
```

最後にfloatingの数式をIQフォーマットの数式に変更します。以下の部分になりますが、

```
    sin_ang = sin(TWO_PI * v->ang);
```

```
    cos_ang = cos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
```

乗算と三角関数の部分を変更する必要があります。加減算はそのままで大丈夫です。乗算は、_IQmpy( )、三角関数は_IQsin( )や_IQcos( )です。以下のように変更します。

```
    sin_ang = _IQsin(_IQmpy(TWO_PI, v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI, v->ang));

    v->de = _IQmpy(v->ds, cos_ang) + _IQmpy(v->qs, sin_ang);
    v->qe = _IQmpy(v->qs, cos_ang) - _IQmpy(v->ds , sin_ang);
```

たったこれだけで、floatingのコードがIQフォーマットのコードに変換ができました。下に元のコードと、変更したコードを記します。

Floatingを使用した元コード	IQmathを使用したIQフォーマット用コード
#include “math.h”	#define GLOBAL_Q 25 #include “math.h” #include “IQmathLib.h”
#define TWO_PI 6.28318530717959	#define TWO_PI _IQ(6.28318530717959)
void park_calc(PARK *v)	void park_calc(PARK *v)
{	{
float cos_ang , sin_ang;	_iq cos_ang , sin_ang;
sin_ang = sin(TWO_PI * v->ang);	sin_ang = _IQsin(_IQmpy(TWO_PI, v->ang));
cos_ang = cos(TWO_PI * v->ang);	cos_ang = _IQcos(_IQmpy(TWO_PI, v->ang));
v->de = (v->ds * cos_ang) + (v->qs * sin_ang);	v->de = _IQmpy(v->ds,cos_ang) + _IQmpy(v->qs ,sin_ang);
v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);	v->qe = _IQmpy(v->qs, cos_ang) - _IQmpy(v->ds, sin_ang);
}	}

網掛けの部分が、変更した部分です。如何でしょうか？変更も非常に簡単で、しかも変更したコードを見てもそれ程、違和感を感じない、見やすいコードになっていると思います。

固定小数点演算を行う場合、どのIQフォーマットをとるか（小数点をどこにあるか）？という事を考える必要があります。上の例では、特に何も考えずにGLOBAL_Qを設定しましたが。使用するアプリケーションによって、慎重に決定する必要があります。IQフォーマットでは扱う事のできる精度と範囲はトレードオフの関係にあります。参考までに、以下に、GLOBAL_Qが20, 24, 28の時の範囲と精度を表します。

GLOGBAL_Q	最大値	最小値	精度
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

表 7:GLOBAL_Q 値(20/24/28 の場合のみ)と取れる値のレンジと精度

もちろん、設計する前に理論的に計算してGLOBAL_Qを設定しておくのが最も望ましいですが、実際には、Try and Errorで決定していく場合も多いかと思います。そんな場合、IQmath環境では、GLOBAL_Qを変更するだけで、同じファイルもし

くは全てのファイルのIQフォーマットを変える事ができますので、調整が簡単にできます。もちろん、局所的に精度やレンジが必要になる変数は、IQフォーマットの変換関数等で対応可能です。

IQmathのライブラリとヘッダ・ファイルは、

C:\TI\controlSUITE\libs\math\IQmath\v160\lib\IQmath_fpu32.lib(FPU付デバイス用)
C:\TI\controlSUITE\libs\math\IQmath\v160\lib\IQmath.lib(FPU無デバイス用)

C:\TI\controlSUITE\libs\math\IQmath\v160\include\IQmathLib.h

ドキュメントは
C:\ti\controlSUITE\libs\math\IQmath\v160\doc
にあります(controlSUITEをデフォルトの場所にインストールした場合)ので、ご参照下さい。この章では、サンプル・コードは提示しません。後のHRPWMのサンプル・コードにて、IQmathを使用してますので(F2802x/F2803x用)、こちらを参照下さい。

コラム：乗算は、IQmathが常に最適か？

この章で解説しましたように、IQmathは固定小数点演算を実行する強力なライブラリです。コードの書きやすさ、見やすさという観点では、疑う余地のない強力なライブラリです。一方、実行速度という観点からすると、もう一工夫できるケースがあります。

_IQmpyは、32bit * 32bit = 64bitを算出し、それをシフトして32bitを取り出すという事を行います。つまり、32bitの固定小数点を非常にまじめに実行しているのです。この演算について、一つ考えどころがあります。C28xは、32bit * 32bit = 64bitの演算を32bit * 32bitの上位32bitの解と32bit*32bitの下位32bitの解との２回にわけて演算します。ストアするレジスタの都合上、この演算には3サイクルかかります。ここで、本当に下位32bitの解が必要か？という疑問がでます。仮に下位32bitの解が必要なければ、32bit * 32bit = 上位32bitになり１サイクルで演算できます。この場合、シフトによって、下位数ビット(Q値に依存します)は、0詰めされてしまいますので、精度が落ちる形になります。しかし、その分高速です。この演算を行うために、Intrinsics関数が用意されています。それが、

long __IQmpy(long A, long B, int N);　// _ではなく__(アンダースコア２つ)です。
意味:A * BのQNフォーマット固定小数点演算

です。実は、IQmathの_IQmpyもこのIntrinsics関数を利用していますが、ここのint Nを32にしてしまえば、32bit * 32bitの上位32bitがとりだせます。

6 ePWM の基礎

6.1 この章の目的

Piccolo MCUは、主にモータ、インバータ等のパワー・エレクトロニクス・アプリケーションをターゲットとして設計されたMCUです。これらアプリケーションにとって、PWM出力は最も重要な機能の一つになります。この章では、Piccolo MCUのPWM機能をもつePWMモジュールについて解説します。ePWMの詳細は、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]のEnhanced Pulse Width Modulator (ePWM) Module及びHigh-Resolution Pulse Width Modulator (HRPWM)の章*
F2802x/ F2803x用	● *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide[SPRUGE9]* ● *TMS320x2802x, 2803x Piccolo High Resolution Pulse Width Modulator (HRPWM) Reference Guide[SPRUGE8]*

をご参照下さい。

6.2 eEV(ePWM/eCAP/eQEP)の全体像

一般的に32bit MCUに搭載されている高機能PWMは、三相モータ用に作られている事が多いです。C28xのePWMモジュールは、三相モータはもちろんなのですが、電源関連にも力を入れている関係で、モータ用三相インバータだけでなく、様々なパワー・トポロジーに対応できるPWMパターンを作りだす事ができます。その分、設定できる（設定しなくてはいけない）レジスタが多いですが、それは、逆に言えば柔軟な設定ができる証拠です。

また、PWM関連モジュールは、キャプチャ機能とA/B相エンコーダ機能を合わせて持っている事が多く、この機能を選択すると、この機能が使えなくなるなど困るといったケースがよく見受けられます。ePWMを含むeEVモジュールでは、PWM機能、キャプチャ機能、エンコーダ機能は、それぞれ独立したモジュール(ePWM/eCAP/eQEP)が用意されていますので、かなり自由に機能を使う事ができます。

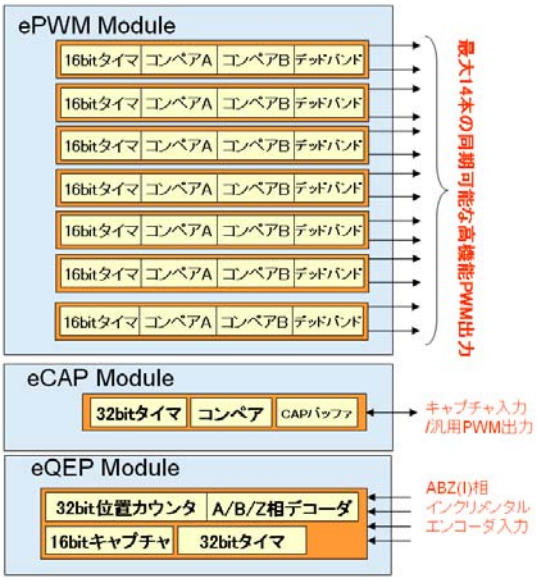


図 62:F28035 MCU の eEV(ePWM/eCAP/eQEP)全体像

図 62(F28035の場合)にeEVの全体像の図を示します。F2802x、F2803x、F2806xに搭載されているePWM/eCAP/eQEPは搭載されているモジュールの本数はそれぞれに違いますが、各モジュールの機能としては、全く同じです。どのデバイスがどのモジュールを何本搭載しているかは、データシートを参照下さい。先ほども少し書きましたが、eEVはePWM, eCAP, eQEPの３つの独立したモジュールから成り立っており、以下のような特徴を持っています。

- ePWMユニットの特徴
  - モジュール間で同期可能な1モジュールあたり１つの16bitタイマ
  - 位相シフトPWMに対応可能な各タイマの位相同期
  - 各タイマにつき2本のデッドバンド付き相補PWM出力
  - 多様な設定が可能な割り込み及びADCトリガ・タイミング
  - 高分解能(150psec)PWM出力:HRPWM、高分解能(150psec)PFMも出力できます。

- eCAPユニットの特徴
  - 1モジュールあたり、1つの独立した、ePWM内のタイマとも同期可能な32bitタイマ
  - キャプチャ入力又はPWM出力として選択可能
  - 各タイマに4つのタイムスタンプ用キャプチャ・バッファ
  - (F2806xは、HRCAPという高分解能キャプチャも搭載しています)

- eQEPユニットの特徴
  - 1モジュールあたり、１組のエンコーダ入力用ユニット
  - 32bit位置カウンタ
  - 各モジュールにA/B/Z(I)相エンコーダのデコーダ回路搭載
  - 速度測定を容易にする16bitキャプチャ及び32bitタイマ

6.3 ePWM モジュールの全体像

それでは、ePWMモジュールの詳細を解説していきます。ePWMモジュールはとにかく柔軟な設定ができるように設計されているため、一見、かなり複雑なモジュールになっています。

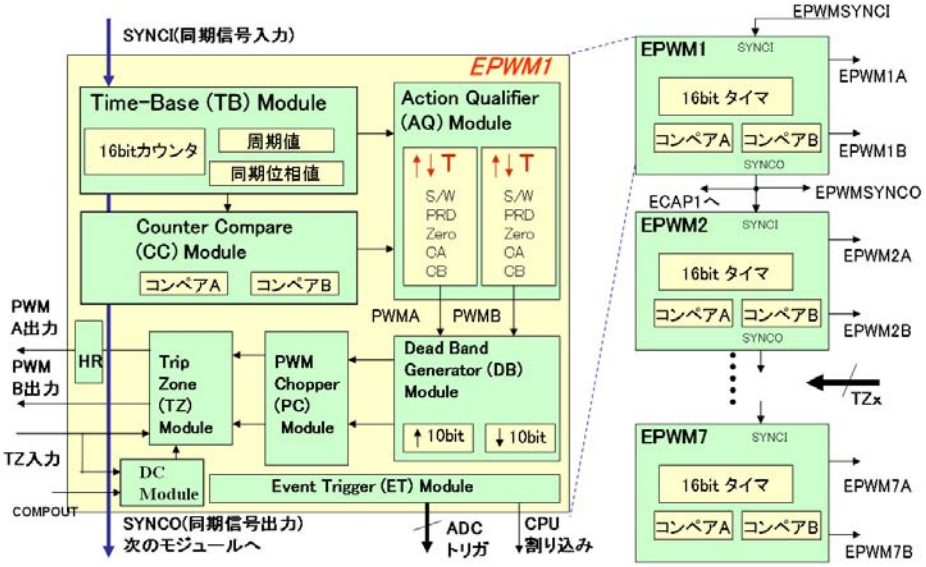


図 63:ePWM モジュールの全体像(図は F28035 の場合のモジュール数)

図 63にePWMモジュールの全体像を示します。ePWMモジュールは、基本的には16bitタイマとコンペア２つをベースにして２つのPWM出力を行う小さいユニットを最大で

F28069の場合	8ユニット

F28035の場合	7ユニット
F28027の場合	4ユニット

備えています。シリーズの中でも型番によって異なっていますので、詳細はデータシートを確認下さい。これら各ユニットのタイマは隣のモジュールのタイマと同期がとれるように、同期信号で結ばれています。図の右側はそれを表しています。
図の左側は、その１ユニット分を拡大した図になっています。１つのユニットは、以下のモジュールから成り立っています。

- Time-Base(TB) Module
  16bitタイマカウンタをもち、カウント動作、タイマの同期動作を行います。
- Counter Compare (CC) Module
  コンペアを２つもち、上記のTB Moduleのカウンタ値に対しコンペア動作を行います。
- Action Qualifier(AQ) Module
  各イベントに対するPWM基本波形を生成します。
- Dead Band Generator(DB) Module
  デッド・バンド(デッド・タイム)生成を行います。
- PWM Chopper(PC) Module
  一旦生成したPWMをさらに高いキャリア周波数にチョッピングします。
- Trip Zone(TZ) Module
  外部信号によりPWM出力をシャットダウンしたり、リミッタ機能を実現します。
- Event Trigger(ET) Module
  CPU割り込み及びADCトリガのタイミングを管理するモジュールです。
- Digital Compare(DC) Module
  このモジュールは、Piccoloシリーズから追加されたモジュールです。Piccoloシリーズは従来のF280xシリーズ等と比べてコンパレータが追加されています。DCモジュールでは、コンパレータとTZ入力を組み合わせて、PWMトリップ、ADC SOC発生などを行うモジュールです。このドキュメントではこのモジュールは解説しません。
- HRPWM Module
  150psec高分解能PWM出力機能です。Piccoloシリーズでは、従来のF280xシリーズと比べてAutoConversionモード、HRPFMモード、対称型PWMのサポートが追加されていますので、より使いやすいモジュールになっています。HRPWM機能は全てのデバイス、全てのユニットでサポートされているわけではありません。どのデバイスのどのユニットでサポートされているかは、データシートを参照して下さい。

6.4 Time-Base (TB)と Counter Compare (CC)モジュール

それでは、その各モジュールをPWM生成の流れとともに解説していきます。図 64にTBモジュールとCCモジュールのブロック図を示します。

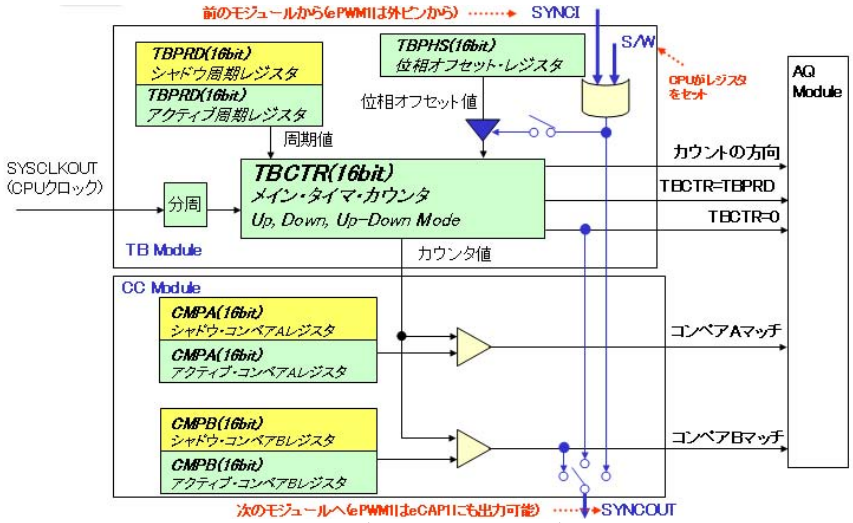


図 64:TB モジュールと CC モジュール

TBモジュールは、主に以下の要素から成り立っています。

- TBCTR : 16ビットのタイマカウンタ (Up, Down, Up-Downモード)
- TBPRD(アクティブ)とTBPRD(シャドウ) : タイマの周期値を設定
- TBPHS :各モジュールのタイマ同士で同期を取る時に使用する位相オフセットレジスタ。

まずタイマカウンタ、TBCTRについて説明します。TBCTRは単なるタイマとして使用する事もありますが、主な目的はPWM生成のための三角波の役割をします。PWMを生成するには、三角波とコンペアを使用する三角波比較型が最も一般的です。このePWMでも基本的にはその生成方法をとっています。但しデジタルですので、実際に三角波形を信号として生成しているのではなく、タイマカウンタで代用しています。さて、そのタイマのカウント方法(三角波の形に相当)により以下の3種類から選択できます。カウンタの源クロックはSYSCLKOUT(=CPU動作周波数)を分周（ユーザーが設定可能）したクロック、TBCLKになります。図 65に設定可能なカウント・モードの図を示します。

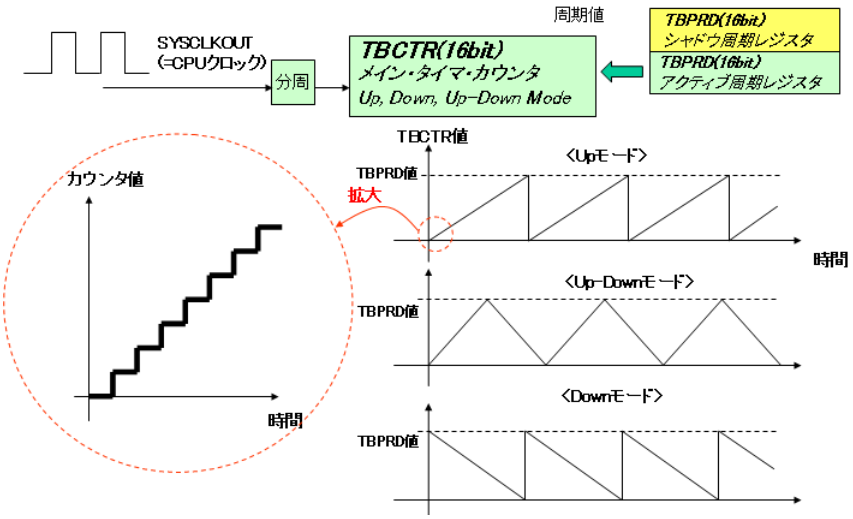


図 65:カウント・モード

- Upモード

このモードでは、入力クロック毎にTBCTR値が１つずつ増えていきます。TBCTRがTBPRDと一致した時、次のクロックでTBCTRは０になります。そしてまたクロック毎にTBCTR値が１つずつ増えていきます。この動作を繰り返します。

- Up-Downモード

  このモードでは、入力クロック毎にTBCTR値が１つずつ増えていきます。TBCTRがTBPRDと一致した時、次のクロックで今度はTBCTR値が１つずつ減っていきます。その後TBCTR値が０に達したとき、次のクロックで再びTBCTR値が１つずつ増えていきます。この動作を繰り返します。

- Downモード

  このモードでは、入力クロック毎にTBCTR値が１つずつ減っていきます。TBCTRが０に達した時、次のクロックでTBCTRはTBPRD値になります。そしてまたクロック毎にTBCTR値が１つずつ減っていきます。この動作を繰り返します。

ここで一点注意していただきたいのは、デジタルPWMの三角波はタイマを使用しているので、アナログ回路による三角波のように連続ではなく、階段状になるという事は理解しておいて下さい。さて、図 64/図 65の中のTBPRDレジスタにアクティブ、シャドウという言葉が書いてありますが、TBPRDレジスタにはアクティブとシャドウの２つのレジスタがあります。PWM生成に関連するレジスタでは一般的な手法なのですが、そのアクティブとシャドウという考え方について、少し触れておきます。

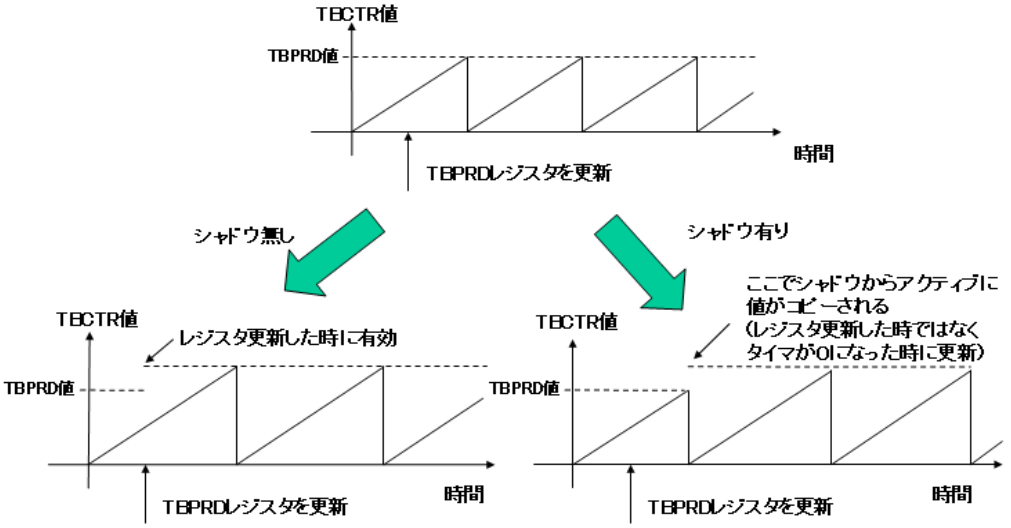


図 66:シャドウ・レジスタの役割

図 66を見てください。TBPRDはタイマの周期を決定します。これはPWMではキャリア周期に相当します。PWMのキャリア周期を変更する時、ある決まったタイミングで変更される事を期待する場合が多いです。例えば、シャドウ用のレジスタがなくアクティブ・レジスタだけだった場合を考えて見ます。図の左下がそれに相当します。TBPRDレジスタがひとつしかないため、ユーザーがこのレジスタを更新したら、その場でTBPRDの値が更新されます。つまりその瞬間、タイマの周期が変更されてしまいます。一方、シャドウ・レジスタがある場合(同図の右下)は、ユーザーはあるタイミングでTBPRDレジスタを更新します。しかし、実はこの更新はシャドウ・レジスタを更新しただけで、アクティブ・レジスタはこの時はまだ更新されていません。実際の周期値を担っているのはシャドウ・レジスタではなくアクティブ・レジスタの方なのです。つまり、この時点ではまだ実際の周期値は更新されていません。しかし、あるタイミング(ユーザーが設定)でこのシャドウ・レジスタからアクティブ・レジスタに自動的に値がコピーされます。この時、初めて実際の周期値が更新されます。図の例では、タイマが0になった時にコピーされた場合を示しています。

つまり、シャドウ/アクティブレジスタとは、ユーザーがいつレジスタを書き換えても、ある決まったタイミングで実際に反映するような仕組みをもてるのです。これ以降、シャドウ/アクティブ・レジスタがいろいろなレジスタで出てきますが、仕組みは同じですので、特に説明はしません。TBPRDの場合、シャドウからアクティブへのコピー・タイミングは以下から選択できます。

1. タイマが０
2. シャドウ・レジスタを更新したそのとき(シャドウ無しと同じ)

このTBモジュールには、他のePWMユニット内のタイマと同期するための、非常に重要な機能がありますが、これをここで解説すると少し話しが複雑になってしまいますので、また後ほど解説します。

さて、TBモジュールは三角波を生成しますが、この三角波を次のCCモジュールとAQモジュールに送ります。CCモジュールではこの三角波を元にコンペア動作を行い、AQモジュールではPWM出力の基本波形を決定します。

次にCCモジュールについて解説します。既に図 64に示されていますように、CCモジュールは２つのコンペア・レジスタ(CMPA及びB)を持っています。先ほども触れましたが、PWMは三角波とコンペアがあれば生成できます。この２つのコンペア・レジスタも、主にPWMのDutyを決定するレジスタとして使用します。しかし、ePWMモジュールのPWM生成は一般的な生成方法と違って、少しユニークな仕組みを持っています。その仕組みの詳細は、後ほど解説します。このCCモジュールは非常に単純な動作のみを行います。

TBモジュールからTBCTR値(要はタイマカウンタ値)を受け取り、CMPAレジスタ及びBレジスタとの比較を行い、コンペア・マッチ(TBCTR値とCMPAもしくはB値が同じ状態)の時、後ろのAQモジュールに信号を送ります。つまり、このユニットは単純にタイマと２つのコンペア値との比較だけを行うユニットと考えて下さい。また、このCMPA/Bレジスタは共に、シャドウ/アクティブ・レジスタ方式をとっています。ここのコピーは以下から選択できます。

1. TBCTR=0の時
2. TBCTR=TBPRDの時
3. TBCTR=0又はTBCTR=TBPRD
4. ユーザーがレジスタを更新した時(シャドウ無し)

さて、ここまでTBモジュールとCCモジュールを見てきました。この2つのモジュールからは図 67のように、

1. カウンタ０(TBCTR=0)
2. コンペアAマッチ(TBCTR=CMPA)かつTBCTRはアップカウント中(Up、Up-Dowmモードのみ)
3. コンペアBマッチ(TBCTR=CMPB)かつTBCTRはアップカウント中(UP、Up-Downモードのみ)
4. カウンタ周期値(TBCTR=TBPRD)
5. コンペアBマッチ(TBCTR=CMPB)かつTBCTRはダウンカウント中(Up-Down、Downモードのみ)
6. コンペアAマッチ(TBCTR=CMPA)かつTBCTRはダウンカウント中(Up-Dowm、Downモードのみ)

の最大６種類のイベントがタイマの１周期中に発生します。

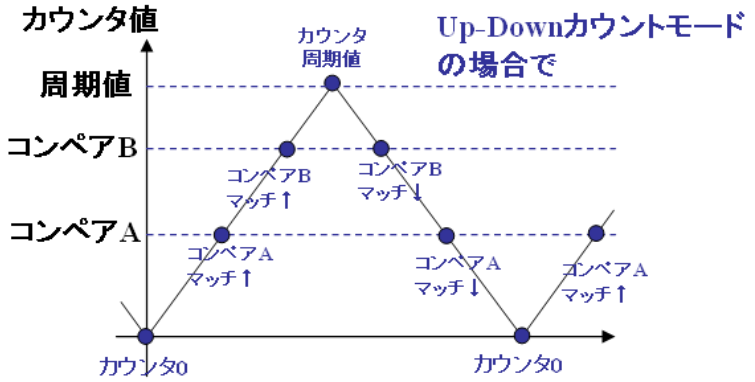


図 67:TB モジュールと CC モジュールによる 6 種類のイベント

TB/CCモジュールはこれらの６種類のイベントを次のAQモジュールに送ります。そしてAQモジュールではこのイベント信号を元にPWMの基本波形を生成します。

6.5 Action Qualifier(AQ)モジュール

前節にて、TB/CCモジュールの解説をしました。この節ではいよいよePWMのユニークなPWM生成方法を解説していきます。そのＰWM生成における基本波形を生成するのがAQモジュールです。TB/CCモジュールから、このAQモジュールにはタイマとコンペアに関する６種類のイベント信号が送られてくる事を前節にて解説しました。

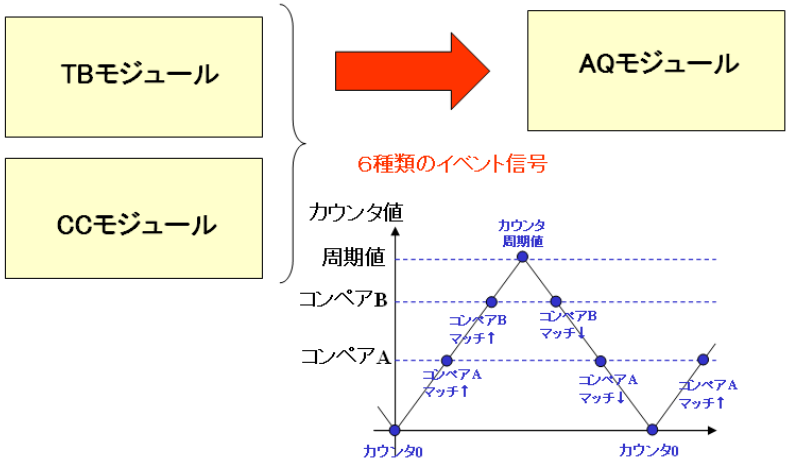


図 68:TB/CC モジュールから AQ モジュールに送られる 6 種類のイベント

AQモジュールではこの６種類の信号をもとにPWM基本波形を作ります。ここで基本波形と書いてあるのにはわけがあります。ここで一旦２本のPWM波形を作りますが、このAQモジュールで生成するPWM波形はまだePWMユニット内の中間的(一時的)な波形です。ここで生成したPWMをその後のモジュールでさらに加工していく事になります。ここでは、その２つの中間PWM波形をAQOUTA/B(リファレンス・マニュアルには、特にこの信号に特定の名前がついていません。あくまでもこのドキュメントでの名前です)とします。

図 69:AQ モジュールでの基本波形生成設定

図 69 にAQモジュールの図を示します。AQモジュールでは、TB/CCモジュールから送られてくる、カウンタ０、カウンタ周期値、コンペアAマッチ↑(アップカウント中)、コンペアAマッチ↓(ダウンカウント中)、コンペアBマッチ↑(アップカウント

中)、コンペアBマッチ↓(ダウンカウント中)の６種類のイベント、及びユーザーが好きなときに生成する事のできるS/W信号（あるレジスタ中の特定ビットに 1をセットする事で生成可能）の計7種類のイベント時に、AQOUT A/B出力に対し、以下の４つの動作をそれぞれ独立して選択する事ができます。

1. High出力(図では↑)
2. Low出力(図では↓)
3. トグル出力(Lowの場合はHighに、Highの場合はLowに)(図ではT)
4. 何もしない(図では×)

また、AQOUTAとBに対して、それぞれ独立して別々に設定することができます。それでは、具体的な設定例を見てみましょう。

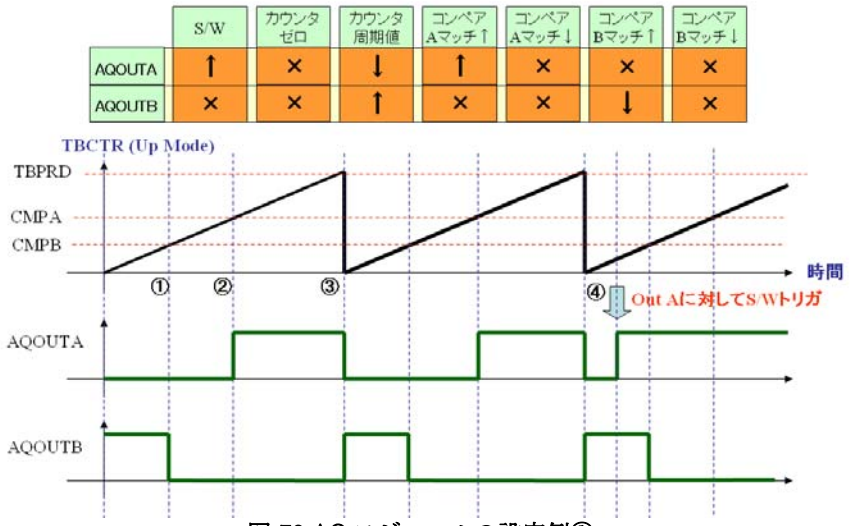


図 70:AQ モジュールの設定例①

図 70の設定例を見てみて下さい。この例では、TBモジュールのTBCTRはUp Modeに設定しています。AQモジュールでは、AQOUTAに対してはS/Wトリガで↑、カウンタ周期値で↓、コンペアAマッチ↑で↑に、その他は×に設定しています。一方、AQOUTBに対してはカウンタ周期値で↑、コンペアBマッチ ↑で↓に、その他は×に設定しています。

順番に見ていきましょう。最初にAQOUTAはLowにAQOUTBはHighになっています。TBCTRがアップカウントしていき、①の時点でコンペアBマッチ↑・イベントが発生します。このイベントでは、AQOUTAは×設定、AQOUTBは↓設定になっています。×は何もしないという事ですのでAQOUTAはここでは何も変化しません。一方AQOUTBは↓という事はLow出力設定ですので、このタイミングでLowに変化します。さらにTBCTRはアップカウントし、②の時点でコンペアAマッチ↑イベントが発生します。ここではAQOUTAは↑設定、AQOUTBは×設定ですので、AQOUTAはHighに変化し、AQOUTBは何も変化しません。さらにTBCTRはアップカウントし、③の時点でカウンタ周期値イベントが発生します。AQOUTAは↓設定、AQOUTBは↑設定ですので、AQOUTAはLowに変化し、AQOUTBはHighに変化します。これを繰り返します。④の時点では、この時、ユーザーがS/Wトリガを発生させるレジスタに値を書き込んでAQOUTAに対するS/Wトリガを発生したとします。この時AQOUTAは↑設定ですので、Highに変化します。S/WトリガはAQOUTAとAQOUTB用に別々に用意されていますので、AQOUTA用のトリガはAQOUTBには影響を与えません。このように設定しておけば、AQOUTAのPWM Duty比はCMPAを、AQOUTBのPWM Duty比はCMPBを変更する事で望みのDutyにする事ができます。

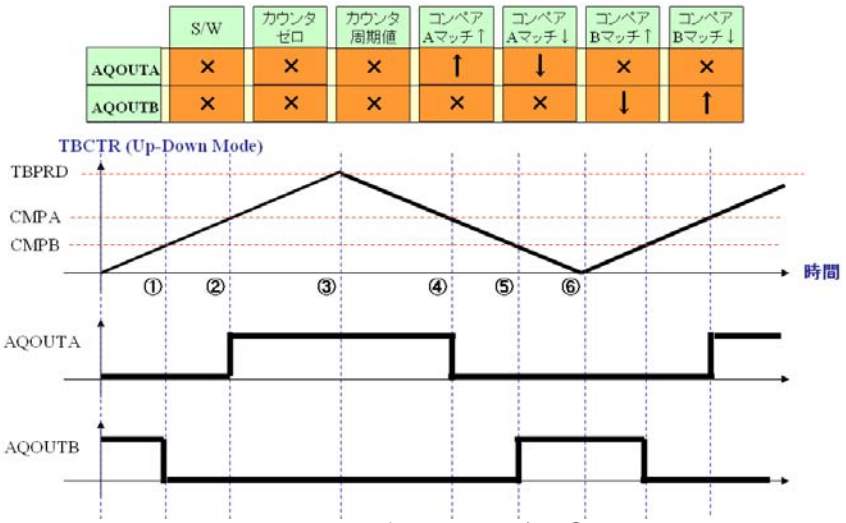


図 71:AQ モジュールの設定例②

もうひとつ例を見てみましょう。図 71は、TBCTRをUp-Downモードに設定した場合の例です。この例ではAQOUTAに対してはコンペアAマッチ↑で↑に、コンペアAマッチ↓で↓に設定しています。一方、AQOUTBに対してはコンペアBマッチ↑で↓に、コンペアBマッチ↓で↑に設定しています。こちらも順番に見ていきましょう。はじめにAQOUTAはLow、AQOUTBはHighになっています。TBCTRがカウントアップしていき①のタイミングでコンペアBマッチ↑イベントが発生します。この時AQOUTAは×、AQOUTBは↓設定ですので、AQOUTAは変化せず、AQOUTBはLowに変化します。次に②のタイミングでAQOUTAは↑、AQOUTBは×設定ですので、AQOUTAはHighに変化し、AQOUTBは変化しません。次に③のタイミングでカウンタ周期値イベントが発生します。この時AQOUTA/Bとも×設定ですので、変化しません。TBCTRはUP-Downモードですので、ここからはダウンカウントを始めます。④のタイミングでコンペアAマッチ↓イベントが発生します。この時AQOUTAは↓設定なのでLowに、AQOUTBは×設定ですので変化しません。次の⑤ではコンペアBマッチ↓イベントが発生します。この時AQOUTAは×設定なので変化せず、AQOUTBは↑設定なのでHighに変化します。最後に⑥ではカウンタ・ゼロ・イベントが発生します。この時はAQOUTA/B共に×設定ですので両方とも変化しません。これを繰り返す事になります。このような設定にすれば、CMPAでAQOUTAのDutyを、CMPBでAQOUTBのDutyを設定する事ができます。

ここにあげた例はあくまで一例です。今まで解説してきたように、このAQモジュールでは６種類のイベント+S/Wイベントでそれぞれ独立にPWM信号変化の設定ができますので、非常に柔軟な設定ができます。

さて、ここで作ったAQOUTAとAQOUTBはまだ中間波形にすぎません。この中間波形を次のDead Band Generator(DB) Moduleに送り、さらにこの波形を加工します。

6.6 Dead-Band Generator(DB)モジュール

前節にて２つのPWM基本波形を生成しました。この基本波形はDead Band Generator(DB)モジュールに送られ、ここでさらに波形を加工する事ができます。DBモジュールはその名のとおり２つのPWM出力に対してデッドバンド(デッドタイム)を付加します。図 72のようなブリッジ回路などは、ハイサイドとローサイドの２つのパワー素子で構成し、このパワー素子を２つの相補関係にあるPWM信号により交互にONするように動作させます。

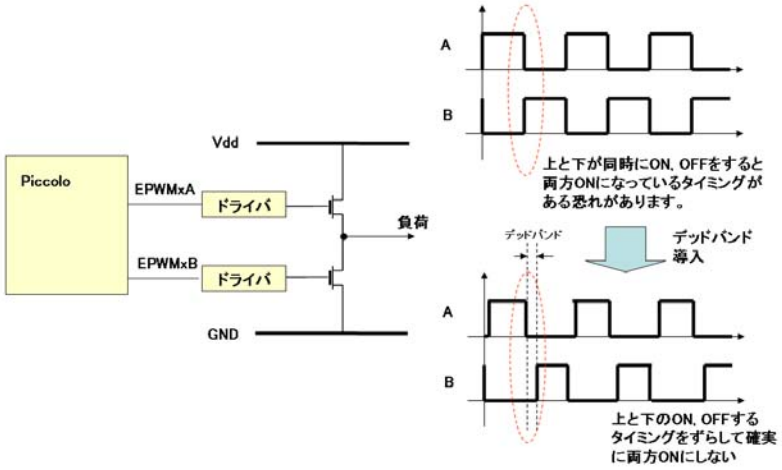


図 72:デッド・バンド(タイム)の挿入

しかしながら、ハイサイドがONからOFFになる同じタイミングでローサイドがOFFからONに変化すると、同時にONになるタイミングがある恐れがあります。同時にONという事は短絡する事を意味し、素子の破壊など重大な事故につながる恐れがあります。これを防ぐために通常デッド・バンドを付加し（両方とも確実にOFFになる時間を設ける）ハイサイドとローサイドのON、OFFするタイミングをずらします。

さて、ePWMに搭載されているDB Moduleはこのデッドバンドを付加するためのモジュールですが、このモジュールもまた非常に柔軟な設定ができるように設計されています。図 73にそのDBモジュールの図を示します。

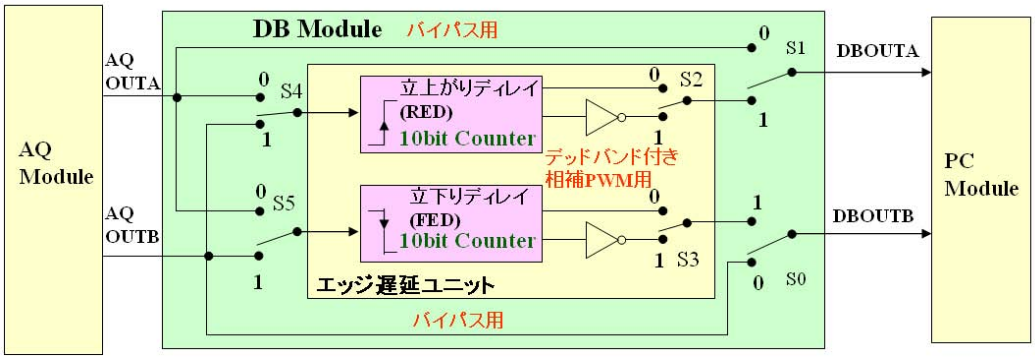


図 73:DB モジュール

DBモジュールも２つの出力を持っていて、次のPC Moduleにその波形を送ります。ここでは、その２つの出力をDBOUTAとDBOUTBとします。DBモジュールはAQモジュールからAQOUTA/Bの２本の基本PWM波形を受け取り、その波形に対してデッド・バンドを付加したり、信号を反転したりする事ができます。

DBモジュールには、AQOUTA/Bから受け取った波形をそのまま（何もせずに）次のモジュールに送るためのバイパス用ラインと、デッド・バンド付加用のエッジ遅延ユニットがあります。エッジ遅延ユニットを通る信号は、エッジ遅延を付加する他に、信号を反転する事もできます。どの信号をどのパスを通すか設定するために、6つのユーザー選択可能なスイッチ（上図のS0,S1,S2,S3,S4,S5）があります。

エッジ遅延ユニットでは、立ち上がり及び立ち下りエッジに対し、それぞれ独立したディレイを設定する事ができます。このディレイの設定範囲は、TBCLK(TBCTRのクロック)で10bit分まで設定できます。また、このディレイ生成に使うクロックは、TBCLKの倍の速度に設定する事もできます。例えば、F2803x CPUを60MHzで動作させ、TBCLKも同じ60MHzで設定した場合、その倍の120MHz(=8.33nsec)単位でディレイを設定できます。

さて、ここまでの説明では、実際にどうやって使うのかがなかなか想像できないと思いますので、代表的な相補PWM出力の設定例(S0:1, S1:1, S2:0, S3:1, S4:0, S5:0)を解説します。図 74にその設定を行った時のパスとPWM波形を示します。

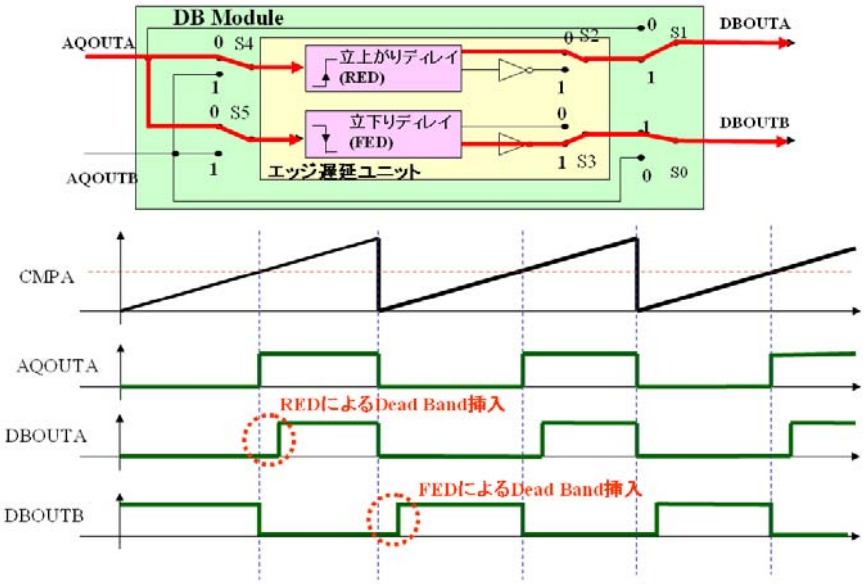


図 74:DB 設定例

この設定では、AQOUTAだけを使います。DBOUTAにはAQOUTAの立ち上がりエッジにディレイを挿入して出力、DBOUTBには同じAQOUTAの立下りエッジにディレイを挿入し、さらに反転して出力しています。この場合はAQOUTBは何も使用しませんので、AQモジュールでのAQOUTBに対する設定はする必要はありません。この例では、TBモジュールはUp-Modeに設定し、AQモジュールではコンペアAマッチ↑イベント発生時にHigh出力に、カウンタ・ゼロ・イベント発生時にLow出力に設定してあります。このように設定する事で、図のAQOUTAのような基本波形が生成されます。DBOUTAは、このAQOUTAの立ち上がりエッジのタイミングでディレイを挿入していますので、図のDBOUTAのような波形になります(REDはRising Edge Delayの略で、立ち上がりディレイの事です)。一方、DBOUTBはAQOUTAの立ち下がりエッジのタイミングでディレイを挿入しています。そしてそれをさらに反転していますので、図のDBOUTBのような波形となります。

さて、この例では、コンペアはCMPAしか使用していませんので、CMPBが余っています。この余ったCMPBは、ADコンバータのトリガタイミングや割り込みタイミングを決めたりする事ができます。この機能は別のモジュールが担当していますので、また後ほど解説します。

6.7 PWM Chopper(PC)モジュール

DBモジュールから、DBOUTA/B信号が生成され、その信号は次のPC Moduleに送られます。このPCモジュールは非常にユニークなモジュールで、DBOUTA/Bにて生成されたPWM波形を高周波化します。パルス・トランス式のゲート・ドライブを使用している時などに便利な機能です。このモジュールは特に必要ない時はディセーブルにする事ができます。

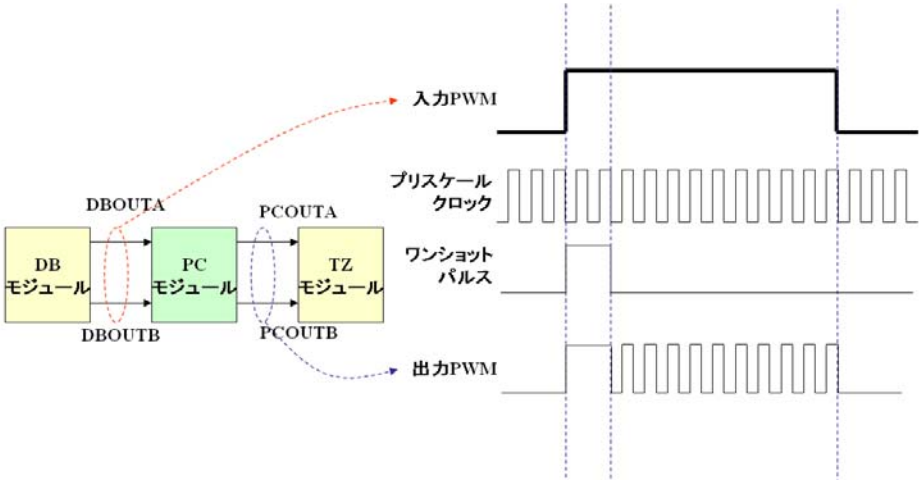


図 75:PC モジュール

図 75にPCモジュールの図を示します。このPCモジュールはDBモジュールから送られてくるDBOUTAとDBOUTBに対して(同図の一番上の波形(入力PWM))、ユーザー設定可能な高周波のクロック(同図のプリスケールクロック)とANDをとります。また、パルス・トランスを確実にONにするために、エッジの最初の部分は、少し長いユーザー設定可能なパルス(同図のワンショット・パルス)に設定する事ができます。これにより入力されたDBOUTA/Bが同図の出力PWMのような波形になります。(このモジュールを使用されているユーザーはあまり多くないように見受けられます。)

6.8 Trip Zone(TZ)モジュール

次にTZモジュールについて解説します。ePWMは主にパワーMOSFETやIGBTなどのパワーデバイスを制御するために用いられます。このようなアプリケーションでは、不測の事態に備え様々な保護機能が必要になります。このTZモジュールは、外部信号により、CPUを通すことなくPWMをシャットダウンできる機能です。また、単にPWMをシャットダウンするだけではなく、リミッタとしての動作機能も持ちます。このドキュメントでは、このモジュールは詳しく触れません。詳細はリファレンス・ガイドを参照下さい。

6.9 Digital Compare(DC)モジュール

次にDCモジュールです。DCモジュールは、前節のTZモジュールと連動して動作します。Picoclo世代になって、アナログ・コンパレータが追加されました。このアナログコンパレータ入力とTZ入力との組み合わせにより、TZモジュールにトリップ信号を出力します。また、ADCのトリガをかける等もできます。このドキュメントでは、このモジュールはかなり複雑なため、詳しくは触れません。詳細はリファレンス・ガイドを参照下さい。

6.10 Event Trigger(ET)モジュール

Event Trigger(ET) Moduleは、主にCPU割り込みとADC変換トリガという２つの機能を持っています。

1. CPUに対する割り込み

   デジタル制御においては、一般的に、ある決まった周期(制御周期)で制御演算を行う必要があります。ＰWM制御の場合は、通常、この制御周期はPWMのキャリア周期と同じ、もしくは定数倍になります。つまり、制御周期はPWM生成のタイミング、つまりPWM生成用タイマと同期する事になります。また、制御周期毎に制御演算を行うためには、CPUに対して、制御演算を開始するよう信号を発生させなければなりせん。通常、これは割り込みが使用されます。ETモジュールではこのCPU割り込みをタイマ周期中のどのタイミングで発生するかを設定できます。

2. AD変換トリガ

   モータ制御や電源制御において、ADコンバータは主にフィードバックする電流や電圧を検出するのに使用されます。アプリケーションによっては、この電流や電圧をどのタイミングで検出するかが重要になる場合が多々あります。ETモジュールではこのADコンバータの変換開始トリガを送るタイミングをタイマ周期中のどのタイミング(位相)で発生するかを設定できます。

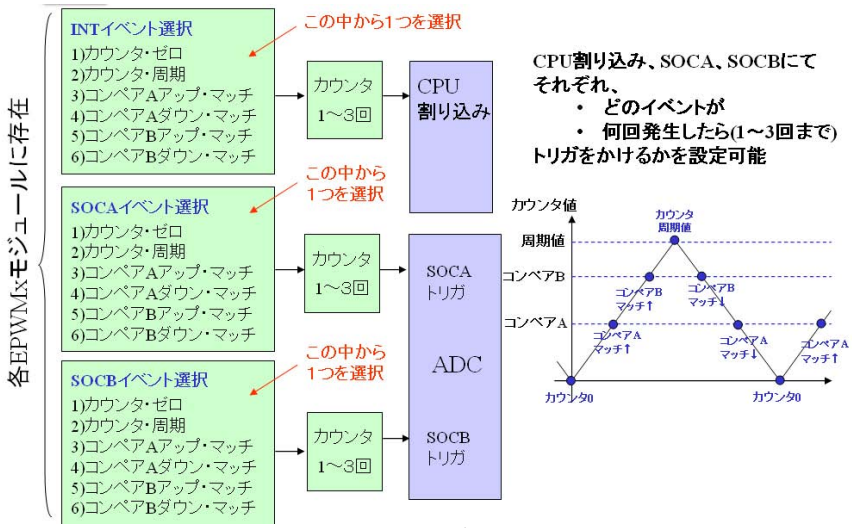


図 76:ET モジュール

図 76にETモジュールの設定できる事をまとめた図を示します。ETモジュールもAQモジュールと同様にカウンタ０、コンペアAマッチ↑、コンペアBマッチ↑、カウンタ周期値、コンペアBマッチ↓、コンペアAマッチ↓の計６イベントの中からCPU割り込みと、ADC変換開始トリガのタイミングを選択できます。

CPU割り込みは、各EPWMxモジュールにつき、一箇所だけ、この６種類のイベントから選択できます。さらに、この選択したイベントが何回発生したらCPU割り込みを発生させるかを選択(1回～３回)する事ができます（つまり間引きです）。これにより、制御周期がPWMキャリア周期よりも遅い場合に、CPUの負担を減らす事ができます。

ADC変換開始トリガの事を、ePWMモジュールではSOC(Start Of Conversion)と呼んでいます。各EPWMxモジュールにつき、SOCAとSOCBという２種類のトリガを使用できます。SOCAとSOCBのADCでの使用方法については、ADコンバータの章を参照して下さい。このSOCAとSOCBはそれぞれ独立してタイミングを設定する事ができます。また、CPU割り込みと同じように、SOCAとSOCBも何回選択したイベントが発生したらSOCA/Bを発生させるかを選択(１回～３回)する事ができます。

6.11 High-Resolution PWM(HRPWM)モジュール

High-Resolution PWM機能は、Piccolo MCUが持つ最大の特徴のひとつで、デジタル制御電源を実現する上で重要な機能です。通常、プロセッサに搭載されているPWMの分解能はCPUの動作周波数が限界になります。例えば、F2803x MCUシリーズの場合はCPU動作周波数は60MHzですので、通常は16.7nsecが分解能の限界になります。HRPWMはこの従来までの限界を突破し150psec(TYP)の分解能を可能にします。一点、注意点としましては、基本的にはA側出力のみHR対応であり、デッドバンド・モジュールは、150psec単位ではありません。

つまり、HRPWMをブリッジ回路で使用する場合は、

1. デッド・バンド生成回路はドライバ側に持たせる
2. ハイサイドはHRモード(150psec)/ローサイドは通常モード(16.7nsec)単位で制御する。

という選択肢になります。従来の固定小数点F280xシリーズ、F2823xシリーズ、Delfinoシリーズ(F2833x, C2834x)は、HR機能はPWMとフェーズ・シフトのみの対応でしたが、Piccoloからは、PFMにも対応しています。また、従来のHRPWMは、希望するDutyからコンペア・レジスタの値を計算する必要がありましたが、PiccoloではAutoconversionモードという新しい機能が追加されたため、この計算が大幅に減り、さらに使いやすくなりました。

6.12 各タイマの同期機能

最後にePWMの各タイマの同期機能について解説します。PiccoloのePWMモジュールは、１個のタイマからは２本のPWM出力が得られ、最大4～8個のタイマを搭載しています。例えばBLDCモータ等の３相モータ制御を例にとれば、6chのPWMを使って３相インバータを制御する必要があります。6chのPWMが必要となれば、３個のタイマが必要となります。三相インバータを制御するための6chのPWMはそれぞれバラバラのタイミングで動作するのではなく、全く同じタイミングで同期して

いる必要があります。各PWMをどのように同期させるかは、アプリケーションによって異なりますが、ePWMモジュール内の各タイマは、非常に柔軟に同期をする事ができます。

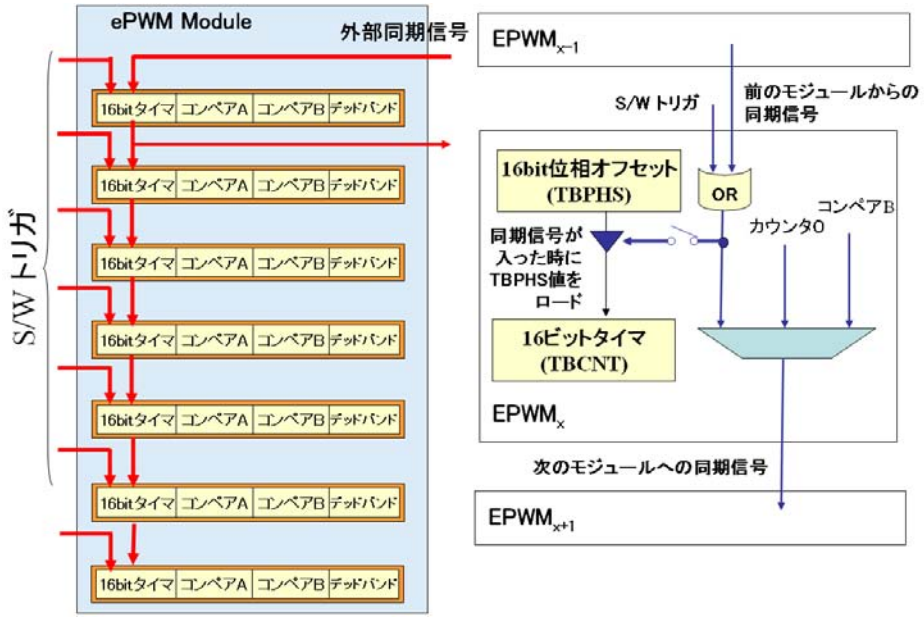


図 77:各タイマの同期機能(F28035 例)

図 77に、各EPWMxモジュールのタイマ間同期についての図を示します。図の左側が全体像を示したものであり、右図が、その中から前後関係にある３つのモジュールを抜き出し、拡大した図になります。

各EPWMxモジュールは、その前後のEPWMモジュールの同期信号とつながっています。例えば、EPWM2のタイマは、EPWM1から同期信号を受け取る事ができると同時に、EPWM3に同期信号を出力する事ができます。この同期信号の入力は必ず前のモジュール(EPWM1の場合は外部から入力する事ができます)となり、出力は次のモジュール(EPWM1の場合は外部に、またはeCAPモジュールに出力する事もできます)になります。例えば、EPWM1の同期信号出力をEPWM3に直接送ることはできません。この場合は、必ずEPWM1→ EPWM2→EPWM3と同期信号を接続しなければなりません。また、同期信号入力は、ユーザーが望みのタイミングで、望みのEPWMxモジュールに対して、対応するレジスタのビットをセットする事で発生させる事もできます(図ではS/Wトリガ)。

この同期信号が発生した時に、位相オフセット(TBPHS)値をタイマ(TBCTR)に自動的にロードする事ができます。位相オフセット(TBPHS)はユーザーが設定するレジスタとなりますので、これは、同期信号が発生した時に、タイマの値を、ユーザーが希望するカウンタ値に、自動的に変更できる事を意味します。それでは、具体的な例を見てみましょう。

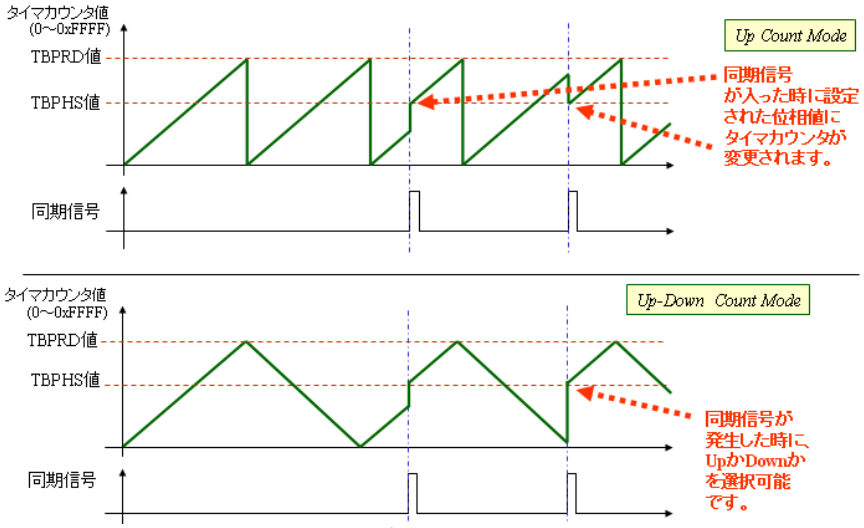


図 78:各タイマの同期

図 78は同期信号が発生した時のタイマカウンタ値(TBCTR)の動きを示した図です。図の上がUpカウントモードの時、下がUp-Downカウント・モードの時の図になります。まず、上の図を見て下さい。同期信号が入った時に、タイマカウンタ値が

TBPHS値に変わっている事がわかると思います。また、下の図はUp-Downカウント・モードになります。同期信号が入った時には同じようにTBPHS値に値が変わっていますが、変わった後、Upカウントをするのか、Downカウントをするのかを指定する必要があります。これを指定するレジスタが用意されていますので、Up-Downモードの時は注意して下さい。

この同期機能により、マルチフェーズDC/DC等で用いられる、ある決まった位相差をもつPWMや、フェーズシフトPWM等も簡単に作ることができます。また、外部ピンからの信号に同期する事もできます。

7 ePWM モジュールの使用例

7.1 この章の目的

前章にて、eEV(ePWM/eCAP/eQEP)の全体像及びePWMの各機能の詳細を説明しました。この章では、ePWMの使用方法の代表的な3つの例について具体的な設定方法を解説します。今までのサンプル・コードでは、F28027/28035は60MHz動作、F28069は80MHz動作させてきましたが、今回の章は、動作周波数が異なると同じ文章で解説しにくいため、F28069は60MHz動作に速度を落としたコードで説明します。ご了承下さい。このドキュメントに付属されているサンプル・コードは、ヘッダ・ファイルが含まれていません。デバイスの動作速度の変更は、ヘッダ・ファイル内のファイルの変更が必要なため、ユーザーにて、変更が必要となります。変更方法は、1.6章の”このドキュメントに付属されているサンプル・コード例について”を参照下さい。

7.2 同じキャリア周波数の 2ch の Duty の異なる PWM 出力

それでは、まず、最も簡単な例から見ていきましょう。目的のPWM波形を図 79に示します。

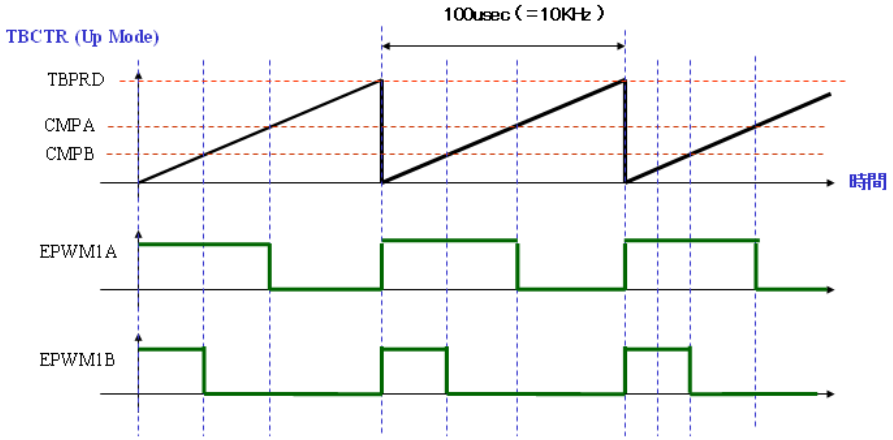


図 79:同じキャリア周波数の 2ch の Duty の異なる PWM 出力波形

EPWM 1 を使って、のこぎり形の三角波に対し、２つのコンペアを用いて同じキャリア周波数(10KHz)のDutyの異なる2chのPWM出力を得ます。

それでは、Projectを作成しましょう。今回はコピーではなく、新規に作成しましょう。プロジェクトの構成は基本的には今までと同じです。新規にディレクトリを一個作成して、ヘッダ・ファイル

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ
F2802xの場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_commonフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_headersフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\DSP28x_Project.h C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_Device.h

をその作成したディレクトリの下にコピーしてください。そして、この作成したディレクトリにCCSから新規Project(付属のサンプルコードではPwmProject1)を作成して下さい。作成したら、不必要なファイルをExcludeしていきます。

F28069の場合	F2806x_common ― cmd F28069.cmdを残して、他は全てExclude F2806x_common ― lib

	全てExclude F2806x_common ― source F2806x_CodeStartBranch.asm F2806x_DefaultIsr.c F2806x_PieCtrl.c F2806x_PieVect.c F2806x_SysCtrl.c F2806x_usDelay.asm を残して他は全てExclude F2806x_headers ― cmd F2806x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28069_RAM_lnk.cmdをExclude
F28035の場合	DSP2803x_common ― cmd F28035.cmdを残して、他は全てExclude DSP2803x_common ― lib 全てExclude DSP2803x_common ― source DSP2803x_CodeStartBranch.asm DSP2803x_DefaultIsr.c DSP2803x_PieCtrl.c DSP2803x_PieVect.c DSP2803x_SysCtrl.c DSP2803x_usDelay.asm を残して他は全てExclude DSP2803x_headers ― cmd DSP2803x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28035_RAM_lnk.cmdをExclude
F28027の場合	DSP2802x_common ― cmd F28027.cmdを残して、他は全てExclude FlashProjectの時と同じく、 SECTIONS{}欄内の **.text** : >> FLASHA | FLASHC | FLASHD. **PAGE** = 0 の行にて、FLASHDの次の文字がドットになっていますが、コンマが正しいです。 **.text** : >> FLASHA | FLASHC | FLASHD, **PAGE** = 0 ここを修正しないと、Warningがでますので、修正して下さい。 DSP2802x_common ― lib 全てExclude

	DSP2802x_common — source DSP2802x_CodeStartBranch.asm DSP2802x_DefaultIsr.c DSP2802x_PieCtrl.c DSP2802x_PieVect.c DSP2802x_SysCtrl.c DSP2802x_usDelay.asm を残して他は全てExclue DSP2802x_headers — cmd DSP2802x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28027_RAM_lnk.cmdをExclude

include pathの設定も必要です。この設定は、前と同じです。念のため、以下に記載しておきます。

F28069の場合	F2806x_common — include F2806x_headers — include
F28035の場合	DSP2803x_common — include DSP2803x_headers — include
F28027の場合	F2802x_common — include F2802x_headers — include 一番上のプロジェクト名(今回の例では、PwmProject1つまり、プロジェクト・フォルダ直下のディレクトリです)。

F28069の場合は、
F2806x_common\include\F2806x_Exapmles.h
を変更して、60MHz動作に速度を落としましょう。このファイルをダブルクックして、エディタにて立ち上げてください。
変更箇所は、

#define DSP28_PLLCR 16 // Uncomment for 80 MHz devices [80 MHz = (10MHz * 16)/2]
をコメントアウトして、

#define DSP28_PLLCR 12
のコメントをはずして下さい。これで、PLLの設定が変更され、80MHz動作から60MHz動作に変わります。
このドキュメントに付属のサンプル・コードでは、Header Filesが含まれていませんので、この変更を必ず行ってください。

次に、main.cを作成して、以下のように記述して下さい。

Main.c

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");
interrupt void Epwm1Isr(void);
void ePWM1Config(void);
```

```
void main(void){

   DINT;
   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
   InitFlash();

   InitPieCtrl();
   IER=0x0000;
   IFR=0x0000;
   InitPieVectTable();
   EALLOW;
   PieVectTable.EPWM1_INT  = Epwm1Isr;
   EDIS;

   ePWM1Config();
   EINT;

   while(1);
}

void ePWM1Config(void){

   EALLOW;
   /*Stop PWM Timer*/
   SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

   /*GPIO Configuration*/
   GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
   GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;

   /*TB Module Configuration*/
   EPwm1Regs.TBCTL.bit.CLKDIV = 0;
   EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
   EPwm1Regs.TBCTL.bit.CTRMODE = 0;
   EPwm1Regs.TBCTR = 0;
   EPwm1Regs.TBPRD = 5999;

   /*CC Module Configuration*/
   EPwm1Regs.CMPA.half.CMPA = 2000;
   EPwm1Regs.CMPB = 1000;
   EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
   EPwm1Regs.CMPCTL.bit.SHDWBMODE = 0;
   EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;
   EPwm1Regs.CMPCTL.bit.LOADBMODE = 0;

   /*AC Module Configuration */
   EPwm1Regs.AQCTLA.bit.CAU = 1;
```

```
    EPwm1Regs.AQCTLA.bit.ZRO = 2;
    EPwm1Regs.AQCTLB.bit.CBU = 1;
    EPwm1Regs.AQCTLB.bit.ZRO = 2;

    /*DB Module Confiruation */
    EPwm1Regs.DBCTL.bit.OUT_MODE = 0;

    /*ET Module Configuration*/
    EPwm1Regs.ETSEL.bit.INTSEL = 1;
    EPwm1Regs.ETSEL.bit.INTEN = 1;
    EPwm1Regs.ETPS.bit.INTPRD = 1;
    EPwm1Regs.ETCLR.bit.INT = 1;

    /*PIE Interrupt/CPU Interrupt Configuration*/
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    IER |= M_INT3;

    /*Start Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    EDIS;

}

interrupt void Epwm1Isr(void){
    static int i = 0;

    if(i==0){
        EPwm1Regs.CMPA.half.CMPA = 2000;
        EPwm1Regs.CMPB = 1000;
        i++;
    }else{
        EPwm1Regs.CMPA.half.CMPA = 3000;
        EPwm1Regs.CMPB = 2000;
        i = 0;
    }

    EPwm1Regs.ETCLR.bit.INT = 1;
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

それでは、コードを解説していきます。まず、ePWMの初期設定の関数ePWM1Config()から解説します。

Main.cよりePWM1Config()関数

```
void ePWM1Config(void){

    EALLOW;
    /*Stop PWM Timer*/
```

```
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

    /*GPIO Configuration*/
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;

    /*TB Module Configuration*/
    EPwm1Regs.TBCTL.bit.CLKDIV = 0;
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
    EPwm1Regs.TBCTL.bit.CTRMODE = 0;
    EPwm1Regs.TBCTR = 0;
    EPwm1Regs.TBPRD = 5999;

    /*CC Module Configuration*/
    EPwm1Regs.CMPA.half.CMPA = 2000;
    EPwm1Regs.CMPB = 1000;
    EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = 0;
    EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;
    EPwm1Regs.CMPCTL.bit.LOADBMODE = 0;

    /*AC Module Configuration */
    EPwm1Regs.AQCTLA.bit.CAU = 1;
    EPwm1Regs.AQCTLA.bit.ZRO = 2;
    EPwm1Regs.AQCTLB.bit.CBU = 1;
    EPwm1Regs.AQCTLB.bit.ZRO = 2;

    /*DB Module Confiruation */
    EPwm1Regs.DBCTL.bit.OUT_MODE = 0;

    /*ET Module Configuration*/
    EPwm1Regs.ETSEL.bit.INTSEL = 1;
    EPwm1Regs.ETSEL.bit.INTEN = 1;
    EPwm1Regs.ETPS.bit.INTPRD = 1;
    EPwm1Regs.ETCLR.bit.INT = 1;

    /*PIE Interrupt/CPU Interrupt Configuration*/
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    IER |= M_INT3;

    /*Start Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    EDIS;

}
```

EPWMは小さなモジュールが集まって構成されていますので、それぞれのモジュール毎に、設定を見ていきましょう。まず、最初にTBモジュールの設定を考えてみましょう。
TBモジュールで考えなければならいのは、

- **タイマ・クロック(TBCLK)の速度をどうするか？**
- **タイマのカウント・モードはどうするか？**
- **タイマの初期値、周期値はどうするか？**
- **タイマの同期はどうするか？**

まず、タイマ・クロック速度から考えてみましょう。TBCTRのクロックはSYSCLKOUT(=CPU動作周波数)を分周したクロックを使用します。この分周率は、TBCTLのCLKDIV及びHSPCLKDIVで設定でき、SYCLKOUT/(CLKDIV*HSPCLKDIV)になります。表 8にCLKDIVとHSPCLKDIVについて示します。

フィールド名	設定値	意味	フィールド名	設定値	意味
CLKDIV	000b	/1 (デフォルト値)	HSPCLKDIV	000b	/1
	001b	/2		001b	/2(デフォルト値)
	010b	/4		010b	/4
	011b	/8		011b	/6
	100b	/16		100b	/8
	101b	/32		101b	/10
	110b	/64		110b	/12
	111b	/128		111b	/14

表 8: CLKDIV と HSPCLKDIV フィールド(TBCTL レジスタ)の設定(設定値欄の"b"は２進数をあらわす記号)

今回は10KHzのキャリア周波数を使います。SYSCLKOUT(=CPU動作速度)は60MHzですので、60M/10K=6000で、6000カウントを１周期とすると10KHzのPWMが生成される計算になります。タイマは16bit(65536)ありますので、6000カウントは可能です。そのためこの例では分周しなくてもカウントは可能ですので、CLKDIVもHSPCLKDIVも分周無し(/1)設定で使用します。ヘッダ・ファイルを利用すれば、

```
EPwm1Regs.TBCTL.bit.CLKDIV = 0 ;
EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0 ;
```

と記述できます。
タイマのカウント・モードはUpカウント・モードを選択します。カウント・モードは、同じTBCTLレジスタのCTRMODE(表 9)にて設定します。

フィールド名	設定値	意味	設定値	意味
CTRMODE	00b	Upモード	10b	Up-Downモード
	01b	Downモード	11b(デフォルト値)	カウント停止

表 9: CTRMODE フィールド(TBCTL レジスタ)の設定

CTRMODEは00bでUpモードになりますので、

```
EPwm1Regs.TBCTL.bit.CTRMODE = 0 ;
```

と記述できます。
タイマカウンタを担うレジスタはTBCTRというレジスタです。タイマの初期値は単純にこのTBCTRに初期値を書いておきます。初期値は0にしておきます。

```
EPwm1Regs.TBCTR = 0 ;
```

と記述できます。
タイマの周期値は、TBPRDというレジスタに設定します。周期値の計算は先ほどしました。6000カウントです。TBPRD=6000と設定したいところですが、カウンタは0からのスタートになりますので、1引いて5999に設定します。

```
EPwm1Regs.TBPRD = 5999;
```

と記述できます。
今回はEPWM1しか使用しませんので、同期機能は使用しません。同期機能はもう一つの例で掲載していますので、ここでは特に設定はしません。さて、今回の例では、TBモジュールの設定はこのくらいで良いでしょう。

次にCCモジュールの設定を行いましょう。CCモジュールでは、

- **コンペアA、コンペアBの値をいくつにするか？**
- **コンペアA、コンペアBのアクティブ・レジスタへのコピーするタイミングはどうするか？**

を設定します。コンペアAはCMPAレジスタ、コンペアBはCMPBレジスタにて設定します。コンペア値の初期値は、ＰＷＭの初期状態になりますので、本来は目的に応じてきちんと考える必要がありますが、今回は特に目的のアプリケーションがあるわけではありませんので、適当に初期値を設定しておきましょう。タイマカウンタは0～5999の範囲で動作しますので、コンペア値ももちろんこの範囲内にしておきます。

```
EPwm1Regs.CMPA.half.CMPA = 2000 ;
EPwm1Regs.CMPB = 1000;
```

ここで、CMPAについては注意が必要です。CMPAはHRPWM対応のコンペア・レジスタとなっているため、ヘッダ・ファイルでは、CMPAとCMPAHRを合わせて32bitのレジスタとして定義しています。そのため、CMPAのみを変更したい場合は、先程のような記述をする必要があります。
EPwm1Regs.CMPA = 1000；
としても、**正しく設定されません**ので注意して下さい。
次に、シャドウ・レジスタを使用するかどうかを設定できますので、その設定をしておきます。それが、CMPCTL(Compare Control)レジスタのSHDWAMODE(CMPA用)、 SHDWBMODE(CMPB用)です(表 10)。

フィールド名	設定値	意味	フィールド名	設定値	意味
SHDWAMODE	0	シャドウ使用	SHDWBMODE	0	シャドウ使用
	1	シャドウ未使用		1	シャドウ未使用

表 10: SHDWAMODE,SHDWBMODE フィールド(CMPCTL レジスタ)の設定

今回は両方とも、シャドウ・レジスタを使用しましょう。

```
EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm1Regs.CMPCTL.bit.SHDWBMODE = 0;
```

となります。次にどのタイミングでこのシャドウ・レジスタからアクティブ・レジスタにロード(コピー)するかですが、同レジスタのLOADAMODE(CMPA用), LOADBMODE(CMPB用)にて設定できます(表 11)。

フィールド名	設定値	意味	フィールド名	設定値	意味
LOADAMODE	00b	カウンタ0	LOADBMODE	00b	カウンタ0
	01b	カウンタ周期値		01b	カウンタ周期値
	10b	カウンタ0又はカウ		10b	カウンタ0又はカウ

		ンタ周期値			ンタ周期値
	11b	ロード無し		11b	ロード無し

表 11: LOADAMODE, LOADBMODE フィールド(CMPCTL レジスタ)の設定

今回は、カウンタ0を選択しましょう。このように設定する事で、カウンタが0（つまり次の周期の最初）に達する前にコンペア値を変更しておけば、カウンタが0になった時に自動的にアクティブ・レジスタにコピーされます(つまり次の周期に反映されます)。

```
EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;
EPwm1Regs.CMPCTL.bit.LOADBMODE = 0;
```

さて、CCモジュールの設定はこれくらいで良いでしょう。

次にPWM波形の基本波形を設定するAQモジュールを設定しましょう。AQモジュールはAQOUTA出力、AQOUTB出力という中間出力に対して、それぞれ、どのイベント時に信号を変化させるかを設定します。これを設定するのが、AQOUTA用がAQCTLA(AQ Output A Control)、AQOUTB用がAQCTLB(AQ Output B Control)レジスタです。AQCTLA及びAQCTLBは全く同じ構造のレジスタになっていて、それぞれ、

- CBD :コンペアBマッチ(カウンタ・ダウン)時の出力設定
- CBU :コンペアBマッチ(カウンタ・アップ)時の出力設定
- CAD :コンペアAマッチ(カウンタ・ダウン)時の出力設定
- CAU :コンペアAマッチ(カウンタ・アップ)時の出力設定
- PRD : カウンタ＝周期値の時の出力設定
- ZRO : カウンタ=0の時の出力設定

を設定できます。それぞれの設定は表 12のようになります。

フィールド名	設定値	意味	設定値	意味
CBD,CBU,CAD, CAU,PRD,ZRO	00b	何もしない(デフォルト値)	10b	セット(High出力)
	01b	クリア(Low出力)	11b	トグル(High→Low, Low→High)

表 12: 各 AQCTLA, AQCTLB の CBD,CBU,CAD,CAU,PRD,ZRO フィールドの設定

さて、ここでもう一度、図 79を見てください。今回のPWM出力は、コンペアAでEPWM1A出力のDutyを、コンペアBでEPWM1BのDutyを設定します。そのため、AQOUTA出力では、ZROの時に出力をHighにし、CAUの時にLowに設定します。一方AQOUTB出力では、ZROの時に出力をHighにし、CBUの時にLowに設定します(PRDとZROのタイミングは１クロックずれています。今回は特にこのあたりを気にしませんので、Lowにするタイミングが PRDでもZROでもどちらでもかまいませんが、アプリケーションによっては重要になる場合も有り得ますので、十分考慮して下さい)。これらの設定をまとめますと、表 13: AQCTLAとAQCTLBの設定
のようになります。

レジスタ	フィールド名	今回の設定	レジスタ	フィールド名	今回の設定
AQCTLA	CBD	00b(何もしない)	AQCTLB	CBD	00b(何もしない)
	CBU	00b(何もしない)		CBU	01b(クリア)
	CAD	00b(何もしない)		CAD	00b(何もしない)
	CAU	01b(クリア)		CAU	00b(何もしない)
	PRD	00b(何もしない)		PRD	00b(何もしない)
	ZRO	10b(セット)		ZRO	10b(セット)

表 13: AQCTLA と AQCTLB の設定

00bはデフォルト値になりますので、設定を省き、(製品に向けての開発であれば、デフォルト値の部分もきちんと設定する事をお勧めします。)、以下のように設定します。

```
EPwm1Regs.AQCTLA.bit.CAU = 1;
EPwm1Regs.AQCTLA.bit.ZRO = 2;
EPwm1Regs.AQCTLB.bit.CBU = 1 ;
EPwm1Regs.AQCTLB.bit.ZRO = 2 ;
```

AQモジュールには、他にS/Wトリガの設定などができますが、今回はS/Wトリガは使用しませんので、設定はこの辺で良いでしょう。

次にDBモジュールの設定をしましょう。DBモジュールでは、主に以下のような事項について設定します。

- **どのようにディレイを付加するか？**
- **立下りディレイ及び立ち上がりディレイは何クロックにするか？**
- **反転（極性）はどうするか？**
- **どの信号を出力するか**

今回の例では、デット・バンドを付加したり、出力を反転する必要はなく、OUTA/B出力をそのまま出力(バイパス)すればOKです。図で表すと、図 80のようになります。

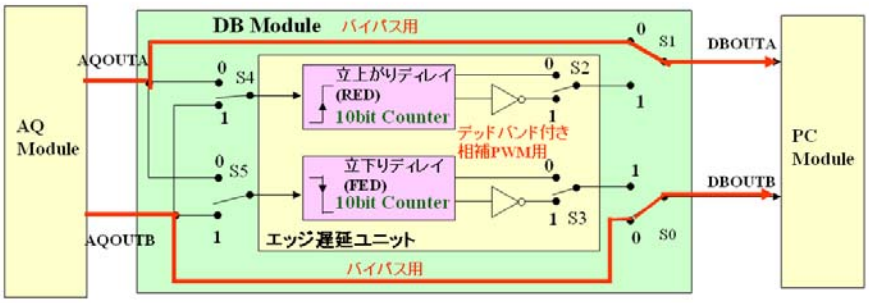


図 80:DB の設定

図のスイッチS0, S1, S2, S3, S4, S5に相当するのが、DBCTL(DB Generator Control)レジスタのPOLSEL, OUTMODE, INMODEになります。これらは2bitのフィールドになり、
図のスイッチで考えますと、
POLSEL[1bit :0bit] = [S3 : S2]
OUTMODE[1bit :0bit] = [S1 : S0]
INMODE[1bit:0bit]=[S5:S4]
に相当します。今回は、S2, S3,S4,S5はDon't careで、S1=0, S0=0に設定する必要がありますので、

```
EPwm1Regs.DBCTL.bit.OUT_MODE = 0;
```

に設定します。今回はディレイは付加しませんので、設定はこれだけでよいでしょう。今回の例では、この出力をそのままピンに出力すればよいですので、CPモジュール、TZモジュール、HRPWMモジュールも使用しません。これらの設定はこの例ではする必要はありません。

さて、毎周期同じDutyのPWMを出力させても何もおもしろくもありませんので、毎周期毎にDutyを変更する事を考えて見ましょう。毎周期変更するという事は、毎周期割り込みを発生させ、その割り込みの中でDutyを変更すれば良いことになります。割り込みの設定を行うのはETモジュールになります。

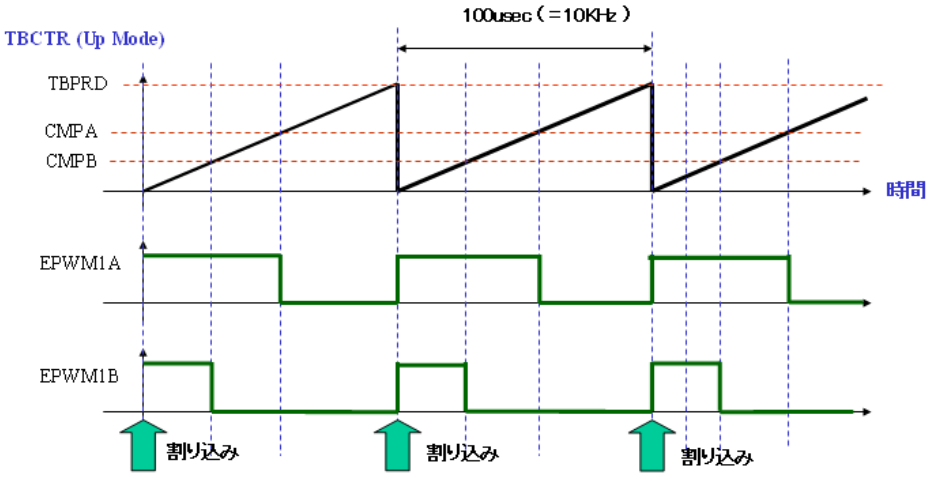


図 81:割り込み発生位置

CPU割り込みは図 81のように、タイマカウンタ0の時に毎回発生させる事にしましょう。割り込みのタイミングを設定するのは、ETSELレジスタのINTSELです。また、同レジスタのINTENにてこの割り込みのペリフェラル・レベルの許可/不許可の設定ができます。

フィールド名	設定値	意味	設定値	意味
INTSEL	000b	予約	100b	コンペアAマッチ(アップカウント)
	001b	カウンタ=0	101b	コンペアAマッチ(ダウンカウント)
	010b	カウンタ=周期値	110b	コンペアBマッチ(アップカウント)
	011b	カウンタ=0、又は、カウンタ=周期値	111b	コンペアBマッチ(ダウンカウント)
INTEN	0	割り込み不許可	1	割り込み許可

表 14: INTSEL フィールドと INTEN フィールド(ETSEL レジスタ)の設定

今回はカウンタ=0の時に割り込みを発生させたいですので、

```
EPwm1Regs.ETSEL.bit.INTSEL = 1；
EPwm1Regs.ETSEL.bit.INTEN = 1；
```

と設定します。また、ETモジュールでは、設定した条件が何回発生したら、割り込みを発生させるかを設定できます。これを設定するのが、ETPSレジスタのINTPRD(表 15)です。

フィールド名	設定値	意味	設定値	意味
INTPRD	00b	ディセーブル	10b	2回で割り込み
	01b	1回で割り込み	11b	3回で割り込み

表 15: INTPRD フィールド(ETPS レジスタ)の設定

今回の例では、毎回割り込みを発生させたいですので、

```
EPwm1Regs.ETPS.bit.INTPRD = 1；
```

と設定できます。

必要になる場合とならない場合があるのですが、念のため、EPWM1のペリフェラル・レベル割り込みフラグをクリアしておきましょう。EPWM1の割り込みをクリアするためには、ETCLRレジスタのINTフィールド(表 16)をセットする必要があります。

フィールド名	設定値	意味	設定値	意味
INT	0	何もしない（影響なし）	1	割り込みフラグをクリア

表 16:INT フィールド(ETCLR レジスタ)の設定

```
EPwm1Regs.ETCLR.bit.INT = 1;
```

と記述できます。

さて、ここでイネーブルにした割り込みはあくまでもペリフェラル・レベルの割り込みです。既に解説しましたが、C28xにはペリフェラル・レベル、PIEレベル、CPUレベルの３つの割り込みレベルがあり、これら全てを正しく設定しなければなりません。EPWM1割り込みは、PIEレベルでは、EPWM1_INTという名前で定義されています。図 82はF28035のPIE割り込みの割り当て表です。この表は各デバイスのデータシートに掲載されています。

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1.y	WAKEINT (LPM/WD) 0xD4E	TINT0 (TIMER 0) 0xD4C	ADCINT9 (ADC) 0xD4A	XINT2 Ext. int. 2 0xD48	XINT1 Ext. int. 1 0xD46	Reserved – 0xD44	ADCINT2 (ADC) 0xD42	ADCINT1 (ADC) 0xD40
INT2.y	Reserved – 0xD5E	EPWM7_TZINT (ePWM7) 0xD5C	EPWM6_TZINT (ePWM6) 0xD5A	EPWM5_TZINT (ePWM5) 0xD58	EPWM4_TZINT (ePWM4) 0xD56	EPWM3_TZINT (ePWM3) 0xD54	EPWM2_TZINT (ePWM2) 0xD52	EPWM1_TZINT (ePWM1) 0xD50
INT3.y	Reserved – 0xD6E	EPWM7_INT (ePWM7) 0xD6C	EPWM6_INT (ePWM6) 0xD6A	EPWM5_INT (ePWM5) 0xD68	EPWM4_INT (ePWM4) 0xD66	EPWM3_INT (ePWM3) 0xD64	EPWM2_INT (ePWM2) 0xD62	EPWM1_INT (ePWM1) 0xD60
INT4.y	Reserved – 0xD7E	Reserved – 0xD7C	Reserved – 0xD7A	Reserved – 0xD78	Reserved – 0xD76	Reserved – 0xD74	Reserved – 0xD72	ECAP1_INT (eCAP1) 0xD70
INT5.y	Reserved – 0xD8E	Reserved – 0xD8C	Reserved – 0xD8A	Reserved – 0xD88	Reserved – 0xD86	Reserved – 0xD84	Reserved – 0xD82	EQEP1_INT (eQEP1) 0xD80
INT6.y	Reserved – 0xD9E	Reserved – 0xD9C	Reserved – 0xD9A	Reserved – 0xD98	SPITXINTB (SPI-B) 0xD96	SPIRXINTB (SPI-B) 0xD94	SPITXINTA (SPI-A) 0xD92	SPIRXINTA (SPI-A) 0xD90
INT7.y	Reserved – 0xDAE	Reserved – 0xDAC	Reserved – 0xDAA	Reserved – 0xDA8	Reserved – 0xDA6	Reserved – 0xDA4	Reserved – 0xDA2	Reserved – 0xDA0
INT8.y	Reserved – 0xDBE	Reserved – 0xDBC	Reserved – 0xDBA	Reserved – 0xDB8	Reserved – 0xDB6	Reserved – 0xDB4	I2CINT2A (I2C-A) 0xDB2	I2CINT1A (I2C-A) 0xDB0
INT9.y	Reserved – 0xDCE	Reserved – 0xDCC	ECAN1_INTA (CAN-A) 0xDCA	ECAN0_INTA (CAN-A) 0xDC8	LIN1_INTA (LIN-A) 0xDC6	LIN0_INTA (LIN-A) 0xDC4	SCITXINTA (SCI-A) 0xDC2	SCIRXINTA (SCI-A) 0xDC0
INT10.y	ADCINT8 (ADC) 0xDDE	ADCINT7 (ADC) 0xDDC	ADCINT6 (ADC) 0xDDA	ADCINT5 (ADC) 0xDD8	ADCINT4 (ADC) 0xDD6	ADCINT3 (ADC) 0xDD4	ADCINT2 (ADC) 0xDD2	ADCINT1 (ADC) 0xDD0
INT11.y	CLA1_INT8 (CLA) 0xDEE	CLA1_INT7 (CLA) 0xDEC	CLA1_INT6 (CLA) 0xDEA	CLA1_INT5 (CLA) 0xDE8	CLA1_INT4 (CLA) 0xDE6	CLA1_INT3 (CLA) 0xDE4	CLA1_INT2 (CLA) 0xDE2	CLA1_INT1 (CLA) 0xDE0
INT12.y	LUF (CLA) 0xDFE	LVF (CLA) 0xDFC	Reserved – 0xDFA	Reserved – 0xDF8	Reserved – 0xDF6	Reserved – 0xDF4	Reserved – 0xDF2	XINT3 Ext. Int. 3 0xDF0

図 82:PIE 割り込み一覧における EPWM1_INT

表より、EPWM1_INTはINT3.1に割り当てられています。F2802x/F2806xもEPWM1_INTはINT3.1に割り当てられています。一度データシートを確認してみてください。これをイネーブルにするには、

```
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
```

と記述できます。さらに、CPUレベルの割り込みをイネーブルにしましょう。PIEでINT3.1に割り当てられているという事は、CPUレベルではINT3に割り当てられています。これをイネーブルにするためには、

```
IER |= M_INT3 ;
```

です。割り込みをイネーブルにするためには、これに付け加えてINTMをイネーブルにしなければいけませんが、これはグローバル割り込みになりますので、ここで設定せずに、別のところで設定しましょう。さて、これでEPWM1の設定は終了です。EPWM１の設定自体はこれで終了ですが、設定する前にタイマを止める事と、設定し終わった後にタイマを開始する必要があります。EPWMのタイマのカウント動作停止/開始はPCLKCR0レジスタのTBCLKSYNCで制御します(このレジスタは、ePWMのマニュアルではなく、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]*のSystem Control and Interrupts章

F2803x用	*TMS320F2803x Piccolo System Control and Interrupts[SPRUGL8]*
F2802x用	*TMS320F2802x/TMS320F2802xx Piccolo System Control and Interrupts[SPRUFN3]*

に掲載されています)。このフィールドの詳細を表 17に示します。

フィールド名	設定値	意味	設定値	意味
TBCLKSYNC	0	TBCTRのカウント停止(デフォルト値)	1	TBCTRのカウント開始

表 17:TBCLKSYNC フィールド(PCLKCR0 レジスタ)

このTBCLKSYNCをクリア/セットする事で、全EPWMモジュールのTBCTRがカウント停止/開始します。これにより、ePWMを設定中にPWM出力が動作するのを防いだり、クロック供給開始から各TBモジュールのタイマが同期する事ができます。スタートする場合は、

```
EALLOW ;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
EDIS ;
```

止める場合は、

```
EALLOW ;
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0 ;
EDIS ;
```

と記述します。このレジスタはEALLOW保護されているレジスタですのでEALLOW/EDISが必要になります。EPWMもレジスタによっては、EALLOW保護がかかっていますので、今回の例では、最初にePWM設定前にEALLOWを実行し、設定が終わったらEDISを実行しています。

また、EPWM1AとEPWM1BからPWM波形をピンに出力しますので、GPIOの設定もしておかなければなりません。EPWM1AとEPWM1Bはそれぞれ、GPIO0とGPIO1に配置されています。GPIO0及び1ピンを何のピンとして使うかを設定するのは、GPAMUX1レジスタのGPIO0及びGPIO1フィールドです(表 18)。

フィールド名	設定値	意味	設定値	意味
GPIO0	00b	GPIO0	10b	データシートを参照下さい
	01b	EPWM1A	11b	データシートを参照下さい
GPIO1	00b	GPIO1	10b	データシートを参照下さい
	01b	EPWM1B	11b	データシートを参照下さい

表 18: GPIO0 及び GPIO1 フィールド(GPAMUX1 レジスタ)の設定

それぞれEPWM1A及びEPWM1Bとして使用しますので、

```
EALLOW;
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;
EDIS;
```

と設定します（コード中はEALLOW/EDISは既に実行されていますので、省略しています）。GPAMUX1レジスタはEALLOW保護されていますので注意して下さい。

さて、これでEPWMの設定は終了しましたので、次にEPWN1_INTが発生した時のISR、Epwom1Isr()を見ていきましょう。

Main.cからEpwm1Isr()

```
interrupt void Epwm1Isr(void){
   static int i = 0;

   if(i==0){
      EPwm1Regs.CMPA.half.CMPA = 2000;
      EPwm1Regs.CMPB = 1000;
      i++;
   }else{
      EPwm1Regs.CMPA.half.CMPA = 3000;
      EPwm1Regs.CMPB = 2000;
      i = 0;
   }

   EPwm1Regs.ETCLR.bit.INT = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

このISRでは、CMPAとCMPBの変更を行っています。CMPAとCMPBに対して2000, 1000と3000, 2000が交互にロードされます。周期が6000(0～5999)ですので、結果として、EPWM1AとEPWM1B出力には、33%,16.7%→50%,33%のDutyが交互に出力されるはずです。結果は後で見てみましょう。この関数はISRですので、Cコンパイラに対して、このコードがISRである事を知らせるために、interruptというキーワードが必要になります。これは忘れないで下さい。また、ペリフェラル・レベルの割り込みフラグ(今回はEPWM1のETFLGレジスタのINTビットです)は自動的にクリアされませんので、ユーザーがクリアする必要があります(EPwm1Regs.ETCLR.bit.INT = 1;にてこのETFLGのINTビットをクリアしています)。また、PIEのグループ３に対してPIEACKをクリアする事も忘れないで下さい。ここでは、

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
```

にて、PIEグループ3のPIEACKをクリアしています。

さて、後は残ったmain()関数を見てみましょう。

mainコード (main.c)

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");
interrupt void Epwm1Isr(void);
void ePWM1Config(void);

void main(void){

   DINT;
   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
   InitFlash();
```

```
    InitPieCtrl();
    IER=0x0000;
    IFR=0x0000;
    InitPieVectTable();
    EALLOW;
    PieVectTable.EPWM1_INT  = Epwm1Isr;
    EDIS;

    ePWM1Config();
    EINT;

    while(1);
}
```

基本的には、新しい事はありませんので、それぞれの意味を是非考えてみてください。
それでは、虫マークにてデバッグを開始して、実行してみましょう。

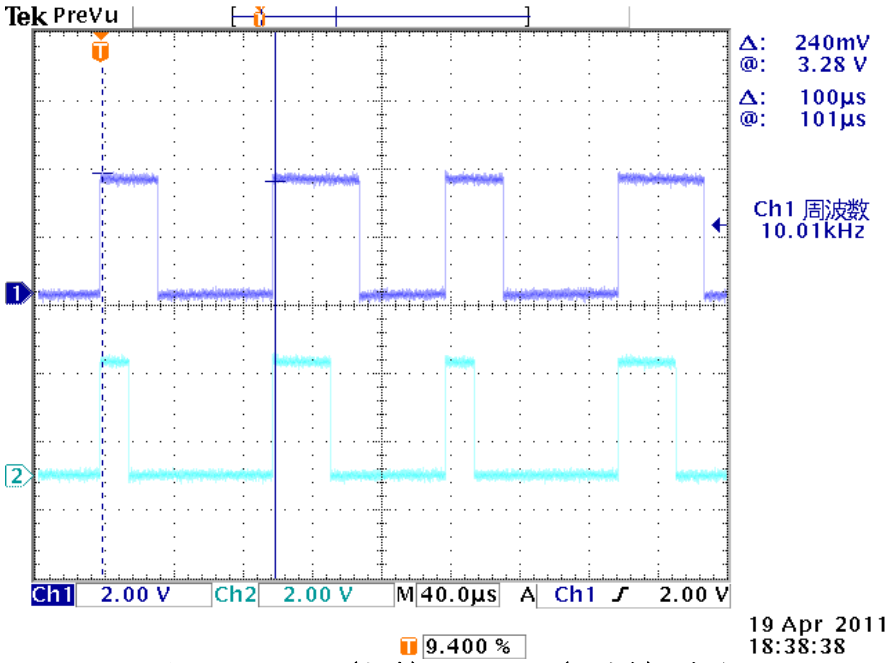


図 83: EPWM1A(上:青)と EWPM1B(下:水色)の波形

図 83は実際にこのコードを動作させた時のEPWM1AとEPWM1Bの波形をオシロスコープでとった図です。EPWM1A(上)がプログラム通り、50%,33%のDutyを繰り返している事がわかると思います。またEPWM1B(下)も、プログラム通り、33%, 16.7%のDutyを繰り返している事がわかります。そして、ＰWM周波数も10KHzになっています。

7.3 6ch のデッドバンド付相補 PWM 出力

次に三相インバータの制御などで最も一般的な6chのデッドバンド付相補PWM出力例を示します。

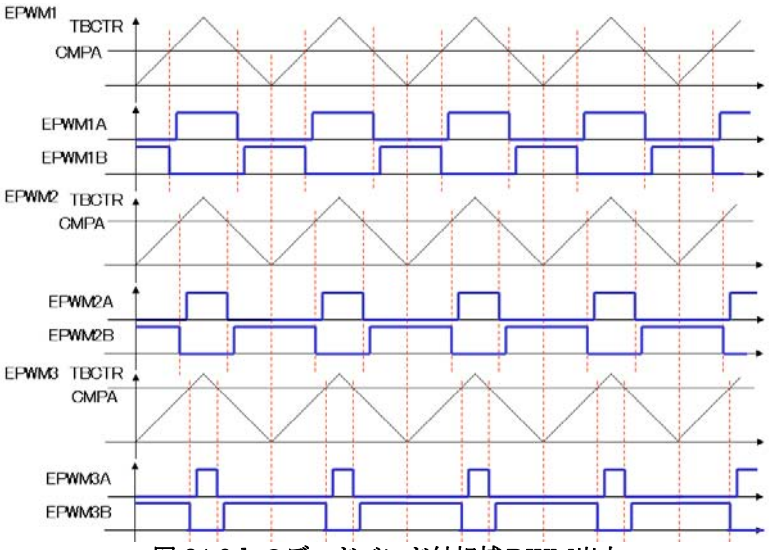


図 84:6ch のデッドバンド付相補ＰWM出力

今回の目標とするPWM波形を図 84に示します。ＰWMキャリア周波数は5KHzとしておきます。3相6chのPWMになりますので、３本のEPWMモジュール(EPWM1/2/3)を用います。出力ピンとしては、EPWM1A/B, EPWM2A/B, EPWM3A/Bを使用します。EPWM1/2/3のタイマは全て同じタイミングで同期します。それでは、Projectを作成していきます。前節のProjectと構成はほとんど同じですので、ディレクトリをコピーして、それを改造してProjectを作成しましょう。コピーして、CCSにImportしたら、適当にProject名を変更してください。その他にはProjectに設定に変更はありません。今回は、基本的には、main.cを書き換えるだけです。

それでは、今回のmain.cは、以下のような記述にして下さい。

Main.c

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");
interrupt void Epwm1Isr(void);
void ePWM123Config(void);

void main(void){

   DINT;
   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();

   InitPieCtrl();
   IER=0x0000;
```

```
    IFR=0x0000;
    InitPieVectTable();
    EALLOW;
    PieVectTable.EPWM1_INT  = Epwm1Isr;
    EDIS;

    ePWM123Config();
    EINT;

    while(1);
}

void ePWM123Config(void){

    EALLOW;

    /*Stop PWM Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

    /*GPIO Configuration*/
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 1;

    /*TB Module Configuration*/
    EPwm1Regs.TBCTL.bit.CTRMODE = 2;    // Up-Down Mode
    EPwm2Regs.TBCTL.bit.CTRMODE = 2;
    EPwm3Regs.TBCTL.bit.CTRMODE = 2;

    EPwm1Regs.TBCTL.bit.PHSEN = 0;        // Disable Phase Sync
    EPwm2Regs.TBCTL.bit.PHSEN = 1;        // Enable Phase Sync
    EPwm3Regs.TBCTL.bit.PHSEN = 1;        // Enable Phase Sync

    EPwm1Regs.TBCTL.bit.SYNCOSEL = 1;   // SyncOut timing --> TBCTR = 0
    EPwm2Regs.TBCTL.bit.SYNCOSEL = 0;   // SyncOut timing == SyncIn timing
    EPwm3Regs.TBCTL.bit.SYNCOSEL = 3;   // Disable SyncOut

    EPwm2Regs.TBPHS.half.TBPHS = 0;
    EPwm3Regs.TBPHS.half.TBPHS = 0;

    EPwm1Regs.TBCTL.bit.CLKDIV = 0;
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
    EPwm2Regs.TBCTL.bit.CLKDIV = 0;
    EPwm2Regs.TBCTL.bit.HSPCLKDIV = 0;
    EPwm3Regs.TBCTL.bit.CLKDIV = 0;
    EPwm3Regs.TBCTL.bit.HSPCLKDIV = 0;
```

```
EPwm1Regs.TBPRD = 6000;
EPwm2Regs.TBPRD = 6000;
EPwm3Regs.TBPRD = 6000;

EPwm1Regs.TBCTR = 0;
EPwm2Regs.TBCTR = 2000;
EPwm3Regs.TBCTR = 4000;

/*CC Module Configuration*/
EPwm1Regs.CMPA.half.CMPA = 1000;
EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm1Regs.CMPCTL.bit.LOADAMODE = 1;

EPwm2Regs.CMPA.half.CMPA = 2000;
EPwm2Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm2Regs.CMPCTL.bit.LOADAMODE = 1;

EPwm3Regs.CMPA.half.CMPA = 3000;
EPwm3Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm3Regs.CMPCTL.bit.LOADAMODE = 1;

/*AC Module Configuration */
EPwm1Regs.AQCTLA.bit.CAD = 1;
EPwm1Regs.AQCTLA.bit.CAU = 2;

EPwm2Regs.AQCTLA.bit.CAD = 1;
EPwm2Regs.AQCTLA.bit.CAU = 2;

EPwm3Regs.AQCTLA.bit.CAD = 1;
EPwm3Regs.AQCTLA.bit.CAU = 2;

/*DB Module Confiruation */
EPwm1Regs.DBCTL.bit.POLSEL = 2;
EPwm1Regs.DBCTL.bit.OUT_MODE = 3;
EPwm1Regs.DBCTL.bit.IN_MODE = 0;
EPwm1Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm2Regs.DBCTL.bit.POLSEL = 2;
EPwm2Regs.DBCTL.bit.OUT_MODE = 3;
EPwm2Regs.DBCTL.bit.IN_MODE = 0;
EPwm2Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm3Regs.DBCTL.bit.POLSEL = 2;
EPwm3Regs.DBCTL.bit.OUT_MODE = 3;
EPwm3Regs.DBCTL.bit.IN_MODE = 0;
EPwm3Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm1Regs.DBRED = 120;
EPwm1Regs.DBFED = 240;
```

```
    EPwm2Regs.DBRED = 120;
    EPwm2Regs.DBFED = 240;

    EPwm3Regs.DBRED = 120;
    EPwm3Regs.DBFED = 240;

  /*ET Module Configuration*/
  EPwm1Regs.ETSEL.bit.INTSEL = 1;
  EPwm1Regs.ETSEL.bit.INTEN = 1;
  EPwm1Regs.ETPS.bit.INTPRD = 1;
  EPwm1Regs.ETCLR.bit.INT = 1;

  /*PIE Interrupt/CPU Interrupt Configuration*/
  PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
  IER |= M_INT3;

  /*Start Timer*/
  SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

  EDIS;

}

interrupt void Epwm1Isr(void){

  static int i = 0;

  if(i==0){
    EPwm1Regs.CMPA.half.CMPA = 1000;
    EPwm2Regs.CMPA.half.CMPA = 2000;
    EPwm3Regs.CMPA.half.CMPA = 3000;
    i++;
  }else if(i==1){
    EPwm1Regs.CMPA.half.CMPA = 2000;
    EPwm2Regs.CMPA.half.CMPA = 3000;
    EPwm3Regs.CMPA.half.CMPA = 4000;
    i++;
  }else{
    EPwm1Regs.CMPA.half.CMPA = 3000;
    EPwm2Regs.CMPA.half.CMPA = 4000;
    EPwm3Regs.CMPA.half.CMPA = 5000;
    i = 0;
  }

  EPwm1Regs.ETCLR.bit.INT = 1;
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

それでは、EPWM1/2/3の設定をまず解説しましょう。ePWM123Config()関数から見ていきます。

main.cよりePWM123Config()関数

```
void ePWM123Config(void){

  EALLOW;

  /*Stop PWM Timer*/
  SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

  /*GPIO Configuration*/
  GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
  GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;
  GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 1;
  GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 1;
  GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 1;
  GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 1;

  /*TB Module Configuration*/
  EPwm1Regs.TBCTL.bit.CTRMODE = 2;    // Up-Down Mode
  EPwm2Regs.TBCTL.bit.CTRMODE = 2;
  EPwm3Regs.TBCTL.bit.CTRMODE = 2;

  EPwm1Regs.TBCTL.bit.PHSEN = 0;        // Disable Phase Sync
  EPwm2Regs.TBCTL.bit.PHSEN = 1;        // Enable Phase Sync
  EPwm3Regs.TBCTL.bit.PHSEN = 1;        // Enable Phase Sync

  EPwm1Regs.TBCTL.bit.SYNCOSEL = 1;   // SyncOut timing --> TBCTR = 0
  EPwm2Regs.TBCTL.bit.SYNCOSEL = 0;   // SyncOut timing == SyncIn timing
  EPwm3Regs.TBCTL.bit.SYNCOSEL = 3;   // Disable SyncOut

  EPwm2Regs.TBPHS.half.TBPHS = 0;
  EPwm3Regs.TBPHS.half.TBPHS = 0;

  EPwm1Regs.TBCTL.bit.CLKDIV = 0;
  EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
  EPwm2Regs.TBCTL.bit.CLKDIV = 0;
  EPwm2Regs.TBCTL.bit.HSPCLKDIV = 0;
  EPwm3Regs.TBCTL.bit.CLKDIV = 0;
  EPwm3Regs.TBCTL.bit.HSPCLKDIV = 0;

  EPwm1Regs.TBPRD = 6000;
  EPwm2Regs.TBPRD = 6000;
  EPwm3Regs.TBPRD = 6000;

  EPwm1Regs.TBCTR = 0;
  EPwm2Regs.TBCTR = 2000;
  EPwm3Regs.TBCTR = 4000;

  /*CC Module Configuration*/
```

```
EPwm1Regs.CMPA.half.CMPA = 1000;
EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm1Regs.CMPCTL.bit.LOADAMODE = 1;

EPwm2Regs.CMPA.half.CMPA = 2000;
EPwm2Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm2Regs.CMPCTL.bit.LOADAMODE = 1;

EPwm3Regs.CMPA.half.CMPA = 3000;
EPwm3Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm3Regs.CMPCTL.bit.LOADAMODE = 1;

/*AC Module Configuration */
EPwm1Regs.AQCTLA.bit.CAD = 1;
EPwm1Regs.AQCTLA.bit.CAU = 2;

EPwm2Regs.AQCTLA.bit.CAD = 1;
EPwm2Regs.AQCTLA.bit.CAU = 2;

EPwm3Regs.AQCTLA.bit.CAD = 1;
EPwm3Regs.AQCTLA.bit.CAU = 2;

/*DB Module Confiruation */
EPwm1Regs.DBCTL.bit.POLSEL = 2;
EPwm1Regs.DBCTL.bit.OUT_MODE = 3;
EPwm1Regs.DBCTL.bit.IN_MODE = 0;
EPwm1Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm2Regs.DBCTL.bit.POLSEL = 2;
EPwm2Regs.DBCTL.bit.OUT_MODE = 3;
EPwm2Regs.DBCTL.bit.IN_MODE = 0;
EPwm2Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm3Regs.DBCTL.bit.POLSEL = 2;
EPwm3Regs.DBCTL.bit.OUT_MODE = 3;
EPwm3Regs.DBCTL.bit.IN_MODE = 0;
EPwm3Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm1Regs.DBRED = 120;
EPwm1Regs.DBFED = 240;

EPwm2Regs.DBRED = 120;
EPwm2Regs.DBFED = 240;

EPwm3Regs.DBRED = 120;
EPwm3Regs.DBFED = 240;

/*ET Module Configuration*/
EPwm1Regs.ETSEL.bit.INTSEL = 1;
EPwm1Regs.ETSEL.bit.INTEN = 1;
```

```
    EPwm1Regs.ETPS.bit.INTPRD = 1;
    EPwm1Regs.ETCLR.bit.INT = 1;

    /*PIE Interrupt/CPU Interrupt Configuration*/
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    IER |= M_INT3;

    /*Start Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    EDIS;

}
```

まず、

SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

にて、EPWMのクロックを停止します。これは前節のサンプル・コードと同じです。

次にGPIOの設定を次のコードで行っています。

```
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 1;
```

フィールド名	設定値	意味	設定値	意味
GPIO0	00b	GPIO0	10b	データシートを参照下さい
	01b	EPWM1A	11b	データシートを参照下さい
GPIO1	00b	GPIO1	10b	データシートを参照下さい
	01b	EPWM1B	11b	データシートを参照下さい
GPIO2	00b	GPIO2	10b	データシートを参照下さい
	01b	EPWM2A	11b	データシートを参照下さい
GPIO3	00b	GPIO3	10b	データシートを参照下さい
	01b	EPWM2B	11b	データシートを参照下さい
GPIO4	00b	GPIO4	10b	データシートを参照下さい
	01b	EPWM3A	11b	データシートを参照下さい
GPIO5	00b	GPIO5	10b	データシートを参照下さい
	01b	EPWM3B	11b	データシートを参照下さい

表 19:GPIO0～5(EPWM1x～3x の MUX)

表 19に、GPIO0～5(EPWM1x～3x)のGPAMUX1レジスタ設定を示します。前回はGPIO0/1(EPWM1A/1B)を設定しましたが、同様にGPIO2～5もそれぞれのビットを設定して、EPWMxA/B出力として設定しています。

次に、ePWMレジスタの設定をTBモジュールからはじめましょう。前回と異なる点は、

- タイマのカウントモードがUp-Downモード
- 各タイマの同期が必要

の２点です。まず、タイマのカウント・モードを設定するのは、TBCTLレジスタのCTRMODEです。EPWM1/2/3全てのタイマをUp-Downモードにするには、表 9より、

```
EPwm1Regs.TBCTL.bit.CTRMODE = 2;
EPwm2Regs.TBCTL.bit.CTRMODE = 2;
EPwm3Regs.TBCTL.bit.CTRMODE = 2;
```

となります。次にEPWM1/2/3の各タイマ間の同期をとらなければなりません。この３つのタイマの同期をとるためには、EPWM1のタイマをマスタとして扱い、EPWM1からあるタイミングで同期信号をEPWM2とEPWM3に送れば実現できます。EPWM1の同期信号出力はEPWM2の同期信号入力につながっていますし、EPWM2の同期信号出力はEPWM3の同期信号入力に接続されています。ユーザーが設定しなければならないのは、以下の２点です。

1. 同期信号入力に対して、位相オフセット(TBPHS)値のロードを許可する事
2. 次のEPWMモジュールにどのタイミングで同期信号を出力するか

まず、1.を設定するのが、TBCTLレジスタのPHSEN(表 20)です。

フィールド名	設定値	意味	設定値	意味
PHSEN	0	TBPHSロードの不許可	1	TBPHSロードの許可

表 20:PHSEN フィールド(TBCTL レジスタ)の設定

EPWM1はマスタとなりますので、同期信号を送るだけです。そのためTBPHSのロードは必要ありません。EPWM2はEPWM1から同期信号を受け取りEPWM3はEPWM2から同期信号を受け取りますので、EPWM2とEPWM3はこのTBPHSのロード(同期信号が入った時)を許可します。そのため、

```
EPwm1Regs.TBCTL.bit.PHSEN = 0;
EPwm2Regs.TBCTL.bit.PHSEN = 1
EPwm3Regs.TBCTL.bit.PHSEN = 1;
```

と設定します。
次に2.ですが、これを設定するのが同レジスタのSYNCOSEL(表 21)です。

フィールド名	設定値	意味	設定値	意味
SYNCOSEL	00	同期入力をそのまま出力	10b	コンペアBマッチ
	01b	カウンタ0	11b	同期出力を不許可

表 21:SYNCOSEL フィールド(TBCTL レジスタ)の設定

今回は、EPWM1のカウンタ0のタイミングで全ての同期をとる事にしましょう。そのため、EPWM1のSYNCOSELの設定は、

```
EPwm1Regs.TBCTL.bit.SYNCOSEL = 1;
```

とします。EPWM2はEPWM1の同期出力を受け取って、そのまま同じ信号をEPWM3に出力しますので、

```
EPwm2Regs.TBCTL.bit.SYNCOSEL = 0;
```

となります。EPWM3は同期信号を受け取るだけです同期信号出力を不許可にしておきます。

```
EPwm3Regs.TBCTL.bit.SYNCOSEL  = 3;
```

これで、EPWM1/2/3の同期信号が接続できました。同期信号が入力された時に、TBPHSの値がタイマ(TBCTR)にロードされますので、TBPHSの設定を行いましょう。EPWM1からはカウンタ０のタイミングで同期信号が生成されますので、それを受け取るEPWM2/3はそのタイミングでタイマ(TBCTR)を0にすれば、３つのタイマが完全に同期することになります（注 :厳密なタイミングの話をしますと、この同期機能には
TBCLK＝SYSCLKOUTの場合：2TBCLK
TBCLK≠SYSCLKOUTの場合 :1TBCLK
の遅れがあります。この例では厳密なタイミングを求めたサンプルではないため、0と設定しています。そのため、実際には2TBCLKだけずれ が生じています。）。そのため、EPWM2/3のTBPHSは0にすれば良い事がわかります。EPWM1はマスタですのでTBPHSの設定は必要ありません(TBPHSはHRPWM用のTBPHSHRと一緒に32bitで定義されています。そのため、.halfが必要になります。)。

```
EPwm2Regs.TBPHS.half.TBPHS = 0;
EPwm3Regs.TBPHS.half.TBPHS = 0;
```

さて、これでタイマ同期の設定が完了しました。後は、前回と同じように、クロックの分周率とカウンタ、周期値の初期値を設定しておきましょう。クロックの分周率は、前回と同じ1分周にして、TBCLK=60MHzで動作させましょう。
次に、周期値を設定します。、キャリア周波数が5KHzとなりますので、
60MHz/5KHz = 12000クロック
を周期とします。今回は、Up-Downカウントモードを使用しますので、周期値はその半分の6000である事に注意してください。また、Up-Downモードですので、1引く必要はありません。ePWM1/2/3全てに同じ設定を行います。

```
EPwm1Regs.TBCTL.bit.CLKDIV = 0 ;
EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0 ;
EPwm2Regs.TBCTL.bit.CLKDIV = 0 ;
EPwm2Regs.TBCTL.bit.HSPCLKDIV = 0 ;
EPwm3Regs.TBCTL.bit.CLKDIV = 0 ;
EPwm3Regs.TBCTL.bit.HSPCLKDIV = 0 ;
EPwm1Regs.TBPRD = 6000 ;
EPwm2Regs.TBPRD = 6000 ;
EPwm3Regs.TBPRD = 6000 ;
```

次にカウンタの初期値ですが、全て0からスタートさせては同期がとれているのが当たり前ですので、同期の機能が正しく働いている事を確認するために、わざと別々の値を設定しておきましょう。

```
EPwm1Regs.TBCTR = 0;
EPwm2Regs.TBCTR = 2000;
EPwm3Regs.TBCTR = 4000;
```

と設定します。同期の機能が正しく働いていなければ、EPWM1/2/3の各PWM出力の位相がずれたままで出力されます。これが全く同じ位相でPWMが出力されていれば、同期の機能が正しく働いていることになります。これで、TBモジュールの設定は終了です。

次にCCモジュールの設定に移りましょう。今回の例では、相補PWMを出力しますので、コンペアA(CMPA)しか使用しません。今回の例ではADCは使っていませんが、ここで余ったコンペアB(CMPB)は、ADCのトリガタイミングとして使用する事ができます。もちろん、今回もシャドウ・レジスタを使用します。また、今回もコンペアをアクティブ・レジスタにアップデートするタイミングはカウンタ0を使用しましょう(表 10、表 11を参照)。コンペアA値はとりあえず適当に設定しておきましょう。そのため、CCモジュールの設定は以下のようになります。

```
EPwm1Regs.CMPA.half.CMPA = 1000;
EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;

EPwm2Regs.CMPA.half.CMPA = 2000;
EPwm2Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm2Regs.CMPCTL.bit.LOADAMODE = 0;

EPwm3Regs.CMPA.half.CMPA = 3000;
EPwm3Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm3Regs.CMPCTL.bit.LOADAMODE = 0;
```

次はAQモジュールの設定です。今回の例では、図 85のようにコンペアAのみでAQOUTA(AQモジュールによる中間出力)を作り、そのAQOUTAをDBモジュールにてデッドバンド付相補PWMに仕上げます。

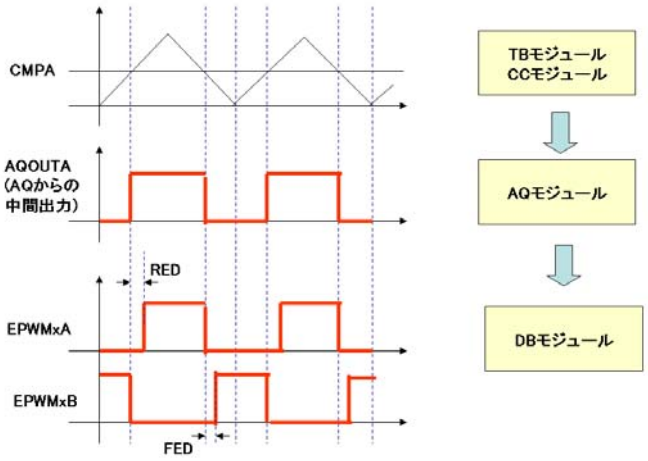


図 85:相補ＰWMの生成過程

この図より、AQOUTAを生成するためのAQCTLAの設定は、表 22:AQCTLAレジスタの設定のようになります。

レジスタ	フィールド名	今回の設定	フィールド名	今回の設定
AQCTLA	CBD	00b(何もしない)	CAU	10b(セット)
	CBU	00b(何もしない)	PRD	00b(何もしない)
	CAD	01b(クリア)	ZRO	00b(何もしない)

表 22:AQCTLA レジスタの設定

そのため、

```
EPwm1Regs.AQCTLA.bit.CAD = 1;
EPwm1Regs.AQCTLA.bit.CAU = 2;
EPwm2Regs.AQCTLA.bit.CAD = 1;
EPwm2Regs.AQCTLA.bit.CAU = 2;
EPwm3Regs.AQCTLA.bit.CAD = 1;
EPwm3Regs.AQCTLA.bit.CAU = 2;
```

と記述できます。

次にDBモジュールを設定しましょう。相補PWMにとってここが一番の要になります。相補PWMを作成する時は、DBを図86のように設定します。

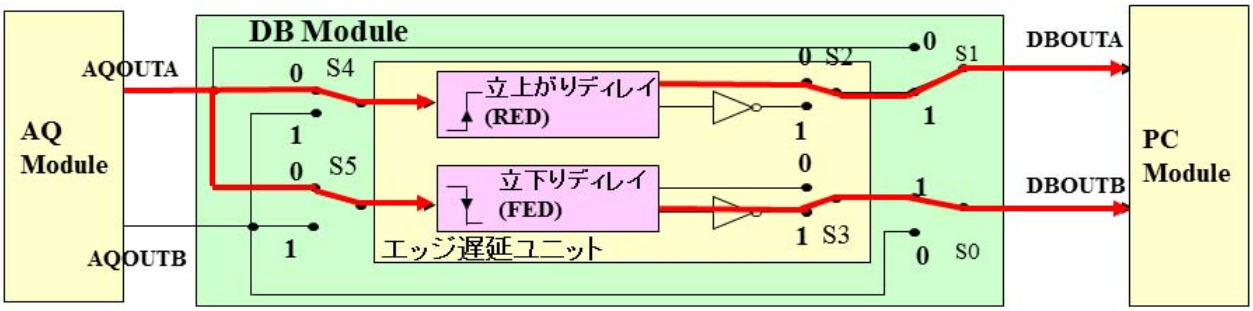


図 86:DB の設定

AQOUTAを使って、一方は立ち上がりディレイを付加し出力。もう一方は立下りディレイを付加し反転して出力します。DBモジュール内のスイッチS0:S1:S2:S3:S4:S5は1:1:0:1:0:0になります。このスイッチを制御するDBCTLレジスタのPOLSEL, INMODE, OUTMODEは、

OUTMODE[1bit:0bit]=[S1：S0]
POLSEL[1bit :0bit] = [S3：S2]
INMODE[1bit :0bit] = [S5：S4]

という関係にあります。また、Piccoloシリーズでは、DBCTLレジスタのHALFCYCLEフィールドにて、デッド・バンドの設定可能な分解能をTBCLKの倍に設定する事ができます(表 23)。例えば、TBCLK=60MHz(=16.67nsec)であれば、デッド・バンドは倍の120MHz(=8.33nsec)単位で設定が可能です。

フィールド名	設定値	意味	設定値	意味
HALFCYCLE	0	Dead Bandの設定分解能=TBCLK	1	Dead Bandの設定分解能はTBCLKの２倍

表 23:HALFCYCLE フィールド(DBCTL レジスタ)の設定

これらにより、デッド・バンドの設定は、以下のようになります。今回は、HALFCYCLE=1(Enable)で使用してみましょう。

```
EPwm1Regs.DBCTL.bit.OUT_MODE = 3;
EPwm1Regs.DBCTL.bit.POLSEL = 2;
EPwm1Regs.DBCTL.bit.IN_MODE=0 ;
EPwm1Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm2Regs.DBCTL.bit.OUT_MODE = 3;
EPwm2Regs.DBCTL.bit.POLSEL = 2;
EPwm2Regs.DBCTL.bit.IN_MODE=0
EPwm2Regs.DBCTL.bit.HALFCYCLE = 1;

EPwm3Regs.DBCTL.bit.OUT_MODE = 3;
EPwm3Regs.DBCTL.bit.POLSEL = 2;
EPwm3Regs.DBCTL.bit.IN_MODE=0
EPwm3Regs.DBCTL.bit.HALFCYCLE = 1;
```

さて、デッド・バンドを付加しますので、デッド・バンド幅を設定しましょう。デッド・バンド幅を決定するのは、RED(Riging Edge Delay:立ち上がりエッジに対するディレイ)はDBREDレジスタのDEL、FED(Falling Edge Delay:立下りエッジに対するディレイ)は、DBFEDレジスタのDELです。それぞれTBタイマのクロック・サイクル(HALFCYCLE=0の場合)か2倍高速のサイクル(HALFCYCLE=1)で10bit(0～1023)のデッドバンド幅を設定する事ができます。TBタイマクロックは、今16.67nsec(=60MHz)に設定していて、HALFCYCLE=1になっていますので、単位は8.33nsecになっています。例えば、デッド

バンドを3usecにしたければ、この場合は3usec÷8.33nsec = 360をこのDELに設定しておけば実現できます。それでは、ここではREDを1usec(1u÷8.33n=120)、FEDを2usec(2u÷8.33n=240)で設定してみましょう。

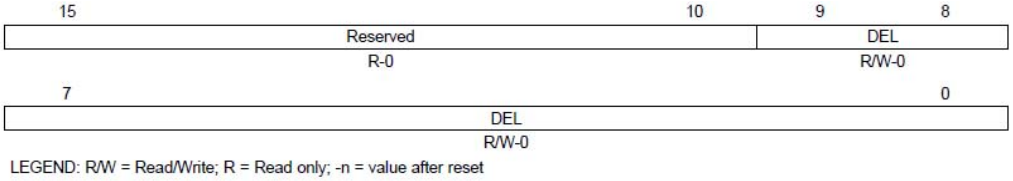


図 87:DBRED レジスタと DBFED レジスタ（同じビット・フィールド構成をしています）

DBREDもDBFEDもビットフィールド構成は全く同じで図 87のように、0～9ビットがDELで10～15ビットが予約となっていますが。しかしヘッダ・ファイルでは、DELビッド・フィールドが定義されていなく、単なる16bitレジスタ扱いになっていますので、

```
EPwm1Regs.DBRED = 120;
EPwm1Regs.DBFED = 240;
EPwm2Regs.DBRED = 120;
EPwm2Regs.DBFED = 240;
EPwm3Regs.DBRED = 120;
EPwm3Regs.DBFED = 240;
```

となります。

さて、次にETモジュールにて割り込みの設定をしましょう。今回も前回と同じように、タイマカウンタ０のタイミングで毎回割り込みを発生させましょう。全てのタイマは同期しているはずですので、割り込みはひとつのモジュールだけが発生させれば良い事になります。ここでは、マスタの役割をしているEPWM1のETモジュールを使って割り込みを発生させましょう。また、PIEとCPUレベルの割り込み設定もしておきましょう。

```
/*ET Module Configuration*/
EPwm1Regs.ETSEL.bit.INTSEL = 1;
EPwm1Regs.ETSEL.bit.INTEN = 1;
EPwm1Regs.ETPS.bit.INTPRD = 1;
EPwm1Regs.ETCLR.bit.INT = 1;

/*PIE Interrupt/CPU Interrupt Configuration*/
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
IER |= M_INT3;
```

これで全てのePWMの設定が終了しました。最後にタイマをスタートさせます。設定の最初にEALLOWを実行していますので、最後にEDISを実行するのを忘れないようにしてください。

```
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

EDIS;
```

次にEPWN1_INTが発生した時のISRをみてみましょう。

Main.cからEpwm1Isr()
interrupt void Epwm1Isr(void){

```
  static int i = 0;

  if(i==0){
    EPwm1Regs.CMPA.half.CMPA = 1000;
    EPwm2Regs.CMPA.half.CMPA = 2000;
    EPwm3Regs.CMPA.half.CMPA = 3000;
    i++;
  }else if(i==1){
    EPwm1Regs.CMPA.half.CMPA = 2000;
    EPwm2Regs.CMPA.half.CMPA = 3000;
    EPwm3Regs.CMPA.half.CMPA = 4000;
    i++;
  }else{
    EPwm1Regs.CMPA.half.CMPA = 3000;
    EPwm2Regs.CMPA.half.CMPA = 4000;
    EPwm3Regs.CMPA.half.CMPA = 5000;
    i = 0;
  }

  EPwm1Regs.ETCLR.bit.INT = 1;
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

このISRでは、EPWM1/2/3のそれぞれのCMPAを3種類の値にて変更を行っています。DutyをePWM1は83.3%→66.7%→50%に、ePWM2は66.7%→50%→33.3%、ePWM3は50%→33.3%→16.7%に変化させています。この関数では、特に新しい内容はありませんので、コードを読んでください。

さて、後は残ったmain()関数を見ていきましょう。

Main()関数

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");
interrupt void Epwm1Isr(void);
void ePWM123Config(void);

void main(void){

  DINT;
  InitSysCtrl();
  memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();

  InitPieCtrl();
  IER=0x0000;
  IFR=0x0000;
```

```
    InitPieVectTable();
    EALLOW;
    PieVectTable.EPWM1_INT  = Epwm1Isr;
    EDIS;

    ePWM123Config();
    EINT;

    while(1);
}
```

前回と比べて、新しい事はありませんので、不明な点がある場合は、前節をもう一度読んでみて下さい。

それでは、実行してみましょう。動作させて、オシロスコープにてPWM出力を確認します。まず、図 88が、EPWM1A/2A/3A出力の波形です。

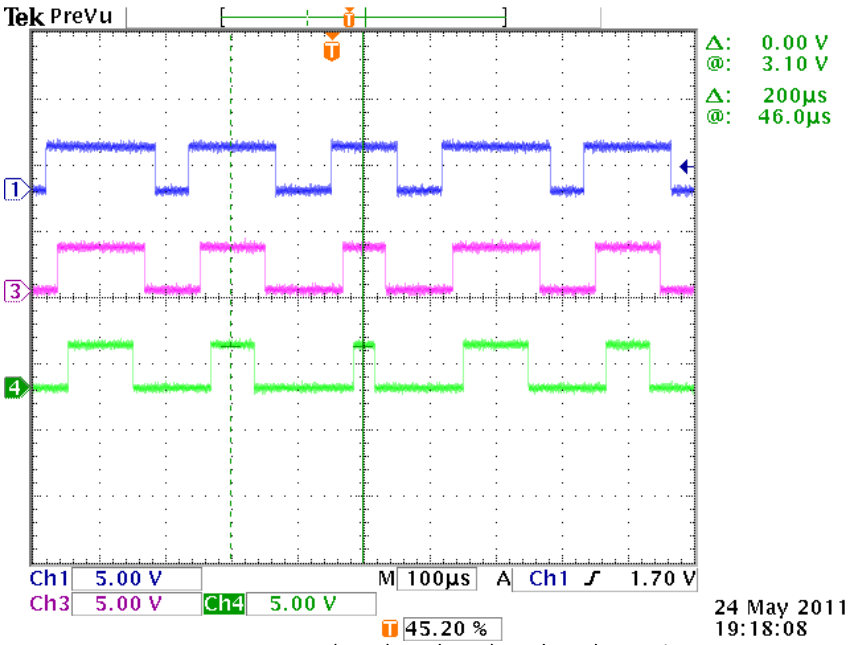


図 88:EPWM1A(CH1)/2A(CH3)/3A(CH4)出力波形

EPWM1A/2A/3Aの0同期が取れている事が確認できます。またPWMキャリア周波数も5KHzで動作しています。Dutyも全て設定通りに動作している事が確認できます。次に、図 89は、EPWM1AとEPWM1B出力の波形です。

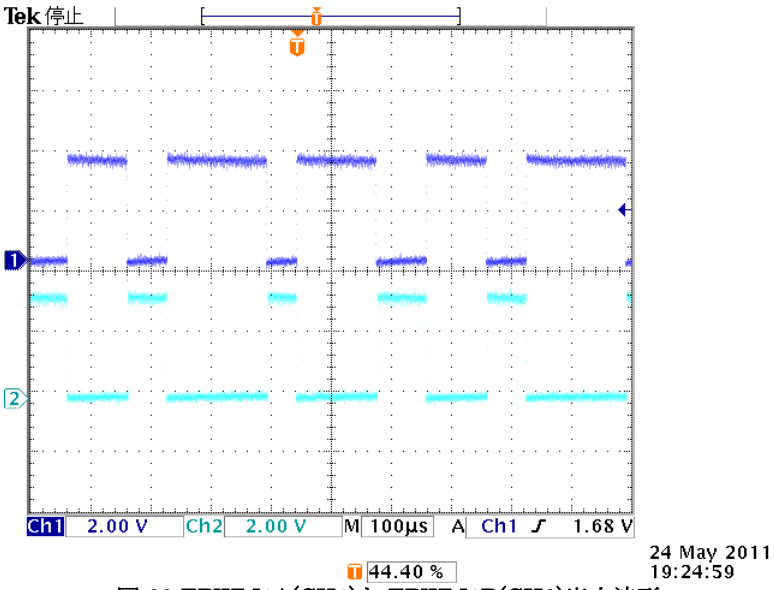


図 89:EPWM1A(CH1)と EPWM1B(CH2)出力波形

2つの出力が相補関係になっている様子がわかります。次にデッドバンドを見てみましょう。図 90にREDデッドバンドの波形を、図 91にFEDデッドバンドの波形を示します。

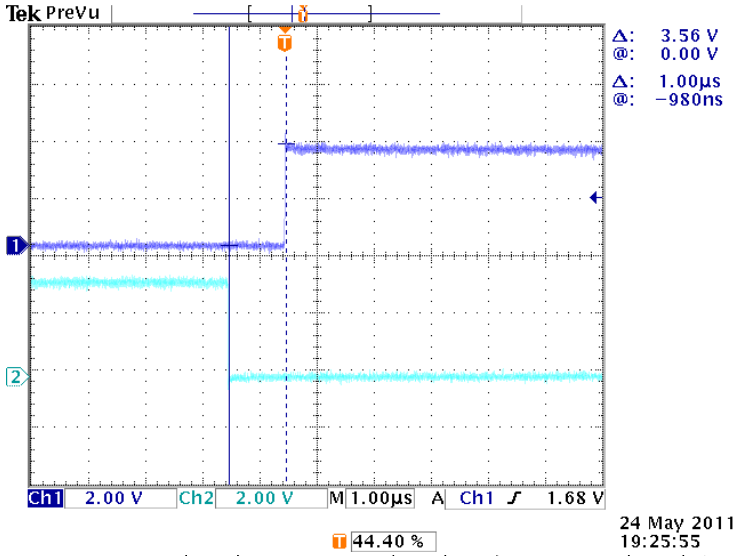


図 90:EPWM1A(CH1)と EPWM1B(CH2)のデッドバンド(RED)波形

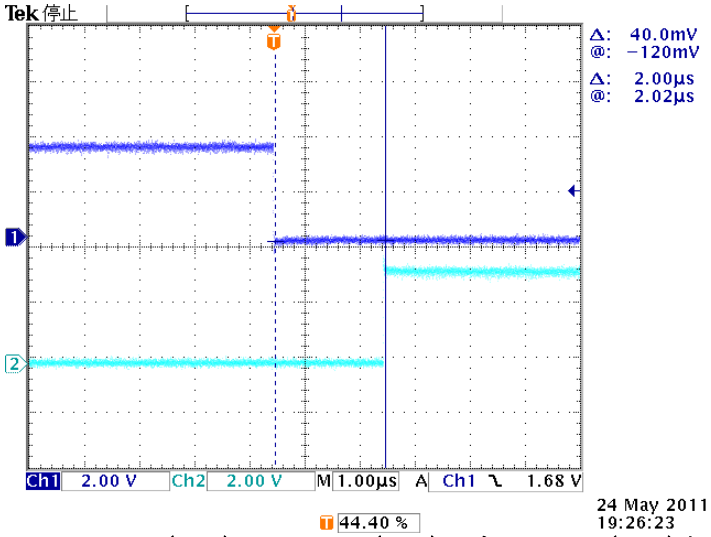


図 91:EPWM1A(CH1)と EPWM1B(CH2)のデッドバンド(FED)波形

設定通りに1usec(RED)/2usec(FED)のデッド・バンドが確認できます。

7.4 HRPWM(High Resolution PWM)使用例

PWMの使用例の最後に、HRPWMの例を解説します。HRPWMは、スイッチング周波数が高いアプリケーション(スイッチング電源等)用にＴＩが開発した、150psecという高分解能を持つＰWMモジュールです。この高分解能の仕組みは公開していませんので、このドキュメントでも、使い方についてのみ解説します。HRPWMモジュールは、PWM Dutyだけではなく、PWMの周波数も150psec単位(TYP)で分解能を刻めます。つまり、150psecのPFM(同時にdutyも150psec単位)も実現できます。さらに、フェーズシフトＰWMの場合は、150psec単位で位相を制御できます。

HRPWMを使う上で、考慮しなければならない事があります。それがデッド・バンドです。ePWMモジュールは一つのタイマにA出力とB出力の２つの出力があります。しかしながら、HRモードをもっているのは、A出力のみで(B出力にA出力の単純な反転を出力する事はできます)、B出力は通常の分解能(=CPUのクロック周期)のＰWMになります。

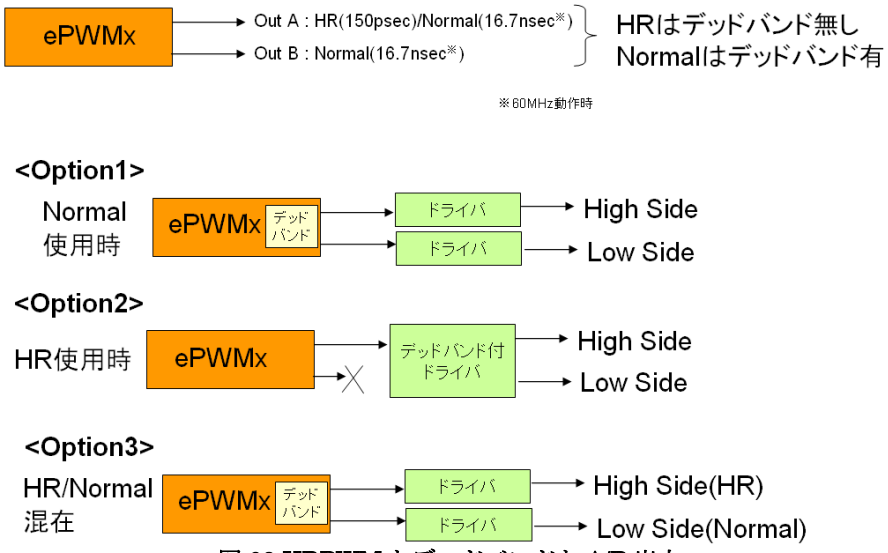


図 92:HRPWM とデッドバンドと A/B 出力

図 92を見てください。PiccoloのePWMをブリッジ回路に使う場合の３つのオプションを示しています。

1. Option 1
   これは、HRを使わないA/B出力とも通常の出力モードです。もちろん内蔵のデッド・バンドが問題なく使用できます。

2. Option 2
   これは、HRを使う通常のパターンです。HRはA出力しかないため、A出力のみを使い、デッド・バンドを付加できるドライバを使用して、High Side/Low Sideに分けます。

3. Option 3
   これは、HRを使いながら、内蔵デッド・バンドも使うパターンです。このパターンは注意が必要です。HR出力を持っているのは、A出力のみですので、この場合は、High SideがHRモード、Low Sideが通常モードになり、分解能がそれぞれ違います。また、デッドバンドは、通常モードですので、A出力のDutyによって、AとBのエッジ間隔(つまりデッド・バンド)は、その都度微妙に変わります。図で表すと図 93になります。非絶縁DC/DCのように、制御側のFET(High Side)と整流側のFET(Low Side)といったように、役割が明確の場合には、制御側のFETにHRを、整流側のFETにはNormalをという方法は今までの筆者の経験上、有効です。

図 93:Option 3 の波形

また、非常に重要な点として、先ほどから記述している150psec(この分解能ステップをMEPステップと呼んでいます)。という分解能はあくまでもTYP値であり、デバイスのばらつき、温度、動作電圧によりこの値は変わる可能性があり、ワーストケースでは310psecまで分解能が落ちる可能性があります。

このように、現在の分解能が分からないと、使い勝手が非常に悪くなってしまいます。そこで、ＴＩは、このHRPWMに対してキャリブレーション・ライブラリを用意しています。通常、デバイスのばらつきについては、最初の一回だけキャリブレーションすれば、補正できます。温度と動作電圧は、通常は、それ程変動が激しいものではありません。そのため、このキャリブレーション・ライブラリを、頻繁にコールする必要性は、ほとんどの使用例ではないでしょう。ほとんどのケースでは、アプリケーションに影響をあまり与えないバック・グラウンド・ループにて、秒単位の間隔でコールすれば十分でしょう。このキャリブレーション・ライブラリを定期的にコールすることで、現在の分解能(MEPステップ)がわかります(つまり、分解能が150psecなのか、180psecなのかがわかります。正確には、1TBCLKあたり、何個のMEPステップがあるかを測定します)。このキャリブレーション・ライブラリ関数が、測定した値を、特定のレジスタに入力してくれます。ユーザーは希望するDutyを指定されたフォーマットでコンペア・レジスタに書き込むと、あとは、HRPWMモジュールがこのキャリブレーション・ライブラリ関数が測定したデータを使って自動的に波形を生成してくれます。

今回の例は一番シンプルな例を示します。目標とする波形は、図 94です。

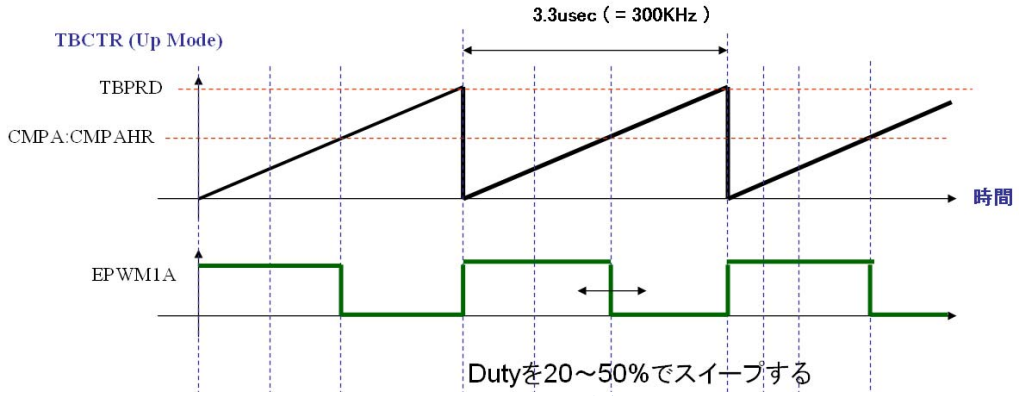
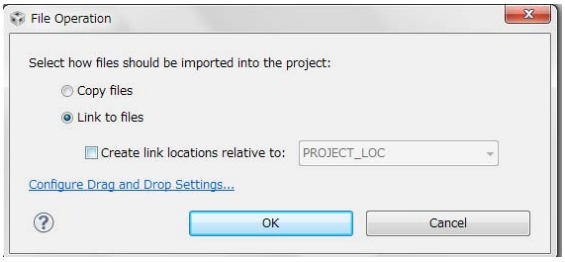


図 94:HRPWM 出力例

一般的に制御アプリケーションではPWMの分解能は最低10bit(=1024)が目安になると言われています。300KHzのスイッチング周波数では、HRモードを使用しない場合は、60MHz/300KHz = 200(=7.6bit)しかありませんので、分解能が足りない計算になります。そこで、HRモードを使用すると、3.3usec(300KHz) / 150psec = 22222(=14.4bit)と、十分過ぎる分解能がとれます。

それでは、Projectを作成します。前節のProjectと構成はほとんど同じですので、ディレクトリをコピーして、それを改造してProjectを作成しましょう。コピーして、CCSにImportしたら、適当にProject名を変更してください。Projectの構成を少し変更します。

F28069の場合	F2806x_common ― lib SFO_TI_Build_V6b_FPU.libのExlcudeを解除
F28035の場合	DSP2803x_common ― lib SFO_TI_Build_V6b.libのExcludeを解除 IQmath.libをProjectに追加します(F28035のv126には、IQmathがHeader Filesに入っていないため、IQmathフォルダからもってきます) Project→Add Filesを選択して下さい。 ファイルを選択するウィンドウが表示されますので、 C:\TI\controlSUITE\libs\math\IQmath\v160\lib\IQmath.lib を選択して下さい。以下の図ようなウィンドウが非常されますので、図のように、 Link to Filesを選択し、 Creat link locations relative to:を非選択 にして下さい。

	 ちなみに、Copy filesとLink to filesの違いですが、Copy filesは、指定したファイルをProjectの下にコピーします。Link to filesは、単にリンクするだけで、ファイル自体をProjectの下にコピーはしません。Create link locations relative to:は、このファイルをリンクする時に、相対パスにする場合はチェックをいれます。今回は、絶対パスを使いたかったので、チェックを外しました。 さらに、Cコンパイラのオプションにおいて、Includeパスの追加が必要です。以下のパスを追加して下さい(パスを指定する際は、Workspaceではなく、File Systemを選択して下さい。これは、追加するパスがWorkspaceよりも上にあるためです)。 "C:\TI\controlSUITE\libs\math\IQmath\v160\include"
F28027の場合	F2802x_common ― lib IQmath.libとSFO_TI_Build_V6b.libのExcludeを解除

F28035とF28027は
SFO_TI_BuildV6.lib
SFO_TI_BuildV6b.lib
の２つの似た名前のファイルがある事に注意してください。Excludeを解除するのは、bバージョンのSFO_TI_BuildV6b.libです。b無しとb有りは、AUTOCONVモードを使う時に動作が若干異なります。特別な理由が無ければ、bバージョンを使用下さい。（このb有とb無の違いはHRPWMモジュールのリファレンス・ガイドに掲載されています)。

それでは、コードを見てみましょう。今回のコードは小数点演算を使っています。F2806xではFPUを搭載していますので浮動小数点演算が使えます。一方、F2802xとF2803xはFPUを搭載していませんので、固定小数点演算が必要になりますので、IQmathを利用します。そのため、#ifdef, #ifndefを使って、一部のコードはF2802x/F2803xとF2806xとで分かれています。最初の行の

```
#define USE_F28069
```

をコメントアウトすると、IQmathを使ったF2802x/F2803x用のコードに、コメントアウトしないと、浮動小数点を使ったF2806x用のコードになります。赤がF2802x/F2803x用のコード、紫がF2806x用のコードと色分けをしました。

main.c

```
#define USE_F28069

#include "DSP28x_Project.h"
#include "SFO_V6.h"
#include <string.h>
#ifndef USE_F28069
#define  GLOBAL_Q 22  // range = +-512
#include "IQmathLib.h"
#endif

#define PWM_PERIOD   200
```

```
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");

interrupt void Epwm1Isr(void);
void ePWM1Config(void);
void wait(void);

#ifndef USE_F28069
_iq Duty = _IQ(0.2);
_iq DutyShadow = _IQ(0.2);
#else
float Duty = 0.2;
float DutyShadow = 0.2;
#endif

int MEP_ScaleFactor = 0;
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        {  &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs};

void main(void){
  int dir = 0;
  int status;

  DINT;
  InitSysCtrl();
  memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();
  InitPieCtrl();
  IER=0x0000;
  IFR=0x0000;
  InitPieVectTable();
  EALLOW;
  PieVectTable.EPWM1_INT  = Epwm1Isr;
  EDIS;

  status = SFO_INCOMPLETE;
  while(status== SFO_INCOMPLETE){
    status = SFO();
    if (status == SFO_ERROR) {
      while(1);
    }
  }
  ePWM1Config();
  EINT;

  while(1){
    status = SFO();
```

```
        if (status == SFO_ERROR) {
            while(1);
        }
        wait();

        //Sweep duty
#ifndef USE_F28069
        if(dir == 0){
            DutyShadow += _IQ(0.0001);
            if(DutyShadow > _IQ(0.5)){
                dir = 1;
            }
        }else{
            DutyShadow -= _IQ(0.001);
            if(DutyShadow < _IQ(0.2)){
                dir = 0;
            }
        }
#else
        if(dir == 0){
            DutyShadow += 0.0001;
            if(DutyShadow > 0.5){
                dir = 1;
            }
        }else{
            DutyShadow -= 0.001;
            if(DutyShadow < 0.2){
                dir = 0;
            }
        }
#endif
        DINT;
        Duty = DutyShadow;
        EINT;
    }
}

void ePWM1Config(void){

    EALLOW;

    /*Stop PWM Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

    /*GPIO Configuration*/
    GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;

    /*TB Module Configuration*/
```

```
    EPwm1Regs.TBCTL.bit.CLKDIV = 0;
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
    EPwm1Regs.TBCTL.bit.CTRMODE = 0;
    EPwm1Regs.TBCTR = 0;
    EPwm1Regs.TBPRD = PWM_PERIOD-1;

    /*CC Module Configuration*/
    EPwm1Regs.CMPA.half.CMPA = 0;
    EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
    EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;

    /*AC Module Configuration */
    EPwm1Regs.AQCTLA.bit.CAU = 1;
    EPwm1Regs.AQCTLA.bit.ZRO = 2;
    EPwm1Regs.AQCTLB.bit.CAU = 1;
    EPwm1Regs.AQCTLB.bit.ZRO = 2;

    /*DB Module Confiruation */
    EPwm1Regs.DBCTL.bit.OUT_MODE = 0;

    /*ET Module Configuration*/
    EPwm1Regs.ETSEL.bit.INTSEL = 1;
    EPwm1Regs.ETSEL.bit.INTEN = 1;
    EPwm1Regs.ETPS.bit.INTPRD = 1;
    EPwm1Regs.ETCLR.bit.INT = 1;

    /*HR Module Configuraton*/
    EPwm1Regs.HRCNFG.bit.SWAPAB = 0;
    EPwm1Regs.HRCNFG.bit.AUTOCONV = 1;
    EPwm1Regs.HRCNFG.bit.HRLOAD = 0;
    EPwm1Regs.HRCNFG.bit.CTLMODE = 0;
    EPwm1Regs.HRCNFG.bit.EDGMODE = 2;

    /*PIE Interrupt/CPU Interrupt Configuration*/
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    IER |= M_INT3;

    /*Start Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    EDIS;

}

#ifdef USE_F28069
#pragma INTERRUPT(Epwm1Isr, HPI);
#endif

interrupt void Epwm1Isr(void){
#ifndef USE_F28069
```

```
   EPwm1Regs.CMPA.all = _IQtoIQ16( _IQmpyI32(Duty,(long)PWM_PERIOD));
#else
   EPwm1Regs.CMPA.all = (unsigned long)(65536.0 * ((float)EPwm1Regs.TBPRD * Duty));
#endif

   EPwm1Regs.ETCLR.bit.INT = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}

void wait(void){
   long i;

   for(i = 0; i<0x50000 ; i++){
      asm(" NOP ");
      asm(" NOP ");
      asm(" NOP ");

   }
}
```

それでは、まず、ｅＰＷＭの設定を解説しましょう。この部分は、F2802x/2803x/2806x共通になります。

ePWM1Config()関数

```
void ePWM1Config(void){

   EALLOW;

   /*Stop PWM Timer*/
   SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

   /*GPIO Configuration*/
   GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
   GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;

   /*TB Module Configuration*/
   EPwm1Regs.TBCTL.bit.CLKDIV = 0;
   EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
   EPwm1Regs.TBCTL.bit.CTRMODE = 0;
   EPwm1Regs.TBCTR = 0;
   EPwm1Regs.TBPRD = PWM_PERIOD-1;

   /*CC Module Configuration*/
   EPwm1Regs.CMPA.half.CMPA = 0;
   EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
   EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;

   /*AC Module Configuration */
```

```
    EPwm1Regs.AQCTLA.bit.CAU = 1;
    EPwm1Regs.AQCTLA.bit.ZRO = 2;
    EPwm1Regs.AQCTLB.bit.CAU = 1;
    EPwm1Regs.AQCTLB.bit.ZRO = 2;

    /*DB Module Confiruation */
    EPwm1Regs.DBCTL.bit.OUT_MODE = 0;

    /*ET Module Configuration*/
    EPwm1Regs.ETSEL.bit.INTSEL = 1;
    EPwm1Regs.ETSEL.bit.INTEN = 1;
    EPwm1Regs.ETPS.bit.INTPRD = 1;
    EPwm1Regs.ETCLR.bit.INT = 1;

    /*HR Module Configuraton*/
    EPwm1Regs.HRCNFG.bit.SWAPAB = 0;
    EPwm1Regs.HRCNFG.bit.AUTOCONV = 1;
    EPwm1Regs.HRCNFG.bit.HRLOAD = 0;
    EPwm1Regs.HRCNFG.bit.CTLMODE = 0;
    EPwm1Regs.HRCNFG.bit.EDGMODE = 2;

    /*PIE Interrupt/CPU Interrupt Configuration*/
    PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
    IER |= M_INT3;

    /*Start Timer*/
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    EDIS;

}
```

まず、最初の以下の行ですが、この辺は既におなじみになってきたと思います。

```
EALLOW;

/*Stop PWM Timer*/
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;

/*GPIO Configuration*/
GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1;
GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1;
```

EALLOW保護を解除し、EPWMクロックを停止し、GPIO0をEPWM1A出力に設定しています。今回、モニタ用にEPWM1B出力も出してみたいと思いますので、GPIO1をEPWM1B出力に設定します。。EPWM1Bには、EPWM1AのHR機能無し版を出力します。EPWM1Aと1Bを比較することで、HR機能の機能を実感しやすくなると思います。

```
/*TB Module Configuration*/
EPwm1Regs.TBCTL.bit.CLKDIV = 0;
```

```
EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0;
EPwm1Regs.TBCTL.bit.CTRMODE = 0;
EPwm1Regs.TBCTR = 0;
EPwm1Regs.TBPRD = 199;
```

次に、TBモジュールの設定を行います。まず、重要な点としては、HRPWMを使用するときは、SYSCLKOUTが50MHz以上で、SYSCLKOUT=TBCLKである事が必要です。そのため、CLKDIVとHSPCLKDIVは0に設定し、SYSCLKOUT=TBCLKに設定します（一般的には、SYSCLKOUT≠TBCLKの環境でＨＲモードが必要になるケースはほとんど無いと思いますので、これの制限が問題になるケースはほとんどないと思います。）。次に、CTRMODE(カウント・モード)の設定ですが、今回は、図 94の波形ですので、Upカウントモード(CTRMODE=0)で使用します。カウンタの初期値(TBCTR)は0にしておきましょう。次に周期ですが、300KHzですので、60MHz/300KHz = 200となりますので、1を引いて199を設定します。

次の３行ですが、

```
/*CC Module Configuration*/
EPwm1Regs.CMPA.half.CMPA = 0;
EPwm1Regs.CMPCTL.bit.SHDWAMODE = 0;
EPwm1Regs.CMPCTL.bit.LOADAMODE = 0;
```

今回は、CMPAのみ使用して、CMPBは使用しません。HRモードを使う場合は必ずCMPAレジスタを使います(後ほど、もう少し詳しく解説します)。シャドウ・レジスタを使用して、TBCTR=0で値のシャドウからアクティブ・レジスタへのコピーを行います。

次の

```
/*AC Module Configuration */
EPwm1Regs.AQCTLA.bit.CAU = 1;
EPwm1Regs.AQCTLA.bit.ZRO = 2;
EPwm1Regs.AQCTLB.bit.CAU = 1;
EPwm1Regs.AQCTLB.bit.ZRO = 2;
```

ですが、図 94の波形から、ZRO(TBCTR=0)でHigh出力(2)、CMPAマッチでLow出力(1)に設定しています。AQOUTAがEPWM1A用(HR使用)、AQOUTBがEPWM1B用(HR未使用、モニタ用)になります。両方同じ設定にして比較できるようにしています。

次のデッド・バンド・モジュールの設定ですが、今回はデッド・バンドは使用せずにそのままスルーさせます。スルーさせるには、既にePWMの最初のProjectで解説したように、DBCTLレジスタのOUT_MODEを0に設定にします。

```
/*DB Module Confiruation */
EPwm1Regs.DBCTL.bit.OUT_MODE = 0;
```

ETモジュールの設定は、今までと同じです。TBCTR=0で毎回割り込みをかける設定です。

```
/*ET Module Configuration*/
EPwm1Regs.ETSEL.bit.INTSEL = 1;
EPwm1Regs.ETSEL.bit.INTEN = 1;
EPwm1Regs.ETPS.bit.INTPRD = 1;
EPwm1Regs.ETCLR.bit.INT = 1;
```

次に、今回のメインのHRモジュールの設定を行っています。

```
/*HR Module Configuraton*/
EPwm1Regs.HRCNFG.bit.SWAPAB = 0;
EPwm1Regs.HRCNFG.bit.AUTOCONV = 1;
EPwm1Regs.HRCNFG.bit.HRLOAD = 0;
EPwm1Regs.HRCNFG.bit.CTLMODE = 0;
EPwm1Regs.HRCNFG.bit.EDGMODE = 2;
```

まず、SWAPABビットの設定です。HRモードは、基本的にはA出力しかできませんが、A出力とB出力を交換する事ができます(表 24)。また、今回は特に設定していませんが、B出力をA出力の単なる反転を出力する設定もあります(SELOUTBフィールドです、詳細はリファレンス・ガイドをご参照下さい)。

フィールド名	設定値	意味	設定値	意味
SWAPAB	0	A出力とB出力を入れ替えない	1	A出力とB出力を入れ替える

表 24:HRCNFG レジスタの SWAPAB フィールド

今回は、入れ替えませんので0に設定しています。

次のAUTOCONVモードですが、基本的にはPiccoloでは、AUTOCONV=1以外で使用する必要性はほぼ無いに等しいと思います。これは、前バージョンのHRモジュール(F280x/F2823x/F2833x/C2834x)シリーズとの互換のためにあるビットと考えて下さい。前バージョンのHRモジュールは、150psec単位のコンペア値とTBCLK単位のコンペア値という２種類のコンペア値があり、希望するDutyを出力するために、それぞれ単位の違うコンペア値の値をソフトにて演算する必要がありました。PiccoloのHRモジュールは、その演算を行う回路を搭載していますので、より簡単に設定できるようになっています。その機能をEnableにするのが、このAUTOCONVビットです。また、このAUTOCONVをセットすることで、前バージョンでは対応していなかった、up-downカウント・モードでのHRモードや、HRPFMにも対応する事ができます。今回ももちろんAUTOCONVモードを使用しますので、1に設定します。

フィールド名	設定値	意味	設定値	意味
AUTOCONV	0	自動変換機能をDisable	1	自動変換機能をEnable

表 25:HRCNFG レジスタの AUTOCONV フィールド

次のHRLOADの説明をする前に、HRモードを使うときのコンペア・レジスタについて解説します。HRモードを使用しない場合は、CMPAレジスタがコンペア・レジスタになりますが、このCMPAレジスタは実は32bitレジスタで、図 95のように、上位16bitがCMPAレジスタ（同じ名前で紛らわしいですが）、下位16bitがCMPAHRレジスタ(内下位8bitはreserved)になっています。ＨＲモードを使わない場合は、上位16bitのCMPAレジスタを使用しますが、HRモードを使う時は、下位16bitのCMPAHRレジスタも使用します。

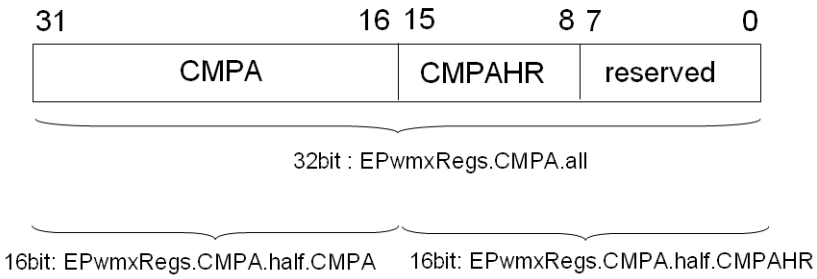


図 95:32bit CMPA レジスタ

HRLOADレジスタは、このCMPAHRレジスタのシャドウ・レジスタから、アクティブ・レジスタへのコピータイミングを設定します。表 26にHRLOADの設定を示します。

フィールド名	設定値	意味	設定値	意味
HRLOAD	00b	TBCTR=0でロード	01b	TBCTR=TBPRDでロード
	10b	TBCTR=0 及びTBPRD でロード	11b	予約

表 26:HRCNFG レジスタの HRLOAD フィールド

CMPAレジスタのシャドウからアクティブへのコピーはTBCTR=0にて設定(EPwm1Regs.CMPCTL.bit.LOADAMODE = 0)していますので、このHRLOADも00bで設定します。

次に、CTLMODEを設定しています。HRPWMの150psec単位の分解能は、次の3つの対象があります。

1. Duty(CMPAHRで制御)
2. 周期(TBPRDHRで制御)
3. フェーズシフト量(TBPHSHRで制御)

このCTLMODEフィールドは、この３つのうちどれが対象かを設定します。

フィールド名	設定値	意味	設定値	意味
CTLMODE	0	CMPAHRかTBPRDHRで制御(つまりPWM/PFM制御)	1	TBPHSHRで制御（つまりフェーズ・シフト制御)

表 27:HRCNFG レジスタの CTLMODE フィールド

今回は、Duty制御ですので、0を設定します。

最後にEDGEMODEです。HRモードでは、HR制御をどのエッジに対して行うかの設定ができます。その選択をするのがEDGEMODEです。

フィールド名	設定値	意味	設定値	意味
EDGEMODE	00b	HRPWM機能はDisable	01b	立ち上がりエッジに対してHR制御
	10b	立ち下りエッジに対してHR制御	11b	両エッジに対してHR制御

表 28:HRCNFG レジスタの EDGEMODE フィールド

図 96にどのようなＰWM波形に対してどのEDGEMODEを使うかの図を示します。こちらを見ていただくと、EDGEMODEの意味がより理解できると思います。

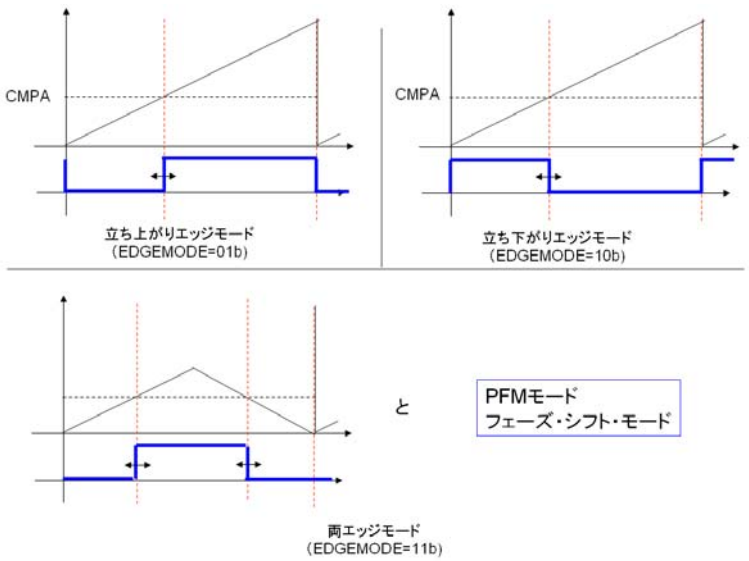


図 96:EDGEMODE と波形

今回の目的波形から、立ち下がりエッジ・モード(EDGEMODE=10b)を選択しています。各カウント・モードに対して、HRが利かない領域が存在します。HRが利かない領域は、通常のTBCLK単位の分解能になります。HRが利かない領域は、リファレンス・ガイドに詳細が記載されていますので、ご参照下さい。今回使用しているUpカウント・モードで今回の使用方法では、TBCTR=0～3の間はHRが利きません。つまり、3TBCLK以下の短いパルスは、HRモードは利かない事になります。ここはご注意下さい。

さて、これでHRモジュールの設定が終わりましたので、後は割り込みの設定と、PWMクロックのスタート処理を行います。

```
/*PIE Interrupt/CPU Interrupt Configuration*/
PieCtrlRegs.PIEIER3.bit.INTx1 = 1;
IER |= M_INT3;

/*Start Timer*/
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;

EDIS;
```

次にmain関数を解説します。ここでは、HRモジュールを使う上で非常に重要なキャリブレーション関数SFO()の使い方も解説します。このmain関数は、小数点演算が一部入っています。そのため、F2806xでは浮動小数点を、F2802x/2803xはIQmathを使っています。

main()関数

```
#define USE_F28069

#include "DSP28x_Project.h"
#include "SFO_V6.h"
#include <string.h>
#ifndef USE_F28069
#define  GLOBAL_Q 22  // range = +-512
#include "IQmathLib.h"
#endif

#define PWM_PERIOD   200

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(Epwm1Isr, "ramfuncs");

interrupt void Epwm1Isr(void);
void ePWM1Config(void);
void wait(void);

#ifndef USE_F28069
_iq Duty = _IQ(0.2);
_iq DutyShadow = _IQ(0.2);
#else
float Duty = 0.2;
float DutyShadow = 0.2;
#endif

int MEP_ScaleFactor = 0;
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        {  &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs};

void main(void){
```

```
  int dir = 0;
  int status;

  DINT;
  InitSysCtrl();
  memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();
  InitPieCtrl();
  IER=0x0000;
  IFR=0x0000;
  InitPieVectTable();
  EALLOW;
  PieVectTable.EPWM1_INT  = Epwm1Isr;
  EDIS;

  status = SFO_INCOMPLETE;
  while(status== SFO_INCOMPLETE){
    status = SFO();
    if (status == SFO_ERROR) {
      while(1);
    }
  }
  ePWM1Config();
  EINT;

  while(1){
    status = SFO();
    if (status == SFO_ERROR) {
      while(1);
    }
    wait();

    //Sweep duty
#ifndef USE_F28069
    if(dir == 0){
      DutyShadow += _IQ(0.0001);
      if(DutyShadow > _IQ(0.5)){
        dir = 1;
      }
    }else{
      DutyShadow -= _IQ(0.001);
      if(DutyShadow < _IQ(0.2)){
        dir = 0;
      }
    }
#else
    if(dir == 0){
      DutyShadow += 0.0001;
      if(DutyShadow > 0.5){
```

```
            dir = 1;
        }
    }else{
        DutyShadow -= 0.001;
        if(DutyShadow < 0.2){
            dir = 0;
        }
    }
#endif
    DINT;
    Duty = DutyShadow;
    EINT;
  }
}
```

まず、ヘッダのincludeです。

```
#include "DSP28x_Project.h"
#include "SFO_V6.h"
#include <string.h>

#ifndef USE_F28069
#define  GLOBAL_Q 22  // range = +-512
#include "IQmathLib.h"
#endif
```

HRPWMを使う時は、キャリブレーション関数のSFO()関数をコールしますが、そのヘッダファイルのSFO_V6.hを必ずincludeして下さい。

F2802x/F2803xの場合はIQmathを使用しますので、GLOBAL_Qのdefineを行い、IQ値を設定しています。今回の変数にて一番大きい値は、理論上は200です。少し余裕を持って、±512のレンジが使えるIQ22フォーマットを採用しました。その次にIQmathのヘッダ、IQmathLib.h をインクルードします。この順番に注意してください。IQmathLib.hをincludeする前にGLOBAL_Qの設定は行ってください。

次に見る箇所は変数の宣言で、F2802x/2803xとF2806xで分かれています。

```
#ifndef USE_F28069
_iq Duty = _IQ(0.2);
_iq DutyShadow = _IQ(0.2);
#else
float Duty = 0.2;
float DutyShadow = 0.2;
#endif
```

まず、F2802x/F2803xの場合は、２つの_iq型(IQmathの変数型)のグローバル変数を宣言しています。_IQ(浮動小数点)は、浮動小数点フォーマットから固定小数点フォーマットへの変換マクロです。この２つの変数は、Dutyの指令値として使っています。1.0 = 100%, 0 = 0%として使います。初期値として、20%の0.2を入れておきます。2つ用意しているのには、理由があります。実際のCMPAレジスタへの値の入力はEPWM1割り込みルーチンにて行っています。その時Duty変数を使用しますが、仮に、Duty変数を途中計算している時に割り込みが入ってしまうと、途中値がCMPAレジスタに入ってしまいます。そのため、計算時はDutyShadow変数を使いこの値が確定したら、割り込み禁止処理/DutyShadow→Dutyのコピー/割り込み許可を行います。後ほど、この処理をしている行がでてきます。F2806xの場合は、単純にfloatにて宣言しています。

次の

```
int MEP_ScaleFactor = 0;
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        { &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs};
```

は、HRPWMキャリブレーション用のSFO関数が使用するグローバル変数で、必ず定義する必要があります。２行目の*ePWM[PWM_CH]ですが、これは、使用するHRPWMのチャネル数に応じた設定が必要です。F2802xでは最大4本、F2803xでは最大7本、F28069では最大8本のHRPWMチャネルがありますが、今回のこの設定では、4本のHRチャネルを使えるように設定しています。ここで、ヘッダファイルの

F28069の場合	F2806x_common\include\ SFO_V6.h
F28035の場合	DSP2803x_common\include\SFO_V6.h
F28027の場合	F2802x_common\include\SFO_V6.h

を開いてください。以下の行があると思います。

```
//===========================================================================
// USER MUST UPDATE THIS CONSTANT FOR NUMBER OF HRPWM CHANNELS USED
//===========================================================================
#define PWM_CH  5   // Equal # of HRPWM channels PLUS 1
                    // i.e. PWM_CH is 9 for 8 channels, 7 for 6 channels, etc.
```

このPWM_CHがデフォルトでは5に設定されていると思いますが、この値は、ユーザーが何番目のPWMをHRモードで使用するかによって、値を変更する必要があります。ここの値は、HRモードで使用するPWMモジュールの番号の一番大きい番号+1を設定します。HRモードのチャネル総数ではない事に注意してください。

例えば、PWM1とPWM2を通常モードで使用して、PWM3をＨＲモード、PWM4以降はは全て通常のモードの場合は、ここは4以上を設定します。デフォルト値は5ですので、PWM1～4は、HRモードを使用する可能性がある事を意味します。

さて、ここで、

```
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        { &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs};
```

この行に戻りますが、この行では、最初は、必ず&EPwm1Regsを、次からは、&EPwm1Regs、&EPwm2Regs, ...と、上で見たSFO_V6.hの下の行にて、定義したPWM_CH-1の番号分だけ、&EPwmxRegsを設定します。

```
#define PWM_CH  5   // Equal # of HRPWM channels PLUS 1
```

例えば、デフォルトでは、PWM_CH=5ですので、

```
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        { &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs};
```

と設定します。例えば、PWM_CH=8 (PWM7までHRモード)の場合は、

```
volatile struct EPWM_REGS *ePWM[PWM_CH] =
        {  &EPwm1Regs, &EPwm1Regs, &EPwm2Regs, &EPwm3Regs, &EPwm4Regs, &EPwm5Regs , &EPwm6Regs,
&EPwm7Regs };
```

と設定します。今回は、EPWM1しかHRモードで使用しませんが、4まで設定しておきました。深い意味はありません（デフォルトのままです）が、F2802xのPWMチャネルが最大4チャネルなので、その設定にしておきました。実際に使用する際は、用途に応じて変更して下さい。

Mainの最初の方は、今まで解説した内容と変わりありませんので、省略します。
以下の行を解説します。

```
status = SFO_INCOMPLETE;
while(status== SFO_INCOMPLETE){
  status = SFO();
  if (status == SFO_ERROR) {
    while(1);
  }
}
```

この行は、SFO()関数を最初にコールしている箇所です。SFO()関数自体は、HRPWMのステップ(MEPステップ)が1TBCLK（正確にはSYSCLKOUTですが、TBCLK=SYSCLKOUTで使用するため）あたり何個のMEPステップがあるかを測定します。SFO()関数は、その結果を、EPWM1モジュールにあるHRMSTEPレジスタに格納します。AUTOCONVモードがEnableの時は、このHRMSTEPレジスタに格納された値をもとに、実際に使われるコンペア値を自動計算します。この行では、一回このSFO()関数をコールする事で、最初のHRMSTEP値を確定させます（この値がないと、最初のHRの計算が正しく計算できないためです）。SFO()関数がSFO_ERRORを返す事はSYSCLKOUTを50MHz以上で使う場合は通常はありません。このSFO_ERRORは、”MEPステップは8bit(0～255)ありますがMEPステップ幅が小さすぎて255個では1TBCLKに辿りつけない”という意味になります。SYSCLKOUTが50MHz以下で使用する場合は、このERRORが発生する可能性があります。そのため、HRモードを使う場合は、50MHz以上で御使用下さい。この例では、SFO_ERRORが起きた場合は永久ループにしています。

次に、ePWM1Config()をコールして、PWMの設定/スタートを行って、グローバル割り込みを許可しています。

```
ePWM1Config();
EINT;
```

この後は、永久whileループに入ります。

```
  while(1){
    status = SFO();
    if (status == SFO_ERROR) {
      while(1);
    }
    wait();

    //Sweep duty
#ifndef USE_F28069
    if(dir == 0){
      DutyShadow += _IQ(0.0001);
      if(DutyShadow > _IQ(0.5)){
        dir = 1;
      }
    }else{
      DutyShadow -= _IQ(0.001);
      if(DutyShadow < _IQ(0.2)){
```

```
            dir = 0;
        }
    }
#else
    if(dir == 0){
        DutyShadow += 0.0001;
        if(DutyShadow > 0.5){
            dir = 1;
        }
    }else{
        DutyShadow -= 0.001;
        if(DutyShadow < 0.2){
            dir = 0;
        }
    }
#endif
    DINT;
    Duty = DutyShadow;
    EINT;
}
```

ループの最初にSFO()をコールしています。初回のSFO()コールにて、デバイスのばらつきと、初期電圧・温度に対しては抑えられますが、初回だけでは、温度と電圧の変動に対応できません。この様にバック・グランド・ループで、常にSFO()をコールする事によって、温度と電圧の変動によって、変動するMEPステップ値に対応できるようになります。SFO()をコールした後は、Duty指令値を20%～50%の間でゆっくり更新しています。F2802x/F2803xはIQmathを使って、F2806xは浮動小数点演算を行っています。指令値更新速度を調整するために、下のwait()関数を途中でコールしています。

```
void wait(void){
  long i;

  for(i = 0; i<0x50000 ; i++){
    asm(" NOP ");
    asm(" NOP ");
    asm(" NOP ");

  }
}
```

この関数は単にＮＯＰループで、時間を適当に費やすコードです。Duty変数の計算が終わったら、割り込み禁止/DutyShadow→Dutyへのコピー/割り込み許可を行っています。割り込み禁止処理をはさんでいる理由は、Dutyを更新しているコードはC言語ではたったの一行ですが、アセンブラレベルでは一行とは限りません。このDutyは割り込み処理関数にて使用しますので、中途半端に使われると困ります。そのため、アップデートが完全に終わるまで割り込みを禁止する処理を入れています。

最後に、割り込み処理関数を見てみましょう。

EPwm1Isr()割り込みサービス・ルーチン
#ifdef USE_F28069 #pragma INTERRUPT(Epwm1Isr, HPI); #endif

```
interrupt void Epwm1Isr(void){
#ifndef USE_F28069
   EPwm1Regs.CMPA.all = _IQtoIQ16( _IQmpyI32(Duty,(long)PWM_PERIOD));
#else
   EPwm1Regs.CMPA.all = (unsigned long)(65536.0 * ((float)EPwm1Regs.TBPRD * Duty));
#endif

   EPwm1Regs.ETCLR.bit.INT = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
}
```

このISRでは、mainの永久ループにて計算されたDuty指令値変数"Duty"を元に計算した値をCMPAレジスタに格納しています。最初に、F2806xの場合には、
#pragma INTERRUPT(Epwm1Isr, HPI);
という一行が加えられています。これは、この関数にて浮動小数点演算を使用しているため入れています。無くてもかまいませんが、この一行を加えることで、割り込み応答が速くなります。一方、若干の制約もでます。この詳細は12.10章にて解説していますので、そちらをご参照下さい。

さて、HRモードでの32bitCMPAレジスタの使い方ですが、AUTOCONVモードを使っている場合は、非常に単純です。IQ16フォーマットの固定小数点フォーマットで、希望するコンペア値を格納します。例を挙げます。

TBPRD(つまりキャリア周期)=199とします。TBPRDには希望する周期値-1をいれますので、実際の周期値は200です。
Duty = 0.3321(33.21%)が指令値とします。

200*0.3321 = 66.42

この66.42をIQ16フォーマットにして、32bit CMPAに格納します。IQmathを使うと、
周期値は整数です。DutyはGLOBAL_Q（この例ではQ22）フォーマットの固定小数点です。
周期＊Dutyを計算する場合、整数＊IQフォーマットになります。この計算をするIQmath関数は、

_iq _IQmpyI32(GLOBAL_Qフォーマットの_iq型, 整数)

です。そのため、_IQmpyI32(Duty, (long)PWM_PERIOD)になります。この計算結果の出力フォーマットはGLOBAL_Qフォーマットです。これをIQ16フォーマットに変換します。GLOBAL_QフォーマットからIQ16フォーマットへの変換関数は、
_iq16 _IQtoIQ16(_iq)
です。
これらをまとめると、_IQtoIQ16(_IQmpyI32(Duty,**(long)**PWM_PERIOD))になるのです。

F2806xの場合は、この行は、以下のようになっています。

EPwm1Regs.CMPA.all = (**unsigned long**)(65536.0 * ((**float**) PWM_PERIOD) * Duty));

F2806xの場合はDuty変数は浮動小数点(float)になっていますので、これを固定小数点に変換しなくてはなりません。浮動小数点をIQ16フォーマットに変換するには、2^16(=65536.0)をかけて、(unsigned long)にキャストします。
この例では、
PWM_PERIOD ＊Duty
までは、浮動小数点演算を行い、これをIQ16フォーマットに変換しています。

それでは、実際に動作確認してみましょう。図 97は、このコードを動作させてEPWM1A出力とEPWM1B出力をオシロスコープにて取った波形です。

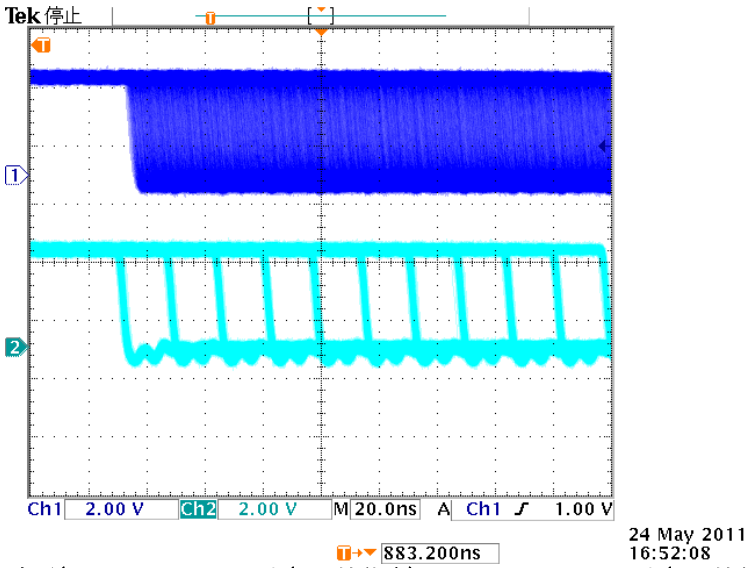


図 97:HRPWM 出力波形(CH1:EPWM1A 出力(HR 機能有)、CH2:EPWM1B 出力(HR 機能無、比較モニタ用)

この波形は、立ち上がりエッジでディレイ・トリガをかけ、立ち下りエッジを拡大した波形です。Tektronix製のオシロスコープを使用しており、パーシスタントを∞秒に設定しています(つまり、永遠に重ねあわされた波形です)。CH1がEPWM1A出力、CH2がEPWM1B出力です。EPWM1Aは150psec HRPWMとして使用しており、EPWM1Bは同じコンペアを使っていますが、HR機能がない16.7nsec分解能の通常のPWM出力になっています。動作させているコードは、Dutyを少しずつスイープさせていますので、それぞれの分解能が見ることができます。EPWM1B出力は、16.7nsecの分解能ステップがはっきり見れますが、EPWM1A出力は、さすがにこのレンジでは全く分解能を見る事ができません。これから、非常に高い分解能が実現できている事がわかります。実際にオシロスコープで観察して、その驚異的な分解能を体験下さい。

8 AD コンバータ

8.1 この章の目的

この章では、Piccoloに搭載されているADCの概要と、その使用例を示します。ADCの詳細は

、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]のAnlog-to-Digital Converter and Comparator章*
F2802x/F2803x用	*TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide[SPRUGE5]*

を参照下さい。ADCに関しましたは、誠に恐縮ですが、重要な不具合が数点報告されています。必ず、各デバイスのErrataに目を通してください。このドキュメントでは、Errataについて少しふれますが、全てを解説するわけではありませんので、必ずErrataをご確認ください。。

8.2 Piccolo のＡＤコンバータの概要と SOC

F2802x/2803x/2806xに搭載されているＡＤコンバータは基本的には同じモジュールが搭載されています(リビジョンによって、少しアップデートがあります)。ここでは、その概要を解説します。従来のF281x/F280x/F2823x/F2833xに搭載されているＡＤＣとは全く違うモジュールになっていますので、ご注意下さい。図 98にPiccolo ADCのブロック図を示します。

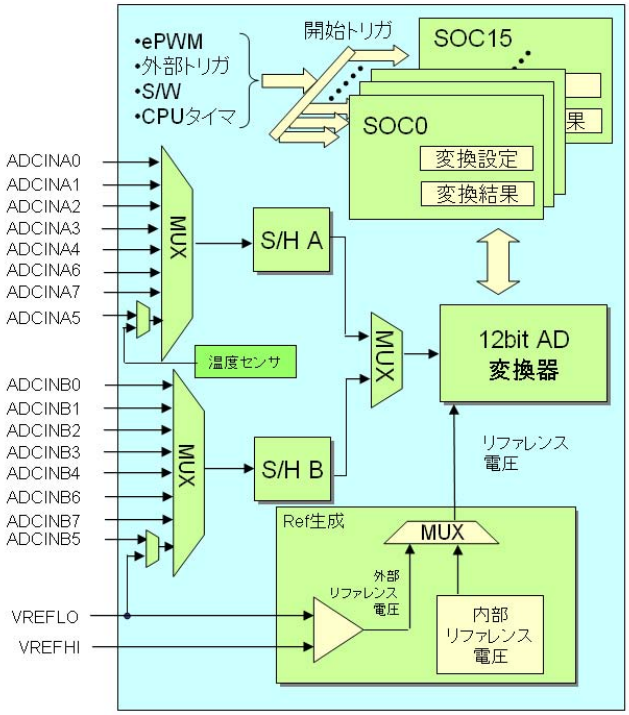


図 98:Piccolo の ADC ブロック図

PiccoloのADCは、入力としては、A0～A7とB0～B7の最大計16本の入力チャネルがあります。しかし、型番及びパッケージによっては、全てのADCINチャネルがピンに出ているわけではありませんので、ご使用のデバイスでは、どのADCINチャネルがピンに出ているかはご確認下さい。特徴として、

- A0～A7/B0～B7の最大16ch入力
- 12bit高速ＡＤＣ(最速のデバイスで216.67nsec変換時間です、デバイスにより異なりますので、各デバイスのデータシートを参照下さい)
- ２つのS/H回路による２入力同時サンプリング
- 各チャネルの変換シーケンスを柔軟に設定可能

- 温度センサ
- 内部リファレンス回路
- 16chの入力の変換設定を柔軟に個別に設定可能なSOC

等があげられます。このADCは、非常に柔軟な変換設定が出来る事が特徴な反面、一般的なＡＤＣの設定と少し異なっている部分もあります。その最大の特徴の変換設定を行うSOCについて説明します。このSOCは従来のF28xシリーズに搭載されていたシーケンサの役割を担うものです。SOCはSOC0～15と、同じモジュールが16個用意されていて、それぞれ個別に設定する事が可能です。このSOCはリファレンスガイドには明記していませんが、恐らくStart Of Conversionの略と思われます。

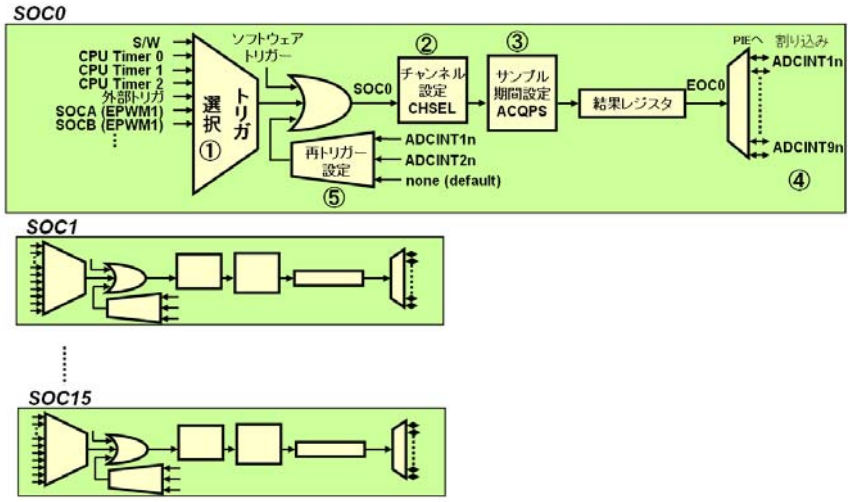


図 99:SOC のブロック図

図 99にSOCのブロック図を示します。SOCで設定する事は、
①どのトリガ信号でこのSOCを起動するか？
選択できるトリガ信号は、

- S/W(特定レジスタにCPUが値を書き込む事でトリガ発生)
- CPUタイマ0～3
- 外部トリガ信号
- EPWM1～xのSOCA/B信号(xはデバイスによって番号が異なります(EPWMの搭載本数が違うため))。

から、どれか1つを設定できます。

②どのチャネルを変換するか？
各SOCにつき、ひとつの変換チャネルを設定できます。A0～A7/B0～B7を自由に選択できます。他のSOCと同じ入力でもかまいません。

③サンプル・ホールド期間(S/H回路にチャージする時間)を何クロックにするか？
一般的なMCUに内蔵されているADCでは、サンプル・ホールド期間は一定かもしくは全チャネル一括設定が多いですが、このADCはSOC毎にサンプリング期間を設定する事ができます。設定範囲は、最大で７～64ADCCLKです。注意としましては、デバイスによって、この範囲が変わりますので、データシートを参照下さい(同じシリーズで変換時間が遅いデバイスは、このサンプリング期間の最小値が大きくなっています)。データシートにて規定されているサンプリング期間よりも短い設定ができてしまいますので、ご注意ください。また、7～64の中で、数点設定してはいけない値があります。ここはリファレンス・ガイドを参照下さい。

④変換終了割り込みを発生させるか？何番の割り込みにするか？

ここは、正確にはSOCレジスタの設定ではありませんが、使い方の意味を考えると、ここで解説するのが適しているので、ここで解説します。各SOCは変換終了後に割り込みをかける事ができます。ADC変換終了割り込みはADCINT1～9まで9本用意されています。各SOCにて変換終了したら、ADCINT1～9のうちどの割り込みをかけるかを設定する事ができます。

⑤変換終了割り込みによる再トリガーを受け付けるか？
変換開始トリガは、S/W, CPUタイマ,外部トリガ,ePWMから選択できますが、そのほかに、自分を含むSOC変換が終了した時にトリガをかけたいケースがあります。例えば、永久変換ループを組みたい時などです。各ＳＯＣは、ADCINT1かADCINT2（これらは、設定したSOCが変換終了した時に発生できます。④を参照してください）を変換開始トリガとして設定する事ができます。
⑥同時サンプリングを行うか？
２つのS/H回路を搭載しているため、2入力同時サンプリングができます。各SOCにてこの同時サンプリングを行うかどうかを設定できます。同時サンプリングで注意が必要な点として、同時にサンプリングできるペアが決まっている事です。同時にサンプリングできるのは、A0とB0、A1とB1といったように、AとBで同じ番号のペアのみです。ここはご注意下さい。
　それでは、少しSOCの設定例を挙げてみます。

1. SOC例１

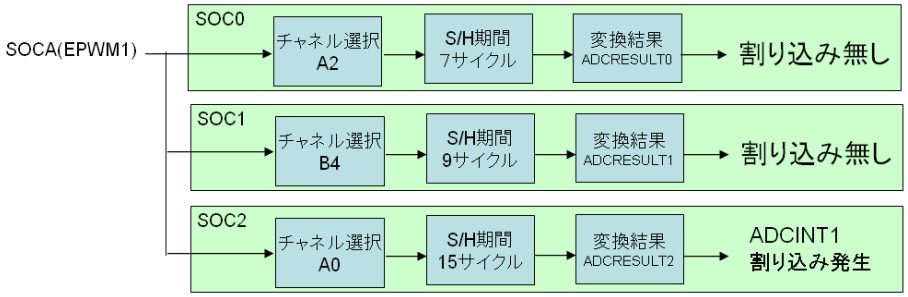


図 100:SOC 例１

図 100を見てください。この例では、SOC0,1,2の３つのSOCを設定しています。SOC0も1も2も全て、EPWM1のSOCAトリガ信号を変換開始トリガとして設定しています。となると、どのSOCから変換を開始するかという、アービトレーションの解説をしなければなりませんが、これは後ほど説明しますので、今は、SOC0→SOC1→SOC2の順番になると仮定しておきます。まず、SOC0が変換開始します。設定により、A2入力を7サンプル・ホールド期間で変換を開始します。変換結果は、SOCと同じ番号のADCRESULTxレジスタに格納されますので、SOC0の変換結果はADCRESULT0レジスタに格納されます。ここでは割り込みは発生させない設定になっています。続いて、SOC1がB4入力を9サンプル・ホールド期間で変換開始し、変換結果は、ADCRESULT1レジスタに格納されます。このSOC1も変換終了割り込みは発生させない設定です。続いてSOC2がA0入力を15サンプル・ホールド期間で変換開始し、変換結果はADCRESULT2に格納されます。このSOC2は、ADCINT1割り込みを発生させる設定になっていますので、このSOC2の変換が終了したら、ADCINT1割り込みが発生します。

2. SOC例2

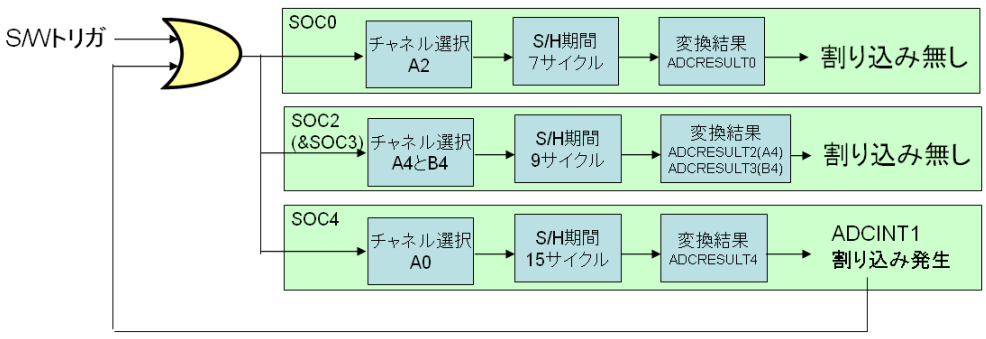


図 101:SOC 例 2

次の図 101を見てください。例1と大きく違うのは２箇所です。SOC2が同時サンプリングモードになっている事と、SOC4の変換終了割り込みADCINT1が再度変換トリガをかける設定になっています。同時サンプリングモードで重要な点は、先ほど解説したように、変換できるペアはAとBの同じチャネルだという点です。それから、2つの連続した番号(偶数番号とその番号+1のペア)のSOCを使います(設定は、一般的には偶数番号のSOCのみに行います)。

この例では、開始トリガはS/Wを設定しています。SOC0→SOC2(同時サンプリング)→SOC4と変換され、SOC4の変換が終了すると、ADCINT1割り込みが発生し、その割り込み信号が再度、SOC0/2/4の変換をトリガします。このように設定すると、最初にS/Wでトリガをかけると、永久（止めなければ）に連続変換する設定になります。

8.3 各ＳＯＣのプライオリティ制御

複数のSOCがそれぞれ独立した設定ができますが、変換器自体は１つしかないため、各SOCが同時に起動したり、あるSOCが起動している時に別のSOCが起動した時など、リソースの競合がおきます。これを解決するために、プライオリティ制御の機能があります。プライオリティ制御は主に次の２つのモードがあります。

- Round Robinモード
- High Priorityモード

まず、Round Robinモードから解説します。

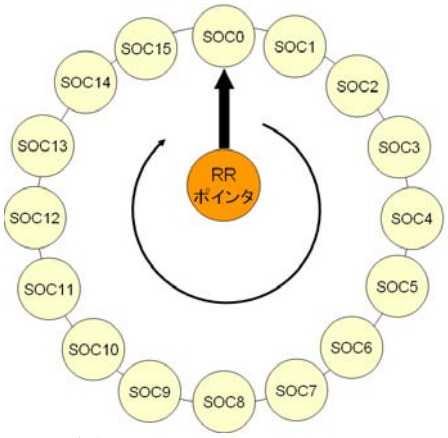


図 102:Round Robin モード

図 102にRound Robinモードの図を示します。SOCはSOC0～SOC15までの計16個がありますが、それらが時計周りに順番に実行されるのが、Round Robinモードになります。一番最近の変換がどのSOCで行われたかを示すのがRRポインタ(SOCPRICTLレジスタのRRPOINTERフィールド)です。このモードでは、このRRポインタがどこにあるかで変換順番が決定されます。複数のSOCリクエストが入ったり、複数の保留リクエストがある場合は、このRRポインタの右回りの順番に実行されます。図 103にRoubd Robinモードの動作例を示します。

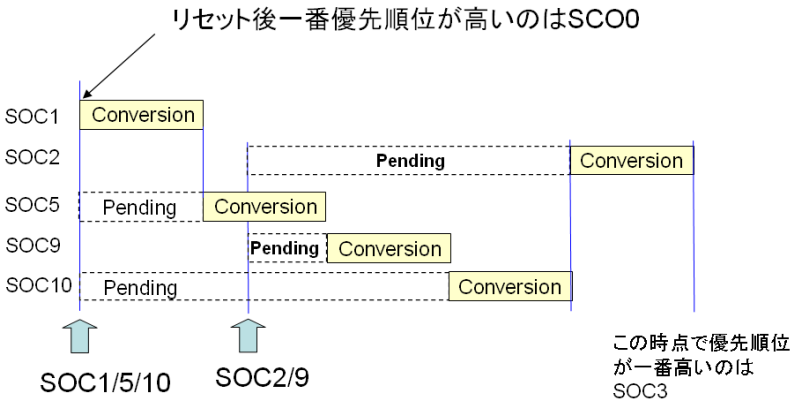


図 103:Round Robin モードの動作例

RRポインタは、リセット直後はSOC0が最初の権利があるように自動的に設定されます。図では、SOC1/2/5/9/10の5個のSOCがあるタイミングで変換開始トリガが来た例です。まず最初に、SOC1/5/9に変換トリガが入ります。最初の権利はSOC0にありますが、SOC0はリクエストが来ていないため、右回りで一番近いSOC1に権利が移り、変換が実行されます。この時、SOC5/10は保留になります。SOC1の変換が終了した時、保留になっているのは、SOC5/10です。SOC1の右回りで一番近いのはSOC5ですので、SOC5の変換が開始されます。SOC5が変換を行っている時に、SOC2/9のトリガがかかりました。SOC5が変換終了した時点では、保留になっているのは、SOC2/9/10になります。SOC5の右回りはSOC9になりますので、SOC9が次に変換されます。その次は同様にSOC10、最後にSOC2が実行されます。重要な点は、どの順番でトリガが入ったかではなく、次の変換を開始する時にどのSOCが保留になっていて、その時RRポインタがどこを指しているかで決定される点です。このように、どれがどのような順番で変換されるかは、わかりにくくなりますので、主な用途は変換タイミングがしっかりと決まっている主要な値ではなく、変換タイミングがそれほど重要ではない値の変換が主な使用方法となります。もしくは、非常に明快な設定を行う事で、それぞれの変換タイミングが確実になるよう設定するかです(例えば、ひとつのトリガで全てのSOCがトリガされるだけの設定等)。

次に、High Priorityモードを解説します。Round Ronbinモードは、それぞれが同じプライオリティであったのとは違い、各SOCのプライオリティが明確に決まっています。このモードでは、SOC0が一番プライオリティが高く、番号が大きくなるについれて、プライオリティが下がり、SOC15が一番プライオリティが低くなります。このプライオリティは固定で再設定はできません。一点注意しなくてはならないのは、あるSOCを変換中に、そのSOCよりもプライオリティが高いSOCトリガが入った場合は、今の変換を止めるわけではないという点です。あくまでも、変換が終了した時点で保留になっているSOCのプライオリティ、又は変換されていない時に複数のSOCトリガが同時に入った場合のプライオリティである事に注意してください。

High PriorityモードとRound Robinモードは、混在する事ができます。

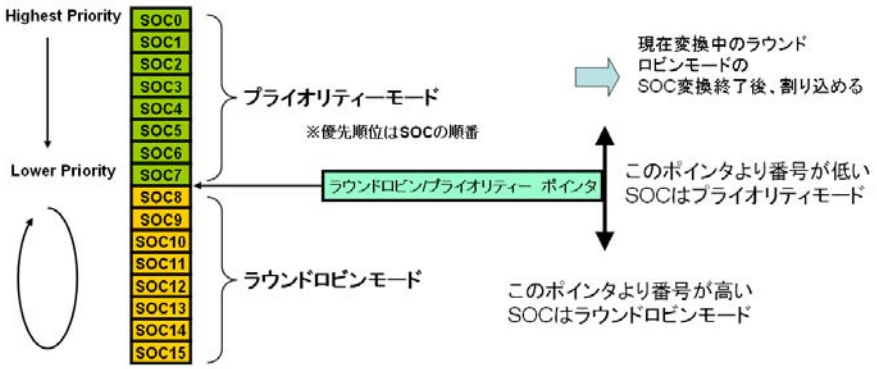


図 104:High Priority モードと Round Robin モードの混在

図 104にHigh PriorityモードとRound Robinモードを混在するケースを示します。この両モードを混在する時は、ラウンドロビン/プライオリティー ポインタ(SOCPRICTLレジスタのSOCPRIORITYフィールド)にてSOC0～15のどこかにその境界を設定します。このポインタより、小さい番号のSOCはHigh Priorityモードに、大きい番号のSOCはRoubd Robinモードに設定されます。それでは両モードを混在した時の動作例を図 105に示します。

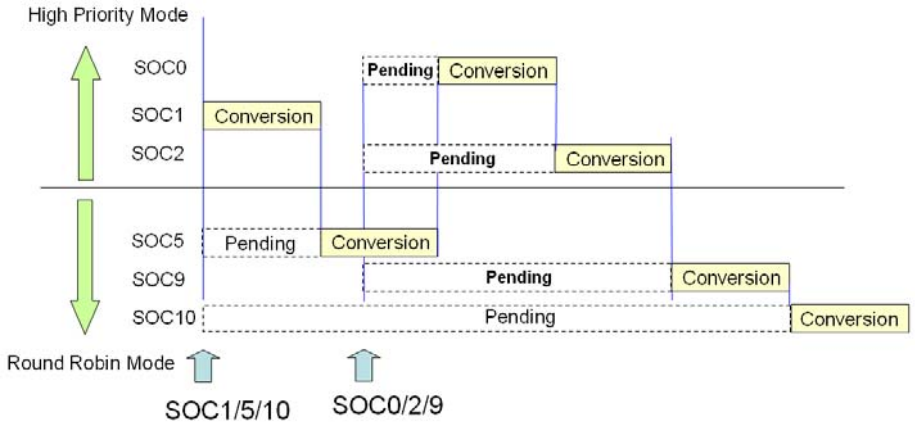


図 105:High Priority モードと Round Robin モードの混在動作例

図では、SOC0/1/2/5/9/10を使用しています。SOC0/1/2をHigh Priorityモードに、SOC5/9/10をRound Robinモードに設定しています。まずSOC1/5/10のトリガが入りました。SOC1がHigh Priorityモードになりますので、まずSOC1が変換されます。SOC1が終わった時点で保留になっているのはSOC5/10です。この２つはRound Robinモードです。Round Robinモードはリセット直後はSOC0が最初の権利がありますが、両モードが混在している時は、Round Robinモードの一番番号が小さいSOCが最初の権利をもっています。この例では、SOC5になりますので、SOC5が次に変換されます。このSOC5を変換中にSOC0/2/9のトリガが入りました。SOC5の変換が終了した時点で保留中なのは、SOC0/2/9/10です。SOC0/2がHigh Priorityモードですので、この２つが先に変換されます。High Priorityモードでは、番号が小さい方がプライオリティが高いですので、SOC0が変換され、その次にSOC2が変換されます。SOC2の変換終了後、保留中なのは、Round RobinモードのSOC9/10です。Roubd Robinモードでは、先ほどSOC5が変換されましたので、次の権利はSOC9になります。SOC9が変換され、次にSOC10が変換されます。

C28xアプリケーションでは、変換タイミングが重要なアプリケーションが多いですので、このような複雑な変換シーケンスを組む事は少ないかもしれませんが、このように柔軟に設定できるSOCですので、用途に応じて設定をして下さい。

8.4 ADC 使用例

それでは、ADCを実際に使用した例を示します。例として、先ほど説明した例を似たような図 106を例題にとります。一点、注意があります。Piccolo ADCのErrataに、Initial Conversionという不具合があります。この不具合では、開始トリガによって変換開始する、連続したSOCシーケンスにて、最初の変換結果が正しくない可能性があるというものです。例えば、あるトリガによって、SOC0→SOC1→SO2→SOC3と連続して変換されるとします。この場合、SOC0の変換結果は、正しくない可能性がありますので、破棄する必要があります。今回の例では、本当は4chだけ変換をすればよいのですが、この不具合のため、最初に、ダミーのSOCを入れています。SOC0→SOC1→SOC2(&SOC3)→SOC4となりますが、SOC0がダミーの変換です。この不具合は、リセット解除後の最初のAD変換値という意味ではありません。開始トリガ毎の最初の変換値が対象となります。また、今回の例では、最初にS/Wトリガを入れて、永久変換ループを組むことになります。この場合、一見、最初のS/Wトリガによる変換値のみが対象のように見えますが、SOC4の変換終了時のADCINT1発生による再トリガには、少しディレイがあり、この再トリガによる最初の変換も、この不具合の対象となります。

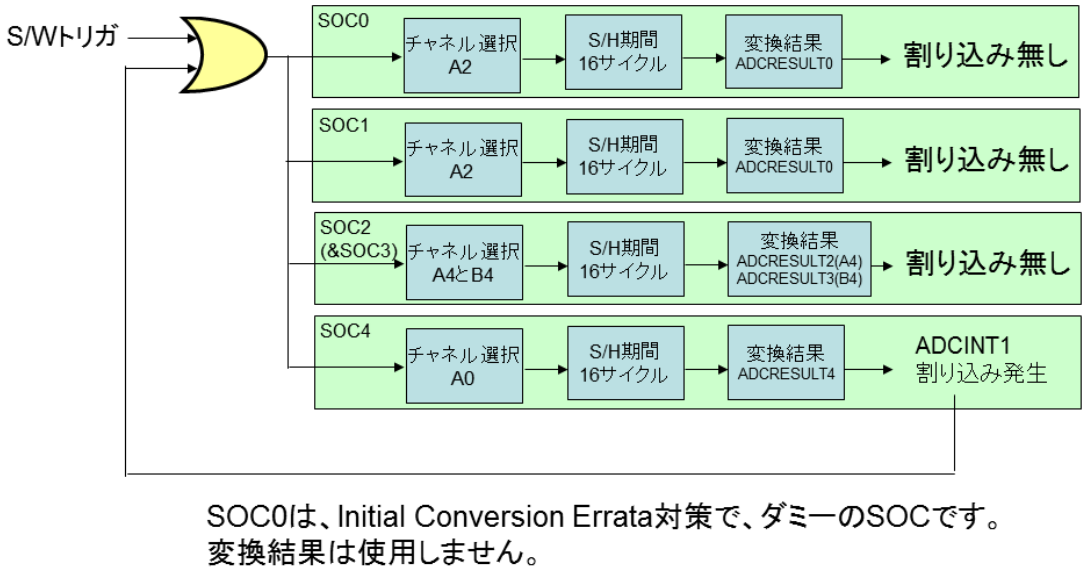


図 106:使用例(図 101 と同じ)

さて、Projectは前のものをコピーして修正しても、新規に作成してもどちらでもかまいません、Projectの設定は以下のとおりです。今までと基本的に同じですが、念のため書いておきます。PWMのProjectをコピーしている場合は、F2806xの場合はCPU動作周波数が60MHz設定になっているはずです。60MHz動作のままでもかまいませんが、80MHz動作に戻してもかまいません。

新規に作成する場合は、以下をProjectディレクトリの下にコピーして下さい。

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ
F2802xの場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_commonフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_headersフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\DSP28x_Project.h C:\ti\contorlSUITE\device_support\f2802x\v200\F2802x_Device.h

ヘッダ・ファイルのExclude設定は以下の通りです。

F28069の場合	F2806x_common ― cmd F28069.cmdを残して、他は全てExclude F2806x_common ― lib 全てExclude F2806x_common ― source F2806x_Adc.c F2806x_CodeStartBranch.asm F2806x_DefaultIsr.c F2806x_PieCtrl.c F2806x_PieVect.c F2806x_SysCtrl.c F2806x_usDelay.asm を残して他は全てExclue

	F2806x_headers ― cmd 　　F2806x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28069_RAM_lnk.cmdをExclude
F28035の場合	DSP2803x_common ― cmd 　　F28035.cmdを残して、他は全てExclude DSP2803x_common ― lib 　　全てExclude DSP2803x_common ― source 　　DSP2803x_Adc.c 　　DSP2803x_CodeStartBranch.asm 　　DSP2803x_DefaultIsr.c 　　DSP2803x_PieCtrl.c 　　DSP2803x_PieVect.c 　　DSP2803x_SysCtrl.c 　　DSP2803x_usDelay.asm 　　を残して他は全てExclue DSP2803x_headers ― cmd 　　DSP2803x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28035_RAM_lnk.cmdをExclude
F28027の場合	DSP2802x_common ― cmd 　　F28027.cmdを残して、他は全てExclude 　　FlashProjectの時と同じく、 　　SECTIONS{}欄内の 　　**.text**　　: >> FLASHA | FLASHC | FLASHD.　**PAGE** = 0 　　の行にて、FLASHDの次の文字がドットになっていますが、コンマが正しいです。 　　**.text**　　: >> FLASHA | FLASHC | FLASHD,　**PAGE** = 0 　　ここを修正しないと、Warningがでますので、修正して下さい。 DSP2802x_common ― lib 　　全てExclude DSP2802x_common ― source 　　DSP2802x_Adc.c 　　DSP2802x_CodeStartBranch.asm 　　DSP2802x_DefaultIsr.c 　　DSP2802x_PieCtrl.c 　　DSP2802x_PieVect.c 　　DSP2802x_SysCtrl.c

	DSP2802x_usDelay.asm を残して他は全てExclue DSP2802x_headers ― cmd DSP2802x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28027_RAM_lnk.cmdをExclude

インクルード・パスの設定は今までと同じです。

F28069の場合	F2806x_common ― include F2806x_headers ― include
F28035の場合	DSP2803x_common ― include DSP2803x_headers ― include
F28027の場合	F2802x_common ― include F2802x_headers ― include 一番上のプロジェクト名(今回の例では、PwmProject1つまり、プロジェクト・フォルダ直下のディレクトリです)。

それでは、コードを示します。

Main.c

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(AdcIsr, "ramfuncs");

interrupt void AdcIsr(void);
void AdcConfig(void);

void main(void){

  DINT;
  InitSysCtrl();
  memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();
  InitPieCtrl();
  IER=0x0000;
  IFR=0x0000;
  InitPieVectTable();
  EALLOW;
  PieVectTable.ADCINT1  = AdcIsr;
  EDIS;

  EINT;
  AdcConfig();
  AdcRegs.ADCSOCFRC1.all = 0x0017; // Start SOC0, 1, 2(&3), 4
```

```
    while(1);

}

void AdcConfig(void){

    InitAdc();
    EALLOW;

    //ADC Control Register Configuration
    AdcRegs.ADCCTL1.bit.INTPULSEPOS = 1;            //Disable early interrupt trigger

    //Interrupt/Overflow Flag clear
    AdcRegs.ADCINTFLGCLR.all = 0x1FF;      //ADC Int flag
    AdcRegs.ADCINTOVFCLR.all = 0x1FF;     //ADC Int ovfl clear

    //ADCINT1 interrupt configuration
    AdcRegs.INTSEL1N2.bit.INT1CONT = 1;     // Enable Continuous mode
    AdcRegs.INTSEL1N2.bit.INT1E = 1;            // ADCINT1 Enable
    AdcRegs.INTSEL1N2.bit.INT1SEL = 4;       // Select EOC4 for ADCINT1 trigger

    //SOCPRICTL configuration
    AdcRegs.SOCPRICTL.bit.ONESHOT = 0;         // Disable oneshot mode
    AdcRegs.SOCPRICTL.bit.SOCPRIORITY = 0;  // All ch is Round Robin mode

    //Simultaneous sampling mode configuration
    AdcRegs.ADCSAMPLEMODE.bit.SIMULEN0 = 0; // CH0 & 1: Single conversion mode
    AdcRegs.ADCSAMPLEMODE.bit.SIMULEN2 = 1; // CH2 & 3: Simultaneous conversion mode
    AdcRegs.ADCSAMPLEMODE.bit.SIMULEN4 = 0; // CH4 & 5: Single conversion mode

    //ADCINTSOCSEL configuration
    AdcRegs.ADCINTSOCSEL1.bit.SOC0 = 1;     // ADCINT1 will trigger SOC0
    AdcRegs.ADCINTSOCSEL1.bit.SOC1 = 1;    // ADCINT1 will trigger SOC1
    AdcRegs.ADCINTSOCSEL1.bit.SOC2 = 1;     // ADCINT1 will trigger SOC2
    AdcRegs.ADCINTSOCSEL1.bit.SOC4 = 1;     // ADCINT1 will trigger SOC4

    //SOC overflow clear
    AdcRegs.ADCSOCOVFCLR1.all = 0xFFFF;

    //SOC0 configuration(dummy : See Initial Conversion Errata)
    AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 0;    // Software trigger
    AdcRegs.ADCSOC0CTL.bit.CHSEL = 2;       // ADCINA2
    AdcRegs.ADCSOC0CTL.bit.ACQPS = 15;      // Sample window = 16

    //SOC1 Configuration
    AdcRegs.ADCSOC1CTL.bit.TRIGSEL = 0;    // Software trigger
    AdcRegs.ADCSOC1CTL.bit.CHSEL = 2;       // ADCINA2
    AdcRegs.ADCSOC1CTL.bit.ACQPS = 15;     // Sample window = 16
```

```
   //SOC2(&SOC3) configuration--simultaneous sampling
   AdcRegs.ADCSOC2CTL.bit.TRIGSEL = 0;   // Software trigger
   AdcRegs.ADCSOC2CTL.bit.CHSEL = 4;     // ADCINA4 and B4
   AdcRegs.ADCSOC2CTL.bit.ACQPS = 15;     // Sample window = 16

   //SOC4 configuration
   AdcRegs.ADCSOC4CTL.bit.TRIGSEL = 0;   // Software trigger
   AdcRegs.ADCSOC4CTL.bit.CHSEL = 0;     // ADCINA0
   AdcRegs.ADCSOC4CTL.bit.ACQPS = 15;    // Sample window = 16

   PieCtrlRegs.PIEIER1.bit.INTx1 = 1;
   IER |= M_INT1;

   EDIS;

}


unsigned int BufResult1[10];
unsigned int BufResult2[10];
unsigned int BufResult3[10];
unsigned int BufResult4[10];

interrupt void AdcIsr(void){

   static int count = 0;

   BufResult1[count] = AdcResult.ADCRESULT1;
   BufResult2[count] = AdcResult.ADCRESULT2;
   BufResult3[count] = AdcResult.ADCRESULT3;
   BufResult4[count] = AdcResult.ADCRESULT4;

   if(count++ > 9){
      count = 0;
   }
   AdcRegs.ADCINTFLGCLR.all = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

それではADCの設定関数、AdcConfig()から解説していきます。

AdcConfig()関数

```
void AdcConfig(void){

   InitAdc();
   EALLOW;

   //ADC Control Register Configuration
```

```
AdcRegs.ADCCTL1.bit.INTPULSEPOS = 1;            //Disable early interrupt trigger

//Interrupt/Overflow Flag clear
AdcRegs.ADCINTFLGCLR.all = 0x1FF;      //ADC Int flag
AdcRegs.ADCINTOVFCLR.all = 0x1FF;     //ADC Int ovfl clear

//ADCINT1 interrupt configuration
AdcRegs.INTSEL1N2.bit.INT1CONT = 1;     // Enable Continuous mode
AdcRegs.INTSEL1N2.bit.INT1E = 1;          // ADCINT1 Enable
AdcRegs.INTSEL1N2.bit.INT1SEL = 4;       // Select EOC4 for ADCINT1 trigger

//SOCPRICTL configuration
AdcRegs.SOCPRICTL.bit.ONESHOT = 0;        // Disable oneshot mode
AdcRegs.SOCPRICTL.bit.SOCPRIORITY = 0;  // All ch is Round Robin mode

//Simultaneous sampling mode configuration
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN0 = 0; // CH0 & 1: Single conversion mode
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN2 = 1; // CH2 & 3: Simultaneous conversion mode
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN4 = 0; // CH4 & 5: Single conversion mode

//ADCINTSOCSEL configuration
AdcRegs.ADCINTSOCSEL1.bit.SOC0 = 1;     // ADCINT1 will trigger SOC0
AdcRegs.ADCINTSOCSEL1.bit.SOC1 = 1;    // ADCINT1 will trigger SOC1
AdcRegs.ADCINTSOCSEL1.bit.SOC2 = 1;     // ADCINT1 will trigger SOC2
AdcRegs.ADCINTSOCSEL1.bit.SOC4 = 1;     // ADCINT1 will trigger SOC4

//SOC overflow clear
AdcRegs.ADCSOCOVFCLR1.all = 0xFFFF;

//SOC0 configuration(dummy : See Initial Conversion Errata)
AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 0;    // Software trigger
AdcRegs.ADCSOC0CTL.bit.CHSEL = 2;       // ADCINA2
AdcRegs.ADCSOC0CTL.bit.ACQPS = 15;      // Sample window = 16

//SOC1 Configuration
AdcRegs.ADCSOC1CTL.bit.TRIGSEL = 0;    // Software trigger
AdcRegs.ADCSOC1CTL.bit.CHSEL = 2;       // ADCINA2
AdcRegs.ADCSOC1CTL.bit.ACQPS = 15;     // Sample window = 16

//SOC2(&SOC3) configuration--simultaneous sampling
AdcRegs.ADCSOC2CTL.bit.TRIGSEL = 0;    // Software trigger
AdcRegs.ADCSOC2CTL.bit.CHSEL = 4;      // ADCINA4 and B4
AdcRegs.ADCSOC2CTL.bit.ACQPS = 15;      // Sample window = 16

//SOC4 configuration
AdcRegs.ADCSOC4CTL.bit.TRIGSEL = 0;    // Software trigger
AdcRegs.ADCSOC4CTL.bit.CHSEL = 0;       // ADCINA0
AdcRegs.ADCSOC4CTL.bit.ACQPS = 15;     // Sample window = 16
```

```
    PieCtrlRegs.PIEIER1.bit.INTx1 = 1;
    IER |= M_INT1;

    EDIS;

}
```

まず、最初に

```
    InitAdc();
```

と、InitAdc()関数をコールしています。この関数は、

F28069の場合	F2806x_common\soruce\F2806x_Adc.c
F28035の場合	DSP2803x_common\source\DSP2803x_Adc.c
F28027の場合	F2802x_common\source\F2802x_Adc.c

にある関数で、ADCのパワーアップを行うための関数で、使う前に必ずコールして下さい。F2806xの場合は、パワーアップの他に、ADCクロックをCPUクロックの1/2に設定しています。これは、F2806xのADCクロックの最大値が40MHz(SPRS698Aから抜粋です。更新される可能性がありますので、必ず最新のデータシートをご参照下さい)のためです。ちなみに、F2802xとF2803xのRev.0シリコンはADCクロックは必ずCPUクロックと同じ(同じにしか設定できません)ですが、Rev.Aシリコン以降は、CPUクロックの1/2に設定する事ができるようになっています（この背景の詳細は、各シリーズのErrataをご参照下さい）。F2802xとF2803xの場合は、この関数では、パワーアップのみです。

次にEALLOWを発行しています。ADCレジスタはEALLOW保護がかかっているレジスタがあるためです。

次の

```
AdcRegs.ADCCTL1.bit.INTPULSEPOS = 1;        //Disable early interrupt trigger
```

ですが、このビットは、ADC早期割り込みを許可するかどうかを設定しています。割り込みが起きる場合、各モジュールからリクエストが入ってもCPUは直ぐに反応できず（パイプライン動作をしているためすぐに実行を移せなかったり、コンテキストセーブがあるためです）、多少のディレイがあります。ADCの場合で考えてみますと、ADCの割り込みは変換終了割り込みです。変換が終了してから割り込みリクエストがかかり、それから少しディレイがあって、CPUが割り込み処理ルーチンを実行します。このディレイがもったいない、とＴＩは考えたわけです。そのため、変換終了してから割り込みリクエストをするのではなく、変換を開始する時に割り込みリクエストを行い、CPUがそれに反応するディレイの間に変換を行い、ＣＰＵが割り込み処理ルーチンに入る頃に丁度、変換が終わっている、というのが最初のコンセプトでした。しかし、F2802x/F2803xでCPUクロック=ADCクロック設定の場合に合わせて設計していたため、F2806xやF2802x/F2803xのRev.Aの1/2モードの時のように、CPUクロック≠ADCクロック設定の場合は、それがうまくいきません。割り込み処理ルーチンに入っても変換がまだ終わっていない可能性があります。この機能を使うときは、変換終了フラグを確認してから、変換値をとりにいくべきでしょう。今回は、この早期割り込みは使わずに、一般的な変換終了した時点で割り込みリクエストがかかるように設定しました。このビットの説明を表 29に示します。

フィールド名	設定値	意味	設定値	意味
INTPULSEPOS	0	変換を開始した時に割り込みリクエストをかける（早期割り込み）	1	変換が終了した時に割り込みをかける。(正確には、結果レジスタに変換結果が格納される1サイクル前)

表 29:ADCCTL1 レジスタの INTPULSEPOS フィールド

次に、ＡＤＣ割り込みフラグと、割り込みオーバーフロー・フラグのクリアを行っています。これらのレジスタは、リセット解除後はもちろんクリアされていますので、念のためという意味合いが強いです。

```
//Interrupt/Overflow Flag clear
AdcRegs.ADCINTFLGCLR.all = 0x1FF;      //ADC Int flag
AdcRegs.ADCINTOVFCLR.all = 0x1FF;      //ADC Int ovfl clear
```

ADCはADCINT1～ADCINT9の9本の割り込みをかける事ができ、それぞれ、独立したPIE割り込み番号を持っています。この9本のうちADCINT1とADCINT2は他の７本と違い、SOC再トリガをかける事ができる特別な割り込み信号です。

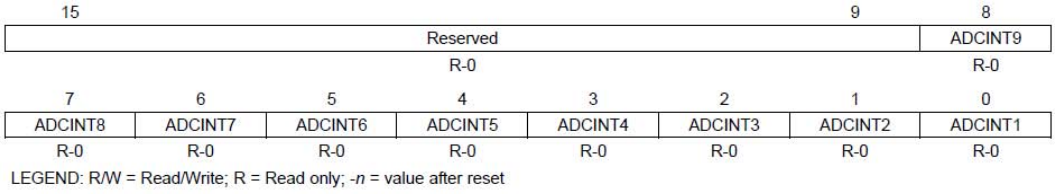


図 107:ADCINTFLG レジスタ

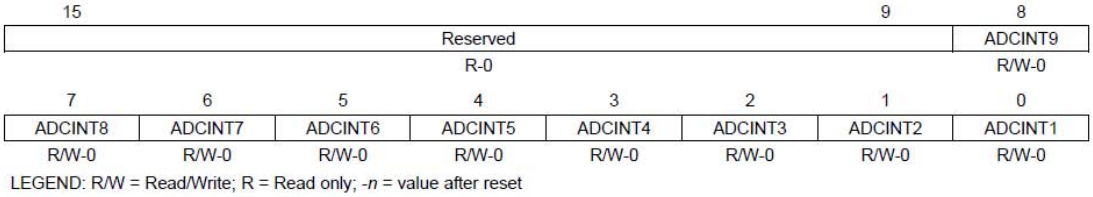


図 108:ADCINTFLGCLR レジスタ(1 を書く事で ADCINTFLG レジスタの対応するビットがクリア)

図 107にADC割り込みフラグレジスタ(ADCINTFLG)を、図 108にそのクリア・レジスタ(ADCINTFLG)を示します。ADCINTx割り込みがリクエストされると、ADCINTFLGレジスタの対応するビットに1が立ちます。このビットをクリアするためには、ADCINTFLGCLRレジスタの対応するビットに1を書きます。今回は初期化のため、全てのビットに1をたてて、全てクリアしています。

次のオーバーフロー・フラグですが、割り込みが保留中に、再度割り込みが入ったかどうかを示すフラグです（要は処理が間に合わなかった場合です）。図 109にADC割り込みオーバーフロー・レジスタ(ADCINTOVF)を、図 110にそのクリア・レジスタを示します。割り込み保留中(ADCINTFLGの対応するビットが1である時)に、同じ割り込みリクエストが起きた場合に、ADCINTOVFレジスタの対応するビットが1になります。これをクリアするには、ADCINTOVFCLRレジスタの対応するビットに1を書き込みます。今回は初期化のため、全てのビットに1を書き、全てをクリアしてます。

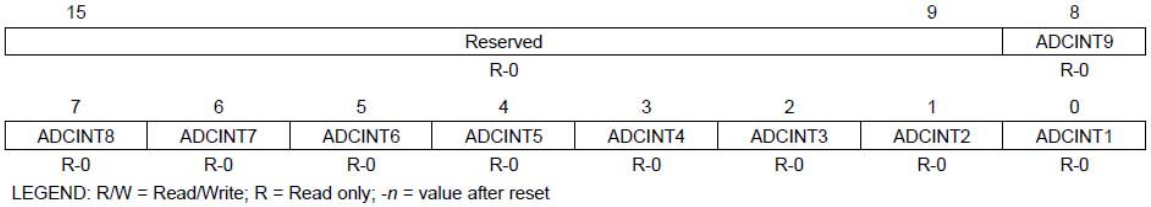


図 109:ADCINTOVF レジスタ

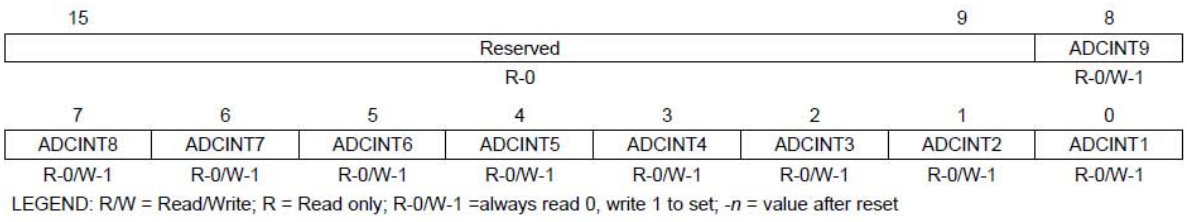


図 110:ADCINTOVFCLR レジスタ(１を書く事で ADCINTOVF レジスタの対応するビットがクリア)

次に、ADCINT1の設定を行います。今回のコードでは図 106より、SOC4の変換にてADCINT1割り込みをかけます。これらの設定はINTSELxNyというレジスタで行います(xとyが番号で、2つのADCINTxの設定が1つレジスタで行えます。例えば、INTSEL1N2は、ADCINT1とADCINT2の設定です)。今回は、ADCINT1の設定ですので、INTSEL1N2レジスタにて設定を行います。

```
//ADCINT1 interrupt configuration
AdcRegs.INTSEL1N2.bit.INT1CONT = 1;      // Enable Continuous mode
AdcRegs.INTSEL1N2.bit.INT1E = 1;         // ADCINT1 Enable
AdcRegs.INTSEL1N2.bit.INT1SEL = 4;       // Select EOC4 for ADCINT1 trigger
```

まず、INT1CONTフィールドを説明します。表 30にその詳細を示します。

フィールド名	設定値	意味	設定値	意味
INTxCONT (今回使っているのは INT1CONT)	0	ADCINTFLG レジスタのADCINTxフラグがたっている場合は、ADC割り込みリクエストを出さない	1	ADC割り込みリクエストを毎回出す

表 30:INTSELxNy レジスタの INTxCONT フィールド

このビットは、ADC割り込みリクエストが保留中でも、再度割り込みリクエストを出すかどうかを設定します。今回のようなシンプルな例では、通常動作時では保留中に再度割り込みが起きるような事はありません。今回はデバッグ目的で1を書いています。デバッグしている場合は、ブレークポイントにて停止したりします。0で設定してしまうと、ブレーク中でも、ADCは動いていますので、後続割り込みが発生しなくなってしまいます。

次のINT1Eフィールドですが、このフィールドは、単にADC割り込みの許可ビットです（表 31）。今回はADCINT1割り込みを使いますので、1を設定しています。

フィールド名	設定値	意味	設定値	意味
INTxE (今回使っているのは INT1E)	0	ADCINTx割り込みの不許可	1	ADCINTx割り込みの許可

表 31:INTSELxNy レジスタの INTxE フィールド

次のINT1SELフィールドですが、ここは、どのSOCがこの割り込みを発生させるかを設定します。EOCxはSOCxの変換終了タイミングです(早期割り込みを使っている場合は、SOCxの変換開始タイミング)。

フィールド名	設定値	意味	設定値	意味
INTxSEL (今回使っているのは INT1SEL)	00h	EOC0	01h	EOC1
	02h	EOC2	03h	EOC3
	04h	EOC4	05h	EOC5
	06h	EOC6	07h	EOC7
	08h	EOC8	09h	EOC9
	0Ah	EOC10	0Bh	EOC11
	0Ch	EOC12	0Dh	EOC13
	0Eh	EOC14	0Fh	EOC15
	1xh	禁止		

表 32：INTSELxNy レジスタの INTxSEL フィールド

今回のコードでは、SOC4の終了にてADCINT1を発生させますので、このINT1SELフィールドは4に設定します。

次にSOCのプライオリティに関するモードを設定します。各SOCには、Round RobinモードとHigh Priorityモードの２種類ある事は既に解説しました。今回は、全てRound Robinモードを使用してみます。この設定を行うのがSOCPRICTLレジスタです。

```
//SOCPRICTL configuration
AdcRegs.SOCPRICTL.bit.ONESHOT = 0;     // Disable oneshot mode
AdcRegs.SOCPRICTL.bit.SOCPRIORITY = 0;          // All ch is Round Robin mode
```

まず、ONESHOTフィールドですが、これは、One shotモードの許可/不許可を設定します(表 33)。One shotモードは、１回だけ変換を行うモードで、最初に設定したトリガがはいると変換を行いますが、そのトリガが再度入っても変換は行われません。今回はこのモードは使いませんので、0(不許可)に設定しています。このONESHOTビット・フィールドはF2802x/F2803xの場合は、Rev.0シリコンには存在しなく(このビットは予約扱いでONESHOTモードはありません)、Rev.Aのみ存在するビットです。0を書いていますので、Rev.0の場合でも影響はありません。また、F2802x/F2803xのADCリファレンス・ガイドにはこのONESHOTビット・フィールドの情報はまだ記載されていなく、Errataに記載されています。詳細はこちらをご参照下さい。

フィールド名	設定値	意味	設定値	意味
ONESHOT	0	One shotモードの不許可	1	One shotモードの許可

表 33:SOCPRICTL レジスタの ONESHOT フィールド(F2802x/2803x Rev.0 シリコンでは予約)

次のSOCPRIORITYフィールドは、Round RobinとHigh Priorityモードの境界を決めます。その設定内容を表 34に示します。今回は、全てのSOCをRound Robinモードで使用しますので、0を設定しています。

フィールド名	設定値	意味	設定値	意味
SOCPRIORITY	00h	全てのSOCはRound Robinモード	01h	SOC0:High Priorityモード SOC1～15: Round Robinモード
	02h	SOC0～1:High Priorityモード SOC2～15:Round Robinモード	03h	SOC0～2:High Priorityモード SOC3～15:Round Robinモード
	04h	SOC0～3:High Priorityモード SOC4～15:Round Robinモード	05h	SOC0～4:High Priorityモード SOC5～15:Round Robinモード
	06h	SOC0～5:High Priorityモード SOC6～15:Round Robinモード	07h	SOC0～6:High Priorityモード SOC7～15:Round Robinモード
	08h	SOC0～7:High Priorityモード SOC8～15:Round Robinモード	09h	SOC0～8:High Priorityモード SOC9～15:Round Robinモード
	0Ah	SOC0～9:High Priorityモード SOC10～15:Round Robinモード	0Bh	SOC0～10:High Priorityモード SOC11～15:Round Robinモード
	0Ch	SOC0～11:High Priorityモード SOC12～15:Round Robinモード	0Dh	SOC0～12:High Priorityモード SOC13～15:Round Robinモード
	0Eh	SOC0～13:High Priorityモード SOC14～15:Round Robinモード	0Fh	SOC0～14:High Priorityモード SOC15:Round Robinモード
	10h	全てのSOCはHigh Priorityモード	その他	禁止

表 34:SOCPRICTL レジスタの SOCPRIORITY フィールド

次の

```
//Simultaneous sampling mode configuration
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN0 = 0; // CH0 & 1: Single conversion mode
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN2 = 1; // CH2 & 3: Simultaneous conversion mode
AdcRegs.ADCSAMPLEMODE.bit.SIMULEN4 = 0; // CH4 & 5: Single conversion mode
```

ですが、同時サンプリング・モードかシングル・サンプリングモードかの設定を行っています。この設定を行うのが、ADCSAMPLEMODEレジスタです。ADCSAMPLEMODEレジスタを図 111に、その中のSIMULENxフィールドを表 35に示します。

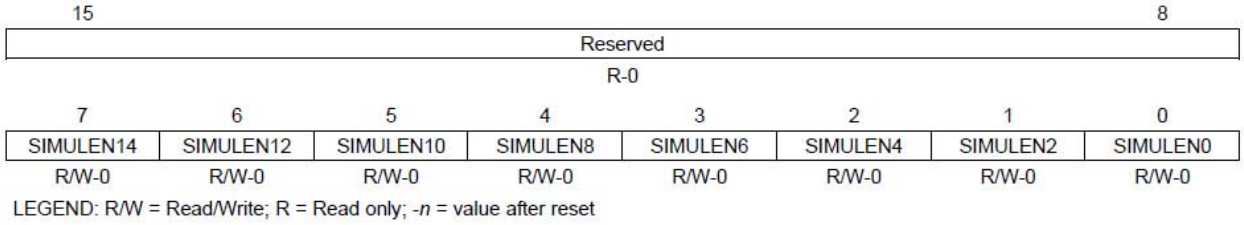


図 111:ADCSAMPLEMODE レジスタ

フィールド名	設定値	意味	設定値	意味
SIMULENx	0	SOCxとSOC(x+1)はシングル・サンプリング・モード	1	SOCxとSOC(x+1)は同時サンプリングモード

表 35:ADCSAMPLEMODE レジスタの SIMULENx フィールド

今回は、
SOC0：シングル・サンプリング・モード（Initial Conversion Errata対策のダミーSOC）
SOC1: シングル・サンプリング・モード
SOC2(と3) :同時サンプリング・モード
SOC4:シングル・サンプリング・モード
で使いますので、SIMULEN0とSIMULEN4は0に、SIMULEN2は1に設定しています。

次の

```
//ADCINTSOCSEL configuration
AdcRegs.ADCINTSOCSEL1.bit.SOC0 = 1;    // ADCINT1 will trigger SOC0
AdcRegs.ADCINTSOCSEL1.bit.SOC1 = 1;    // ADCINT1 will trigger SOC1
AdcRegs.ADCINTSOCSEL1.bit.SOC2 = 1;    // ADCINT1 will trigger SOC2
AdcRegs.ADCINTSOCSEL1.bit.SOC4 = 1;    // ADCINT1 will trigger SOC4
```

は、ADCINT1の再トリガを行うかの設定をしています。これを行うのが、ADCINTSOCSEL1レジスタとADCINTSOCSEL2レジスタです。ADCINT1～9のうち、ADCINT1とADCINT2だけが、SOCに再トリガをかける事ができます。ADCINTSOCSEL1/2レジスタを図 112と図 113に示します。ADCINTSOCSEL1がSOC0～SOC7用で、ADCINTSOCSEL2がSOC8～SOC15用です。その中のSOCxフィールドを表 36に示します。

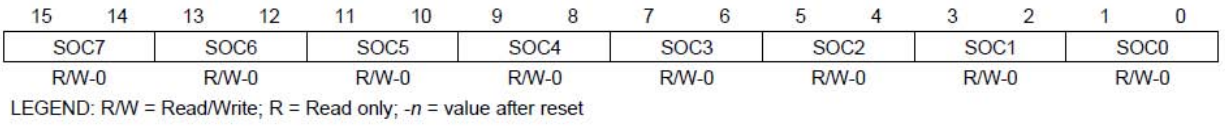


図 112:ADCINTSOCSEL1 レジスタ

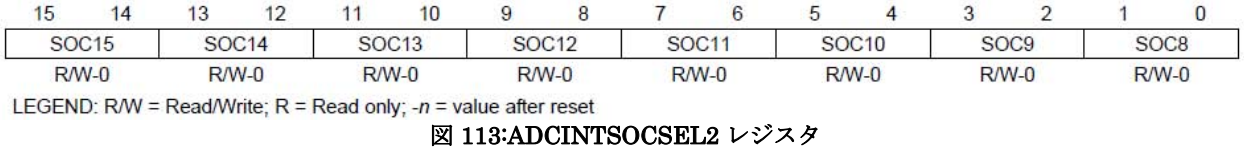


図 113:ADCINTSOCSEL2 レジスタ

フィールド名	設定値	意味	設定値	意味
SOCx	00b	ADCINTはSOCxをトリガしません。このSOCxをトリガするのは、ADCSOCxCTLレジスタのTRIGSELで決定されます。	01b	ADCINT1がSOCxをトリガします。ADCSOCxCTLレジスタのTRIGSELは無視されます。

	10b	ADCINT2がSOCxをトリガします。ADCSOCxCTLレジスタのTRIGSELは無視されます。	11b	禁止

表 36:ADCINTSOCSEL1/2 レジスタの SOCx フィールド

今回の例では、SOC4の変換終了でADCINT1を発生させ、そのADCINT1が再度SOC0/1/2/4をトリガする形になりますので、SOC0、SOC1、SOC2、SOC4フィールドをそれぞれ1に設定しています。

次の

```
//SOC overflow clear
AdcRegs.ADCSOCOVFCLR1.all = 0xFFFF;
```

は、SOCオーバーフロー・フラグのクリアをしています。先ほどは割り込みのオーバーフロー・フラグがありましたが、今度はSOCのオーバーフロー・フラグです。各SOCは設定した何らかのトリガ信号により変換を開始しますが、必ずしもすぐに変換されるわけではありません。他のSOCが変換中だっりすると、保留状態になります。この時保留中のSOCに再度トリガが発生した場合は、このSOCオーバーフロー・フラグがたちます。このフラグは、ADCSOCOVF1レジスタにあります。ちなみに、何故か1しかありません。恐らく単なるネーミング・ミスでしょう。ADCSOCOVF1レジスタを図 114に示します。この対応するビットが1の時は、オーバーフローが発生した事を意味します。このADCSOCOVF1レジスタのフラグをクリアするのが、ADCSOCOVFCLR1レジスタです。対応するビットに1を書くと、ADCSOCOVF1レジスタの対応するフラグがクリアされます。今回は念のための初期化で、ADCSOCOVFCLR1レジスタの全てのビットに1を書き、ADCSOCOVF1レジスタをクリアしています。

15	14	13	12	11	10	9	8
SOC15	SOC14	SOC13	SOC12	SOC11	SOC10	SOC9	SOC8
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
7	6	5	4	3	2	1	0
SOC7	SOC6	SOC5	SOC4	SOC3	SOC2	SOC1	SOC0
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 114:ADCSOCOVF1 レジスタ

15	14	13	12	11	10	9	8
SOC15	SOC14	SOC13	SOC12	SOC11	SOC10	SOC9	SOC8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
SOC7	SOC6	SOC5	SOC4	SOC3	SOC2	SOC1	SOC0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 115:ADCSOCOVFCLR1 レジスタ

次の、

```
//SOC0 configuration(dummy : See Initial Conversion Errata)
AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 0;   // Software trigger
AdcRegs.ADCSOC0CTL.bit.CHSEL = 2;     // ADCINA2
AdcRegs.ADCSOC0CTL.bit.ACQPS = 15;    // Sample window = 16

//SOC1 Configuration
AdcRegs.ADCSOC1CTL.bit.TRIGSEL = 0;   // Software trigger
AdcRegs.ADCSOC1CTL.bit.CHSEL = 2;     // ADCINA2
AdcRegs.ADCSOC1CTL.bit.ACQPS = 15;    // Sample window = 16

//SOC2(&SOC3) configuration--simultaneous sampling
AdcRegs.ADCSOC2CTL.bit.TRIGSEL = 0;   // Software trigger
```

```
AdcRegs.ADCSOC2CTL.bit.CHSEL = 4;     // ADCINA4 and B4
AdcRegs.ADCSOC2CTL.bit.ACQPS = 15;    // Sample window = 16

//SOC4 configuration
AdcRegs.ADCSOC4CTL.bit.TRIGSEL = 0;   // Software trigger
AdcRegs.ADCSOC4CTL.bit.CHSEL = 0;     // ADCINA0
AdcRegs.ADCSOC4CTL.bit.ACQPS = 15;    // Sample window = 16
```

ですが、各SOCの設定を行っています。各SOCの設定は、ADCSOCxCTLレジスタ(xはSOC番号)にて行います。SOC2は同時サンプリングモードになっています。同時サンプリングの場合は、SOCx(今回はSOC2)とSOCx+1(今回はSOC3)の２つのSOCが使われますが、設定は偶数番号のみ(今回はSOC2)行います。奇数番号(今回はSOC3)は設定をする必要はありません

まず、最初にTRIGSELフィールドを設定します。実は今回は、ADCINT1の再トリガを設定していますので、このフィールドの設定は無視されます。しかし、このドキュメントにこのフィールドの説明を記述したかったので登場させました。ご了承下さい。このフィールドにて、何の信号がこのSOCをトリガするかを決定します。このフィールドの詳細を表 37に示します。

フィールド名	設定値	意味	設定値	意味
TRIGSEL	00h	ソフトウェア・トリガのみ	01h	CPUタイマ0(TINT0)
	02h	CPUタイマ1(TINT1)	03h	CPUタイマ2(TINT2)
	04h	外部割み込み XINT2(XINT2SOC)	05h	ePWM1(ADCSOCA)
	06h	ePWM1(ADCSOCB)	07h	ePWM2(ADCSOCA)
	08h	ePWM2(ADCSOCB)	09h	ePWM3(ADCSOCA)
	0Ah	ePWM3(ADCSOCB)	0Bh	ePWM4(ADCSOCA)
	0Ch	ePWM4(ADCSOCB)	0Dh	ePWM5(ADCSOCA)
	0Eh	ePWM5(ADCSOCB)	0Fh	ePWM6(ADCSOCA)
	10h	ePWM6(ADCSOCB)	11h	ePWM7(ADCSOCA)
	12h	ePWM7(ADCSOCB)	その他	禁止

表 37:ADCSOCxCTL レジスタの TRIGSEL フィールド(x は SOC 番号),デバイスによって ePWM の搭載本数が異なる為、全てのデバイスで全てのトリガが使用できるわけではありません。

今回は0に設定しました。ソフトウェア・トリガとは、後ほど詳細を解説しますが、ADCSOCFRC1レジスタの指定ビットに1を書く事でトリガをかける事ができます。このソフトウェア・トリガは、このTRIGSELフィールドをどれに設定しても(禁止は除く)使う事ができます。

次のCHSELフィールドは、どの入力チャネルを変換するかを設定します。このフィールドの設定は、シングル・サンプリング・モードか同時サンプリング・モードか(ADCSAMPLEMODEレジスタにて設定)のどちらにSOCが設定されているかで、その意味が異なってきます。その詳細を表 38に示します。今回は、SOC0(ダミー)、SOC1、SOC4がシングル・サンプリング・モード、SOC2が同時サンプリングモードに設定されています。SOC0とSOC1はADCINA2を使いますので2を、SOC4はADCINA0を使いますので0を、SOC2はADCINA4とADCINB4の同時サンプリングなので4をそれぞれ設定しています。

フィールド名	設定値	意味(シングル・サンプリングの場合)	意味(同時サンプリングの場合)
CHSEL	00h	ADCINA0	ADCINA0 & ADCINB0
	01h	ADCINA1	ADCINA1 & ADCINB1
	02h	ADCINA2	ADCINA2 & ADCINB2
	03h	ADCINA3	ADCINA3 & ADCINB3
	04h	ADCINA4	ADCINA4 & ADCINB4
	05h	ADCINA5	ADCINA5 & ADCINB5
	06h	ADCINA6	ADCINA6 & ADCINB6
	07h	ADCINA7	ADCINA7 & ADCINB7
	08h	ADCINB0	禁止

	09h	ADCINB1	禁止
	0Ah	ADCINB2	禁止
	0Bh	ADCINB3	禁止
	0Ch	ADCINB4	禁止
	0Dh	ADCINB5	禁止
	0Eh	ADCINB6	禁止
	0Fh	ADCINB7	禁止

表 38:ADCSOCxCTL レジスタの CHSEL フィールド

次にサンプル・ホールド回路をどれくらいの期間閉じているかの設定をACQPSフィールドにて設定します。サンプル・ホールド回路をどれくらい閉じる必要があるかは、使用方法と目的によって異なります。基本的にはサンプリング・キャパシタにどれくらいの時間で信号が充電されるかから考えますが、フィルタ目的で、長めにとるケースもあります。一般的にMCUに搭載されているADCのサンプル・ホールド期間は一定、もしくは設定できても全て共通という事が多いですが、このPiccoloのADCは、SOC毎に異なるサンプル・ホールド期間を設定する事ができます。このフィールドの詳細を表 39に示します。最小値が06h(7サイクル)であることに注意して下さい。飛び飛びに設定してはいけない値もありますので、ご注意下さい。また、この値の設定可能な範囲はデバイスによって異なります。た例えば、F280200、F28020、F28021のACPQS設定範囲は、14サイクル(0Dh)～64サイクル(3Fh)です。これらデバイスでは、00h～0Chは設定してはいけません（設定自体はできてしまいますし、それなりに動作してしまいますので、注意して下さい。）。必ず最新のデータシートを参照下さい(ADCの電気的仕様の欄に記載されています)。また、06h/07hを設定する場合は、必ずErrataを参照下さい。この値に設定した場合、不具合につかまる可能性があります。

今回の例では、特に深い意味はありませんが、SOC0/1/2/4を全て16サイクルに設定しました。今回使用するボードのADCINxピンに直接電圧をかけるだけの場合、これらピンは、単にピンがでているだけや、キャパシタが接続されているだけのピンになりますので、サンプル・ホールド期間が短いと、そんなに正しく変換はできない可能性があります。おそらく、今回のサンプル・コードの設定では、サンプル・ホールド期間が足りず、さほど正確な値を変換しないと思います（変換結果が前のチャネルの変換値に影響をうけるはずです）。各ピンに0Vか3Vをかけて頂ければ、設定したチャネルが変換されているか確認できると思います。あまりにも、値がかけ離れている場合や、前にとった値の影響を強く受ける場合(要はサンプル・ホールド時間が足りてない)は、この値を増やしてみてください（禁止されている値を設定しないように、注意下さい）。今回は、ADCのS/Wとしての使い方を示しているだけになりますので、この辺りは、あまり深く考えないことにします。

フィールド名	設定値	意味	設定値	意味
ACQPS	00h～05h	禁止	06h	7サイクル
	07h	8サイクル	08h	9サイクル
	09h	10サイクル	0Ah	11サイクル
	…	…	…	…
	…	…	…	…
	3Eh	63サイクル	3Fh	64サイクル
	注意1:10h～14h, 1Dh～21h, 2Ah～2Eh, 37h～3Bhは禁止 注意2:デバイスによって、設定できる範囲が異なります。詳細はデータシートを参照下さい。			

表 39:ADCSOCxCTL レジスタの ACQPS フィールド

これで、今回のADCの基本設定は終了です。最後にPIEレベル、CPUレベルの割り込み設定を行い。EDISにてEALLOW保護レジスタの再保護を行います。それを行っているのが下記の箇所です。

```
PieCtrlRegs.PIEIER1.bit.INTx1 = 1;
IER |= M_INT1;
EDIS;
```

ADCINT1とADCINT2の2つはPIE割り込みの2箇所に割り当てられています(ADCINT1はINT1.1とINT10.1、ADCINT2はINT1.2とINT10.2)。ハードとしては、どちらを使用してもかまわわないのですが、ヘッダ・ファイルの定義上は、INT1.1と

INT1.2の方を使用していますので、今回はINT1.1のADCINT1を使用します。INT10.1とINT10.2を使用したい場合は、ヘッダ・ファイルの変更が必要になります、ご注意下さい。

次に、割り込み処理ルーチン、AdcIsr()関数を解説します。

割り込み処理リーチン　AdcIsr()関数

```
unsigned int BufResult1[10];
unsigned int BufResult2[10];
unsigned int BufResult3[10];
unsigned int BufResult4[10];

interrupt void AdcIsr(void){

   static int count = 0;

   BufResult1[count] = AdcResult.ADCRESULT1;
   BufResult2[count] = AdcResult.ADCRESULT2;
   BufResult3[count] = AdcResult.ADCRESULT3;
   BufResult4[count] = AdcResult.ADCRESULT4;

   if(count++ > 9){
      count = 0;
   }
   AdcRegs.ADCINTFLGCLR.all = 1;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

この割り込みは、SOC4の変換が終了した時に発生するように(つまり、SOC0→SOC1→SOC2(&3)→SOC4の順番で変換が終わった時)ADCの設定を行いました。この処理ルーチンは、実際には大した事はしていません、SOC1/2/3/4用にそれぞれ10個のバッファを用意しておき(SOC0はダミーなので、結果は使用しません)、そのバッファにコピーしているだけになります。ADCの変換結果は、SOCxの番号と同じ番号のADCRESULTxレジスタに右詰(つまり0～11bit目)で格納されます。例えばSOC4であれば、ADCRESULT4レジスタに格納されます。同時サンプリングのSOC2(&3)はSOC2にAチャネルの結果が、SOC3にBチャネルの結果が格納されます。割り込みルーチンを抜ける前に、ADCINTFLGCLRレジスタに1を書き、ADCINT1のペリフェラル・レベルの割り込みフラグをクリアし、PIEACKレジスタにPIEACK_GROUP1を書く事で、PIEのINT1グループのACKをクリアしています。

最後にmain()関数を解説します。

main()関数

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(AdcIsr, "ramfuncs");

interrupt void AdcIsr(void);
```

```
void AdcConfig(void);

void main(void){

   DINT;
   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
   InitFlash();
   InitPieCtrl();
   IER=0x0000;
   IFR=0x0000;
   InitPieVectTable();
   EALLOW;
   PieVectTable.ADCINT1  = AdcIsr;
   EDIS;

   EINT;
   AdcConfig();
   AdcRegs.ADCSOCFRC1.all = 0x0017; // Start SOC0, 1, 2(&3), 4
   while(1);

}
```

ほとんどの箇所は、全章までに解説した事と同じですので、今回新しい項目のみ解説します。
まず、ベクタテーブルの登録を行っている、

PieVectTable.ADCINT1 = AdcIsr;

この部分について少しコメントします。前に、ADCINT1はINT1.1とINT10.1の両方にアサインされていて、ヘッダファイルではINT1.1を使うようになっていると解説しましたが、ここがその部分です。PieVectTable.ADCINT1は、INT1.1のベクタテーブルになっています。

AdcConfig()をコールして、ADCの設定をした後、以下の行で、ADCをスタートしています。今回は、ADCのスタートはソフトウェアにて行っていて、そのスタートを行うレジスタが、ADCSOCFRC1レジスタです。

AdcRegs.ADCSOCFRC1.all = 0x0017; // Start SOC0, 1,2(&3), 4

ADCSOCFRC1レジスタを図 116:ADCSOCFRC1レジスタに示します。

15	14	13	12	11	10	9	8
SOC15	SOC14	SOC13	SOC12	SOC11	SOC10	SOC9	SOC8
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
7	6	5	4	3	2	1	0
SOC7	SOC6	SOC5	SOC4	SOC3	SOC2	SOC1	SOC0
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 116:ADCSOCFRC1 レジスタ

このレジスタの対応するビットに1を書く事で、そのSOCにソフトウェア・トリガをかける事ができます。今回は、SOC0/1/2/4にトリガをかけますので、0x0017(0000 0000 0001 0111b)を書いています。このように、SOC0/1/2/4にトリガをかけると、全てはRound Robinモードに設定されていますので、SOC0→SOC1→SOC2(&3)→SOC4と順番に変換される事になります。

8.5 コードのデバッグ

それでは、コードをCCSにてデバッグしてみましょう。今回は、ADC変換が行われている様子を観察するために、Real-Timeモードというデバッグ方法も紹介します。Piccoloの主なターゲット・アプリケーションはモータや電源等のパワーエレクトロニクス・アプリケーションです。このようなアプリケーションは、ブレーク・ポイントにてプログラムを止めるというデバッグ方法が難しいアプリケーションです(変な状態で止めてしまうと、パワー回路に悪い影響を与えてしまい、回路を破壊してしまう可能性があるため)。このようなアプリケーションをデバッグするのに有効な機能がReal Timeモード・デバッグです。このモードでは、デバイスを止める事なく、レジスタやExpressionsウィンドウのグローバル変数などをリアルタイムでモニタ(あくまでもモニタであり、残念ながらトレースはできません)できたり、値を書き換えたりする事ができます。

それでは、いつものように虫マークをクリックしデバッガを立ち上げ、コードをロードしましょう。ロードし終わったら、

View→Expressions

にてExpressionsウィンドウを立ち上げてください(図 117)。このwindowにて、Name欄に、図 117のように、BufResult1, BufResult2, BufResult3, BufResult4を記入してください。次に、図に指示されているボタンをクリックして、Continuous Refreshモードを有効にします。

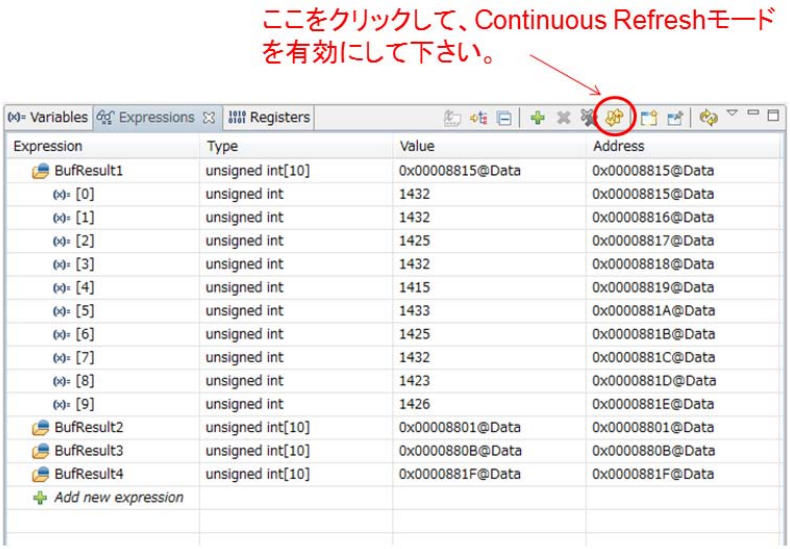


図 117：Expressions Window と Continuous Refresh ボタン

このContinuous Refreshモードは、デバイスがReal Timeデバッグモードの時に、表示されている値をリアルタイムで自動更新を行う機能を有効にしてくれます。ちなみに、この自動更新する期間は、別の場所で設定する事ができます(Window→Preferencesにて Preferences ウィンドウをたちあげ、Code Composer Studio―Debugを選択して下さい。Continous Refresh Intervalr欄にて期間を設定できるます)。但し、設定した期間はあくまでも、最短の更新時間であって、実際にどれくらいの頻度で更新されるかは、表示されているデータの量に依存します。大量にデータをリアルタイムでモニタしている場合は、それだけ更新頻度は遅くなります。さて、次にデバイスをリアルタイム・モードで動作させましょう。デバイスをリアル・タイム・モードにするためには、

Run→Advanced→Enable Silicon Real-time Mode

を選択するか、

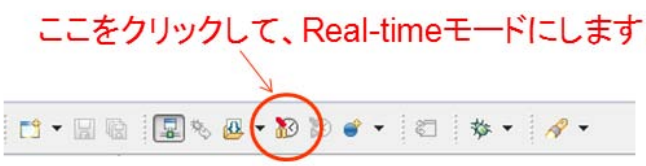


図 118：Real Time モード

図 118に示す箇所をクリックして下さい。

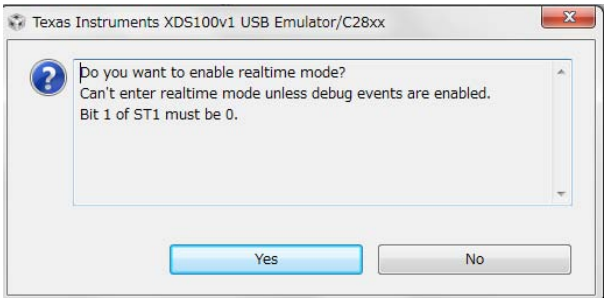


図 119：Real Time モードの確認

図 119のように、聞かれますので、Yesを選択して下さい。この画面は、デバイスをReal Timeモードにする事の確認です。これでデバイスはReal Timeモードになりましたので、RUNさせてください。実行中、先ほど開いたExpressionsウィンドウを見てください。BufResultxの中身が自動更新されていると思います（まれに、Continuous Refreshが動かない時がありますので、そんな時は、一度Conitnuous Refreshをディセーブルにし、再度イネーブルにして下さい）。これは、前述しましたように、BufResultx変数の中身をリアルタイムでモニタしています。今回の例では使いませんが、Expressionsウィンドウの変数のValue欄をCCSから書き換えると、同様にリアルタイムで値を書き込む事もできます。

そでれは、動作確認してみましょう。今回のコードでは、

SOC0→ADCINA2→Initial Conversion Errata対策のため破棄
SOC1→ADCINA2→BufResult1
SOC2/3 → ADCINA4/ADCINB4→BufResult2/3
SOC4→ADCINA0→BufResult4

になっていますので、変換結果を確認してみてください。ボードを何も設定していない場合は、ADC入力ピンがオープンになっていますので、適当な値が表示されると思いますが、ためしに、GNDに接続するか、何かしらの電圧を入力してみてください。変換結果が変わる様子がわかると思います。前に記載しましたが、今回のボードは、ADCの入力段にきちんとした回路が設けられていないため、さほど正確な値を変換してくれないはずです今回のこのドキュメントの目的は、設定の解説ですので、設定したチャネルが変換されていることが確認できれば、よしとします。

Real Timeモードを使っている時に、再度実行しなおしたい場合は注意が必要です。Real Timeモード中にResetをかけてはいけません。もし、かけてしまった場合は、Rude Retryを選択して下さい。Rude Retryとは、なんとなくReal Timeモード(表現が少し良くありませんが)というイメージです。Rude Retryを選択すると、ほとんどのケースで、CCSから再度デバイスがコントロールできるようになりますので、次にReal Timeモードを解除して下さい。復帰できると思います。また、Real Timeモードを使用している時に、ブレーク・ポイントを貼った場合でも、同じようにエラーがでます（復帰は同じくRude Retryです）。Real Timeモードでは、ブレーク・ポイントは使えないと考えてください。

Real Timeモードを一旦解除するためには

Target→Advanced→Enable Silicon Real-time Mode

を再度選択してください（もしくは、図 118のボタンを再度クリックして下さい）。この部分のチェックがはずれ、Real-time Modeがディセーブルになります。これでResetを実行する事ができます。

9 SCI(UART)

9.1 この章の目的

MCUに搭載されているペリフェラルの中で最も一般的なシリアル・ポートのUART機能をもったSCIについて解説します。SCIの詳細につきましては、以下のリファレンス・マニュアルを参照下さい。

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18] の Serial Communication Interface(SCI)章*
F2802x/F2803x用	*TMS320x2802x, 2803x Piccolo Serial Communication Interface(SCI) Reference Guide[SPRUGH1]*

9.2 SCI の概要

UART(Universal Asynchronous Receiver Transmitter)は、最も代表的な非同期式のシリアル・ポートの一つで、図 120の

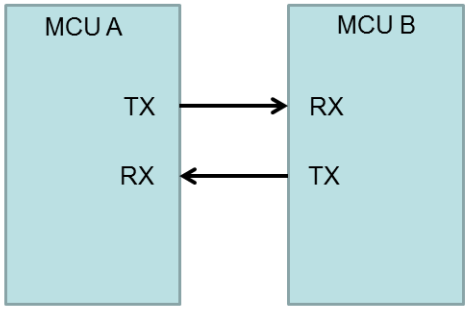


図 120：UART の結線(TX:送信ポート, RX:受信ポート)

ように、送信ポートと受信ポートをそれぞれ接続するだけです（もちろんGNDも接続する必要があります）ので、たった２線で全二重通信が行えます。非常にシンプルで使いやすいシリアル・ポートですので、今後も使われ続けられるシリアル・ポートだと思います。通信速度はビット・レート(bps等)で表すシリアル・ポートが多いですが、UARTでは一般的にはビット・レートではなくボー(baud)・レートという単位が使用されます。ビット・レートとボー・レートの意味は、厳密には異なりますが、UARTの場合は、ほぼ同じと考えて良いでしょう(いろんな意見があるかとは思いますが)。同期クロックがない通信になりますので、送信側も受信側も、お互いにボー・レートを知っている必要があります。お互いにボー・レートを知っていますので、データの開始タイミングがわかれば、その後は決められたボー・レートに従ってデータが送られてくるため、同期クロックが無くても、正しくデータの受信を行う事ができます。UARTのデータ・フォーマットは一般的に、以下の図 121のようになります。

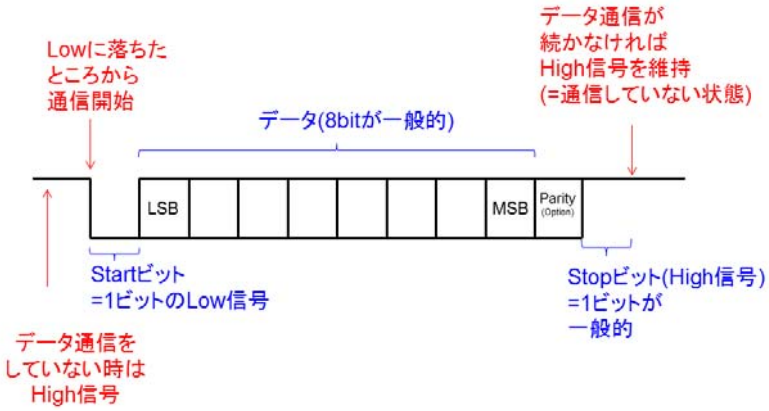


図 121：UART の一般的なフォーマット

データ通信をしていない時は、信号はHigh状態です。送信側は、送信を開始する時に1ビット分のLow信号を出力します。受信側は、このLow信号を検出する事で、通信が開始された事を認識します。この1ビット分のLow信号をStartビットと呼びます。先ほども記述しましたが、UART通信では、送信側と受信側が共に通信ボー・レートを知っていますので、同期クロックがなくても、この最初のLow信号から、指定されたボー・レートでデータが送信されますので、受信側も正しく受信する事ができます。送信側は、Startビットに続いて、実際に送信したいデータをLSBから送信します。データ長は8ビットを使う事が最も一般的ですが、他のビット長が使われる事もあります。PiccoloのSCIでは、1～8ビットまで設定できます。特定のデータが（たとえば図では8ビット）送信されたら、続いて1ビットのパリティ・ビットを送ります。このパリティ・ビットは、オプションになっていて、偶パリティ、奇パリティ、パリティ無(つまりこのビットが無い)から選択できます。パリティ・ビットに続いて、Stopビットと呼ばれるHigh信号が送信されます。このビットは、1ビットの場合が一般的ですが、Piccoloでは1ビットと2ビットの２種類から選択できます。続いて送信するデータが無い場合は、このままHigh信号を維持します。このように、非常にシンプルなフォーマットですので、オシロスコープで波形を見ることで、どのようなデータが送信されているかを把握するのも簡単です。

Piccoloに搭載されているSCIには、以下の特徴があります。

- 設定可能なデータ・フォーマット
  - 1 Startビット
  - 1～8ビットまで選択可能な転送ビット長
  - 偶パリティ/奇パリティ/パリティ無を選択可能
  - 1又は2 Stopビット
- プログラマブルなボーレート
  - 16bitのボー・レート生成用分周器
  - オート・ボー・レート検出
- 独立した送受信FIFO(DMA搭載デバイスでも、DMA転送がサポートされていない事に注意下さい)
  - ４段の送信FIFO
  - ４段の受信FIFO
  - 設定可能な送信FIFO割り込み及び受信FIFO割り込み(何段目のFIFOにて割り込みを入れるかを設定可能)

UART自体が非常に単純な通信のため、SCIペリフェラルの使い方も非常にシンプルですが、このSCIには、マルチ・プロセッサ間通信（通常、UARTは1対1通信ですが、I2CバスやCANバスのように、1つのUARTラインを複数のプロセッサが使う事ができます）を可能にする機能があるため、マニュアルを見ると、少し難しく感じるかもしれません。SCIには以下の2つのモードが存在します。

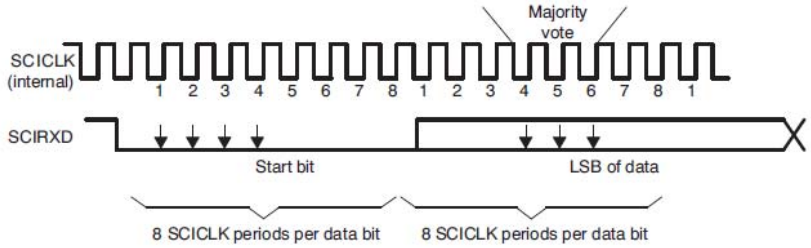
- Idle-Line Multiprocessor Mode（一般的な通常のUART通信はこのモードを使います）
- Address-Bit Multiproessor Mode

両方とも、マルチ・プロセッサ間通信を実現するモードですが、その方法に違いがあります。SCIにおけるマルチ・プロセッサ間通信はあまり利用される事がありませんので、このドキュメントではその中身は割愛します。一般的な通常のUART通信は、Idle-Line Multiprocessor Modeを使うという事だけを、覚えておいて頂ければと思います(つまり、一般的なUART通信しかしない場合は、マニュアルのAddress-Bit Multiprocessor Modeは読まなくても良いという事です)。

9.3 UART のボー・レートと SCI におけるサンプリング・ポイント

UARTを使うに当たって、ボー・レートとクロックの精度を考える必要があります。送信側も受信側も、通信するボー・レートはお互いに知っていますので、理想的には全く同じボー・レートで通信を行いたいところですが、現実はそうではありません。送信側のICと受信側ICが全く同じクロック源を使用している場合は、理想に近い形になるケースもありますが、両社が違うクロック源を使用している事の方が多いと思います。また、MCUでは、クロック源をPLLで逓倍したり、分周したりしますので、理想的なボー・レートを、使用しているクロック源から作れない場合もあります。つまり、全く同じボー・レートで通信を行える方がまれなのです。しかしながら、両ICが全く同じボー・レートで通信を行う必要もありません。多少の誤差は吸収

する事ができます。一般的に、受信側では、1ビット(1ボー)を一回でサンプリングするのではなく、複数回サンプリングします。Piccoloでは、図 122のようになっています。



図のSCICLKは、SCIペリフェラルでボー・レート生成のために使用されるクロックです。このクロックは、CPUクロックを分周したクロックで、ユーザーが設定します。UART通信の1bitは8 SCICLKになります。受信開始時は、4bitの連続したローを検出した場合にStartビットと判断します。後続のビットは、8 SCICLK中の4, 5, 6 SCICLK目(4 SCICLKの最初から6 SCICLKの終わりまで)をサンプリングし、サンプリングした結果を多数決で、1か0かを判断します。ここで重要なのが、サンプリング点は4, 5, 6 SCICLKであって、1,2,3,7,8 SCICLK目は、全く参照されていないという点です。つまり、1,2,3,7,8 SCICLK目の信号は、どうなってもよいのです。これが、多少の誤差が吸収できる仕組みとなります。例えば下の図 123を見てください。

図 122：SCI のデータ・サンプリング・ポイント

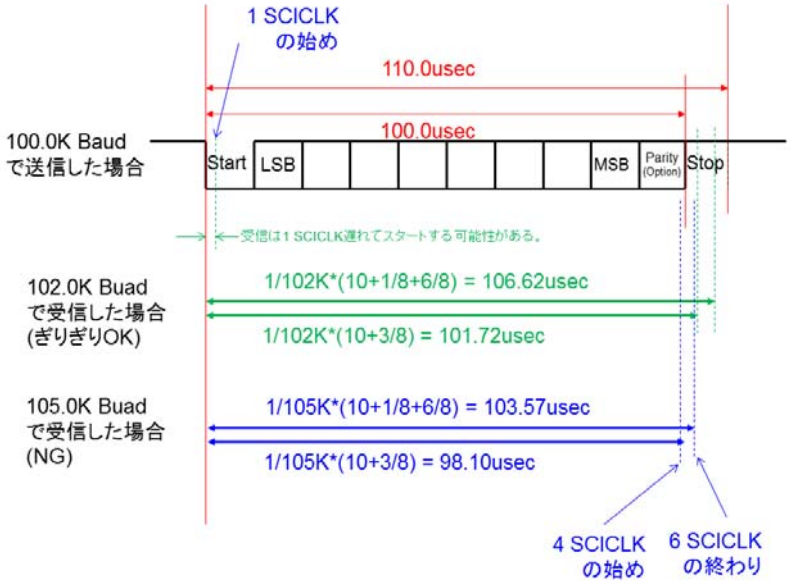


図 123：送信と受信のボー・レート

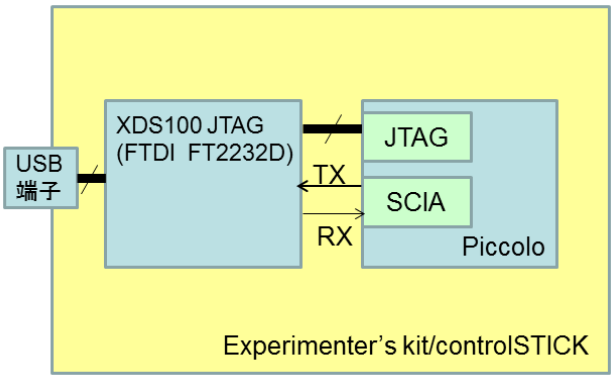
この例では、100.0K ボーで送信側がデータを送信した時を考えています。Start + 8bit data + Parity+ Stop で合計 11 ビットあります。一方、受信側は、何等かの事情で、102.0K ボー、105.0K ボーで、受信した場合の 2 種類を考えています。送信と受信のボー・レートが異なっているため、お互いにどんどんずれていく事になります。一番ずれが生じるのが、最後の 11 ビット目の Stop ビットです。受信側は 4 SCICLK の始めから 6 SCILK の終わりまでに信号をサンプリングしますので、受信側のこの期間に送信側の Stop ビットが正しくサンプリングできれば、正しく通信できることになります。

まず、102.0Kbaudで受信した場合を見てみましょう。受信側は、Low信号を検出するところからスタートします。理想的には、送信と全く同じタイミングで検出できればいいのですが、実際には、1 SCICLK遅れる可能性があります(1 SCICLK中のどこかでサンプリングしているため)。11ビット目のStopビットを検出する一番遅い可能性があるタイミングは、1/102K*(10+1/8+6/8)になりますので、106.62usecになります。一方、一番早いタイミングは、送信側と全く同じタイミングでスタートできた場合で、1/102K*(10+3/8)になりますので、101.72usecになります。11bit目は100usec～110usecまでに送信されていますので、この例は、正しく受信できることになります。同様に105.0Kボーで受信した場合を考えますと、98.10usec～103.57usecとなり、100usec～110usecに入らないため、正しく受信できない可能性がある計算になります。

UARTにおけるサンプリング・ポイントはデバイスにより異なりますので、UART通信の相手側の仕様をよくご確認ください。また、この話は、あくまでも完全に理想的な話で、実際にはクロックにも精度があり、配線により立ち上がり/下がり波形がなまり、クロックにジッタがあったりしますので、さらに余裕を見る必要がありますので、余裕を持った設計が必要です。

9.4 SCI 使用例

それでは、非常にシンプルなSCIの使用例を見ながら、使い方を解説します。図 124にありますように、controlSTICK、Experimenter's kit、C2000 LaunchPadは、オンボートに搭載されているXDS100 v1/v2 JTAG回路(FTDI FT2232を使用しています)にPiccoloのSCIAのSCIRXDA(図ではRX)とSCITXDA(図ではTX)が接続されています。XDS100 v1/v2 JTAG回路には、



2ポートがあり、1ポートをJTAGとして使って、もう一つをUARTと接続する事ができます。

このSCIAの両信号はFTDI FT2232を通してPCとUSBに接続されます。XDS100 JTAGのドライバがインストールされていれば、PCはこのシリアル・ポートをバーチャル・シリアル・ポートとして認識してくれます(F28027のcontrolSTICKの場合は少し設定が必要です)。Windowsのデバイス・マネージャを見ると、何番のシリアル・ポートに登録されたかを確認する事ができます。私が試してみた時は、以下の図 125のように、COM34に登録されています。この番号は、USBを接続する度に変わる可能性がありますので、接続する度に確認してください。

図 124：E と SICA

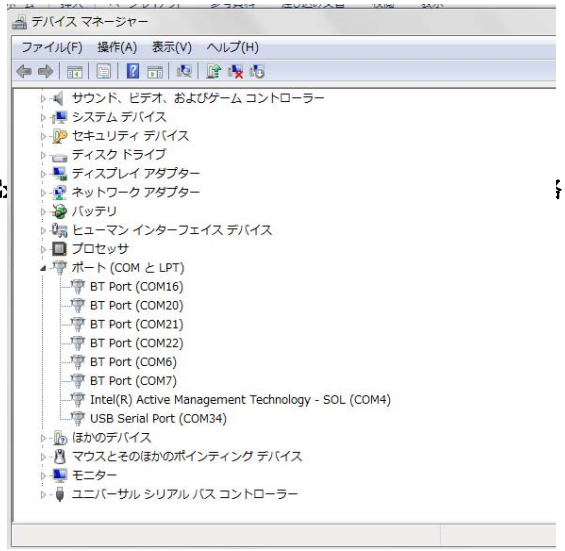


図 125：シリアル・ポート番号の確認

このように、Experimenter's Kit、controlSTICK、C2000 LaunchPadは、ボードのUSBをPCに接続するだけで、PCとUART通信ができる環境となります。今回はこれを利用しましょう。

今回のコードは、PCからUARTで送信したデータをそのまま、エコー・バックするコードを例に挙げます。それでは、Projectを作成します。手順は今までと同様ですが、念のため記載します。まず、Projectを作成するディレクトリの下に以下のHeader Filesをコピーして、新規Projectを作成して下さい。

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ
F2802xの場合	C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_commonフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_headersフォルダ C:\ti\controlSUITE\device_support\f2802x\v200\DSP28x_Project.h C:\ti\controlSUITE\device_support\f2802x\v200\F2802x_Device.h

Header FilesのExclude設定は以下の通りです。

F28069の場合	F2806x_common ― cmd 　　F28069.cmdを残して、他は全てExclude F2806x_common ― lib 　　全てExclude F2806x_common ― source 　　F2806x_CodeStartBranch.asm 　　F2806x_CpuTimers.c 　　F2806x_DefaultIsr.c 　　F2806x_PieCtrl.c 　　F2806x_PieVect.c 　　F2806x_Sci.c 　　F2806x_SysCtrl.c 　　F2806x_usDelay.asm 　　を残して他は全てExclue F2806x_headers ― cmd 　　F2806x_Headers_BIOS.cmdをExclude
F28035の場合	DSP2803x_common ― cmd 　　F28035.cmdを残して、他は全てExclude DSP2803x_common ― lib 　　全てExclude DSP2803x_common ― source 　　DSP2803x_CodeStartBranch.asm 　　DSP2803x_CpuTimers.c 　　DSP2803x_DefaultIsr.c 　　DSP2803x_PieCtrl.c 　　DSP2803x_PieVect.c 　　DSP2803x_Sci.c 　　DSP2803x_SysCtrl.c 　　DSP2803x_usDelay.asm 　　を残して他は全てExclue DSP2803x_headers ― cmd

	DSP2803x_Headers_BIOS.cmdをExclude
F28027の場合	DSP2802x_common ― cmd F28027.cmdを残して、他は全てExclude FlashProjectの時と同じく、 SECTIONS{}欄内の **.text** : >> FLASHA | FLASHC | FLASHD. **PAGE** = 0 の行にて、FLASHDの次の文字がドットになっていますが、コンマが正しいです。 **.text** : >> FLASHA | FLASHC | FLASHD, **PAGE** = 0 ここを修正しないと、Warningがでますので、修正して下さい。 F2802x_common ― lib 全てExclude F2802x_common ― source F2802x_Adc.c F2802x_CodeStartBranch.asm F2802x_Cputimers.c F2802x_DefaultIsr.c F2802x_PieCtrl.c F2802x_PieVect.c F2802x_Sci.c F2802x_SysCtrl.c F2802x_usDelay.asm を残して他は全てExclue F2802x_headers ― cmd F2802x_Headers_BIOS.cmdをExclude Project作成時に自動生成されたF28027_RAM_lnk.cmdをExclude

include pathの設定もして下さい。

F28069の場合	F2806x_common ― include F2806x_headers ― include
F28035の場合	DSP2803x_common ― include DSP2803x_headers ― include
F28027の場合	F2802x_common ― include F2802x_headers ― include 一番上のプロジェクト名(今回の例では、PwmProject1つまり、プロジェクト・フォルダ直下のディレクトリです)。

それでは、コードを見ていきましょう。以下が、コードになります。

main.c
#include "DSP28x_Project.h" #include <string.h>

```
#define USE_F28069  // When using F28069, Please uncomment

interrupt void IsrSciRx(void);
interrupt void IsrTimer0(void);
void SciInit(void);
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadSize;
extern Uint16 RamfuncsRunStart;

void main(void) {

   InitSysCtrl();
   memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
   InitFlash();

   //Interrupt Init
   DINT;
   InitPieCtrl();
   IER=0x0000;
   IFR=0x0000;
   InitPieVectTable();
   EALLOW;
    PieVectTable.TINT0 = IsrTimer0;
   PieVectTable.SCIRXINTA = IsrSciRx;
   EDIS;
   PieCtrlRegs.PIEIER9.bit.INTx1 = 1;
   PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
   IER |= ( M_INT9 | M_INT1) ;

   SciInit();
   InitCpuTimers();
#ifdef USE_F28069
   ConfigCpuTimer(&CpuTimer0, 80, 1000000);          //1sec timer
#else
   ConfigCpuTimer(&CpuTimer0, 60, 1000000);          //1sec timer
#endif
   CpuTimer0Regs.TCR.all = 0x4000;

   EINT;

   while(1){
   }

}

void SciInit(void){

   //GpioInit
   InitSciaGpio();
```

```
    EALLOW;
    SysCtrlRegs.LOSPCP.bit.LSPCLK = 0;                  // LSPCLK = SYSCLKOUT
    EDIS;

    // SW Reset
    SciaRegs.SCICTL1.bit.SWRESET = 0;                   // Reset

    // Format Configuration
    SciaRegs.SCICCR.bit.STOPBITS  = 0;                  // Stop bit = 1
    SciaRegs.SCICCR.bit.PARITY    = 0;                  // Parity = Odd
    SciaRegs.SCICCR.bit.PARITYENA = 1;  // Enable Parity
    SciaRegs.SCICCR.bit.SCICHAR   = 7;  // Character length = 8bit

#ifdef USE_F28069
    // Baud rate configuration
    // Baud rate = 1152000 Baud
    // SYSCLKOUT(=CPU CLK) = 80MHz
    // LSPCLK = SYSCLKOUT / 1 (reset default) = 80MHz
    // SCIHBAUD:SCILBAUD = 80000000 / (115200 * 8 ) -1  = 85.8 = 86 (0x56)
    // When SCIHBAUD:SCILBAUD = 86 --> Actual Baud rate = 80000000/((86+1)*8) = 11494 Baud

    SciaRegs.SCIHBAUD = 0x00;
    SciaRegs.SCILBAUD = 0x56;
#else
    // Baud rate configuration
    // Baud rate = 1152000 Baud
    // SYSCLKOUT(=CPU CLK) = 60MHz
    // LSPCLK = SYSCLKOUT / 1 (reset default) = 60MHz
    // SCIHBAUD:SCILBAUD = 60000000 / (115200 * 8 ) -1  = 64.1 = 64 (0x40)
    // When SCIHBAUD:SCILBAUD = 64 --> Actual Baud rate = 60000000/((64+1)*8) = 115385 Baud

    SciaRegs.SCIHBAUD = 0x00;
    SciaRegs.SCILBAUD = 0x40;
#endif

    //TX FIFO Configuration
    SciaRegs.SCIFFTX.bit.SCIFFENA  = 1;                 // Enable SCI FIFO
    SciaRegs.SCIFFTX.bit.TXFFIENA  = 0;                 // Disable FIFO TX interrupt

    //RX FIFO Configuration
    SciaRegs.SCIFFRX.bit.RXFFIL    = 2;                 // RX FIFO interrupt level = 2
    SciaRegs.SCIFFRX.bit.RXFFIENA  = 1;                 // Enable RX FIFO interrupt

    //Enable SCI
    SciaRegs.SCICTL1.bit.TXENA = 1;                     // Enable TX
    SciaRegs.SCICTL1.bit.RXENA = 1;                     // Enable RX
    SciaRegs.SCICTL1.bit.SWRESET = 1;                   // Re-Enable SCI

    return;
```

```
}

interrupt void IsrSciRx(void){

   while(SciaRegs.SCIFFRX.bit.RXFFST != 0){

      while(SciaRegs.SCIFFTX.bit.TXFFST == 4);
         SciaRegs.SCITXBUF = SciaRegs.SCIRXBUF.bit.RXDT;
      }

      SciaRegs.SCIFFRX.bit.RXFFINTCLR = 1;
      PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;

   return;

}

interrupt void IsrTimer0(void){

   while(SciaRegs.SCIFFRX.bit.RXFFST != 0){
      while(SciaRegs.SCIFFTX.bit.TXFFST == 4);
      SciaRegs.SCITXBUF = SciaRegs.SCIRXBUF.bit.RXDT;
   }

   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

最初の方は、既におなじみになっていると思いますので、解説を省きますが、一点だけ、以下の３行目ですが、

```
#define USE_F28069  // When using F28069, Please uncomment
```

この行は、F2806xを使用する時のみコメントをはずして、F2802x/F2803xの場合は、コメントして下さい。この行は、F2806xが80MHz動作、F2802x/F2803xが60MHz動作という違いを吸収するための、defineです。

さて、今回は、割り込みを２本使います。
一つが、IsrSciRx()で、これは、SCIの受信FIFO割り込みです。もう一つがIsrTimer0()で、CPUタイマ0の割り込みです。受信FIFO割り込みは、FIFOの2段目までデータを受信したら割り込みがかかるように設定しています。このような設定の場合、1byteのデータを受信し、その後データを受信しない場合は、永遠に割り込みがかかず、1byteデータを受信している事に気づきません。それを防ぐために、タイマ割り込みをしかけておき、定期的に受信FIFOをチェックしています。
それでは、SCIの設定をしているSciInit()を解説していきます。まず最初に、

```
   InitSciaGpio();
```

と、InitSciaGpio関数をコールしています。この関数は、

F28069の場合	F2806x_common\soruce\F2806x_sci.c
F28035の場合	DSP2803x_common\source\DSP2803x_sci.c
F28027の場合	F2802x_common\source\F2802x_sci.c

にある関数です。このファイルのこの関数の中身を見てください。GPIO28とGPIO29をSCIRXDAとSCITXDAにアサインしています。この関数にて、一点重要なのが、

```
GpioCtrlRegs.GPAQSEL2.bit.GPIO28 = 3;  // Asynch input GPIO28 (SCIRXDA)
```

この行です。PiccoloのGPIOは、入力モードが４種類あります。使うペリフェラルや目的に応じてこのモードを選択する必要があります。この入力モードを決めるのが、GPxQSELyレジスタ(図 126)で、GPIOピン毎に設定する事ができます。

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
GPIO15	GPIO14	GPIO13	GPIO12	GPIO11	GPIO10	GPIO9	GPIO8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
GPIO7	GPIO6	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 126：GPxQSELy(図は GPAQSEL1)レジスタ

フィールド名	設定値	意味	設定値	意味
GPIOxx	00b(default)	SYSCLKOUTに同期。	01b	3サンプリングのQualfication
	10b	6サンプリングのQualification	11b	非同期。非同期ペリフェラルに対して使用します。SCIの受信ポートは必ずこの設定を使ってください。

表 40：GPxQSELy レジスタの GPIOxx フィールド

表 40：GPxQSELyレジスタのGPIOxxフィールド表 40にGPxQSELyレジスタのGPIOxxフィールドに設定する4モードを記します。リセット値は00bになりますので、何も設定しないとこのモードになります。このモードは、SYSCLKOUTに同期とありますが、最も一般的な通常の設定と考えて頂ければと思います。01bと10bは、Qualification(なかなか良い日本語訳が思いつきません)という機能を使う場合の設定です。GPIOを入力として使う場合、一回のサンプルでHigh/Low入力を判断するのは、あまり良い方法ではない場合があります。例えば、GPIO入力を取得するまさにそのタイミングでノイズが入って、誤った入力信号を認識する事がありえます。そのため、GPIO入力値を取得するには、S/Wにて何回が入力値を読み込み、多数決をとるといった手法が用いられる事があります。このQualification機能は、この入力値を何回か取得するという方法を、H/Wで実装していると考えるとわかりやすいかと思います。01bは3回、10bは6回のサンプリングを行い3/6回のサンプル値が全て一致した場合、入力値が変わったと判断します(1回でも違った場合は前の値を保持)。この機能の詳細は、Reference Guideを参照下さい。さて、本題ですが、11bが、非同期入力です。SCIを使う場合は、受信ポートを必ずこの設定にする必要がありますので、このコードでも3(11b)を設定しています。非同期設定は、ペリフェラル内に同期回路がある、I2C、SPI、eCAN、McBSPでも使います。また、ePWMの /TZx入力信号も、一般的には非同期設定を使います(同期設定でもかまいませんが、ディレイが生じます)。

それでは、もとに戻って、SciInit()関数の続きを解説していきます。次は以下のようにクロックの設定を行っています。

```
EALLOW;
SysCtrlRegs.LOSPCP.bit.LSPCLK = 0;              // LSPCLK = SYSCLKOUT
EDIS;
```

SCIのクロックはCPUのクロック(SYSCLKOUT)を分周したLSPCLKが使われます。LSPCLKの設定は以下の図 127：LOSPCPレジスタ図 127、表 41

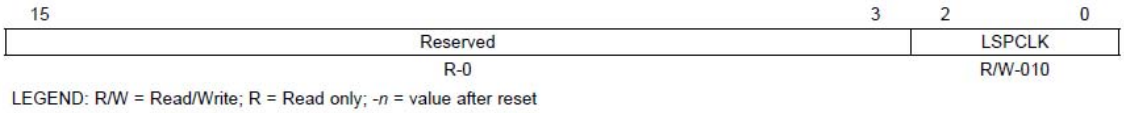


図 127：LOSPCP レジスタ

フィールド名	設定値	意味	設定値	意味
LSPCLK	000b	LSPCLK=SYSCLKOUT/1	100b	LSPCLK=SYSCLKOUT/8
	001b	LSPCLK=SYSCLKOUT/2	101b	LSPCLK=SYSCLKOUT/10
	010b	LSPCLK=SYSCLKOUT/4 (Reset値)	110b	LSPCLK=SYSCLKOUT/12
	011b	LSPCLK=SYSCLKOUT/6	111b	LSPCLK=SYSCLKOUT/14

表 41：LOSPCP レジスタの LSPCLK フィールド

に示すLOSPCPレジスタのLSPCLKフィールドにて設定します。LSPCLKは、SCI以外にもSPIでも使われますので、両方のペリフェラルを使う場合は、よく考えて設定する必要があります。さて、今回は一番高速の0設定にしました。これで、LSPCLK=SYSCLKOUT(F2802x/F2803x = 60MHz, F2806x = 80MHz)となります。

続いて、以下の行にて、UARTモジュールをリセットしています。デバイスのリセット時にSCIもリセットされているので、この行は必須というわけではありませんが、念のためリセットしました。リセットの設定は、SCICTL1レジスタのSW RESETフィールドになります(図 128と表 42)。このフィールドを0にライトする事で、リセットされます。

```
SciaRegs.SCICTL1.bit.SWRESET = 0;          // Reset
```

図 128：SCICTL1 レジスタ

ここで、このSCICTL1レジスタのほかのフィールドも一緒に紹介しておきましょう。TXENA及びRXENAは後で使用します。RX ERR INT ENAは今回は使用しませんが、BKDT(10ビット以上ローが続いた場合のエラー)、PE(パリティ・エラー)、FE(Stopビットが見つからなかった場合)といった、受信エラー割り込みを使う場合にはイネーブルになります。尚、これらのステータスは、SCIRXSTレジスタにて確認できます。TXWAKE, SLEEPは、マルチプロセッサ間通信用のフィールドですので、通常のUART通信を行う場合は、使用しません。

フィールド名	設定値	意味	設定値	意味
SW RESET	0	SCIモジュールのリセット	1	SCIモジュールのリセット解除
RX ERR INT ENA	0	受信エラー割り込みのディセーブル	1	受信エラー割り込みのイネーブル
TXENA	0	SCIの送信ディセーブル	1	SCIの送信イネーブル
RXENA	0	SCIの受信ディセーブル	1	SCIの受信イネーブル

表 42：SCICTL1 レジスタの重要フィールド

次に、データのフォーマットを設定しています。データのフォーマットを決めるのが、SCICCRレジスタです(図 129：SCICCRレジスタ(図 129と表 43)。

```
SciaRegs.SCICCR.bit.STOPBITS  = 0;          // Stop bit = 1
SciaRegs.SCICCR.bit.PARITY    = 0;          // Parity = Odd
```

```
SciaRegs.SCICCR.bit.PARITYENA = 1;  // Enable Parity
SciaRegs.SCICCR.bit.SCICHAR   = 7;  // Character length = 8bit
```

7	6	5	4	3	2	1	0
STOP BITS	EVEN/ODD PARITY	PARITY ENABLE	LOOPBACK ENA	ADDR/IDLE MODE	SCICHAR2	SCICHAR1	SCICHAR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

図 129：SCICCR レジスタ

フィールド名	設定値	意味	設定値	意味
STOP BITS	0	Stopビット＝1ビット	1	Stopビット＝2ビット
EVEN/ODD PARITY	0	奇パリティ	1	偶パリティ
PARITY ENABLE	0	パリティのディセーブル	1	パリティのイネーブル
LOOPBACK ENA	0	ループバック・テスト・モードのイネーブル	1	ループバック・テスト・モードのディセーブル
ADD/IDLE MODE	0	マルチ・プロセッサ間通信における Idle-Lineモード**(通常のUART通信はこちらの設定です)**	1	マルチ・プロセッサ間通信における、Address-bitモード
SCI CHAR[2:0]	000b	データ幅＝1ビット	001b	データ幅＝2ビット
	010b	データ幅＝3ビット	011b	データ幅＝4ビット
	100b	データ幅＝5ビット	101b	データ幅＝6ビット
	110b	データ幅＝7ビット	111b	データ幅＝8ビット

表 43：SCICCR レジスタの重要フィールド

今回の設定は、
Stopビット＝1ビット
パリティ＝奇パリティ
パリティのイネーブル
データ幅＝8ビット
になっています。ADD/IDLE MODEは今回は設定していませんが、リセット値が0ですので、Idle-Lineモードになっています。通常のUART通信はこのモードを使用します。ループバック・モードも、リセット値が0(ディセーブル)なので設定していませんが、このモードは、SCIペリフェラル内で送信/受信がテストできる機能です。

次にボー・レート生成器の設定を行っています。

```
SciaRegs.SCIHBAUD = 0x00;
F2802x/F2803xの場合：SciaRegs.SCILBAUD = 0x40;
F2806xの場合　　：SciaRegs.SCILBAUD = 0x56;
```

SCIのボー・レート生成器の設定は、SCIHBAUDレジスタ(図 130)と、SCILBAUDレジスタ(図 131)にて設定します。

7 15	6 14	5 13	4 12	3 11	2 10	1 9	0 8
BAUD15 (MSB)	BAUD14	BAUD13	BAUD12	BAUD11	BAUD10	BAUD9	BAUD8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

図 130：SCIHBAUD レジスタ

7	6	5	4	3	2	1	0
BAUD7	BAUD6	BAUD5	BAUD4	BAUD3	BAUD2	BAUD1	BAUD10 (LSB)
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

図 131：SCILBAUD レジスタ

SCIHBAUDレジスタにつきましては、大変申し訳ありませんが、現時点でのReference Guideの記載に誤記があります。BAUD8フィールドからBAUD15フィールドまで、8～15とビット番号が書いてありますが、正しくは0～7のビット番号となります(図 130を参照下さい)。SCIのボー・レート生成器は、SCIHBAUDレジスタがHigh側8bit、SCILBAUDレジスタがLow側8bitとなって、16bitレジスタを形成します。生成されるボー・レートとSCIH/LBAUDレジスタのBAUD[15 :0]フィールドの関係式は以下の通りです。

この式から、希望するボー・レートに対するBAUD[15 :0]の設定値の式に変換すると、

となります。今回、ボー・レートとしては115200ボーを使いますので、60MHz動作のF2802x/F2803xの場合で、

となります。レジスタに小数点を設定する事はできませんので、四捨五入で64(0x40)を設定します。BAUD[15 :0]に64を設定した時の実際のボー・レートを計算してみますと、

となります。まず、問題のない誤差範囲です。SCIHBAUDがHigh側、SCILBAUDがLow側ですので、SCIHBAUD = 0. SCILBAUD=0x40と設定します。同様に、F2806xの方も80MHz動作で計算してみますと、

となります。SCIHBAUD = 0, SCILBAUD=0x56(86)と設定します。

次にFIFOの設定をします。このSCIは、FIFOを使うモードと使わないモード(C24xとの互換モード)の２種類があります。通常はFIFOを使うと思います。

```
//TX FIFO Configuration
SciaRegs.SCIFFTX.bit.SCIFFENA  = 1;          // Enable SCI FIFO
SciaRegs.SCIFFTX.bit.TXFFIENA  = 0;          // Disable FIFO TX interrupt

//RX FIFO Configuration
SciaRegs.SCIFFRX.bit.RXFFIL    = 2;          // RX FIFO interrupt level = 2
SciaRegs.SCIFFRX.bit.RXFFIENA  = 1;          // Enable RX FIFO interrupt
```

まず、送信FIFOの設定をします。送信FIFOの設定は、SCIFFTXレジスタにて設定します(図 132, 表 44)。

15	14	13	12	11	10	9	8
SCIRST	SCIFFENA	TXFIFO Reset	TXFFST4	TXFFST3	TXFFST2	TXFFST1	TXFFST0
R/W-1	R/W-0	R/W-1	R-0	R-0	R-0	R-0	R-0
7	6	5	4	3	2	1	0
TXFFINT Flag	TXFFINT CLR	TXFFIENA	TXFFIL4	TXFFIL3	TXFFIL2	TXFFIL1	TXFFIL0
R-0	W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 132：SCIFFTX レジスタ

フィールド名	設定値	意味	設定値	意味
SCIRST	0	SCIのリセット	1	SCIのリセット解除
SCIFFENA	0	FIFOのディセーブル	1	FIFOのイネーブル
TXFIFO Reset	0	送信FIFOのリセット(FIFOポインタがクリアされます)	1	送信FIFOのリセット解除
TXFFST[4:0]	00000b	現在の送信FIFOレベル = 0(空)	00001b	現在の送信FIFOレベル = 1
	00010b	現在の送信FIFOレベル= 2	00011b	現在の送信FIFOレベル = 3
	00100b	現在の送信FIFOレベル=4(満杯)	00101b ～ 1111b	予約
TXFFINT Flag	0	送信FIFO割り込みは起きていません	1	送信FIFO割り込みが起きました(TXFFINT CLRビットにてクリア)
TXFFINT CLR	0	無効	1	TXFFINT Flagのクリア
TXFFIENA	0	送信FIFO割り込みのディセーブル	1	送信FIFO割り込みのイネーブル
TXFFIL[4:0]	送信FIFO割り込みのFIFOレベルを設定します。このフィールドに設定した値とTXFFSTの値が等しいか、もしくはTXFFSTの方が小さい場合に、割り込みを発生する事ができます。			

表 44：SCIFFTX レジスタの各フィールド

先ほど、SCICTL1レジスタにもSCIリセットのフィールドがありましたが、このSCIFFTXレジスタにもSCIRSTというSCIのリセットを行うビットがあります。SCIは、もともとFIFOが無かったペリフェラル(C28xの前身のC24xに搭載されていました)に、FIFOを追加したペリフェラルなので、このような構成になっているものと思われます。SCI通信を行うためには、このSCIRSTもリセット解除する必要があります。このビットはリセット値が1(リセット解除)になっていますので、今回は設定していません。コードでは、まず、SCIFFENAビットに1をライトして、FIFOをイネーブルにしています。FIFOはリセット後はディセーブルになっていますので、FIFOを使う場合は必ずイネーブルにして下さい。今回のコードでは送信割り込みは使用しませんので、TXFFIENAに0をライトして(リセット値が0なので、何もしなくてもかまいませんが)、明示的に送信FIFO割り込みをディセーブルにしています。送信FIFO割り込みを使う場合は、TXFFILにて何段目までFIFOに空きが出来たら割り込みをかけるかを設定して、TXFFIENAに1をライトして送信FIFO割り込みを許可してください。

次に、受信FIFOの設定を行っています。受信FIFOの設定はSCIFFRXレジスタ(図 133,表 45)にて行います。

15	14	13	12	11	10	9	8
RXFFOVF	RXFFOVR CLR	RXFIFO Reset	RXFIFST4	RXFFST3	RXFFST2	RXFFST1	RXFFST0
R-0	W-0	R/W-1	R-0	R-0	R-0	R-0	R-0
7	6	5	4	3	2	1	0
RXFFINT Flag	RXFFINT CLR	RXFFIENA	RXFFIL4	RXFFIL3	RXFFIL2	RXFFIL1	RXFFIL0
R-0	W-0	R/W-0	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 133：SCIFFRX レジスタ

フィールド名	設定値	意味	設定値	意味
RXFFOVF	0	受信FIFOオーバー・フローは起きていません	1	受信FIFOオーバー・フローが起きています。
RXFFOVR CLR	0	無効	1	RXFFOVFビットのクリア
RXFIFO Reset	0	受信FIFOのリセット(FIFOポインタがクリアされます)	1	受信FIFOのリセット解除
RXFFST[4:0]	00000b	現在の受信FIFOレベル = 0(空)	00001b	現在の受信FIFOレベル = 1
	00010b	現在の受信FIFOレベル= 2	00011b	現在の受信FIFOレベル = 3
	00100b	現在の受信FIFOレベル=4(満杯)	00101b ～ 1111b	予約
RXFFINT Flag	0	受信FIFO割り込みは起きていません	1	受信FIFO割り込みが起きました(RXFFINT CLRビットにてクリア)
RXFFINT CLR	0	無効	1	RXFFINT Flagのクリア
RXFFIENA	0	受信FIFO割り込みのディセーブル	1	受信FIFO割り込みのイネーブル
RXFFIL[4:0]	受信FIFO割り込みのFIFOレベルを設定します。このフィールドに設定した値とRXFFSTの値が等しいか、もしくはRXFFSTの方が大きい場合に、割り込みを発生する事ができます。			

表 45：SCIFFRX レジスタの各フィールド

今回は、受信FIFO割り込みを使用しますので、RXFFILに何段目までFIFOにデータがたまったら割り込みを入れるかを設定します。FIFOは全部で4段ですので、今回は半分の2段までデータを受信したら割り込みをかけるように設定しました(RXFFIL=2)。RXFFILの設定をしたら、RXFFIENAに1をライトして、受信FIFO割り込みをイネーブルにします。

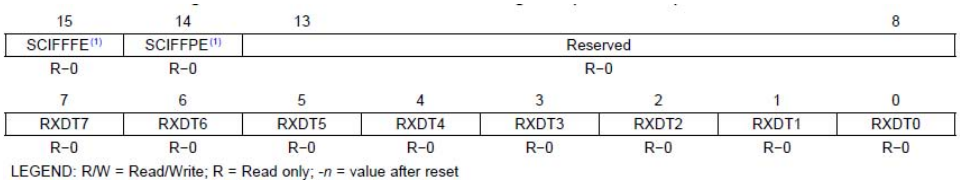
これで、基本的な設定は終わりです。最後にSCIをイネーブルにします。SCIをイネーブルにするフィールドが複数ありますので、注意下さい。

```
SciaRegs.SCICTL1.bit.TXENA = 1;          // Enable TX
SciaRegs.SCICTL1.bit.RXENA = 1;          // Enable RX
SciaRegs.SCICTL1.bit.SWRESET = 1;        // Re-Enable SCI
```

これら、SCICTL1レジスタのフィールドは、既に図 128と表 42に解説していますので、参照下さい。さて、次にSCIの受信FIFO割り込み関数、IsrSciRx()を見てみましょう。この関数は以下のようになっています。

```
interrupt void IsrSciRx(void){

  while(SciaRegs.SCIFFRX.bit.RXFFST != 0){

    while(SciaRegs.SCIFFTX.bit.TXFFST == 4);
      SciaRegs.SCITXBUF = SciaRegs.SCIRXBUF.bit.RXDT;
    }

    SciaRegs.SCIFFRX.bit.RXFFINTCLR = 1;
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;

  return;

}
```

まず、SCIFFRXレジスタのRXFFSTフィールドをチェックして、RXFFSTが0になるまで、whileループを行っています。FIFO受信割り込みは2段に設定しましたので、この割り込みが入ってきた時は、２個のデータがFIFOにはいっているはずです。ここで、わざわざ、RXFFSTをチェックしている理由は、この処理中にさらにデータを受信してきた時に、ついでに、受信するためです。今回のコードは、単に受信したデータをエコー・バック(そのまま返す)するだけですので、受信FIFOのデータを取り出し、そのまま送信FIFOに送るだけです。送信FIFOに送る時に、SCIFFTXレジスタのTXFFSTフィールドをチェックしています。これは、送信FIFOが満杯になっていないかチェックしています。満杯になっていなければ、送信FIFOに受信したデータをライトしています。受信FIFOのデータ取得レジスタは、SCIRXBUFレジスタ(図 134)です。このレジスタのRXDT[7 :0]からFIFOの受信データをリードします。このレジスタの15bit目(SCIFFFEフィールド)にて、フレーム・エラー(Stopビットが見つからない)の検出を、14bit目(SCIFFPEフィールド)にて、パリティ・エラーの検出ができますので、16bitで値をリードして、この２つのビットで、エラーをチェックという事もできます。今回は、特にエラーはケアしていません。

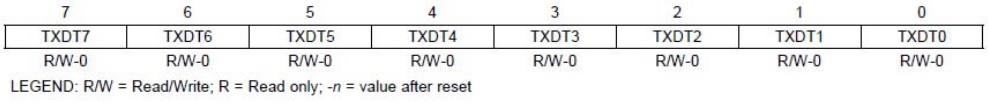


15	14	13					8
SCIFFFE(1)	SCIFFPE(1)	Reserved					
R-0	R-0	R-0					
7	6	5	4	3	2	1	0
RXDT7	RXDT6	RXDT5	RXDT4	RXDT3	RXDT2	RXDT1	RXDT0
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

図 134：SCIRXBUF レジスタ

送信FIFOへのライトは、SCITXBUFレジスタ(図 135)を使います。こちらは、シンプルに8ビットのレジスタです(16bitライトアクセスで問題ありません)。



7	6	5	4	3	2	1	0
TXDT7	TXDT6	TXDT5	TXDT4	TXDT3	TXDT2	TXDT1	TXDT0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

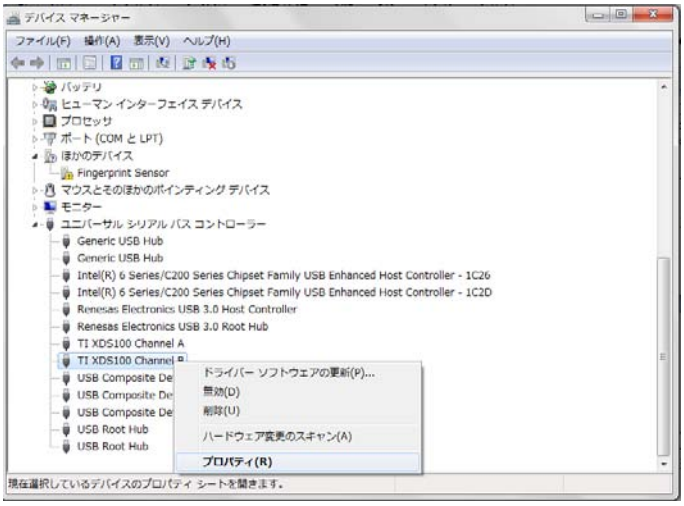
図 135：SCITXBUF レジスタ

最後に、受信FIFO割り込みフラグのクリアと、PIEACKのクリアを行っています。

さて、先ほども解説しましたが、今回は、受信FIFOの割り込みレベルを2段に設定しましたので、1個しかデータを受信していない時は、受信FIFO割り込みがはいらない事になります。これの対策として、CPUタイマ0を使って、定期的(このコードでは、1秒周期です)に、受信FIFOをチェックしています。この処理を行っているのが、CPUタイマ0割り込みのIsrTimer0関数です。中身は先ほど解説した内容とほぼ同じですので、解説は省きます。

それでは、ビルド、ロード、スタートをして、動作確認をしてみましょう。まず、ボードが正しく設定されているか確認して下さい。

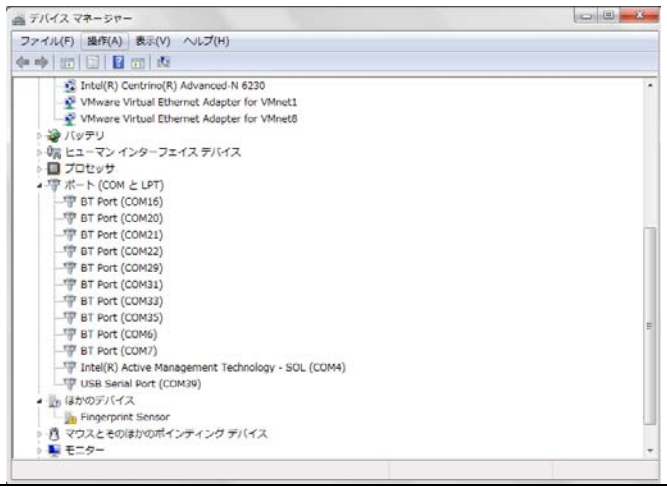
C2000 LaunchPad	S4スイッチ(204-211と書かれている大きなスイッチです。)をONに設定して下さい。
controlSTICK	ハードウェアの設定は必要ありませんが、Windowsのドライバの設定が必要です。Windowsにてデバイス・マネージャを立ち上げてください。 下図はWindows7の場合ですが、Windows XPやVistaの場合でも同様です。



ユニバーサル シリアル バス コントローラのTI XDS100 Channel Bを右クリックして、プロパティを選択して下さい。

上図のように、プロパティ・ウィンドウが表示されますので、詳細設定タブのセットアップ欄の”VCPをロードする”にチェックをいれてOKを押してください。次に、controlSTICKをPCから抜いて、再度挿してください。

上図のように、デバイス・マネージャのポート(COMとLPT)に、USB Serial Port (COMxx)が追加



	されたと思います。このポート番号を使用します。
Experimenter's Kit	[全てのボードが対象] マザーボード側のJ9をショート。J9には、ピンが立っていませんので、ピンを立てるか、ワイヤー等でショートして下さい。 [F28069カード] 設定は必要ありません。 [F28027カード] ボードのリビジョンと出荷時期によって異なるかもしれませんが、R10抵抗が実装されている場合には、このR10抵抗を取り除く必要があります。R10抵抗は、U3の右横にあります。 [F28035カード] SW1スイッチをOFFに設定して下さい。

PC側は、ハイパー・ターミナル等(Windows 7にはハイパー・ターミナルはありませんので、何等かのターミナル・ソフトをインストールして使用下さい)のターミナル・ソフトを開き、ポート番号を指定して、

ボー・レート：115200
データ長：8bit
パリティ：奇パリティ(odd)
ストップ・ビット：1ビット
フロー制御 :無

という設定(設定項目は、使用されているターミナル・ソフトによって異なります)にて、通信を開始してください。ターミナルで入力したキャラクタがそのまま表示されるはずです。1個だけ入力した場合は、１秒後に表示され、２個連続して入力した場合は、すぐに表示されると思います。

10 CLA(Control Law Accelerator:制御補償器アクセラレータ)

10.1 この章の目的

F2806x及びF2803xの一部デバイスには、CLA(Control Law Accelerator：制御補償器アクセラレータ)という非常にユニークな機能が搭載されています。このCLAは、C28x CPUと同じ周波数で動作するプログラマブルなアクセラレータで、ほぼCPUといってもよい機能を持っています。CLAのプログラミングは、以前はアセンブラ記述のみのサポートでしたが、コード生成ツールv6.1からC言語記述(いくつか制限事項があります)がサポートされるようになりましたので、使いやすくなりました。CLAの基本的なコンセプトとして、割り込み専用ミニCPUと考えると、理解しやすいと思います。この章では、CLAの基本的な使い方を解説します。CLAの詳細は以下のリファレンス・マニュアルを参照して下さい。

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18]のControl Law Accelerator (CLA)章*
F2802x/F2803x用	*TMS320x2803x Piccolo Conctol Law Accelerator (CLA) [SPRUGE6]*

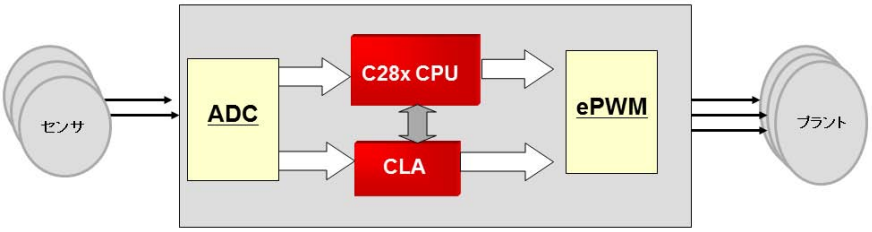


図 136：CLA のコンセプト

10.2 CLA の概要

CLAはF2803x及びF2806xの一部デバイスにてサポートされている機能です。このCLAは、浮動小数点命令セットを持ったプログラマブルなアクセラレータで、ほぼCPUと言ってよい機能を持っています（CPUと言い切って、デュアルCPUとうたうには、少し機能が足りないところがあります(例:スタックがハードで実装されていない等)）。CLAはC28xと並列に、同じ周波数で動作します。CLAは、ADC、ePWM、コンパレータのレジスタに直接アクセスする事ができます。このため、C28xメインCPUを全く使用せずに、ADCから値を取得し、CLAで演算を行い、ePWMにてPWMデューティーのアップデートという、一連のフィードバック処理を完結する事ができます。例えば、モータ制御とPFC(力率改善)制御をワンチップで制御するアプリケーションがあ インのC28x CPUで、処理は単純ですが比較的処理速度が求められるPFC処理はCLAで処理といったように、処理を完全に分担する事もできます。もちろん、C28x CPUが行っていた一部処理をCLAに実行させるといった、コプロセッサ的な使い方もできます。CLAがアクセスできるリソースは、ePWM、ADC(結果レジスタのみ)、コンパレータ、一部メモリに限られるという事に注意して下さい。CLAがあることで、CPUが2つ存在する事になりますので、単純にパフォーマンスが向上する事は簡単にご理解いただけると思います。ここで、もう一つのメリットを紹介します。

図 137を見てください。通常のCPUは、普段は何らかのバック・グランド処理を行っていて、割り込みが入ると、バック・グランド処理を一旦停止し、現状保存(コンテキスト・セーブ）を行い、割り込み処理に移ります。割り込み処理が終了したら、保存しておいたCPUの状態を元に戻します(コンテキスト・リストア)。ここで重要になるのが、現状保存(コンテキスト・セーブ)と回復(コンテキスト・リストア)は、そんなに長い時間ではありませんが、保存で10～20サイクル程度(コード依存)、回復で10～20サイクル程度(コード依存)のサイクルがかかるという点です。

一方、CLAは、バック・グラウンド処理というものがそもそもありません。通常はスリープ状態にあります。イベント(割り込み)が発生した時だけ、CLAが起動し、指定された処理を行い、処理が終了したら再びスリープ状態に戻ります。つまり、割り込み処理しかしないイメージです。このため、メインのC28x CPUで必ず必要なコンテキスト・セーブとコンテキスト・リストアが必要ありません（通常がスリープ状態のため、現状保存する必要がありません）ので、イベントが起きた時に、すぐに処理を始める事ができます。しかも、CLAは32bit単精度FPUを搭載していますので、C28x CPUにFPUが搭載されていない場合

は、C28x CPUよりも、高速に数値演算ができる可能性があります(固定小数点演算よりも、FPUによる浮動小数点演算の方が、演算が速いケースが多いです。これは、小数点の桁調整が必要ないためです)。

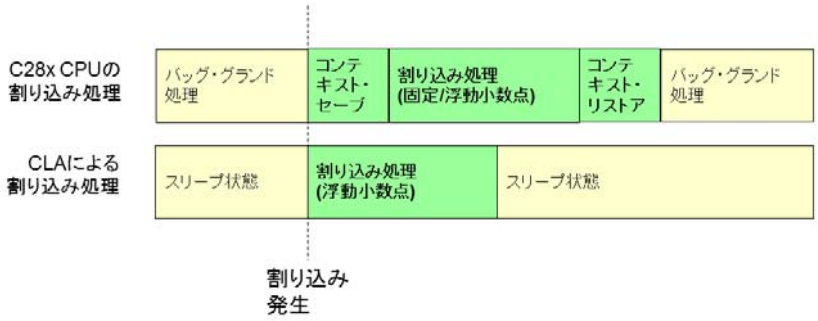


図 137：CPU と CLA の割り込み処理の違い

一般的に、デジタル制御電源アプリケーションは、スイッチング周波数が高く、高速応答が求められるため、制御周期が数100KHzに達する事がしばしばあります。この場合、1制御周期で許されるサイクル数は数100サイクルと、非常に少ないサイクル数になります。C28x CPUはハイ・パフォーマンスなCPUコアですが、当然割り込みのオーバーヘッドがどうしてもかかってしまいます。一方、CLAは割り込みのオーバー・ヘッドがほぼありませんので、10～20サイクルといった、少しのサイクルでも削減できる事は大きな魅力です。一方、CLAはC28x CPUと比べると、デバッグ機能が制限され、またC言語記述にも制限があるため、複雑で大きな処理を記述するのは、少し大変です。そのため、CLAは比較的シンプルで高速な処理を実行させるのに向いていると言えるでしょう（筆者の個人的な意見です）。

10.3 CLA と C28x CPU とメモリ・マッピング

CLAとC28x CPUとメモリ・マッピングの関係について解説します。図 138に、その全体像を示します。

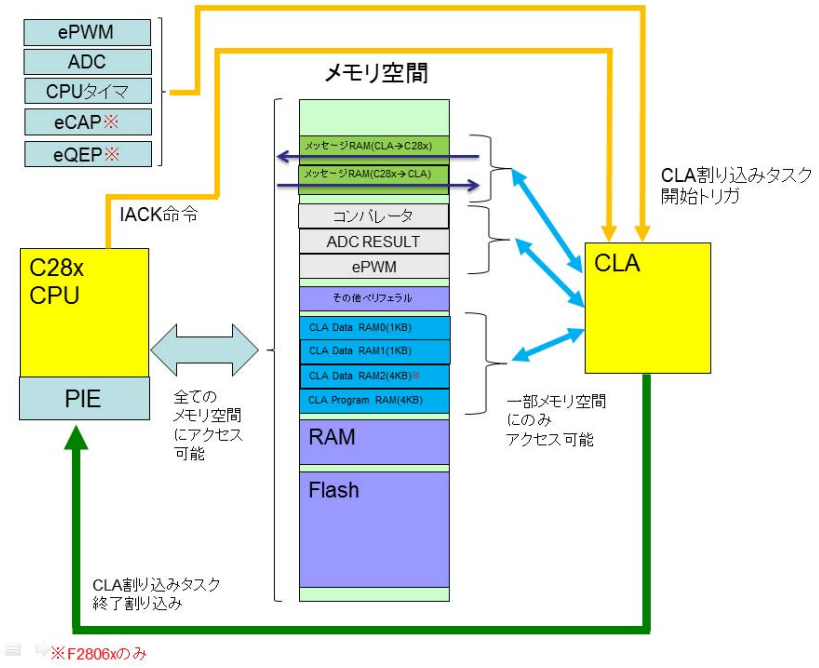


図 138：CLA と C28x CPU とメモリ・マッピング

C28x CPUとCLAは、基本的には同じメモリ空間を共有しています（同じアドレスを使用します）。C28x CPUは、当然、全メモリ空間(0x00_0000～0x3F_FFFF)にアクセスする事ができます。一方、CLAは、一部メモリ空間にのみアクセスする事ができます。CLAがアクセスできるメモリ空間は、

- 一部ペリフェラル・レジスタ(ePWM(HRPWM含む)、ADCのADCRESULT0-15、コンパレータ)
- 共用メモリ

に限られます。共用メモリは、以下の３つがあります。

- CLA Program RAM
  CLAのコード(プログラム)を配置するためのメモリです。CLAを使わない場合は、C28x CPUが通常のRAMとして使用する事ができます。リセット直後はC28x CPUに所属しています。CLAを使う場合は、C28x CPUがCLA用のコードをこの領域にロード（コピー）して、CLAにアクセス権を移します。
- CLA Data RAM
  CLAのデータを配置するためのメモリです。CLAを使わない場合は、C28x CPUが通常のRAMとして使用する事ができます。CLA Data RAMは、複数のブロック(F2803xは2ブロック、F2806xは3ブロック)から構成されており、ブロック毎にCLAに割り当てるかC28x CPUに割り当てるかを選択する事ができます。
- CLA Shared Message RAM
  CLAとC28x CPUとの通信を行うためのメモリで、次の２種類あります。
  - ✧ CLA to CPU Message RAM
    CLAからCPUへデータを送信するためのRAMです。CLAはR/W可能です。CPUはRead Onlyです。
  - ✧ CPU to CLA Message RAM
    CPUからCLAへデータを送信するためのRAMです。CPUはR/W可能です。CLAはRead Onlyです。

これらの共用レジスタ、メモリに対して、C28x CPU及びCLAが同時にアクセスした場合のアービトレーションにつきましては、リファレンス・マニュアルに掲載されていますので、詳細を知りたいユーザーは、リファレンス・マニュアルを参照下さい。

CLAのプログラムはCLA Program RAM上に置く必要があります。CLAはFlashメモリにアクセスする事ができませんので、CLAを起動する前に、C28x CPUがFlashからCLA Program RAMにコードをロード(コピー)する必要があります。リセット直後は、CLA Program RAMはC28x CPUに所属していますので、CLAのコードをロード(コピー)したら、このメモリをCLA所属に変更します。CLA Data Memoryもリセット直後はC28x CPUに所属していますので、必要に応じて、CLAに所属を変えます。CPUとCLAとのデータのやりとりは、CLA Shared Message RAMを使用します。このMessage RAMは、C28x CPU→CLA用と、CLA→C28x CPU用の２つのブロックがあります。

10.4 CLA の割り込みタスク

既に述べましたが、CLAはCLA割り込みタスクが動いていない時はスリープ状態で、イベント発生時にのみ、CLA割り込みタスクが起動して処理を行います。“割り込み“という言葉が非常に紛らわしいですので、ここで以降に出てくる言葉の定義をしておきます。

- CLA割り込みタスク：CLAの処理タスク
- ペリフェラル割り込み：ePWMやADCなどの、C28xのペリフェラル割り込み(PIE割り込み)。

CLA割り込みタスクは、8個まで登録する事ができます。CLA割り込みタスクには、MVECT1～8の8本の割り込みベクタが用意されていて、このMVECTに各CLA割り込みタスクの開始アドレスを設定しておきます。後程詳しく解説しますが、MVECT1～8に登録するアドレスは、CLA Program RAMの開始アドレスからのオフセットアドレスになります（例えば、CLA Program RAMの開始アドレスが0x00_9000番地で、CLA割り込みタスクの開始アドレスが0x00_9200番地の場合は、0x200を登録します）。

CLA割り込みタスクを起動するためには、開始トリガが必要です。開始トリガを発生できるのは、C28x CPUによるIACK命令、ePWM(EPWMx_INT割り込み信号)、ADC(ADCINTx割り込み信号)、CPUタイマ0(TINT0割り込み信号)、eQEP(EQEPx_INT割り込み信号)、eCAP(ECAPx_INT割り込み信号)になります。eQEP及びeCAPはF2806xのみのサポートです。各CLA割り込みタスクに対して、自由に開始トリガが選択できるのではなく、CLA割り込みタスク番号ごとに、選択でき

るトリガが決まっています(C28x CPUによるIACK命令は全ての割り込みタスク番号を起動する事ができます)ので、どのリソースがどの番号を起動できるかは、リファレンス・マニュアルで確認する必要があります。

C28x CPUのIACK命令によるCLA割り込みタスクの起動は、

IACK #CLA割り込みタスク番号 (例: IACK #0x0001 → CLA割り込みタスク1番の起動)

というアセンブラ命令を実行する事で実現できます。C言語にてアセンブラを記述するには、

asm(" アセンブラ命令 ");

ですので、

asm(" IACK #0x0001 ");

と記述すれば、CLA割り込みタスク1番を起動する事ができます。ペリフェラル・ヘッダ・ファイルに、このタスク起動のC言語要マクロ関数が用意されています。これは、後程、紹介します。

CLA割り込みタスクが終了した時に、C28x CPUのPIEにCLA割り込みタスク終了割り込み信号が、必ず入ります。CLAには、この割り込みの許可ビットはありません。必ず割り込み信号が発生しますので、PIEにて、イネーブル、ディセーブルの制御を行います。

10.5 CLA 用の C コンパイラの仕様

CLAは、以前はアセンブラのみのサポートでしたが、コード生成ツールのVer6.1.0からC言語での記述をサポートするようになりました。しかし、CLAは、少し特殊なCPUですので、C言語記述もいくつか制限及び注意点があります。以下にその制限を記します。

- C++はサポートしていません。
- データ型がC28xと違うものがあります(以下にまとめます)

データ型	C28x	CLA
char, signed char, unsigned char	16bit	
short, unsigned short	16bit	
int, signed int, unsigned int	16bit	32bit
long, signed long, unsigned long	32bit	
long long, singed long long, unsigned long long	64bit	32bit
Float	IEEE-32bit	
Double	IEEE-32bit	
long double	IEEE-64bit	IEEE-32bit
ポインタ(Large Memory Modelの場合)	22bit	16bit

特にint型がC28xとCLAでは異なりますので、16bit整数はshort、32bit整数はlongを使った方が無難です。もしくは、typedefにて、int16, int32のようにビット数を明確にして使うべきでしょう。

- FAST_FUNC_CALL pragmaは使用できません。
- Cの標準関数は使用できません。
- far及びioportキーワードは使えません(C28xでもほぼ使いませんので、ほとんど気にする必要はありません)
- 関数コールは一段のみです。２段以上の関数コールはできません。
- 関数コールの引数は最大２つまでです。

- 関数の再帰コールはサポートしていません
- 整数の除算、モジュロ、unsigned整数の比較はサポートしていません
- ヒープはサポートしていません
- 初期値があるグローバル変数、static変数の定義はできません。例えば、

```
unsigned int val = 0x1234;
interrupt void cla_task1(void){
   int i;

   i = val + 123;
   …
   …

}
```

のようなケースです。CLAの処理モデルでは、割り込みしかないというモデルですので、変数の初期化を一切行いません。このようなコードを記述した場合は、コンパイラ・エラーがでます。グローバル変数の初期値は、プログラムにて与える必要があります。初期値が無いグローバル/static変数は使用する事ができます。また、仕様としては、const変数が使える事になっていますが、v6.1.0では、不具合のためconstは使用できません（コンパイラ・エラーがでます）。これは、将来のバージョンで改善されると思います。

10.6 CLA とプロジェクトとリンカ・コマンド・ファイル

CLAを使う場合でも、CCSのProjectは1つになります。つまり、１つのProjectに、C28x CPU用の通常のコードと、CLA用のコードが混在する形になります(ファイルは別々です)。CLA用のCコードは、必ず、.claという拡張子のファイル名にする必要があります。Cコンパイラは.claという拡張子のファイルをCLA用のCコードであると自動的に認識し、.claファイルに対してCLA用のコードを生成してくれます。C28x CPU用のコードと、CLA用のコードを、同じファイルにする事はできません。必ず別ファイルにして下さい。

CLAを使う場合は、リンカ・コマンド・ファイルの記述方法をきちんと理解した方がよいでしょう。Header FilesにCLA用の雛形が用意されていますが、残念ながら、現時点のバージョンでは、完全では無い印象です。必要に応じて、修正が必要になります。

リンカ・コマンド・ファイルの記述方法の基礎は、3.3(24ページ)章にて解説していますので、まず、この内容を理解しておいてください。CLAのCコンパイラは、以下の4つのセクションを自動生成しますので、これらのセクションをリンカ・コマンド・ファイルにて定義する必要があります。

セクション名	Page	初期値を持つセクション?	内容
.bss_cla	1	NO	CLAを使う時のみ生成されるセクションです。CLAのグローバル変数領域です。このセクションは、必ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。
.const_cla	1	YES	CLAを使う時のみ生成されるセクションです。CLAのconst変数領域です。このセクションは、必ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。注意:v6.1.0では、不具合のためconstを使う事ができません(C28x CPU用のC言語ファイルにて定義されているconst定数で、CLAからアクセスできるRAMに配置されている変数は、extern宣言して参照できます)。そのため、このセクションはv6.1.0では生成されません。既にこの不具合はレポートされていますので、今後のバージョンで改善されると思います。念のため、このセクションをリンカ・コマンド・ファイルに記載しておいてください。尚、おそらく、constを使用する場合は、Cla1Progと同様、このセクションのコピー(FlashからCLA Data RAM領域に)が必要になると思います。このドキュメントでは、それは行っていませんので、将来のバージョンでconstが使

			用できるようになって、constを使用する場合は、その使用方法をよく確認して使用下さい。
CLAscratch	1	NO	CLAのスクラッチ・パッド領域(ソフトウェア・スタック、一時領域等)です。このセクションは少し特殊な記述方法が必要で、また、必ずCLA Data RAM領域に配置する必要があります。
Cla1Prog	0	YES	CLAのプログラム領域です。必ずCLA Program RAM領域に配置する必要があります。FlashからCLA Program RAMにコピーする事が必須なため、少し特殊な記述方法が必要です。

また、CLA用のCLA Math Libraryを使用する場合は、さらに以下のセクションが必要になります。

セクション名	Page	初期値を持つセクション?	内容
CLA1mathTables	1	YES	CLAを使う時のみ生成されるセクションです。CLA Math Libraryを使う時に生成されるセクションです。このセクションは、必ずCLA Data RAM領域(CLAの章にて解説します)に配置する必要があります。

次の節から、実際のCLAの使用例を解説しますが、そこで使うリンカ・コマンド・ファイルを作成しておきましょう。F2803xの場合は、ある程度使える雛形がHeader Filesに用意されていますが、残念ながら、F2806xの場合は、RAM用の雛形しかなく、Flash用の雛形がありません。まず、F2806xの場合の例を示します。

F28069用、Flashを使った、CLA用のリンカ・コマンド・ファイル, F28069_CLA_C_sample.cmd

```
CLA_SCRATCHPAD_SIZE=0x100;
_Cla1Prog_Start = _Cla1funcsRunStart;
--undef_sym=__cla_scratchpad_end
--undef_sym=__cla_scratchpad_start

MEMORY
{
PAGE 0 :  /* Program Memory */
   CLAPROG     : origin = 0x009000, length = 0x001000     /* on-chip RAM block L3 (CLA Program RAM) */
   RAML4       : origin = 0x00A000, length = 0x002000     /* on-chip RAM block L4 */
   OTP         : origin = 0x3D7800, length = 0x000400     /* on-chip OTP */
   FLASHH      : origin = 0x3D8000, length = 0x004000     /* on-chip FLASH */
   FLASHG      : origin = 0x3DC000, length = 0x004000     /* on-chip FLASH */
   FLASHF      : origin = 0x3E0000, length = 0x004000     /* on-chip FLASH */
   FLASHE      : origin = 0x3E4000, length = 0x004000     /* on-chip FLASH */
   FLASHD      : origin = 0x3E8000, length = 0x004000     /* on-chip FLASH */
   FLASHC      : origin = 0x3EC000, length = 0x004000     /* on-chip FLASH */
   FLASHA      : origin = 0x3F4000, length = 0x003F80     /* on-chip FLASH */
   CSM_RSVD    : origin = 0x3F7F80, length = 0x000076     /* Part of FLASHA.  Program with all 0x0000 when CSM is in use. */
   BEGIN       : origin = 0x3F7FF6, length = 0x000002     /* Part of FLASHA.  Used for "boot to Flash" bootloader mode. */
   CSM_PWL_P0  : origin = 0x3F7FF8, length = 0x000008     /* Part of FLASHA.  CSM password locations in FLASHA */
   FPUTABLES   : origin = 0x3FD860, length = 0x0006A0     /* FPU Tables in Boot ROM */
   IQTABLES    : origin = 0x3FDF00, length = 0x000B50     /* IQ Math Tables in Boot ROM */
   IQTABLES2   : origin = 0x3FEA50, length = 0x00008C     /* IQ Math Tables in Boot ROM */
   IQTABLES3   : origin = 0x3FEADC, length = 0x0000AA  /* IQ Math Tables in Boot ROM */
   ROM         : origin = 0x3FF3B0, length = 0x000C10     /* Boot ROM */
   RESET       : origin = 0x3FFFC0, length = 0x000002     /* part of boot ROM  */
```

```
   VECTORS    : origin = 0x3FFFC2, length = 0x00003E     /* part of boot ROM  */

PAGE 1 :
   BOOT_RSVD  : origin = 0x000000, length = 0x000050     /* Part of M0, BOOT rom will use this for stack */
   RAMM0      : origin = 0x000050, length = 0x0003B0     /* on-chip RAM block M0 */
   RAMM1      : origin = 0x000400, length = 0x000400     /* on-chip RAM block M1 */
   CLARAM2    : origin = 0x008000, length = 0x000800     /* on-chip RAM block L0 (CLA Data RAM2) */
   CLARAM0    : origin = 0x008800, length = 0x000400     /* on-chip RAM block L1 (CLA Data RAM0) */
   CLARAM1    : origin = 0x008C00, length = 0x000400     /* on-chip RAM block L2 (CLA Data RAM1) */
   RAML5      : origin = 0x00C000, length = 0x002000     /* on-chip RAM block L5 */
   RAML6      : origin = 0x00E000, length = 0x002000     /* on-chip RAM block L6 */
   RAML7      : origin = 0x010000, length = 0x002000     /* on-chip RAM block L7 */
   RAML8      : origin = 0x012000, length = 0x002000     /* on-chip RAM block L8 */
   USB_RAM    : origin = 0x040000, length = 0x000800     /* USB RAM  */
   FLASHB     : origin = 0x3F0000, length = 0x004000     /* on-chip FLASH */

   CLA1_MSGRAMLOW     : origin = 0x001480, length = 0x000080
   CLA1_MSGRAMHIGH    : origin = 0x001500, length = 0x000080

}

SECTIONS
{
   .cinit          : > FLASHA,     PAGE = 0
   .pinit          : > FLASHA,     PAGE = 0
   .text           : > FLASHA,     PAGE = 0
   codestart       : > BEGIN,      PAGE = 0
   ramfuncs        : LOAD = FLASHD,
                     RUN = RAML4,
                     LOAD_START(_RamfuncsLoadStart),
                     RUN_START(_RamfuncsRunStart),
                     LOAD_SIZE(_RamfuncsLoadSize),
                     PAGE = 0
   csmpasswds      : > CSM_PWL_P0, PAGE = 0
   csm_rsvd        : > CSM_RSVD,   PAGE = 0
   .stack          : > RAMM0,      PAGE = 1
   .ebss           : > RAML5,      PAGE = 1
   .esysmem        : > RAML5,      PAGE = 1
   .econst         : > FLASHA,     PAGE = 0
   .switch         : > FLASHA,     PAGE = 0
   IQmath          : > FLASHA,     PAGE = 0
   IQmathTables    : > IQTABLES,   PAGE = 0, TYPE = NOLOAD
   FPUmathTables   : > FPUTABLES,  PAGE = 0, TYPE = NOLOAD

  /* Uncomment the section below if calling the IQNexp() or IQexp()
     functions from the IQMath.lib library in order to utilize the
     relevant IQ Math table in Boot ROM (This saves space and Boot ROM
     is 1 wait-state). If this section is not uncommented, IQmathTables2
     will be loaded into other memory (SARAM, Flash, etc.) and will take
     up space, but 0 wait-state is possible.
```

```
*/
/*
IQmathTables2    : > IQTABLES2, PAGE = 0, TYPE = NOLOAD
{

        IQmath.lib<IQNexpTable.obj> (IQmathTablesRam)

}
*/
 /* Uncomment the section below if calling the IQNasin() or IQasin()
    functions from the IQMath.lib library in order to utilize the
    relevant IQ Math table in Boot ROM (This saves space and Boot ROM
    is 1 wait-state). If this section is not uncommented, IQmathTables2
    will be loaded into other memory (SARAM, Flash, etc.) and will take
    up space, but 0 wait-state is possible.
 */
 /*
 IQmathTables3    : > IQTABLES3, PAGE = 0, TYPE = NOLOAD
 {

         IQmath.lib<IQNasinTable.obj> (IQmathTablesRam)

 }
 */

.reset           : > RESET,     PAGE = 0, TYPE = DSECT
vectors          : > VECTORS,   PAGE = 0, TYPE = DSECT

/* For CLA*/
Cla1Prog         : LOAD = FLASHD,
                   RUN = CLAPROG,
                   LOAD_START(_Cla1funcsLoadStart),
                   LOAD_SIZE(_Cla1funcsLoadSize)
                   RUN_START(_Cla1funcsRunStart),
                   PAGE = 0
CLAscratch       :
                   { *.obj(CLAscratch)
                  . += CLA_SCRATCHPAD_SIZE;
                   *.obj(CLAscratch_end) } > CLARAM1,
                   PAGE = 1

Cla1ToCpuMsgRAM    : > CLA1_MSGRAMLOW,   PAGE = 1
CpuToCla1MsgRAM    : > CLA1_MSGRAMHIGH,  PAGE = 1
.bss_cla         : > CLARAM0,  PAGE = 1
.const_cla       : LOAD = FLASHB,
                   RUN = CLARAM0,
                   LOAD_START(_Cla1ConstLoadStart),
                   RUN_START(_Cla1ConstRunStart),
                   LOAD_SIZE(_Cla1ConstLoadSize),
```

```
                PAGE = 1
  CLA1mathTables      : LOAD = FLASHB,
                RUN = CLARAM1,
                LOAD_START(_Cla1mathTablesLoadStart),
                LOAD_END(_Cla1mathTablesLoadEnd),
                RUN_START(_Cla1mathTablesRunStart),
                LOAD_SIZE(_Cla1mathTablesLoadSize),
                PAGE = 1

}
```

CLA用に必要な個所を、赤字にしました。緑字はコメント・アウトですので、今回は無くても構いません(IQmathを使っている場合で、IQNexp(), IQexp()関数を使う場合にのみ、コメントをはずします)。赤字の箇所を解説していきます。

まず、最初の

CLA_SCRATCHPAD_SIZE=0x100;

は後程出てくる、スクラッチ・パッド領域のサイズを指定していて、0x100ワード確保しています。スクラッチ・パッド領域は、スタック領域のようなものです。CLAのCコンパイラは、このスクラッチ・パッド領域をソフトウェア・スタック(ハードウェアでスタックをサポートしていないため、ソフトウェアでのエミュレーションになります)や、一時領域として使います。どれだけのサイズが必要かは、コード依存ですが、スタックと同様と考えれば、およそサイズも想像できます。

次の

_Cla1Prog_Start = _Cla1funcsRunStart;

ですが、これは、CLA Program RAMの先頭アドレスのラベルを取得しています。各CLA割り込みタスクの先頭アドレスを、MVECT1～8レジスタに設定する事になりますが、このMVECTxレジスタに設定するアドレスは、CLA Program RAMの先頭アドレスからのオフセット・アドレスになります。そのため、プログラムにて、CLA Program RAMの先頭アドレスが必要になります。この定義にて、その先頭アドレスが、_Cla1Prog_Start(C言語では、Cla1Prog_Startです。最初のアンダースコアは、アセンブラ言語の時に必要で、C言語の場合は、最初のアンダースコアが省略されます)というシンボルを使う事で取得する事ができます。_Cla1funcsRunStatというラベルは、後程出てきますが、CLA Program RAMの先頭アドレスを指しています。実は、この一行は、本当は必須ではありません。プログラムにて、_Cla1funsRunStartというラベルを使えば済みますので、ちょっと冗長です。しかし、Header FilesのCLA用のサンプル・コードでは、この方法を使っていますので、互換性を考えて、今回もこの方法をとります。

次の

--undef_sym=__cla_scratchpad_end
--undef_sym=__cla_scratchpad_start

この２行ですが、少々、説明が難しくなります。CLAのCコンパイラは、__cla_scratchpad_endと__cla_scratchpad_startというラベルを持つ、中身が無いがセクション（中身はありませんが、ラベルは必要なのです）を生成します。実態がないため、何も指定しないと、リンク時に省かれてしまい、いろいろと不都合が出てきます。その不都合を解決するために、強制的にリンカにこのシンボルを登録させるための記述です。この原理は、理解しなくても、全く問題ありませんので、単純に、CLAのスクラッチ・パッド領域を定義するために、必須な２行と考え、必ず記述して頂ければと思います。

MEMORY{}欄のPAGE 0(プログラム領域)に、CLAPROGが定義されています。ここが、CLA Program RAM領域になります。また、PAGE 1(データ領域)に、CLARAM0/1/2が定義されています。これは、CLA Data RAM領域になります。同じくPAGE 1には、CLA1_MSGRAMLOW/CLA1_MSGRAMHIGHが定義されています。これは、CLA Shared RAMになります。CLA1_MSGRAMLOWがCLAからCPUへのメッセージRAM(CLA:R/W, CPU:Read Only)、CLA1_MSGRAMHIGHがCPUからCLAへのメッセージRAM(CLA:Read Only, CPU:R/W)になります。

次にSECTIONS{}欄を見ていきます。まず、Cla1Progセクションの配置が定義されています。

```
Cla1Prog        : LOAD = FLASHD,
                  RUN = CLAPROG,
                  LOAD_START(_Cla1funcsLoadStart),
                  LOAD_SIZE(_Cla1funcsLoadSize)
```

```
        RUN_START(_Cla1funcsRunStart),
        PAGE = 0
```

Cコンパイラは、CLA用のコードを、Cla1Progセクションとして生成します。CLAのコードになりますので、CLA Program RAM領域で実行させる必要があります。既に解説していますが、CLAはFlashメモリにアクセスできないため、C28x CPUがCLA起動前に、FlashからCLA Program RAM領域にコピーする必要があります。そのため、まずLOAD=FLASHDにて、最初にコードを配置する場所を指定します。そして、実際に動作する場所をRUN=CLAPROGにて指定します。さらに、コピーする時に、コピー元アドレス(ロード時)、コピー先アドレス(ラン時)、サイズが必要となりますので、それぞれLOAD_START, RUN_START, LOAD_SIZEを使って、そのアドレス・ラベルを取得しています。コピーは自動的に行われませんので、C28x CPUのプログラムにてコピーを記述する必要がある事に注意して下さい。

次の

```
CLAscratch      :
        { *.obj(CLAscratch)
        . += CLA_SCRATCHPAD_SIZE;
        *.obj(CLAscratch_end) } > CLARAM1,
        PAGE = 1
```

ですが、これは、スクラッチ・パッド領域の配置を指定しています。ここの記述については、詳細は割愛します。スクラッチ・パッド領域を記述するためには、必ずこのように記述すると考えてください。この記述では、CLARAM1に配置するように指定しています。配置場所（今回はCLARAM1）は、変更してもかまいません。ただし、必ずCLA Data RAMに配置する必要があります。

次の、

```
Cla1ToCpuMsgRAM   : > CLA1_MSGRAMLOW,  PAGE = 1
CpuToCla1MsgRAM   : > CLA1_MSGRAMHIGH,  PAGE = 1
```

この２行は、CLA Shared RAMの配置を指定しています。Cla1ToCpuMsgRAMとCpuToClaMsgRAMというセクションは、C言語が自動生成するセクションではありません。ユーザーがプログラムにて指定するセクション名です。そのため、セクション名は何でもかまいませんが、今回提示するサンプルコードでは、このセクション名を使いますので、ここでその配置を定義しています。後程提示する、サンプル・コードを見て頂くと、この記述の意味が理解できると思います。

次の、

```
.bss_cla      : > CLARAM0,  PAGE = 1
```

は、CLAのグローバル変数セクションの配置を定義しています。Cコンパイラは、グローバル変数がある場合は、.bss_claセクションを自動生成しますので、必ず、.bss_claをCLA Data RAMのどこかに配置して下さい。

次の、

```
.const_cla    : LOAD = FLASHB,
        RUN = CLARAM0,
        LOAD_START(_Cla1ConstLoadStart),
        RUN_START(_Cla1ConstRunStart),
        LOAD_SIZE(_Cla1ConstLoadSize),
        PAGE = 1
```

は、v6.1.0を使う場合は、実際には必要ありません。CLA Cコンパイラは、仕様としてはconst変数を使用でき、その際は.const_claというセクションを生成します。しかし、v6.1.0はバグにより、constを使用する事ができませんので、このセクションが生成される事はありません。将来のバージョンにてこのバグが修正された時を考え、この記述を入れておきました。

最後の

```
CLA1mathTables   : LOAD = FLASHB,
        RUN = CLARAM1,
        LOAD_START(_Cla1mathTablesLoadStart),
        LOAD_END(_Cla1mathTablesLoadEnd),
        RUN_START(_Cla1mathTablesRunStart),
        LOAD_SIZE(_Cla1mathTablesLoadSize),
        PAGE = 1
```

ですが、これも、今回提示するサンプルコードでは、必要ありません。CLAにはCLAmathライブラリというライブラリが用意されています。このライブラリには、三角関数や除算などの関数が収められています。CLAmathライブラリを使う場合は、このCLA1mathTablesセクションを配置する事が必要になります。今回提示するサンプル・コードはCLAmathライブラリは使用していませんので、この記述はなくてもかまいません。F2806x用の解説はこれで終わりです。

次にF2803x用のリンカ・コマンド・ファイルを解説します。F2803x用には、Header FilesにFlash用のCLAリンカ・コマンド・ファイルが用意されていますので、今回提示するサンプル・コードにおいては、それを使用しても問題ありません。しかし、少し手直しした方がよい項目がありますので、それを解説します。以下にその修正を示します。

F28035_CLA_C.cmd

```
/*
//###########################################################################
//
// FILE:                    F28035_CLA_C.cmd
//
// TITLE:                   Linker Command File For F28035 Device
//
//###########################################################################
// $TI Release: F2803x C/C++ Header Files and Peripheral Examples V126 $
// $Release Date: November 30, 2011 $
//###########################################################################
*/

/* ======================================================
// For Code Composer Studio V2.2 and later
// ---------------------------------------
// In addition to this memory linker command file,
// add the header linker command file directly to the project.
// The header linker command file is required to link the
// peripheral structures to the proper locations within
// the memory map.
//
// The header linker files are found in <base>\DSP2803x_Headers\cmd
//
// For BIOS applications add:      DSP2803x_Headers_BIOS.cmd
// For nonBIOS applications add:   DSP2803x_Headers_nonBIOS.cmd
====================================================== */

/* ======================================================
// For Code Composer Studio prior to V2.2
// ---------------------------------------
// 1) Use one of the following -l statements to include the
// header linker command file in the project. The header linker
// file is required to link the peripheral structures to the proper
// locations within the memory map                          */

/* Uncomment this line to include file only for non-BIOS applications */
/* -l DSP2803x_Headers_nonBIOS.cmd */

/* Uncomment this line to include file only for BIOS applications */
```

```
/* -l DSP2803x_Headers_BIOS.cmd */

/* 2) In your project add the path to <base>\DSP2803x_headers\cmd to the
   library search path under project->build options, linker tab,
   library search path (-i).
/*========================================================== */

/* Define the memory block start/length for the F28035
   PAGE 0 will be used to organize program sections
   PAGE 1 will be used to organize data sections

   Notes:
        Memory blocks on F2803x are uniform (ie same
        physical memory) in both PAGE 0 and PAGE 1.
        That is the same memory region should not be
        defined for both PAGE 0 and PAGE 1.
        Doing so will result in corruption of program
        and/or data.

        L0 memory block is mirrored - that is
        it can be accessed in high memory or low memory.
        For simplicity only one instance is used in this
        linker file.

        Contiguous SARAM memory blocks or flash sectors can be
        be combined if required to create a larger memory block.
*/
_Cla1Prog_Start = _Cla1funcsRunStart;
-heap 0x200
-stack 0x200

// Define a size for the CLA scratchpad area that will be used
// by the CLA compiler for local symbols and temps
// Also force references to the special symbols that mark the
// scratchpad are.
CLA_SCRATCHPAD_SIZE = 0x100;                    /*この行のコメント・アウトをはずす*/
--undef_sym=__cla_scratchpad_end
--undef_sym=__cla_scratchpad_start

MEMORY
{
PAGE 0:   /* Program Memory */
          /* Memory (RAM/FLASH/OTP) blocks can be moved to PAGE1 for data allocation */
   RAMM0      : origin = 0x000050, length = 0x0003B0     /* on-chip RAM block M0 */
   RAML3      : origin = 0x009000, length = 0x001000     /* on-chip RAM block L3 */
   OTP        : origin = 0x3D7800, length = 0x000400     /* on-chip OTP */
   FLASHH     : origin = 0x3E8000, length = 0x002000     /* on-chip FLASH */
   FLASHG     : origin = 0x3EA000, length = 0x002000     /* on-chip FLASH */
   FLASHF     : origin = 0x3EC000, length = 0x002000     /* on-chip FLASH */
```

```
   FLASHE     : origin = 0x3EE000, length = 0x002000     /* on-chip FLASH */
   FLASHD     : origin = 0x3F0000, length = 0x002000     /* on-chip FLASH */
   FLASHC     : origin = 0x3F2000, length = 0x002000     /* on-chip FLASH */
   FLASHA     : origin = 0x3F6000, length = 0x001F80     /* on-chip FLASH */
   CSM_RSVD   : origin = 0x3F7F80, length = 0x000076     /* Part of FLASHA.  Program with all 0x0000 when CSM is in use. */
   BEGIN      : origin = 0x3F7FF6, length = 0x000002     /* Part of FLASHA.  Used for "boot to Flash" bootloader mode. */
   CSM_PWL_P0 : origin = 0x3F7FF8, length = 0x000008     /* Part of FLASHA.  CSM password locations in FLASHA */

   IQTABLES   : origin = 0x3FE000, length = 0x000B50     /* IQ Math Tables in Boot ROM */
   IQTABLES2  : origin = 0x3FEB50, length = 0x00008C     /* IQ Math Tables in Boot ROM */
   IQTABLES3  : origin = 0x3FEBDC, length = 0x0000AA              /* IQ Math Tables in Boot ROM */

   ROM        : origin = 0x3FF27C, length = 0x000D44     /* Boot ROM */
   RESET      : origin = 0x3FFFC0, length = 0x000002     /* part of boot ROM  */
   VECTORS    : origin = 0x3FFFC2, length = 0x00003E     /* part of boot ROM  */

PAGE 1 :   /* Data Memory */
          /* Memory (RAM/FLASH/OTP) blocks can be moved to PAGE0 for program allocation */
          /* Registers remain on PAGE1                                                  */
   BOOT_RSVD  : origin = 0x000000, length = 0x000050     /* Part of M0, BOOT rom will use this for stack */
   RAMM1      : origin = 0x000400, length = 0x000400     /* on-chip RAM block M1 */
   RAML0      : origin = 0x008000, length = 0x000800     /* on-chip RAM block L0 */
   CLARAM0    : origin = 0x008800, length = 0x000400
   CLARAM1    : origin = 0x008C00, length = 0x000400

   CLA1_MSGRAMLOW   : origin = 0x001480, length = 0x000080
   CLA1_MSGRAMHIGH  : origin = 0x001500, length = 0x000080

   FLASHB     : origin = 0x3F4000, length = 0x002000     /* on-chip FLASH */

}

/* Allocate sections to memory blocks.
   Note:
        codestart user defined section in DSP28_CodeStartBranch.asm used to redirect code
                execution when booting to flash
        ramfuncs  user defined section to store functions that will be copied from Flash into RAM
*/

SECTIONS
{

   /* Allocate program areas: */
   .cinit           : > FLASHA      PAGE = 0
   .pinit           : > FLASHA,     PAGE = 0
   .text            : > FLASHC      PAGE = 0
   codestart          : > BEGIN       PAGE = 0
   ramfuncs           : LOAD = FLASHD,
                        RUN = RAMM0,
                        LOAD_START(_RamfuncsLoadStart),
```

```
                LOAD_SIZE(_RamfuncsLoadSize),
                RUN_START(_RamfuncsRunStart),
                PAGE = 0

csmpasswds        : > CSM_PWL_P0  PAGE = 0
csm_rsvd          : > CSM_RSVD    PAGE = 0

/* Allocate uninitalized data sections: */
.stack            : > RAMM1       PAGE = 1
.cio              : > RAML0       PAGE = 1
.sysmem           : > RAMM1       PAGE = 1
.ebss             : > RAML0       PAGE = 1
.esysmem          : > RAML0       PAGE = 1


/* Initalized sections go in Flash */
/* For SDFlash to program these, they must be allocated to page 0 */
.econst           : > FLASHA      PAGE = 0
.switch           : > FLASHA      PAGE = 0

/* Allocate IQ math areas: */
IQmath            : > FLASHA      PAGE = 0          /* Math Code */
IQmathTables      : > IQTABLES,   PAGE = 0, TYPE = NOLOAD

Cla1Prog          : LOAD = FLASHD,
                RUN = RAML3,
                LOAD_START(_Cla1funcsLoadStart),
                LOAD_SIZE(_Cla1funcsLoadSize)
                RUN_START(_Cla1funcsRunStart),
                LOAD_SIZE(_Cla1funcsLoadSize),    /* ミスプリです。Warningが出るので、この行を消去して下さい*/
                PAGE = 0

Cla1ToCpuMsgRAM   : > CLA1_MSGRAMLOW,   PAGE = 1
CpuToCla1MsgRAM   : > CLA1_MSGRAMHIGH,  PAGE = 1
Cla1DataRam0      : > CLARAM0,    PAGE = 1
Cla1DataRam1      : > CLARAM1,    PAGE = 1

CLA1mathTables    : LOAD = FLASHB,
                RUN = CLARAM1,
                LOAD_START(_Cla1mathTablesLoadStart),
                LOAD_END(_Cla1mathTablesLoadEnd),
                RUN_START(_Cla1mathTablesRunStart),
                LOAD_SIZE(_Cla1mathTablesLoadSize),
                PAGE = 1

CLAscratch        :
                { *.obj(CLAscratch)
                . += CLA_SCRATCHPAD_SIZE;
                *.obj(CLAscratch_end) } > CLARAM1,
```

```
                    PAGE = 1

 /*追加開始*/
  .bss_cla         : > CLARAM0,  PAGE = 1
  .const_cla       : LOAD = FLASHB,
                     RUN = CLARAM0,
                     LOAD_START(_Cla1ConstLoadStart),
                     RUN_START(_Cla1ConstRunStart),
                     LOAD_SIZE(_Cla1ConstLoadSize),
                     PAGE = 1
/*追加終了*/

  /* Uncomment the section below if calling the IQNexp() or IQexp()
     functions from the IQMath.lib library in order to utilize the
     relevant IQ Math table in Boot ROM (This saves space and Boot ROM
     is 1 wait-state). If this section is not uncommented, IQmathTables2
     will be loaded into other memory (SARAM, Flash, etc.) and will take
     up space, but 0 wait-state is possible.
  */
  /*
  IQmathTables2    : > IQTABLES2, PAGE = 0, TYPE = NOLOAD
  {

          IQmath.lib<IQNexpTable.obj> (IQmathTablesRam)

  }
  */
   /* Uncomment the section below if calling the IQNasin() or IQasin()
      functions from the IQMath.lib library in order to utilize the
      relevant IQ Math table in Boot ROM (This saves space and Boot ROM
      is 1 wait-state). If this section is not uncommented, IQmathTables2
      will be loaded into other memory (SARAM, Flash, etc.) and will take
      up space, but 0 wait-state is possible.
   */
   /*
   IQmathTables3    : > IQTABLES3, PAGE = 0, TYPE = NOLOAD
   {

           IQmath.lib<IQNasinTable.obj> (IQmathTablesRam)

   }
   */

  /* .reset is a standard section used by the compiler.  It contains the */
  /* the address of the start of _c_int00 for C Code.   /*
  /* When using the boot ROM this section and the CPU vector */
  /* table is not needed.  Thus the default type is set here to  */
  /* DSECT  */
  .reset           : > RESET,      PAGE = 0, TYPE = DSECT
  vectors          : > VECTORS     PAGE = 0, TYPE = DSECT
```

```
}

/*
//===========================================================================
// End of file.
//===========================================================================
*/
```

赤字で記述した箇所が、修正、削除、追加項目です。その内容は、既にF2806x用のところで解説しましたので、割愛致します。

10.7 CLA の使用例

それでは、実際の使用例を見てみましょう。C28x CPUから、2つの変数をCLAに渡して、その２つの加算を行い、結果をC28x CPUに返すという非常にシンプルな例を考えてみます。CLA用のコードは、.claという拡張子でファイルを作成します。プロジェクトは、C28x用のコードとCLAのコードの両方が入ります。

Projectの作成方法は、基本的に今までと同じです。新規プロジェクト用のディレクトリを作成し、以下のヘッダ・ファイルをそのディレクトリにコピーして下さい。

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ
F2803xの場合	C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_commonフォルダ C:\ti\controlSUITE\device_support\f2803x\v126\DSP2803x_headersフォルダ

ヘッダ・ファイルのExclude設定は以下の通りです。

F28069の場合	F2806x_common ― cmd 全てExclude (別途、F28069.cmdを変更したリンカ・コマンド・ファイルを追加します) 前節にて、示したF28069_CLA_C_sample.cmdをプロジェクトに追加して下さい。 F2806x_common ― lib 全てExclude F2806x_common ― source F2806x_CodeStartBranch.asm F2806x_DefaultIsr.c F2806x_PieCtrl.c F2806x_PieVect.c F2806x_SysCtrl.c F2806x_usDelay.asm を残して他は全てExclue F2806x_headers ― cmd F2806x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28069_RAM_lnk.cmdをExclude
F28035の場合	DSP2803x_common ― cmd F28035_CLA_C.cmdを残して、他は全てExclude。このF28035_CLA_C.cmdを前節にて記述したように修正を加えてください。

	DSP2803x_common ― lib 　　全てExclude DSP2803x_common ― source 　　DSP2803x_CodeStartBranch.asm 　　DSP2803x_DefaultIsr.c 　　DSP2803x_PieCtrl.c 　　DSP2803x_PieVect.c 　　DSP2803x_SysCtrl.c 　　DSP2803x_usDelay.asm 　　を残して他は全てExclue DSP2803x_headers ― cmd 　　DSP2803x_Headers_BIOS.cmdをExclude Project作成時に自動生成された28035_RAM_lnk.cmdをExclude

include pathの設定も必要です。この設定は、前と同じです。念のため、以下に記載しておきます。

F28069の場合	F2806x_common ― include F2806x_headers ― include
F28035の場合	DSP2803x_common ― include DSP2803x_headers ― include

コンパイラがCLAをサポートする設定になっているか確認して下さい。Project作成時に正しくデバイスを選択していれば、自動的にサポートする設定になっているはずです。

Project→Propertiesにて、

Build – C2000 Compiler – Processor Optionsにて、

Specify CLA Support (--cla_support)にて、”cla0”という設定になっていれば、正しく設定されています。もし設定されていなければ、設定して下さい。

尚、CCS5.2より前のバージョンでは、.claという拡張子を自動認識してくれず、設定の変更が必要になります。CCS5.2より前のバージョンのCCSを使われている場合は、CCS5.2以降の最新バージョン・アップして下さい。

コラム: CCS5.1/4.xでCLA用のコンパイラを使う場合
このドキュメントでは、基本的には5.2.0以降を推奨していますが、CCS5.1/4.xで何等かの理由により、どうしても試したいユーザーは、以下のように設定を変更して下さい。 Windows→Preferences にて、Preferencesウィンドウを開いてください。 左の欄で、 C/C++ → File Typeを選択して下さい。 CCSに登録されている、拡張子の一覧が表示されます。 Newをクリックして、 Pattern：.cla Type　：C Source File を選択して下さい。 これで、.claファイルがCソース・ファイルとCCSに認識され、CLA対応のCコンパイラが.claファイルを扱えるようになります。

この変更を行わないと、.claファイルはビルドの対象となりません。

それでは、コードを見ていきましょう。コードは２つあります。main.cがC28x CPU用のコードで、ClaCode.claがCLA用のコードです。CLA用のコードは、必ず.claという拡張子にする必要があります。.claという拡張子にする事で、Cコンパイラは、CLA用のコードである事を認識します。
また、CLA用のCコンパイラを使うためには、Cコンパイラのバージョンがv6.1.0以上である必要があります。Cコンパイラのバージョンは、
Project→Propertiesにて
Generalにて、確認できます。v6.1.0以上でない場合、v6.1.0以上のバージョンに設定を変更してください。v6.1.0以上のバージョンが選択できない場合は、v6.1.0以上のバージョンをインストールするようアップデートが必要です。

以下が、コードになります。

main.c (C28x CPUのコード)

```
#include "DSP28x_Project.h"
#include <string.h>

interrupt void Cla1TaskIsr(void);
extern interrupt void Cla1Task1(void);

extern Uint16 Cla1Prog_Start;

extern Uint16 Cla1funcsLoadStart;
extern Uint16 Cla1funcsRunStart;
extern Uint16 Cla1funcsLoadSize;

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;

#pragma DATA_SECTION(ClaAnswer, "Cla1ToCpuMsgRAM")
int16 ClaAnswer;

#pragma DATA_SECTION(ClaInput,"CpuToCla1MsgRAM")
int16 ClaInput[2];

int16 Answer;

void main(void) {

    InitSysCtrl();
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
    InitFlash();
    DINT;
    InitPieCtrl();
    IER = 0x00000000;
    IFR = 0x00000000;
    InitPieVectTable();

    EALLOW;
```

```
    PieVectTable.CLA1_INT1  = Cla1TaskIsr;
    EDIS;

    memcpy(&Cla1funcsRunStart, &Cla1funcsLoadStart, (Uint32)&Cla1funcsLoadSize);

    EALLOW;
    Cla1Regs.MVECT1 = ((Uint16)Cla1Task1 - (Uint16)&Cla1Prog_Start);
    Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_NONE;
    Cla1Regs.MIER.all  = 0x0001;
    Cla1Regs.MCTL.bit.IACKE          = 1;
    Cla1Regs.MMEMCFG.bit.PROGE = 1;
    Cla1Regs.MMEMCFG.bit.RAM0E    = 1;
    Cla1Regs.MMEMCFG.bit.RAM1E    = 1;
    EDIS;

    PieCtrlRegs.PIEIER11.bit.INTx1 = 1;
    IER |= M_INT11;
    EINT;

    ClaInput[0] = 10;
    ClaInput[1] = 20;
    Cla1ForceTask1();

    while(1);

}

interrupt void Cla1TaskIsr(void){

    Answer = ClaAnswer;
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP11;

}
```

ClaCode.cla

```
#include "DSP28x_Project.h"

extern int16 ClaInput[2];
extern int16 ClaAnswer;

int16 func1(int16, int16);

interrupt void Cla1Task1( void ){

  __mdebugstop();
  ClaAnswer = func1(ClaInput[0], ClaInput[1]);

  return;
```

```
}

int16 func1(int16 ArgA, int16 ArgB){

                return(ArgA + ArgB);

}
```

それでは、まず、main.cから見ていきましょう。

```
#include "DSP28x_Project.h"
#include <string.h>
```

DSP28x_Project.hのincludeは既におなじみと思いますので、説明を割愛します。CLAのコードをFlashからCLA Program Memoryにコピーする時に、memcpy標準関数を使いますので、string.hのincludeが必要になります。

```
interrupt void Cla1TaskIsr(void);
extern interrupt void Cla1Task1(void);
```

この２行は単なるPrototype宣言です。 Cla1TaskIsrがC28xのCLA割り込みISR、Cla1Task1がCLA割り込みタスク1番の処理関数になります。Cla1Task1はClaCode.claに記述されていますので、extern宣言にしています(externを省いてもかまいませんが)。

```
extern Uint16 Cla1Prog_Start;
```

次のこの行ですが、CLA割り込みタスクのベクタ登録のために使う宣言です。CLAのベクタ(MVECTxレジスタ)に、CLA割り込みタスク処理の先頭アドレスを登録しますが、この先頭アドレスはCLA Program Memoryの先頭アドレスからのオフセット・アドレスを登録する必要があります。そのため、CLA Program Memoryの先頭アドレスを取得する必要があります。CLA Program Memoryの先頭アドレスのシンボル、Cla1Prog_Startは、F28069用には作成した、F28035用には修正したリンカ・コマンド・ファイルにて定義されていますので、確認してみて下さい。リンカ・コマンド・ファイル内に

```
_Cla1Prog_Start = _Cla1funcsRunStart;
```

という一行があるはずです。この一行にて、Cla1Prog_start(アンダースコアは、アセンブラ上のラベルで、C言語上のラベルでは、アンダースコアが無くなります)というラベルが、CLA Program Memoryの先頭であることを定義しています。main.cでは、この.cmdにて定義されたシンボルを参照しています。

```
extern Uint16 Cla1funcsLoadStart;
extern Uint16 Cla1funcsRunStart;
extern Uint16 Cla1funcsLoadSize;
```

次の4行は、CLAのコードをFlashからCLA Program Memoryにコピーするためのラベルを取得しています。このラベルも先ほどと同じ.cmdファイルにて定義されています。ここの仕組みは、既に紹介した、FlashからRAMへのコピーと全く同じ仕組みです。次の

```
extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
```

は、そのFlashからRAMへのコピーに使う記述で、既に何度も出てきていますので、説明は割愛します。

```
#pragma DATA_SECTION(ClaAnswer, "Cla1ToCpuMsgRAM")
int16 ClaAnswer;

#pragma DATA_SECTION(ClaInput,"CpuToCla1MsgRAM")
int16 ClaInput[2];
```

次に、CLAとCPUとのデータのやりとりをする変数について定義しています。CLAとCPUのデータのやりとりは、CLA Shared Message RAMを使用します。既に解説しましたが、CLA Shared Message RAMは２つのブロックがあります。一つが、CLAからCPUへデータを送るブロック(CLA：R/W, CPU：Read only)です。もう一つが、逆でCPUからCLAへデータを送るブロック(CLA：Read only, CPU: R/W)です。それぞれ、0x80ワード(0x100バイト)あります。この２つのブロックは、アドレスが決まっていますので、リンカにて、アドレスの指定をする必要があります。アドレスを指定するために、

#pragma DATA_SECTION(変数名, “リンカ・コマンド・ファイルのセクション名”)

を使用します。例えば、

```
#pragma DATA_SECTION(ClaAnswer, "Cla1ToCpuMsgRAM")
int16 ClaAnswer;
```

にて、int16 ClaAnswerという変数は、Cla1ToCpuMsgRAMというセクションに配置するよう指示しています。一方、Cla1ToCpuMsgRAMというセクションは、リンカ・コマンド・ファイルにて定義されていますので確認してみましょう。リンカ・コマンド・ファイルにて、

MEMORY{}内に、

```
  CLA1_MSGRAMLOW   : origin = 0x001480, length = 0x000080
```

という一行があります。さらに、
SECTIONS{}内に

```
  Cla1ToCpuMsgRAM    : > CLA1_MSGRAMLOW,   PAGE = 1
```

という一行があります。これらにより、Cla1ToCpuMsgRAMというセクションが、CLA1_MSGRAMLOW(PAGE 1)にマッピングされています。CLA1_MSGRAMLOWは、0x1480番地から0x80ワードの大きさと定義されていますので、このセクションが、CLAからCPUへのShared Message RAMに正しく配置されるよう指示されているのがわかります。この仕組みにより、ClaAnswerは、Shared Message RAMのCLAからCPUへのメッセージRAMに配置されます。ClaInput[]も同様の仕組みにより、今度はCPUからCLAへのメッセージRAMに配置されることになります。

さて、それでは、main関数の中を見てきましょう。

```
    InitSysCtrl();
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
    InitFlash();
    DINT;
    InitPieCtrl();
    IER = 0x00000000;
    IFR = 0x00000000;
    InitPieVectTable();
```

ここまでは、既におなじみにのデバイスの初期化、FlashからRAMへのコピー、PIEの初期化のコードです。今まで何回か解説していますので、詳細は、割愛します。

```
    EALLOW;
```

```
PieVectTable.CLA1_INT1  = Cla1TaskIsr;
EDIS;
```

この行では、C28xのPIE割り込みISRの登録をしています。CLA割り込みタスクが終了した時に、C28x CPUのPIEに割り込み信号が必ず入るようになっています（CLAにて何も設定しなくても、CPUに終了割り込みが通知されます）。今回は、CLA割り込みタスク1番を使用しますので、その終了割り込み(CLA1_INT1, INT11.1)用のISRを設定しています。

```
memcpy(&Cla1funcsRunStart, &Cla1funcsLoadStart, (Uint32)&Cla1funcsLoadSize);
```

次に、memcpy標準関数を使って、CLA用のコードをFlashからCLA Program Memoryにコピーしています。コピー元、コピー先、サイズのラベル参照は、先ほど解説しました。CLAを起動する前に必ず、コピーを行ってください。

```
EALLOW;
Cla1Regs.MVECT1 = ((Uint16)Cla1Task1 - (Uint16)&Cla1Prog_Start);
Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_NONE;
Cla1Regs.MIER.all  = 0x0001;
Cla1Regs.MCTL.bit.IACKE         = 1;
Cla1Regs.MMEMCFG.bit.PROGE = 1;
Cla1Regs.MMEMCFG.bit.RAM0E    = 1;
Cla1Regs.MMEMCFG.bit.RAM1E    = 1;
EDIS;
```

次は、CLAの設定をしています。CLAの設定は、C28x CPUが行います。まず、次の一行にて、CLA割り込みタスク1番のベクタ設定を行っています。

```
Cla1Regs.MVECT1 = ((Uint16)Cla1Task1 - (Uint16)&Cla1Prog_Start);
```

CLA割り込みタスクのベクタは、MVECT1～8レジスタのMVECTフィールド(図 139と表 46)にて設定します。

図 139：MVECT1～8 レジスタ

フィールド名	設定値	意味
MVECT	0x000～0xFFF	CLA割り込みタスクのベクタ・アドレス・オフセット。CLA Program Memoryの先頭アドレスからのオフセットを設定します。

表 46：MVECT1～8 レジスタの MVECT フィールド

MVECTxがx番目のCLA割り込みタスクのベクタとなります。注意点としましては、何度も記載していますが、このレジスタには、CLA Program Memoryの先頭アドレスからのオフセットを設定するという点です。例えば、CLA Program Memoryが0x9000番地から始まっていて、CLA割り込みタスクの先頭アドレスが0x9120番地であれば、このレジスタには0x120を設定します。今回のコードでは、Cla1Prog_StartというCLA Program Memoryの先頭アドレスのシンボルをリンカ・コマンド・ファイルから取得し、Cla1Task1(今回のCLA割り込みタスク1番の処理の先頭アドレス)から減算して、レジスタに格納しています。

```
Cla1Regs.MPISRCSEL1.bit.PERINT1SEL = CLA_INT1_NONE;
```

次のこの行では、CLA割り込みタスクが、何によって起動できるかを設定しています。これを設定するのが、MPISRCSEL1レジスタのPERINTxSELフィールドです(図 140と表 47)。PERINTxSELフィールドは、F2803xとF2806xでは異なります。表は、F28035の場合です。

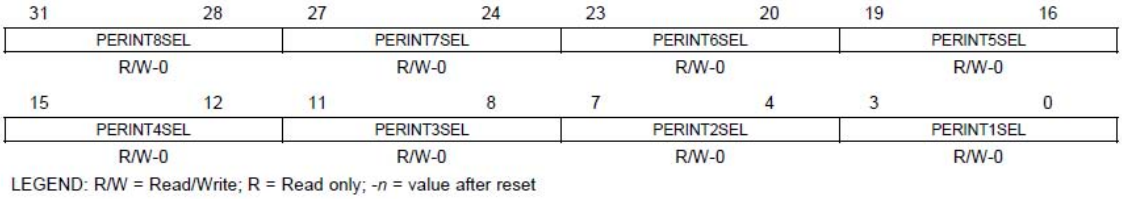


図 140：MPISRCSEL1 レジスタ

フィールド名	設定値	意味
PERINT1SEL	0000b	ADCINT1ペリフェラル割り込みがCLA割り込みタスク1の開始トリガになります。
	0010b	EPWM1_INTペリフェラル割り込みがCLA割り込みタスク1の開始トリガになります、
	xxx1b	CLA割り込みタスク1の開始トリガはありません(C28x CPUのみ開始トリガを発行できます)

表 47：MPISRCSEL1 レジスタの PERINT1SEL フィールド(F28035 の場合)

PERINTxSELが、x番目のCLA割り込みタスク用のフィールドになります。今回は、CLA割り込みタスク1番を使用しますので、PERINT1SELフィールドのみ設定しています。CLA1_INT1_NONEは、1という意味で、DSP2803x_common\include\DSP2803x_Cla_defines.hもしくは、F2806x_common\include\F2806x_Cla_defines.hにて定義されています。今回は、CLA割り込みタスクの起動はC28x CPUが命令にて行いますので、1と設定しています。

```
Cla1Regs.MIER.all  = 0x0001;
```

次のこの行にて、CLA割り込みタスク1番のイネーブルにしています。CLA割り込みタスクのイネーブルは、MIERレジスタのINTxフィールドにて行います(図 141と表 48)。

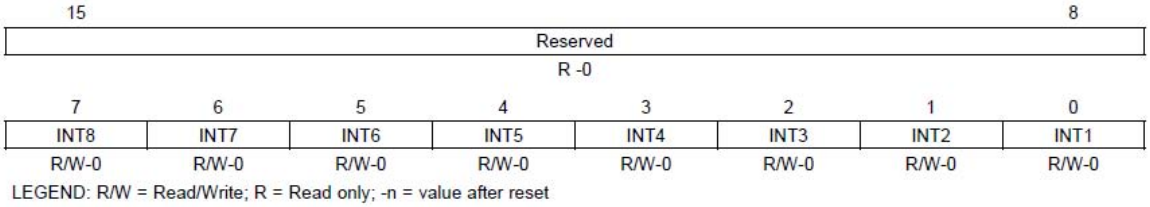


図 141：MIER レジスタ

フィールド名	設定値	意味	設定値	意味
INTx	0	CLA割り込みタスクx番のディセーブル	1	CLA割り込みタスクx番のイネーブル

表 48：MIER レジスタの INTx フィールド

CLA割り込みタスクは、リセット直後は、全てディセーブルになっていますので、イネーブルにしたいタスクのINTxを1にしてイネーブルにします。今回はCLA割り込みタスク1番を使用しますので、INT1フィールドを1に設定しています。

今回は特に使用していませんが、CLA割り込みタスクに対して、起動リクエストが入った場合は、MIFRレジスタにフラグが立ちます。タスクがMIERレジスタによって許可されていて、他のタスクが処理中でなければ、MIFRレジスタがクリアされ、タスクが実行されます。尚、CLAは、割り込みのネストは行いません。他のタスクが処理されている時に、別のタスクのリクエストが入った場合は、その新たにリクエストされたタスクは優先順位に関わらず、必ずペンディング状態になり、前の処理が終了したら、実行されます。処理がペンディング状態の時は、MIFRレジスタのフラグは立ったままになります。現在実行されているタスクがある場合は、MIRUNレジスタのフラグが立ちます。ペンディングになっているタスクがあり、同じタスクのリ

クエストがさらに入った場合は、MIOVFレジスタにフラグが立ちます(後続のリクエストはロストになります)。複数のタスクのリクエストが同時に入ったり、複数のタスクがペンディングになっている時は、番号が小さいタスクが優先的に処理されます。

次に、C28x CPUから、命令にてCLA割り込みタスクを起動できるように設定します。これを設定するのが、MCTLレジスタのIACKEフィールドです(図 142と表 49)。

```
Cla1Regs.MCTL.bit.IACKE          = 1;
```

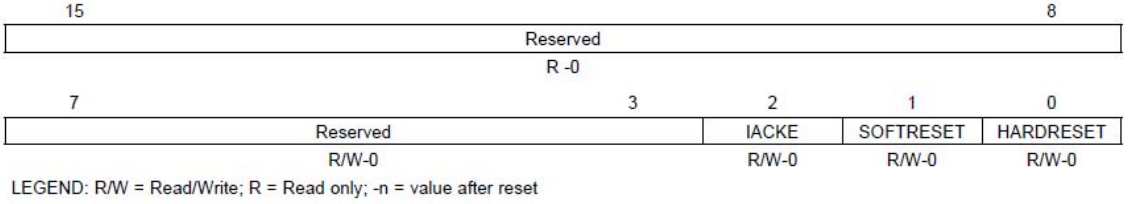


図 142：MCTL レジスタ

フィールド名	設定値	意味	設定値	意味
IACKE	0	C28x CPUによるIACK命令(CLA割り込みタスク起動命令)はディセーブル	1	C28x CPUによるIACK命令のイネーブル

表 49：MCTL レジスタの IACKE フィールド

このIACKEフィールドを1に設定しないと、C28x CPUのIACK命令によるCLA割り込みタスク起動ができませんので、必ず1に設定してください。尚、IACK命令を使わなくても、MIFRCレジスタを設定する事で、同じくC28x CPUからCLA割り込みタスクを起動できますが、MIFRCレジスタを使う場合は、EALLOW命令が必要になりますので、IACK命令を使った方が便利です。

続いて、CLA Program MemoryとCLA Data MemoryをC28x CPUからCLAへと所属を移します。この設定を行うのが、MMEMCFGレジスタのPROGE/RAM0E/RAM1Eフィールドです(図 143、図 144、表 50)。F2806xの場合は、CLA Data Memoryが3ブロックありますので、さらに RAM2Eフィールドがあります。今回のコードでは、F2803x/6x共通のコードという事で、RAM2Eフィールドは設定していません(つまり、F2806xのCLA Data RAM2は、CPU所属のままです)。

```
Cla1Regs.MMEMCFG.bit.PROGE = 1;
Cla1Regs.MMEMCFG.bit.RAM0E  = 1;
Cla1Regs.MMEMCFG.bit.RAM1E  = 1;
```

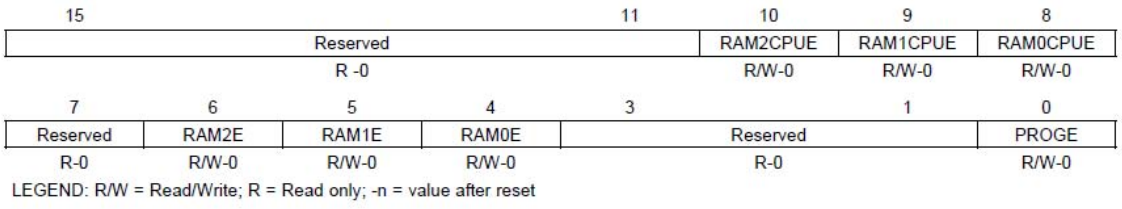


図 143：MMEMCFG レジスタ(F2806x)

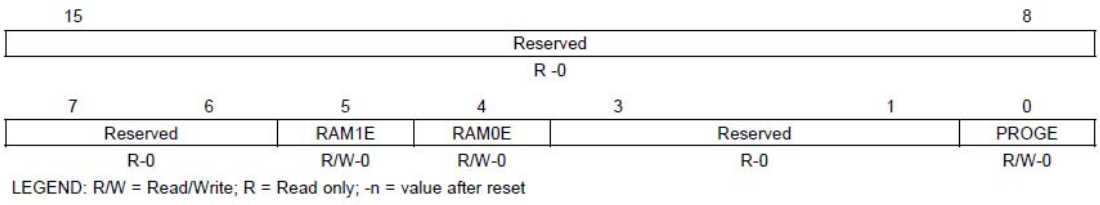


図 144：MMEMCFG レジスタ(F2803x)

フィールド名	設定値	意味	設定値	意味
PROGE	0	CLA Program MemoryはC28x CPUに所属	1	CLA Program MemoryはCLAに所属
RAMxE	0	CLA Data Memory(xブロック)はCPUに所属	1	CLA Data Memory(xブロック)はCLAに所属

表 50：MMEMCFG レジスタの PROGE/RAMxE フィールド

F2806xの方がF2803xよりも後に設計されたため、F2806xでは、少し機能が増えています。そのため、MMEMCFGレジスタにて、F2806xのではフィールドが増えています。PROGE, RAMxEについての基本的な考え方はF2803xとF2806xで共通です。リセット直後は、CLA Program/Dataの両メモリは、C28x CPUに所属しており、CLAはアクセスする事ができません。C28x CPUがCLA用のコード及びデータを、CLA Program/Data Memoryにコピー(今回はProgramしかコピーしていません。これは特に初期化するデータが今回のコードでは無かったためです)した後に、PROGE/RAMxEを1に設定して、CLAにアクセス権を移します。CLAにアクセス権を移した後の動きで、F2803xとF2806xで違いがあります。F2803xの場合は、CLAにアクセス権を移した後は、基本的にはC28x CPUはこれらのメモリにアクセスする事ができません(デバッガによるアクセスはできます)。一方、F2806xの場合は、RAMxCPUEフィールドを1に設定すると、CLAに権利を移した後でも、C28x CPUからリード・ライト・アクセスを行う事ができます(RAMxCPUE=0の場合は、F2803xと同じ動作)。この場合のアービトレーションは、リファレンス・マニュアルを参照下さい。今回は、F2803xとF2806xで共通のコードにしていますので、RAMxCPUEの設定は行っていません(つまり、F2806xでも、CLA Program/Data MemoryはC28x CPUからはアクセスできない設定です)。

さて、これでCLAの設定は終了です。次の行にて、C28x PIEのCLA割り込みの許可設定を行っています。

```
PieCtrlRegs.PIEIER11.bit.INTx1 = 1;
IER |= M_INT11;
EINT;
```

CLAは割り込みタスクが終了した時に、必ずPIEに割り込み信号を出します(CLA1_INTx)。今回は割り込みタスク1番を走らせますので、その処理終了割り込みをイネーブルにします。CLA1_INT1はPIEのINT11.1にアサインされていますので、それをイネーブルにします。

それでは、次にいよいよ、CLA割り込みタスクを起動します。まず、

```
ClaInput[0] = 10;
ClaInput[1] = 20;
Cla1ForceTask1();
```

ClaInput[]は、CPUからCLAへのShared Message RAMです。この領域は、CPUはR/W可能で、CLAはRead Onlyです。10と20を代入して、最後にCla1ForceTask1()にて、CLA割り込みタスク1番を起動します。Cla1ForceTask1()は、DSP2803x_common\include\DSP2803x_Cla_defines.h又は、F2806x_common\include\F2806x_Cla_defines.hにて定義されているマクロ関数で、以下のように、定義されています。

```
#define Cla1ForceTask1()    __asm(" IACK #0x0001")
```

既に何度か解説しましたが、C28x CPUからCLA割り込みタスクを起動するには、IACK命令を使用しますので、そのマクロ関数になります。この命令を実行する事で、CLA割り込みタスク1番が起動されます。起動したら、永久whileになっています。

さて、次にCLA割り込みタスク1番のISR、Cla1TaskIsr()を見てみましょう。このISRは大した事はしていません。

```
interrupt void Cla1TaskIsr(void){

    Answer = ClaAnswer;
```

```
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP11;

}
```

次に解説しますが、CLA割り込みタスク1番は、ClaAnswerという変数に、演算結果を格納してくれます。ClaAnswerは、CLAからCPUへのShared Message RAMになります。このClaAnswer変数を、単純にグローバル変数のAnswerにコピーしています。CLA割り込みタスクによるPIE割り込みは、ペリフェラル・レベルの割り込みフラグは存在しませんので、ペリフェラル・レベルのフラグのクリアは必要ありません。このISRから抜ける前には、PIEACKの11番をクリアするだけで、かまいません。

それでは、次にCLAのコードを見てみましょう。

```
#include "DSP28x_Project.h"

extern int16 ClaInput[2];
extern int16 ClaAnswer;

int16 func1(int16, int16);

interrupt void Cla1Task1( void ){

  __mdebugstop();
  ClaAnswer = func1(ClaInput[0], ClaInput[1]);

  return;

}

int16 func1(int16 ArgA, int16 ArgB){

    return(ArgA + ArgB);

}
```

今回のCLAのコードは非常にシンプルです。まず、ClaInput[](C28x CPUからCLAへのMessage RAM)とClaAnswer(CLAからC28x CPUへのMessage RAM)をExtern参照しています。これら変数の定義は、main.cにて行われています。まず、CLAの割り込みタスクは、必ずinterruptキーワードをつけてください。このキーワードをつけないと、正しく動作しません。今回は、わざわざ関数コールをする程の処理ではありませんが、例として、あえて関数コールを一回しています。注意点としては、CLAのCコンパイラは、関数コールは一段だけです。interruptがついている親関数(つまり、CLA割り込みタスクの元関数)から、一段だけ関数コールができます。今回の例では、func1関数をコールしています。1段しか関数コールできませんので、この例では、func1関数からは別の関数をコールする事はできません。また、関数コールでは引数は最大2つとなりますので注意下さい。

まず、

```
    __mdebugstop();
```

という行があります。CLAは、残念ながら、デバッガからブレーク・ポイントを貼る事ができません。ブレーク・ポイントを使うためには、ブレーク・ポイント用の命令を埋め込んでおく必要があります。その命令がこの__mdebugstop()マクロ関数になります。このブレーク・ポイント命令はJTAG接続している時は、ブレーク・ポイントとして働きますが、JTAG接続していない時は、NOP（何も実行しない命令）扱いになります。そのため、量産用のコードにおいても、この命令をそのまま埋め込んでおいても特に問題はありません(1サイクル損する事にはなりますが)。今回は、CLA用のコードをデバッグしたいですので、この命令を先頭に埋め込んでおきました。続いて、func1関数をコールし、ClaInput[0]とClaInput[1]の加算を行い、結果を、CLA→C28x CPUのShared Message RAM内のClaAnswerに格納して抜けています。

10.8 CLA のデバッグ

それでは、ビルドをしてデバッグしてみましょう。いつものように、虫マークをクリックしてビルドしてデバッグ開始を実行してください。

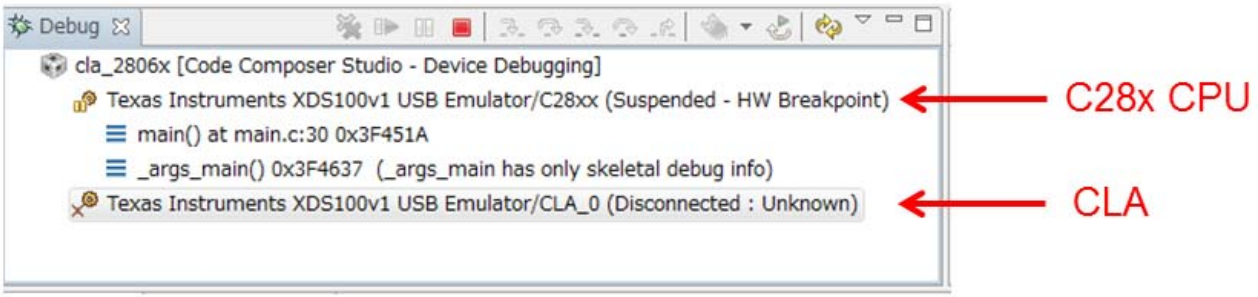


図 145：Debug ウィンドウ

プログラムがFlashにロードされ、Debugウィンドウは、図 145のようになっていると思います。CLAを搭載したデバイスでは、C28x CPUとCLAがデバイス内にて、JTAGチェーンが組まれていますので、２つのCPU(C28x CPUとCLA)がこのDebugウィンドウに表示されます。この２つのCPUは、それぞれ独立してデバッグする事ができます（実際にはCLAのデバッグはいろいろと制約があります）。上がC28x CPUで、下がCLAになります。上のC28x CPUは、自動的にコネクトされていますが、CLAの方は、自動的にコネクトされていません。CLAをコネクトしなくても、C28x CPUもしくはペリフェラルがCLA割り込みタスクを起動すれば、CLAは動作しますので、デバッグが必要ないのであれば、このままコネクトしなくてもかまいません。今回は、CLAのデバッグもしたいと思いますので、CLAのコネクトを行います。図 146のように、Debugウィンドウの、Texas Instruments XDS100v1 USB Emulator/CLA_0(Disconnected：Unknown)を右クリックして、Conncet Targetを選択してください。図 147のように表示され、CLAがコネクトされます。

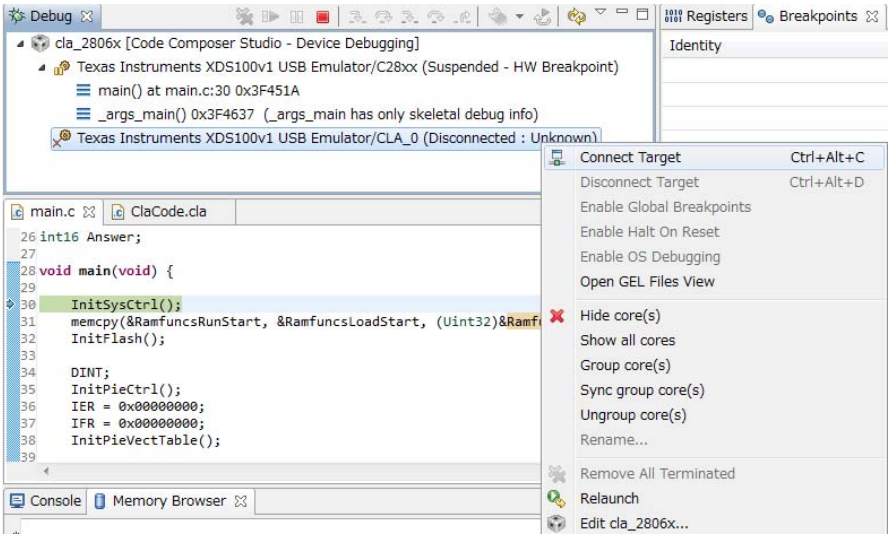


図 146：CLA のコネクト

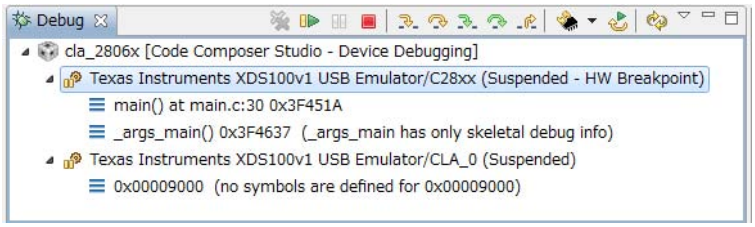


図 147：CLA のコネクト(コネクト後)

このDebugウィンドウにて、C28x CPUとCLAのどちらをデバックするかを選択します。

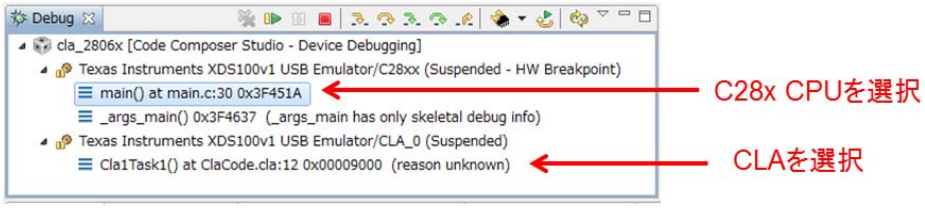


図 148：C28x CPU と CLA の選択

図 148を見てください。Texas Instruments XDS100v1 USB Emulator/C28xx....の行、もしくはその1行下をクリックすると（1行下の方が便利です。Disassemblyウィンドウに現在のPCの場所が表示されます）、C28x CPUをデバッグする事ができます。Texax Instruments XDS100v1 USB Emulator/CLA_0...の行、もしくはその１行下をクリックすると、CLAをデバッグできます。全く同じデバッグ画面を、C28x CPUとCLAにて共有しており、このDebugウィンドウの選択で切り替える事になりますので、今、どちらをデバッグしているか(どちらを選択しているか)を常に確認しながら作業をして下さい。

さて、C28x CPUの方には、プログラムがロードされていますので、シンボリック・デバッグができる状態になっていますが、CLAの方は、何もロードしていませんので、このままでは、シンボリック・デバッグができません。CLAのコードはC28x CPUがプログラムにて、CLA Program RAMにコピーしますので、デバッガがCLAにコードをロードする必要はありません。しかし、シンボリック・デバッグを行うために、シンボル情報が必要ですので、Debugウィンドウにて、CLAの方を選択し（これを忘れないようにして下さい）、

Run→Load→Load Symbols

を選択して下さい。Load Symbolsウィンドウが表示されます。残念ながら、Browse Projectが利用できない(選択しても何も表示されません)ので、Browseをクリックし、今回のProjectのディレクトリの下のDebug—Project名.out（つまり今回の実行ファイル）を選択して下さい。これで、CLA側にシンボル情報がロードされましたので、CLAもC言語レベルでシンボリック・デバッグができるようになります。

さて、準備が整ったところで、デバッグを開始してみましょう。Debugウィンドウにて、C28x CPUの方を選択し、RUNして下さい。数秒まって、今度はDebugウィンドウにて、CLA側を選択して下さい。CLA側を選択すると、interrupt void Cla1Task1(void)関数の最初の行の

__mdebugstop();

にて、止まっていると思います。C28x CPU側にてプログラムを走らせましたので、C28x CPUが、CLAのコードをCLA Program RAMにコピーし、CLA割り込みタスク1番を起動しましたので、CLA側のCla1Task1関数が実行されました。そして、最初の行の__mdebugstop();はブレークポイント命令ですので、ここでブレークがかかった止まったのです。それでは、CLAのコードをデバッグしてみましょう。View→Expressionsを選択して、Expressionsウィンドウ(グローバル変数表示ウィンドウ)を表示させてください。図 149のように、ClaInput, ClaAnswer, Answerを登録して下さい。

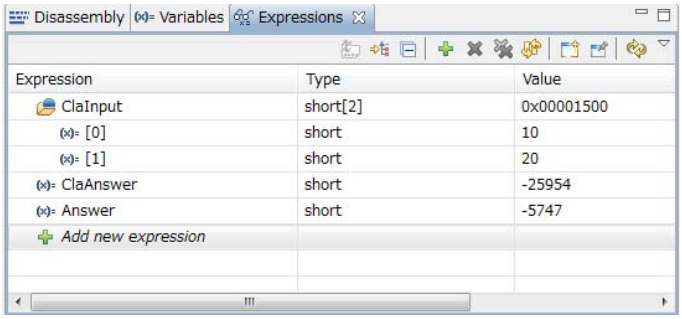


図 149：Expressions ウィンドウ(CLA のデバッグ)

CLAのデバッグにおきましては、CCSからブレーク・ポイントを貼る事はできませんが、Step IntoとStep Returnは使う事ができます(Step OverはStep Intoになってしまうようです)ので、かなり複雑なコードをデバッグするのでなければ、十分でしょう。Step IntoでもRUNでもかまいませんので、CLAのCla1Task()関数を最後まで実行してください。最後まで実行すると、ExpressionsウィンドウのAnswerが30に変化するはずです。Answer変数は、C28x CPUがCLA割り込みタスク1番の終了割り込みISRにてClaAnswer変数から代入していますので、このAnswerが30になったという事は、C28x CPUにおいて、CLA割り込みタスク1番の終了割り込みが入った事を意味し、正しく動作した事になります(C28x側で、Cla1TaskIsr()割り込み関数内にブレークポイントを貼っておくと、良く動作がわかると思います)。さて、CLAのコードを最後まで実行すると、CCSのConsoleウィンドウに、

CLA_0: Can't Single Step Target Program : (Error -2060.....

というエラーが表示されると思います。これはCLAが割り込みタスク処理を終えて、スリープ状態に戻ったため、デバッグができませんという表示です。つまり、これは正しい動作ですので、このエラーメッセージは気にする必要はありません。CLAは、割り込みタスク処理を実行している時にしか、ステップ実行できないという事を覚えておいて下さい（つまり、デバッグするためにはC28x側がCLA割り込みタスクを起動しなければなりません）。

11 DMA(F2806x のみ搭載しています。)

11.1 この章の目的

MCUが行う操作で、代表的な操作がデータの転送です。これは、メモリーメモリ間、メモリーペリフェラル間、ペリフェラルーペリフェラル間(あまり行われませんが)でデータの転送が行われます。DMAが搭載されていないMCUでは、これはＣＰＵが行わなければなりませんので、CPUの処理能力が必要となります。一方、DMAが搭載されているMCUでは、このデータ転送をDMAがCPUの代わりに行ってくれますので、DMAがデータ転送を行っている最中でも、CPUは別の処理を行う事ができます。Piccolo MCUでは、F2806xシリーズにのみ、DMAが搭載されています。この章では、F2806xに搭載されているこのDMAの概略を解説し、サンプルコードを元に使い方を解説します。DMAの詳細は、

F2806x用	*TMS320F2806x Piccolo Technical Reference Manual[SPRUH18] の Direct Memory Access(DMA) Module章*

に掲載されています。

11.2 DMA の概要

DMA(Direct Memory Access)は、CPUとは完全に独立したモジュールで、メモリやペリフェラル間のデータ転送を行うモジュールです。図 150にF2806xに搭載されているDMAのブロック図を示します。

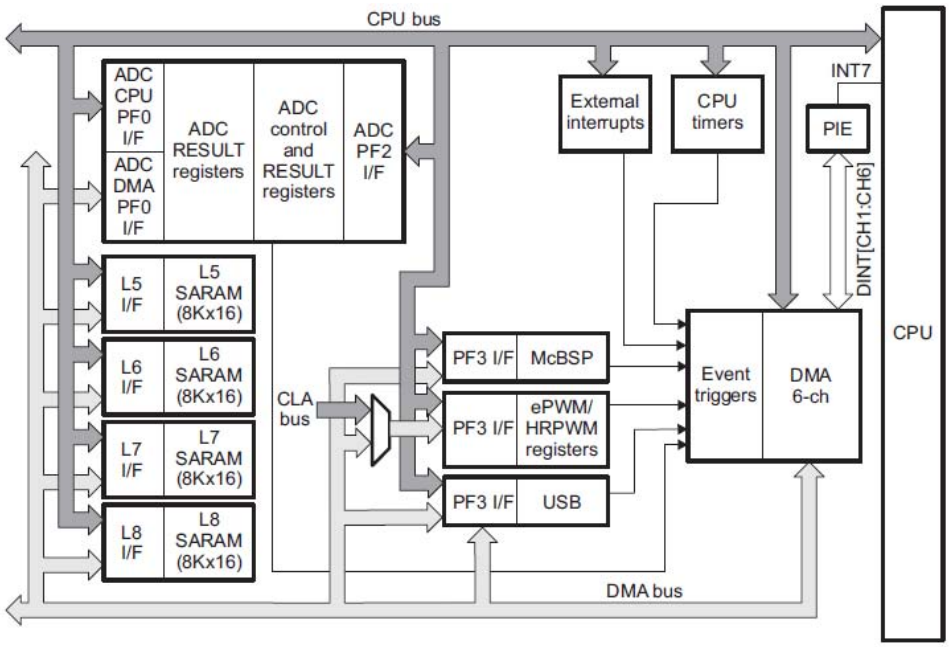


図 150:DMA のブロック図

F2806xに搭載されているDMAは6chのDMAで、それぞれのチャネルはそれぞれ独立して設定を行う事ができます。CPUとは完全に独立したDMA専用バスをもっていますので、アクセスするリソースが競合しなければ（例えば同じL5メモリブロックにDMAとCPUが同時にアクセスする等）、CPUの動作には影響を与えません。DMA専用バスは一本しかありませんので、DMA内の6chは同時には実行できなく、ある規則に基づいてプライオリティ制御されます。一点注意すべき事は、DMAはF2806x MCUの全てのリソースにアクセスできるわけではないという点です。DMAがアクセスできるリソースは、

- L5～L8 SARAM
- ADCのADCREULSTxレジスタ(変換結果の格納レジスタ)
- McBSPの送受信バッファ・レジスタ(DXRx/DRRx)

- ePWM1～8
- USB EP1/2/3送受信

に限られます。ADCRESULTxレジスタ以外の全てのリソースについて、CPUとDMAが同時に同じリソースにアクセスした場合は、DMAに優先権が与えられており、CPUは待たされます。ADCRESULTxレジスタは、CPUとDMAが同時にアクセスした場合でも、コンフリクトは発生しません。これは、CPU用とDMA用にそれぞれ別々の独立したインターフェイスが設けられているためです。SCI(UART)/SPI/I2Cといったシリアル・ポートは、DMAはアクセスできません。その代りに、これらシリアルポートにはFIFOが搭載されています。

DMAの各chに転送開始トリガをかけるリソースは以下の中から選択する事ができます。

- ソフトウェア
- ADCINT1/2
- 外部割込み1/2/3(XINT1/XINT2/XINT3)
- CPUタイマ0/1/2(TINT0/TINT1/TINT1)
- McBSP Aの送信バッファEmptyイベント(MXEVTA)/受信バッファのFullイベント(MREVTA)
- ePWM2～7のSOCA/B(ePWMxSOCA/ePWMxSOCB), ePWM1はトリガをかけられない事に注意下さい。
- USB0 EP1/2/3の送信Emptyイベント/受信Fullイベント(USB0EPxTX/USB0EPxRX)

11.3 DMA 転送の基本単位

DMA転送の最も小さい単位をWordといいます。Wordは16bitと32bitのどちらかを選択する事ができます。このWordをどのように転送するかで、２つの基本ループと、１つの特殊ループが存在します。２つの基本ループが以下のBURSTとTRANSFERで、1つの特殊ループがWRAPです。BURSTとTRANSFERは、一般的な使用方法ですので、すぐにご理解いただけると思いますが、WRAPは少し特殊で一見理解しづらいかもしれません。後ほど、もう少し詳細を解説します。

BURST	1BURSTあたり、1～32 Wordを転送する事ができます。1word転送毎に転送先/転送元のアドレスを自動変更(Burst Step)する事ができます。
TRANSFER	1～65536 BURSTを転送する事ができます。1BUSRT転送毎に転送先/転送元のアドレスを自動更新(Transfer Step)する事ができます。
WRAP	WARPにて指定したBURST転送回数に達した時に、転送先/転送元のアドレス自動更新をTRANSFERで指定した更新ではなく、WRAP機能（後程解説します）を使った自動更新を実行します。

まず、BURSTとTRANSFERについて解説していきます。図 151にBURSTとTRANSFER転送の図を示します。

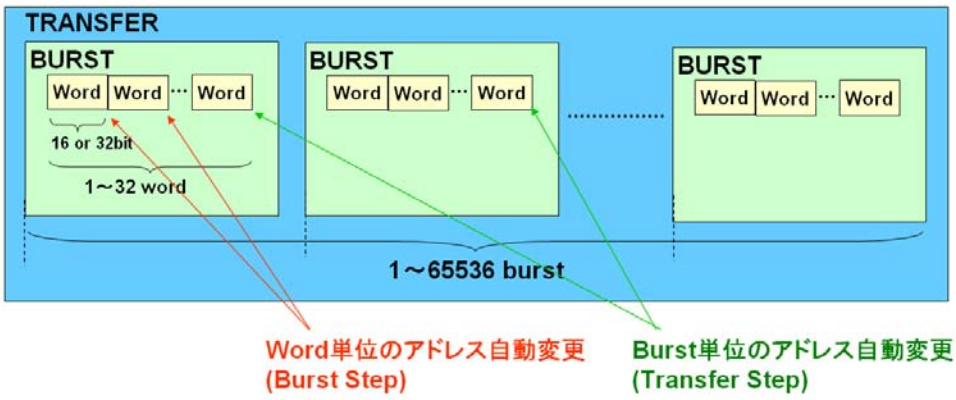


図 151:BURST と TRANSFER 転送

先ほど解説したように、1Wordは16bitもしくは32bitのどちらかを選択できます（DMAの転送速度は、4 Cycle/Word(McBSPは5 Cycle/Word、Burst転送を開始する時には+1サイクル)になります。このため32bitの方が高速に転送できます）。このWordが1～32個集まった転送がBURSTです。このBURST転送における1Word転送毎に、転送先および転送元のアドレスを

Burst Stepと呼ばれる値で自動更新を行う事ができます。自動更新は、転送先と転送元は個別に設定でき、－4095～＋4096まで設定する事ができます。1BURST転送(つまり1～32Word転送)が終了した時はこのBurst Stepではなく、Transfer Stepと呼ばれる値で、自動更新する事ができます。Transfer StepもBurst Stepと同様に、転送先と転送元は個別に設定でき、それぞれに－4095～＋4096まで設定する事ができます。

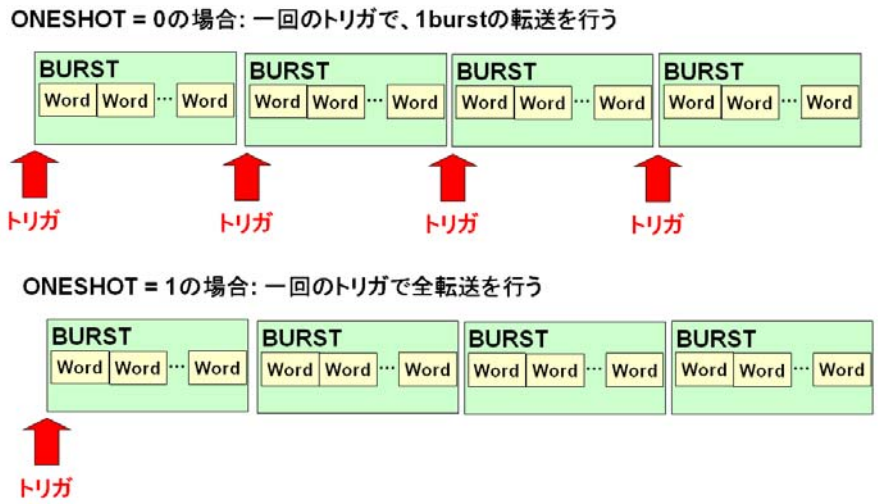


図 152:BURST/TRANSFER 転送における転送開始トリガ

DMAの転送には、転送を開始するトリガが必要です。選択できるトリガは前節にて解説しました。トリガにはONESHOTモードというモードが存在します。図 152に転送とトリガの関係図を示します。ONESHOT=0(ONESHOTモードがディセーブル)の時は、1BURST転送毎にトリガが必要です。１回のトリガで1BURST転送が行われ、停止し、トリガを待ちます。ONESHOT=1(ONESHOTモードがイネーブル)の時は、1回のトリガで全TRANSFER転送が一気に行われます。それでは、BURST/TRANSFER転送の例を見てみましょう。

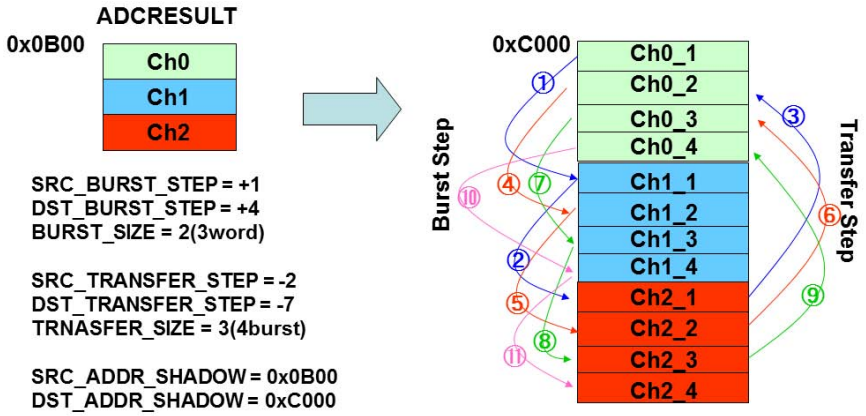


図 153:BURST/TRNASFER 転送の例

図 153に例を示します。この例では、

転送元アドレス	ADCRESULTxレジスタ	開始アドレスは0x0B00
転送先アドレス	内部メモリ	開始アドレスは0xC000

と、ADCRESULTxレジスタから、内部メモリへの転送例になります。転送元アドレスの初期値(SRC_ADDR_SHADOW)は0x0B00です。転送先アドレスの初期値(DST_ADDR_SHADOW)は0xC000です。ここで、レジスタの名前にSHADOWという言葉が入っています。このレジスタには、ActiveレジスタとShadowレジスタがあります（レジスタの名前は違う名前になっています）。ユーザーが設定するレジスタは、SHADOWレジスタの方です。転送を開始する時に、DMAモジュールがSHADOWレジスタ(SRC_ADDR_SHADOW/DST_ADDR_SHADOW)からACTIVEレジスタ(SRC_ADDR/DST_ADDR)にコピーします。

実際にアドレスのポインタとして使われるのは、ACTIVEレジスタの方です。つまり、設定するのはSHADOWレジスタ、現在指しているアドレスを確認するのはACTIVEレジスタです。尚、ACTIVEレジスタはRead Onlyのレジスタになりますので、CPUからのライトはできません。

Burst Stepは、

SRC_BURST_STEP(転送元アドレスのBurst Step)	＋1
DST_BURST_STEP(転送先アドレスのBurst Step)	＋4

に設定されています。BURST_SIZE=2です。この設定は、1BUSRTは3Word(設定数+1)である事を意味します。

Transfer Stepは、

SRC_TRANSFER_STEP(転送元のTransfer Step)	－2
DST_TRANSFER_STEP(転送先のTransfer Step)	－7

に設定されています、TNRASFER_SIZE=3です。この設定ではTRANSFERは4BURST(設定数+1)である事を意味します。

まず、転送開始時に、ユーザーが設定した、転送元/転送先開始アドレス、SRC_ADDR_SHADOW/DST_ADDR_SHADOWレジスタが、ACTIVEレジスタであるSRC_ADDR/DST_ADDRに自動的にコピーされます。図では、1 Wordが16bitか32bitかの設定をしていませんが、デフォルト設定が1 Word=16bitになっていますので、ここでは1 Word = 16bitです。1 Burst=3 Word設定になっていますので、3 Wordの転送が行われます。この時、1 Word転送後のアドレス変更は、Burst Stepが使用されますので、転送元で＋1、転送先で＋3と設定されていますので、1 Word転送後、SRC_ADDR=0x0B01、DST_ADDR=0xC004になります。1 Burst(3 Word)転送終了よりも1転送前まで、このアドレス変更ルールになります。1 Burst転送が終了した時、アドレス変更はBurst Stepではなく、Transfer Stepが使われますので、転送元で-2、転送先で-7になります。このように転送されていき、4 Burst転送されたら終了です。転送されたデータは、図のように配置されることになります。

11.4 DMA 転送の Wrap 機能

前節にて解説したBurst Step/Transfer Step転送ループの外側にさらにWrap転送ループという機能があります。筆者の個人的な感想ではありますが、リファレンス・ガイドを読んだだけでは、このWrap機能は少し理解しにくいように思えます。

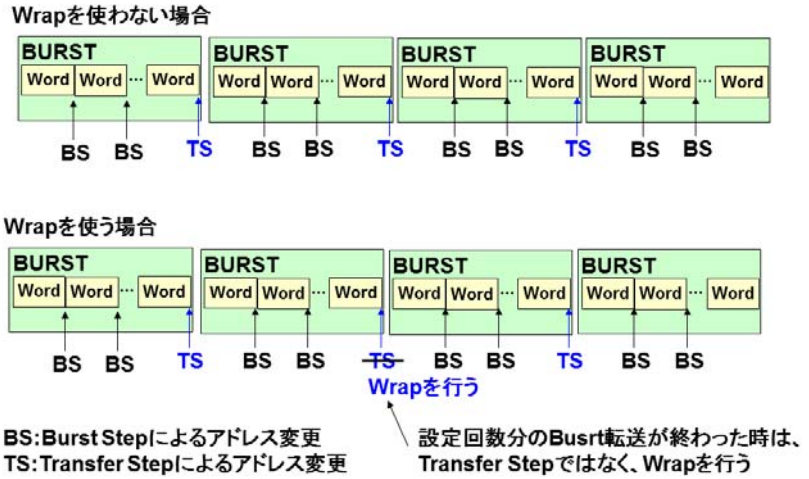
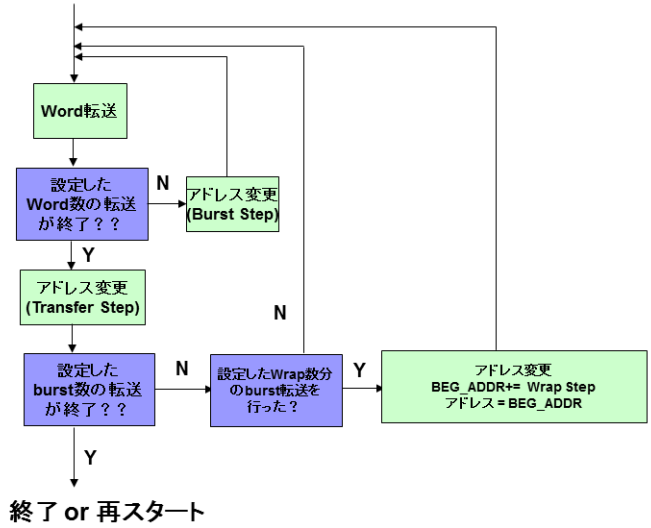


図 154：DMA の Wrap 機能

図 154を見てください。前節にて解説したBurst StepとTrnasfer Stepのみを使用した転送を図示したのが、上の図です。一方、Wrap機能を使った場合が下の図です。上の図では、Word転送毎に、Burst Stepによるアドレス変更が行われ、Burst Sizeにて指定したBurst転送数が終了した時は、Transfer Stepによるアドレス変更が行われます。一方、Wrapを使う場合は、Burst Sizeにて指定したBurst転送数が終了した時のアドレス変更に、違いが出てきます。Wrap機能を使う時は、Wrap Sizeを指定します。このWrap Sizeにて設定された数のBurst転送が終了した時(SRC_WRAP_SIZE/DST_WRAP_SIZEレジスタにて指定します)に、Transfer Stepではなく、Wrap機能によるアドレス変更を行います。Wrapによるアドレス変更は、Burst Step

やTransfer Stepのように、現在のアドレスに±の値を足し合わせるのではありません。Wrap機能が働いた時は、SRC_BEG_ADDR(転送元用)/DST_BEG_ADDR(転送先用)レジスタの値が、現在のアドレス(SRC_ADDR/DST_ADDR)にロードされます。このSRC_BEG_ADDR/DST_BEG_ADDRもSHADOWレジスタとACTIVEレジスタがあり、ユーザーが設定するのは、SHADOWレジスタ(SRC_BEG_ADDR_SHADOW/DST_BEG_ADDR_SHADOW)で、実際に使用されるのはACTIVEレジスタ(SRC_BEG_ADDR/DST_BEG_ADDR)で、転送開始時にShadowレジスタからActiveレジスタに自動的にコピーされます。更に、Wrap機能が働いた時に、SRC_BEG_ADDR/DST_BEG_ADDRレジスタの値が現在のアドレス(SRC_ADDR/DST_ADDR)にロードされる前に、Wrap Step(SRC_WRAP_STEP/DST_WRAP_STEPレジスタにて指定)による



変更が行われます。言葉で書くと、理解しにくいと思いますので、非常に簡潔にしたフローチャートを示します(図 155)。

Wrap機能を使った例を図 156に示します。

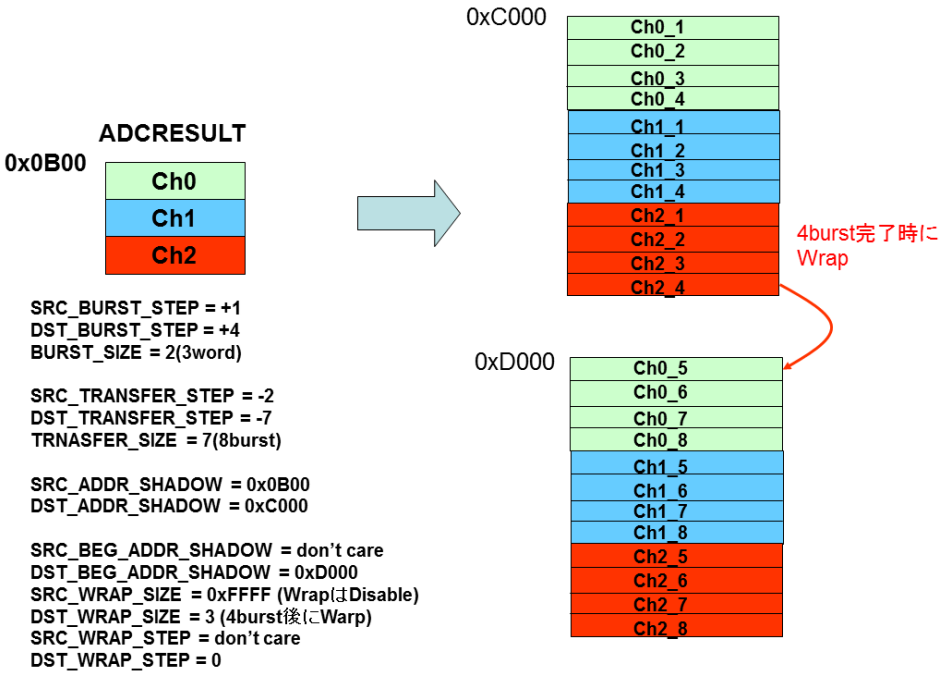


図 156：Wrap を使った例

この例は、先ほどの例に、Wrapを追加したものです、先ほどの例では、ADCRESULTから0xC000から始まるバッファに４回分のデータをチャネル毎に整理して転送しました。今回の例では、前回の例のバッファを２面持っているケースです。転送元のADCRESULTは同じく3ch分ですので、BURST_SIZEは2(3 word)で同じです。また、SRC_BURST_STEP/DST_BURST_STEPも同じです。SRC_TRANSFER_STEP/DST_TRANSFER_STEPも前回と同じですが、TRANSFER_SIZEは前回の倍の7(8 burst)に増えています。これは、バッファが倍になったためです。さて、この例では、半分(ch0_1～ch2_4まで)転送し終わったら、もうひとつの面にアドレスを変更する必要があります。ここで、Wrap機能の登場です。転送元は、同じアドレスの繰り返しですので、Wrap機能は使用しません。Wrap機能を使用しない場合は、WRAP_SIZEをTRANSFER_SIZEよりも大きな値を設定します。通常、最大値の0xFFFFを設定しておきます。0xFFFFに設定すると、どんなTRANSFER_SIZEに対しても、Wrapは働かないため、便利です。さて、転送先ですが、4バースト終了した時点で、アドレス(DST_ADDR)は0xC00Bになっています。DST_WRAP_SIZE=3(4 burst転送後にWrap)設定になっていますので、この時Wrap機能が働きます。Wrap機能では、DST_BEG_ADDR(DST_BEG_ADDR_SHADOWが転送開始時にコピーされたもの)がDST_ADDRにコピーされますので、0xD000となり、2面目のバッファの先頭アドレスに変わります。この時、正確には、DST_ADDRにコピーされる前に、

DST_BEG_ADDR += DST_WRAP_STEP

が実行されてから、DEST_BEG_ADDRがDST_ADDRにコピーされます。今回はDST_WRAP_STEP=0ですので、単にDST_BEG_ADDRがコピーされる事になります。その後は、同じようにBURST/TRNASFER転送になり、ch2_8まで転送が続きます。正直、あまり良い例ではない気がしますが（申し訳ありません、良い例が思いつきませんでした）、このように、あるセットの転送を、複数のバッファに転送する時に便利です。尚、DMAの割り込みは、転送開始時と転送終了時にのみ発生でき、Wrap機能が働いた時に割り込みを発生させる事はできませんので、この機能をPing Pongバッファに使うには、少し不便です。Ping Pongバッファを実現したい場合は、後述するサンプル・コードを参照下さい。

11.5 DMA の各チャネルにおけるプライオリティ制御

DMAは6個のチャネルがありますが、DMAバスが一本のため、それぞれのチャネルは同時に動く事はできません。ここでは、同時に複数のチャネルに転送リクエストが入っている場合の、プライオリティ制御について解説します。DMAのプライオリティ制御は、Round-RobinモードとCh1 High Priorityモード２つのモードがあります。

まず、Round-Robinモードについて解説します。Round-Robinモードでは、全てのチャネルは同じ優先順位であり、リクエストが入っているチャネルの中で順番に実行されます。順番はCH1から始まり、

CH1→CH2→CH3→CH4→CH5→CH6→CH1→CH2→.....

となります。このモードの場合、1 Burst転送毎に（つまり、BURST_SIZEにて指定した1 burst中のWord数を転送し終わったら）、次のチャネルに優先権が移ります。

Ch1 High Priorityモードの場合は、Ch1のみ高優先順位となり、他のCH2～CH6は低優先順位となります。もし、Ch1が動作していない時は、Ch2～Ch6は、先ほど説明したRound-Robinモードと同じ動きをします。Ch1の転送トリガが入った場合は、他のチャネルよりも、Ch1が優先的に実行されます。例えば、Ch4がBurst転送を行っている最中にCh1の転送トリガが入ったとします。この時、Ch4の転送中の1 wordは転送が行われます(この時、Ch1はペンディングとなります)。この1 wordが終了した時(1 burst転送が終了していなくても)にCh4の転送はペンディングとなり、Ch1が転送を開始します。Ch1が全ての転送を終了した時点(つまりTRANSFER_SIZEにて指定したBurst転送が全て終了した時)で、Ch4転送が再開されます。

11.6 DMA の使用例 : Ping Pong バッファ

それでは、DMAの使用例を見てみましょう。今回は、いわゆるPing Pongバッファ(２面のバッファを持ち、交互に使用するバッファ)の実現例を示します。図 157に今回提示する例の内容を示します。CPU Timer0にて、10usec毎にADCに変換開始トリガをかけます。ADCはこのトリガを受けADCINA0(SOC1)/ADCINA1(SOC2)/ADCINA2(SOC3)の変換を行います(実際のコードでは、ADC Initial Conversion Errata対策のために、ADCINA0(SOC0)を最初に変換します)。A2(SOC3)の変換終了時にADCINT1割り込み信号が発生し、このADCINT1割り込み信号がDMAのCH1に転送開始トリガをかけます。DMAは、DmaBuf1とDmaBuf2というPing Pongバッファを持っています。ADCの変換結果を、図のようにチャネル毎に並び替えて転送

し、片方のバッファが一杯になったら、CPUに割り込みをかけます。CPUは、DMA割り込みを受け、各チャネルの4回分の変換値の平均値をとります。このような例を挙げてみたいと思います。

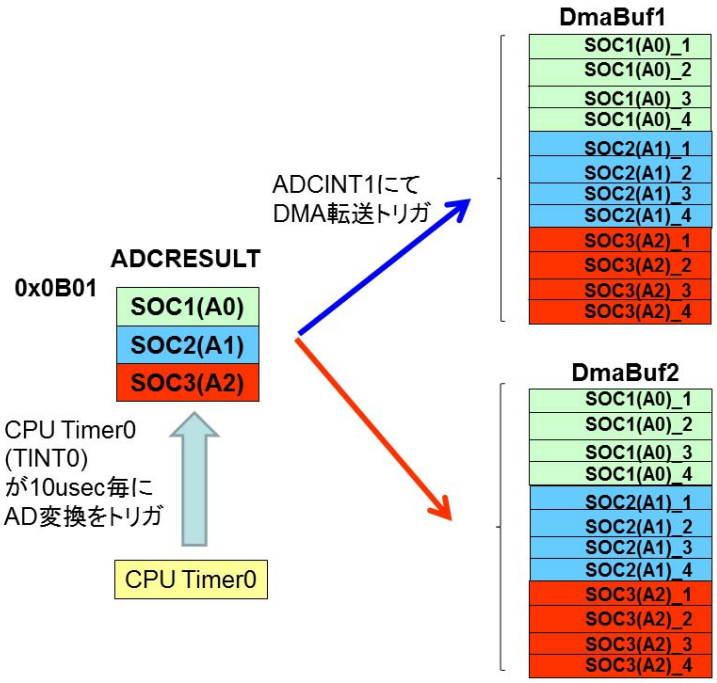


図 157：DMA の Ping Pong バッファ例

それでは、Projectを作成します。手順は今までと同様ですが、念のため詳細に記載します。まず、Projectを作成するディレクトリの下に以下のHeader Filesをコピーして、新規Projectを作成して下さい。

F2806xの場合	C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_commonフォルダ C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headersフォルダ

Header FilesのExclude設定は以下の通りです。

F28069の場合	F2806x_common ― cmd 　　F28069.cmdを残して、他は全てExclude F2806x_common ― lib 　　全てExclude F2806x_common ― source 　　F2806x_CodeStartBranch.asm 　　F2806x_Adc.c 　　F2806x_CpuTimers.c 　　F2806x_DefaultIsr.c 　　F2806x_PieCtrl.c

	F2806x_PieVect.c F2806x_SysCtrl.c F2806x_usDelay.asm を残して他は全てExclue F2806x_headers ― cmd F2806x_Headers_BIOS.cmdをExclude

include pathの設定もして下さい。

F28069の場合	F2806x_common ― include F2806x_headers ― include

それでは、コードを見ていきます。

main.c

```
#include "DSP28x_Project.h"
#include <string.h>

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsRunStart;
extern Uint16 RamfuncsLoadSize;
#pragma CODE_SECTION(DmaCh1Isr, "ramfuncs");

#pragma DATA_SECTION(DmaBuf1, "DMARAML5");
#pragma DATA_SECTION(DmaBuf2, "DMARAML5");
Uint16 DmaBuf1[12];
Uint16 DmaBuf2[12];
Uint16 DmaBufPointer = 0;          // 0：DmaBuf1, 1:DmaBuf2

Uint16 AdcCh0Ave;
Uint16 AdcCh1Ave;
Uint16 AdcCh2Ave;

interrupt void AdcIsr(void);
void AdcConfig(void);
void DmaConfig(void);
interrupt void DmaCh1Isr(void);

void main(void){

  DINT;
  InitSysCtrl();
  memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
  InitFlash();
  InitPieCtrl();
  IER=0x0000;
  IFR=0x0000;
  InitPieVectTable();
  EALLOW;
  PieVectTable.DINTCH1 = DmaCh1Isr;
  EDIS;
```

```
    AdcConfig();
    DmaConfig();
    InitCpuTimers();
    ConfigCpuTimer(&CpuTimer0, 80, 10);
    EINT;
    CpuTimer0Regs.TCR.all = 0x4000;                    // Start CPU Timer0

    while(1);

}

void AdcConfig(void){

    InitAdc();
    EALLOW;
    //ADC Control Register Configuration
    AdcRegs.ADCCTL1.bit.INTPULSEPOS = 1;            //Disable early interrupt trigger

    //Interrupt/Overflow Flag clear
    AdcRegs.ADCINTFLGCLR.all = 0x1FF;       //ADC Int flag
    AdcRegs.ADCINTOVFCLR.all = 0x1FF;      //ADC Int ovfl clear

    //ADCINT1 interrupt configuration
    AdcRegs.INTSEL1N2.bit.INT1CONT = 1;     // Enable Continuous mode
    AdcRegs.INTSEL1N2.bit.INT1E = 1;           // ADCINT1 Enable
    AdcRegs.INTSEL1N2.bit.INT1SEL = 3;       // Select EOC3 for ADCINT1 trigger

    //SOCPRICTL configuration
    AdcRegs.SOCPRICTL.bit.ONESHOT = 0;      // Disable oneshot mode
    AdcRegs.SOCPRICTL.bit.SOCPRIORITY = 0;          // All ch is Round Robin mode

     //Sequential sampling mode configuration
     AdcRegs.ADCSAMPLEMODE.bit.SIMULEN0 = 0; // CH0 & 1: Single conversion mode
     AdcRegs.ADCSAMPLEMODE.bit.SIMULEN2 = 0; // CH2 & 3: Single conversion mode

    //SOC overflow clear
    AdcRegs.ADCSOCOVFCLR1.all = 0xFFFF;

    //SOC0 configuration (dummy : To avoid inital conversion errata)
    AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 1;    // CPU Timer0(TINT0) trigger
   AdcRegs.ADCSOC0CTL.bit.CHSEL = 0;       // ADCINA0
   AdcRegs.ADCSOC0CTL.bit.ACQPS = 15;      // Sample window = 16

    //SOC1 configuration
    AdcRegs.ADCSOC1CTL.bit.TRIGSEL = 1;    // CPU Timer0(TINT0) trigger
   AdcRegs.ADCSOC1CTL.bit.CHSEL = 0;       // ADCINA0
   AdcRegs.ADCSOC1CTL.bit.ACQPS = 15;      // Sample window = 16
```

```
   //SOC2 configuration
   AdcRegs.ADCSOC2CTL.bit.TRIGSEL = 1;   // CPU Timer0(TINT0) trigger
   AdcRegs.ADCSOC2CTL.bit.CHSEL = 1;     // ADCINA1
   AdcRegs.ADCSOC2CTL.bit.ACQPS = 15;    // Sample window = 16

   //SOC3 configuration
   AdcRegs.ADCSOC3CTL.bit.TRIGSEL = 1;   // CPU Timer0(TINT0) trigger
   AdcRegs.ADCSOC3CTL.bit.CHSEL = 2;     // ADCINA2
   AdcRegs.ADCSOC3CTL.bit.ACQPS = 15;    // Sample window = 16

    EDIS;

}

void DmaConfig(void){

   EALLOW;
   //DMA Configuration
   DmaRegs.PRIORITYCTRL1.bit.CH1PRIORITY = 0; // Round-Robin Mode

   //DMA CH1 Configuration
   DmaRegs.CH1.MODE.bit.CHINTE = 1;                  // Interrupt Enable
   DmaRegs.CH1.MODE.bit.CHINTMODE = 0;               // Interrupt at b/o transfer
   DmaRegs.CH1.MODE.bit.DATASIZE = 0;                // Data Size = 16bit
   DmaRegs.CH1.MODE.bit.CONTINUOUS = 1;              // Continuous Mode
   DmaRegs.CH1.MODE.bit.ONESHOT = 0;                 // Disable Oneshot mode
   DmaRegs.CH1.MODE.bit.PERINTE = 1;                 // Enable Peripheral Interrupt Trigger
   DmaRegs.CH1.MODE.bit.PERINTSEL = 1;               // Select ADCINT1 for DMA tranfer trigger

   DmaRegs.CH1.BURST_SIZE.bit.BURSTSIZE = 2;  // Burst size = 3 word
   DmaRegs.CH1.SRC_BURST_STEP = 1;               // SRC Burst Step = +1
   DmaRegs.CH1.DST_BURST_STEP = 4;               // DST Burst Step = +4

   DmaRegs.CH1.TRANSFER_SIZE = 3;                // Transfer size = 4 burst
   DmaRegs.CH1.SRC_TRANSFER_STEP = -2;           // SRC Transfer Step = -2
   DmaRegs.CH1.DST_TRANSFER_STEP = -7;           // DST Transfer Step = -7

   DmaRegs.CH1.SRC_WRAP_SIZE = 0xFFFF;           // SRC Wrap Disable
   DmaRegs.CH1.DST_WRAP_SIZE = 0xFFFF;           // DST Wrap Disable

   DmaRegs.CH1.SRC_ADDR_SHADOW = (Uint32)&AdcResult.ADCRESULT1; // SRC Start address = ADCRESULT1
   DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf1;               // DST Start Address = Dmabuf1[0]

   DmaRegs.CH1.CONTROL.bit.RUN = 1;              // Start DMA CH1
   EDIS;

   PieCtrlRegs.PIEIER7.bit.INTx1 = 1;            // Enable PIE7.1(DMA CH1)
   IER |= M_INT7;                                // Enable INT7
}
```

```
interrupt void DmaCh1Isr(void){

  if(DmaBufPointer == 0){
    EALLOW;
    DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf2;
    EDIS;
    DmaBufPointer = 1;

    AdcCh0Ave = (DmaBuf2[0] + DmaBuf2[1] + DmaBuf2[2] + DmaBuf2[3]) >> 2;
    AdcCh1Ave = (DmaBuf2[4] + DmaBuf2[5] + DmaBuf2[6] + DmaBuf2[7]) >> 2;
    AdcCh2Ave = (DmaBuf2[8] + DmaBuf2[9] + DmaBuf2[10] + DmaBuf2[11]) >> 2;

  }else{
    EALLOW;
    DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf1;
    EDIS;
    DmaBufPointer = 0;

    AdcCh0Ave = (DmaBuf1[0] + DmaBuf1[1] + DmaBuf1[2] + DmaBuf1[3]) >> 2;
    AdcCh1Ave = (DmaBuf1[4] + DmaBuf1[5] + DmaBuf1[6] + DmaBuf1[7]) >> 2;
    AdcCh2Ave = (DmaBuf1[8] + DmaBuf1[9] + DmaBuf1[10] + DmaBuf1[11]) >> 2;
  }

  PieCtrlRegs.PIEACK.all = PIEACK_GROUP7;
}
```

ADCとCPU Timer0の設定は、今まで解説した設定を、少し変更しただけですので、コードを読んでみて下さい。ADCの設定で重要な点は、ADCの変換開始トリガにCPU Timer0(TINT0)を使っているという点です。

```
AdcRegs.ADCSOC0CTL.bit.TRIGSEL = 1;   // CPU Timer0(TINT0) trigger
```

この部分(SOC1/2も同様です)が、その設定になります。また、EOC3(SOC3の変換終了時)に、ADCINT1を生成するように設定しています。これは、

```
AdcRegs.INTSEL1N2.bit.INT1SEL = 3;    // Select EOC3 for ADCINT1 trigger
```

この行になります。また、今回はADCINT1を生成していますので、割り込み信号を生成している事になりますが、CPU割り込み(PIE割り込み)自体はDisableにしています。これは、ADCINT1はDMAにトリガをかけるのが目的であって、割り込みを行いたいわけではないためです。CPUは割り込み応答しませんので、ADCINT1のペリフェラル・レベルの割り込みフラグをクリアするコードがありません。ここで重要になるのが、

```
AdcRegs.INTSEL1N2.bit.INT1CONT = 1;    // Enable Continuous mode
```

ここの設定です。このINTxCONTビットを1にする事で、ADCINTFLGレジスタのADCINTxビット(ペリフェラル・レベルの割り込みフラグ)がクリアされていなくても、ADCINTx信号が毎回生成されます。

CPU Timer0も、割り込み信号(TINT0)を生成する設定になっていますが、CPU割り込み(PIE割り込み)自体はDisableにしています。これも、TINT0をADCに変換開始トリガに使用したいだけのためです。

main()の上に、以下の行があります。

```
#pragma DATA_SECTION(DmaBuf1, "DMARAML5");
#pragma DATA_SECTION(DmaBuf2, "DMARAML5");
Uint16 DmaBuf1[12];
Uint16 DmaBuf2[12];
```

既に解説しましたが、DMAは全てのRAMにアクセスする事はできません。DmaBuf1/DmaBuf2配列変数は、後程解説しますが、DMAの転送先バッファになります。そのため、この２つの配列変数は、DMAがアクセス可能なRAMに配置する必要があります。そのため#pragma DATA_SECTIONを使ってセクション指定しています。DMARAML5がどこに配置されているかは、リンカ・コマンド・ファイルF28069.cmdに記述されていますので、読んでみて下さい。DMAがアクセス可能なL5に配置されているのがわかると思います（どのメモリがDMAアクセス可能かは、データシートのMemory Mapsを参照下さい）。

さて、本題のDMAの設定を見ていきましょう。DMAの設定を行っているのは、DMAConfig()関数です。DMAのレジスタの大部分はEALLOW保護がかかっていますので、まず、EALLOW命令を発行して、EALLOW保護の解除を行っています。

最初に、設定しているのが、チャネルのプライオリティ設定です。

```
DmaRegs.PRIORITYCTRL1.bit.CH1PRIORITY = 0; // Round-Robin Mode
```

チャネルのプライオリティ設定は、PIRORITYCTRLレジスタのCH1PRIORITYビットにて設定します(図 158と表 51)。

図 158：PRIORITYCTRL レジスタ

フィールド名	設定値	意味	設定値	意味
CH1PRIORITY	0	Round-Robinモード	1	CH1 High Priorityモード

表 51：PRIORITYCTRL レジスタの CH1PRIORITY ビット

今回はCH1しか使いませんので、どちらのモードでもかまいませんが、Round-Robinモードの設定にしました。

続いて、CH1の基本的な設定をMODEレジスタに対して設定しています。MODEレジスタは、チャネル毎にあります。MODEレジスタは重要なレジスタのため、今回設定していないビットも含めて、全てのビット・フィールドを解説しておきます(図 159と表 52)。

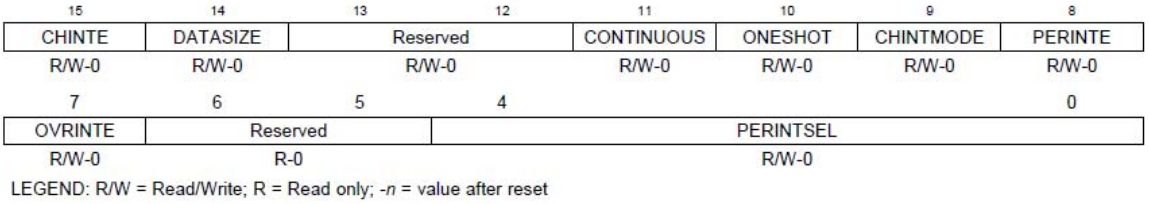


図 159：MODE レジスタ(CH 毎に用意されています)

フィールド名	設定値	意味	設定値	意味
CHINTE	0	DMA チャネル割り込み(DINTCHx)のDisable	1	DMA チャネル割り込み(DINTCHx)のEnable
DATASIZE	0	16bit 転送モード(1 Word = 16bit)	1	32bit転送モード(1ワード = 32bit)
CONTINUOUS	0	TRNASFER_SIZEにて設定したBurst転送が終了したら、DMAは停止します。	1	TRANSFER_SIZEにて設定したBurst転送が終了したら、再度DMAを設定し直し(Shadowレジスタからレジスタのコピーが行

				われます)、転送開始トリガを待ちます。
ONESHOT	0	Burst転送毎に開始トリガが必要です。	1	Burst転送毎に開始トリガは必要ではありません。最初の 1 回の開始トリガにて、TRANSFER_SIZEにて設定した数のBurst転送を全て行います。
CHINTMODE	0	DMA チャネル割り込み(DINTCHx)は、転送開始時(最初の転送開始トリガが入った時)に生成されます。(つまり、転送開始時)	1	DMA チャネル割り込み(DINTCHx)は、TRANSFER_SIZEにて設定した数のBurst転送が完了した時に生成されます(つまり、転送終了時)
OVRINTE	0	オーバーフロー(開始トリガが入り、その転送がまだスタートしてないのに、再度開始トリガが入った場合)割り込みのDisable	1	オーバーフロー割り込みのEnable
PERINTE	0	転送開始トリガの受信Disable	1	転送開始トリガの受信Enable
PERINTSEL	0	ソフトウェアのみ	1	ADCINT1(ADC)
	2	ADCINT2(ADC)	3	XINT1(外部割り込み)
	4	XINT2(外部割り込み)	5	XINT3(外部割り込み)
	6	予約	7	USB0EP1RX(USB)
	8	USB0EP1TX(USB)	9	USB0EP2RX(USB)
	10	USB0EP2TX(USB)	11	TINT0(CPU Timer0)
	12	TINT1(CPU Timer1)	13	TINT2(CPU Timer2)
	14	MXEVTA(McBSP)	15	MREVTA(McBSP)
	16	予約	17	予約
	18	ePWM2SOCA(ePWM2)	19	ePWM2SOCB(ePWM2)
	20	ePWM3SOCA(ePWM3)	21	ePWM3SOCB(ePWM3)
	22	ePWM4SOCA(ePWM3)	23	ePWM4SOCA(ePWM3)
	24	ePWM5SOCA(ePWM3)	25	ePWM5SOCA(ePWM3)
	26	ePWM6SOCA(ePWM3)	27	ePWM6SOCA(ePWM3)
	28	ePWM7SOCA(ePWM3)	29	ePWM7SOCA(ePWM3)
	30	USB0EP3RX(USB)	31	USB0EP3TX(USB)

表 52：MODE レジスタの各ビット・フィールド

さて、今回、MODEレジスタの設定は、以下の通りです。

```
DmaRegs.CH1.MODE.bit.CHINTE = 1;          // Interrupt Enable
DmaRegs.CH1.MODE.bit.CHINTMODE = 0;       // Interrupt at b/o transfer
DmaRegs.CH1.MODE.bit.DATASIZE = 0;        // Data Size = 16bit
DmaRegs.CH1.MODE.bit.CONTINUOUS = 1;      // Continuous Mode
DmaRegs.CH1.MODE.bit.ONESHOT = 0;         // Disable Oneshot mode
DmaRegs.CH1.MODE.bit.PERINTE = 1;         // Enable Peripheral Interrupt Trigger
```

```
DmaRegs.CH1.MODE.bit.PERINTSEL = 1;          // Select ADCINT1 for DMA tranfer trigger
```

CHINTE=1にして、DMA Ch1割り込み(DINTCH1)をイネーブルにしています。そして、CHINTMODE=0となっていますので、このDINTCH1割り込みは、転送が開始される時に生成されます。転送が終了した時に割り込みを生成するのは、イメージしやすいと思いますが、開始時に生成というのは、イメージしにくいかもしれません。この理由は、後程解説します。この設定は今回のサンプル・コードでは、非常に重要な項目になります。

DATASIZE=0ですので、転送の最小単位:Wordは16bitの設定です。CONTINUOUS=1ですので、転送が終了した時に、再度自動的に設定が行われ、開始トリガを待つように設定しています。この設定は、永久ループの設定になります。単発の転送を行いたい時は、このCONTINOUS=0に設定します。今回は、永久に転送を繰り返すコードになりますので、CONTINUOUS=1に設定します。ONESHOT=0ですので、Burst転送毎に開始トリガが必要です。今回は、AD変換が1シーケンス(SOC0→SOC1→SOC2)終了した時点でDMA転送しますので、Burst転送毎にトリガが必要な設定にします。一気に全転送を完了したい時は、このONESHOTを1に設定します。

DMAのCHをEnableにするためには、必ずPERINTE=1に設定する必要があります。このビットを1にしないと、開始トリガを受信できず、転送が開始されません。PERINTSELにて、このチャネルの転送開始トリガを選択します。今回は、ADCINT1割り込み信号が開始トリガとなりますので、1を設定しています。今回のコードでは、オーバーフロー割り込みは使用しませんので、OVRINTEは設定していません(デフォルトの0(ディセーブル)となります)。

続いて、Burst転送の設定です。1 Burst転送におけるWord数は、BURST_SIZEレジスタのBURSTSIZEビット・フィールドに設定します(図 160、 図 160、表 53)。

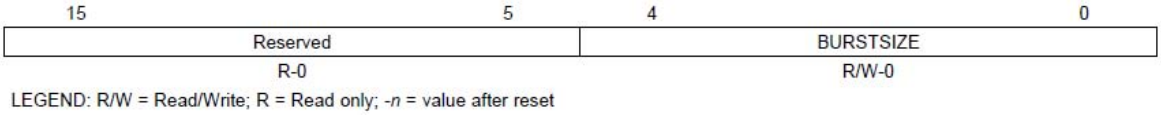


図 160：BURST_SIZE レジスタ

フィールド名	設定値	意味
BURSTSIZE	0～31	1 Burstの転送Word数を設定します。設定した値+1になる事に注意して下さい。 0：1 Word/Burst 1：2 Word/Burst … 31：32 Word/Burst

表 53：BURST_SIZE レジスタの BURSTSIZE ビット・フィールド

このBURSTSIZEに設定された値+1が1 Burstあたりの Word数になることに注意して下さい。今回は、

```
DmaRegs.CH1.BURST_SIZE.bit.BURSTSIZE = 2;  // Burst size = 3 word
```

と設定されていますので、3 Word/Burst設定となります。今回のコードでは、転送元のADCでは、1変換シーケンスは3 Word(SOC0→SOC1→SOC2)となりますので、この設定となっています。次にこの1 Burst中のWord転送毎に、どのようにアドレスを自動変更するかを設定します。この変更量は、転送元(SRC：SouRCeの略)と転送先(DST：DeSTinationの略)で、それぞれ独立して設定できます(図 161、図 162、表 54)。

図 161：SRC_BURST_STEP レジスタ(転送元アドレス用)

図 162：DST_BURST_STEP レジスタ(転送先アドレス用)

フィールド名		設定値	意味
SRCBURSTSTEP	-4096 ～ +4095	1 Burst転送中のWord転送毎の転送元アドレス(SRC_ADDRレジスタ)の変更量を設定します(2の補数表現)。	
DSTBURSTSTEP	-4096 ～ +4095	1 Burst転送中のWord転送毎の転送先アドレス(DST_ADDRレジスタ)の変更量を設定します(2の補数表現)。	

表 54: SRC_BURST_STEP/DST_BURST_STEP レジスタの SRCBURSTSTEP/DSTBURSTSTEP ビット・フィールド

ヘッダ・ファイルでは、SCR_BUSRT_STEP/DST_BURST_STEPレジスタは、ビット・フィールドがない単なる16bitレジスタとしての記述方法になっている事に注意して下さい。今回のコードでは、図 163のように、転送元では＋1、転送先では+4に設定すれば良い事がわかります。そのため以下のように設定しています。

```
DmaRegs.CH1.SRC_BURST_STEP = 1;          // SRC Burst Step = +1
DmaRegs.CH1.DST_BURST_STEP = 4;          // DST Burst Step = +4
```

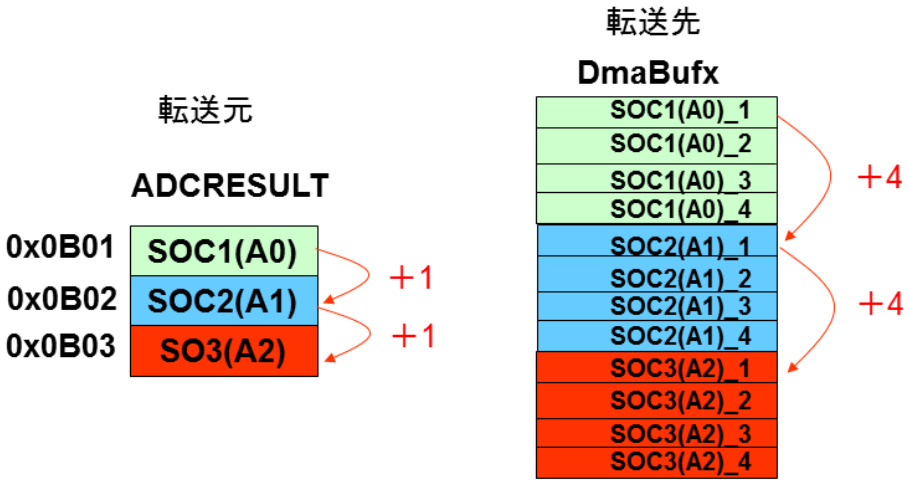


図 163：BURST_STEP 設定

次は、何Burst転送するかと、Burst転送毎のアドレス変更量の設定を行います。何Burst転送するかを設定するのが、TRANSFER_SIZEレジスタです(図 164と表 55)。

図 164：TRANSFER_SIZE レジスタ

フィールド名	設定値	意味
TRANSFERSIZE	0x0000 ～ 0xFFFF	転送するBurst転送数をを設定します。設定した値+1になる事に注意して下さい。 0x0000：1 Burst 0x0001：2 Burst … 0xFFFF：65536 Burst

表 55：TRANSFER_SIZE レジスタ TRANSFERSIZE ビット・フィールド

このレジスタも、設定数+1がBurst転送数になる事に注意して下さい。今回は、1バッファにつき、4回分のADCシーケンス結果を格納しますので、4 Burst設定の

```
DmaRegs.CH1.TRANSFER_SIZE = 3;          // Transfer size = 4 burst
```

と設定しています。このレジスタもヘッダ・ファイルでは、ビット・フィールド無の単なる16bitレジスタ扱いの記述になっている事に注意して下さい。

次に、Burst転送毎の、アドレス変更量を設定します。これらを設定するのが、SRC_TRANSFER_STEPとDST_TRNASFER_STEPです。BURST_STEPと同様に、転送元(SRC)と転送先(DST)でそれぞれ独立して設定する事ができます(図 165、図 166、表 56)。

図 165：SRC_TRANSFER_STEP レジスタ(転送元用)

図 166：DST_TRANSFER_STEP レジスタ(転送先用)

フィールド名	設定値	意味
SRCTRNASFERTSTEP	-4096 ～ +4095	Burst転送毎の転送元アドレス(SRC_ADDRレジスタ)の変更量を設定します(２の補数表現)。
DSTTRANSFERSTEP	-4096 ～ +4095	Burst転送毎の転送先アドレス(DST_ADDRレジスタ)の変更量を設定します(２の補数表現)。

表 56: SRC_TRANSFER_STEP/DST_TRANSFER_STEP レジスタの SRCTRANSFERSTEP/DSTTRANSFERSTEP ビット・フィールド

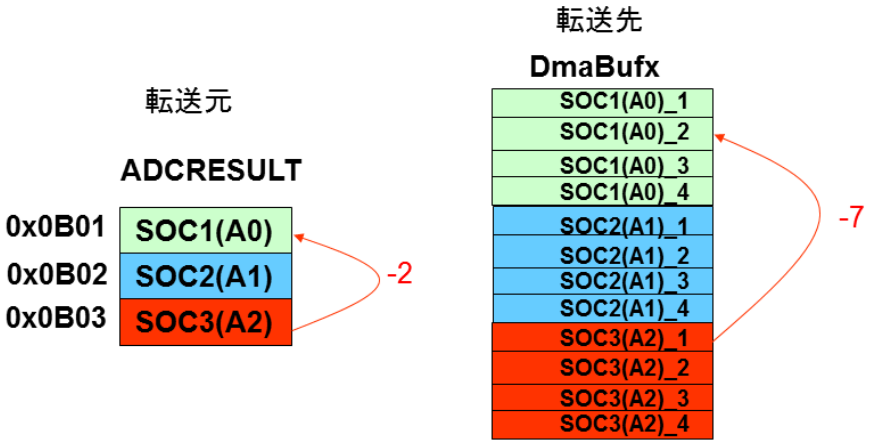


図 167：TRANSFER_STEP の設定

今回のコードでは、1 Burst転送終了時に、図 167のように、アドレスを変更すれば良いですので、以下のように、SRC_TRASFER_STEPを-2に、DST_TRANSFER_STEPを-7に設定しています。

```
DmaRegs.CH1.SRC_TRANSFER_STEP = -2;          // SRC Transfer Step = -2
DmaRegs.CH1.DST_TRANSFER_STEP = -7;          // DST Transfer Step = -7
```

次にWrap機能の設定をしています。今回はWrap機能を使いません。Wrap機能を使わない場合は、WRAP_SIZEを最大値の0xFFFFに設定します。WRAP_SIZEは、BURST_STEPやTRNASFER_STEPと違って、転送元(SRC)と転送先(DST)でそれぞれ独立して設定できます(つまり、SRCとDSTとそれぞれWrapを使うかどうかを独立して設定できます)。SRC_WRAP_SIZEとDST_WRAP_SIZEレジスタを図 168と表 57に示します。

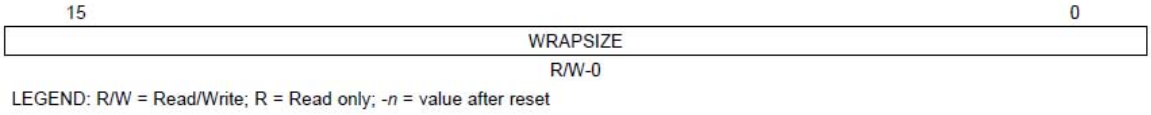


図 168：SRC_WRAP_SIZE と DST_WRAP_SIZE レジスタ

フィールド名	設定値	意味
WRAPSIZE	0x0000 ～ 0xFFFF	何burst転送後にWrap機能を使うかを設定します。 0x0000：1 burst後にWrap機能を使います。 0x0001：2 burst後にWrap機能を使います。 …. 0xFFFF：65536 burst後にWrap機能を使います(Wrap機能のディセーブル)

表 57：SRC_WRAP_SIZE/DST_WRAP_SIZE レジスタの WARPSIZE ビット・フィールド

コードでは、以下のように、0xFFFFにして、Wrap機能をディセーブルにしています。このレジスタも、ヘッダ・ファイルでは、ビット・フィールド無の単なる16bitレジスタ扱いの記述になっている事に注意下さい。

```
DmaRegs.CH1.SRC_WRAP_SIZE = 0xFFFF;  // SRC Wrap Disable
DmaRegs.CH1.DST_WRAP_SIZE = 0xFFFF;  // DST Wrap Disable
```

次に、転送元と転送先の転送開始時の初期アドレスを設定します。この初期アドレスを設定するのが、SRC_ADDR_SHADOW レジスタ(転送元アドレス)と DST_ADDR_SHADOW レジスタ(転送先アドレス)です。SRC_ADDR_SHADOWとDST_ADDR_SADOWレジスタは、転送開始時に、これらレジスタのActiveレジスタである、SRC_ADDRとDST_ADDRレジスタに自動的にコピーされます。実際の転送時には、このActiveレジスタが現在のアドレスを指します。SRC_ADDR/DST_ADDR/SRC_ADDR_SHADOW/DST_ADDR_SHADOWレジスタは全て同じビット・フィールド構成になっています(図 169)。

図 169：SRC_ADDR/DST_ADDR/SRC_ADDR_SHADOW/DST_ADDR_SHADOW レジスタ

ユーザーが設定する初期値は、SHADOWレジスタの方ですので、以下のように、転送元(SRC)側は、ADCのADCRESULT1レジスタのアドレスを、転送先(DST)側は、DmaBuf1配列変数の先頭アドレスを設定しています。

```
DmaRegs.CH1.SRC_ADDR_SHADOW = (Uint32)&AdcResult.ADCRESULT1; // SRC Start address = ADCRESULT1
DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf1;               // DST Start Address = Dmabuf1[0]
```

これで、設定は完了です。設定が完了したら、最後に設定したDMA CH1をEnableにします。DMAチャネルをイネーブルにするためには、CONTROLレジスタのRUNビット(図 170と表 58)を1に設定します。

15	14	13	12	11	10	9	8
Reserved	OVRFLG	RUNSTS	BURSTSTS	TRANSFERSTS	Reserved		PERINTFLG
R-0	R-0	R-0	R-0	R-0	R-0		R-0
7	6	5	4	3	2	1	0
ERRCLR	Reserved		PERINTCLR	PERINTFRC	SOFTRESET	HALT	RUN
R0/S-0	R-0		R0/S-0	R0/S-0	R0/S-0	R0/S-0	R0/S-0

LEGEND: R0/S = Read 0/Set; R = Read only; -*n* = value after reset

図 170：CONTROL レジスタ

フィールド名	設定値	意味	設定値	意味
RUN	0	チャネルの転送を停止します	1	チャネルの転送を開始します(開始トリガーを待ちます)

表 58：CONTROL レジスタの RUN ビット

```
DmaRegs.CH1.CONTROL.bit.RUN = 1;          // Start DMA CH1
```

最後に、以下の行で、EALLOW保護を再保護して、PIEのINT7.1(DINTCH1)をイネーブルにし、CPUレベルでINT7をイネーブルにしています。

```
EDIS;

PieCtrlRegs.PIEIER7.bit.INTx1 = 1;        // Enable PIE7.1(DMA CH1)
IER |= M_INT7;                            // Enable INT7
```

さて、この設定だけでは、Ping Pongバッファの実現はできません。Ping Pongバッファの実現には、DMACH1割り込みのISRである、DmaCh1Isr()関数を見る必要があります。それでは、そのDmaCh1Isrを見てみましょう。

DmaCh1Isr()関数(DMA CH1の割り込みのISR)

```
interrupt void DmaCh1Isr(void){

  if(DmaBufPointer == 0){
    EALLOW;
    DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf2;
    EDIS;
    DmaBufPointer = 1;

    AdcCh0Ave = (DmaBuf2[0] + DmaBuf2[1] + DmaBuf2[2] + DmaBuf2[3]) >> 2;
    AdcCh1Ave = (DmaBuf2[4] + DmaBuf2[5] + DmaBuf2[6] + DmaBuf2[7]) >> 2;
    AdcCh2Ave = (DmaBuf2[8] + DmaBuf2[9] + DmaBuf2[10] + DmaBuf2[11]) >> 2;

  }else{
    EALLOW;
    DmaRegs.CH1.DST_ADDR_SHADOW = (Uint32)DmaBuf1;
    EDIS;
    DmaBufPointer = 0;

    AdcCh0Ave = (DmaBuf1[0] + DmaBuf1[1] + DmaBuf1[2] + DmaBuf1[3]) >> 2;
```

```
    AdcCh1Ave = (DmaBuf1[4] + DmaBuf1[5] + DmaBuf1[6] + DmaBuf1[7]) >> 2;
    AdcCh2Ave = (DmaBuf1[8] + DmaBuf1[9] + DmaBuf1[10] + DmaBuf1[11]) >> 2;
  }

  PieCtrlRegs.PIEACK.all = PIEACK_GROUP7;
}
```

Ping Pongバッファを実現するために重要な事は、3つの重要事項があります。

１つ目が転送元と転送先のアドレスがActive/Shadow方式になっているという事です。転送元、転送先の初期アドレスは、SRC_ADDR_SHADOWとDST_ADDR_SHADOWレジスタに設定しました。これらのレジスタに設定された値は、転送を開始する時に、Activeレジスタであ るSRC_ADDRとDST_ADDRにコピーされ、実際に転送が実行されるアドレスとしては、このActiveレジスタが使用されます。Actieレジスタにコピーされた後(つまり、転送が開始された後)は、SRC_ADDR_SHADOWとDST_ADDR_SHADOWレジスタは、書き換えてもかまいません。Ping Pongバッファを実現するためには、これを利用します。

２つ目が、DMAチャネル割り込みのタイミングです。今回は、転送が開始された時(トリガが入ってShadowレジスタからActiveレジスタに値がコピーされた直後)に割り込みが入ります。この時、最初にShadowレジスタに設定した値は、既にActiveレジスタにコピーされて、その転送が開始されていますので、Shadowレジスタに、次のバッファのアドレス(今回の例では、DmaBuf2)を設定する事ができます。

3つ目が、CONTINUOUSモードを使う事です。CONTINUOUS=1に設定した場合、転送が終了した時に、再びShadowレジスタからActiveレジスタに値が自動的にコピーされ、開始トリガを待ちます。

これらが組み合わさると、Ping Pongバッファが実現できます。以下にその動作をまとめます。今回はPing Pongバッファになっているのは、転送先だけですので、転送先に重点を置いて記載します。

1. DST_ADDR_SHADOWレジスタに初期転送先アドレス(バッファ1のアドレス)を設定して、DMAをスタートします。
2. 転送開始トリガが入り、DST_ADDR_SHADOWレジスタの値がActiveレジスタのDST_ADDRレジスタにコピーされます。そして、転送が開始されます。
3. DINTCH1(DMA CH1割り込み)のタイミングは、転送開始時に設定に設定されているため、ここで割り込みが入ります。
4. 割り込みのISRにて、DST_ADDR_SHADOWレジスタをバッファ2のアドレスに書き換えます。
5. TRANSFER_SIZEにて指定されたBurst数のDMA転送が完了します。CONTINUOUS=1設定になっていますので、再び、TRANSFER_SIZEにて指定されたBurst数のDMA転送をするために、転送開始トリガを待ちます。
6. 転送開始トリガが入り、DST_ADDR_SHADOWレジスタ(この時はバッファ2のアドレス)がActiveレジスタのDST_ADDRにコピーされ、転送が開始されます。今回の転送は、バッファ2に対して転送が実行されます。
7. 転送が開始されたため、DINTCH1割り込みが入ります。この割り込みISRにて、DST_ADDR_SHADOWレジスタの値をバッファ1の値に変更します。
8. TRANSFER_SIZEにて指定されたBurst数のDMA転送が肝要します。CONTINUOUS=1設定になっていますので、再び、TRANSFER_SIZEにて指定されたBurst数のDMA転送をするために、転送開始トリガを待ちます。
9. 転送開始トリガが入り、DST_ADDR_SHADOWレジスタ(この時はバッファ1のアドレス)がActiveレジスタのDST_ADDRにコピーされ、転送が開始されます。今回の転送は、バッファ1に対して転送が実行されます。
10. 3に戻る

このように、Ping Pongバッファが実現できます。今回のコードでは、DmaBufPointerというグローバル変数に、今どちらのバッファを使っているかを保存しておき、その値を元に、DmaBuf1とDmaBuf2のアドレスをDST_ADDR_SHADOWレジスタに設定しています。そして、転送されたADC変換値（各チャネル4回分）の平均値を算出して、AdcChxAve変数に格納しています。

DINTCHx割り込みでは、DMAのペリフェラル・レベルのフラグはありませんので、PIEACKのクリアだけを実行して、ISRを抜けています。

ちなみに、最初の割り込みは、何も転送されていない時におきますので、AdcChxAve変数は最初の一回だけは意味のない値になります。今回は、特にこれは気にしない事にしました。

さて、それではデバッグしてみましょう、Expressionsウィンドウに、DmaBuf1, DmaBuf2, AdcCh0Ave, AdcCh1Ave, AdcCh2Aveを登録して、Real-Timeモードで実行してみて下さい。そして、ADCINA0/ADCINA1/ADCINA2に何かしらの電圧(例えば0V, 3.3V)を接続して、きちんと変換値がDmaBuf1/DmaBuf2に転送されているか確かめてください。今回使用しているボードのADC入力ピンは、ピンがそのまま出ているだけのピンや、キャパシタが接続されているだけのピンになりますので、必ずしも正確な値が変換されるとは限りません(おそらく、今回のサンプル・ホールド時間の設定値が短いので、あまり正確な値は変換できないと思います)が、およその値は確認できると思います。

12 C28x CPU のコード最適化へのヒント

12.1 この章の目的

この章では、C28x MCUのコードを記述する時に注意すべき点や最適化の方法・ヒントについて解説します。

12.2 C 言語におけるデータ型

この節は、最適化とは関係ありませんが、C言語におけるC28xコンパイラのデータ型を確認しておきます。表 59に各データ型とビット数を示します。

char, signed char, unsigned char	16bit
short, unsigned short	16bit
int, signed int, unsigned int	16bit
long, signed long, unsigned long	32bit
long long, singed long long, unsigned long long	64bit
enum	16bit
float	IEEE-32bit
double	IEEE-32bit
long double	IEEE-64bit
ポインタ(Large Memory Modelの場合)	22bit(0x00_0000 – 0x3F_FFFF)

表 59:C28x C コンパイラのデータ型(v5.x, v6.x の場合)

注意すべき点としまして、charが8bitではなく16bitという点です。C28xは、ＴＩが得意とするＤＳＰ(Digital Signal Processor:積和演算を代表とする数値演算を高速に実行するＣＰＵ)技術をベースとしたCPUになっています。MCUは一般的には、1アドレス=1Byteですが、C28xは1アドレス=1Word(=2Byte)になっています。これは、DSPは数値演算を最優先とする傾向があり、8bit演算よりも16bit演算を重視しているためです。

F2803x/F2802xは、FPU(浮動小数点ユニット)を搭載していません。float等の浮動小数点型を使う事はできますが、FPUがないため、浮動小数点演算を実行すると膨大なサイクル数がかかります。サイクル数が気になるような箇所では、浮動小数点型を使う事はお勧めできません。IQmathを使った固定小数点をご使用下さい。

12.3 ビルド・オプション

コード生成ツールのオプションを設定する事で、目的に合わせた最適化を行う事ができます。ビルド・オプションは、Project→propertiesを選択すると、図 171のように、プロジェクトのプロパティが表示されます。この画面の左側で、C/C++ Buildを選択すると、コード生成ツールの多岐にわたるオプションが表示されます。このオプション表示画面は、コンパイラのバージョンによって、異なる可能性があります。ここではよく使われるオプションについて、簡単に説明します。

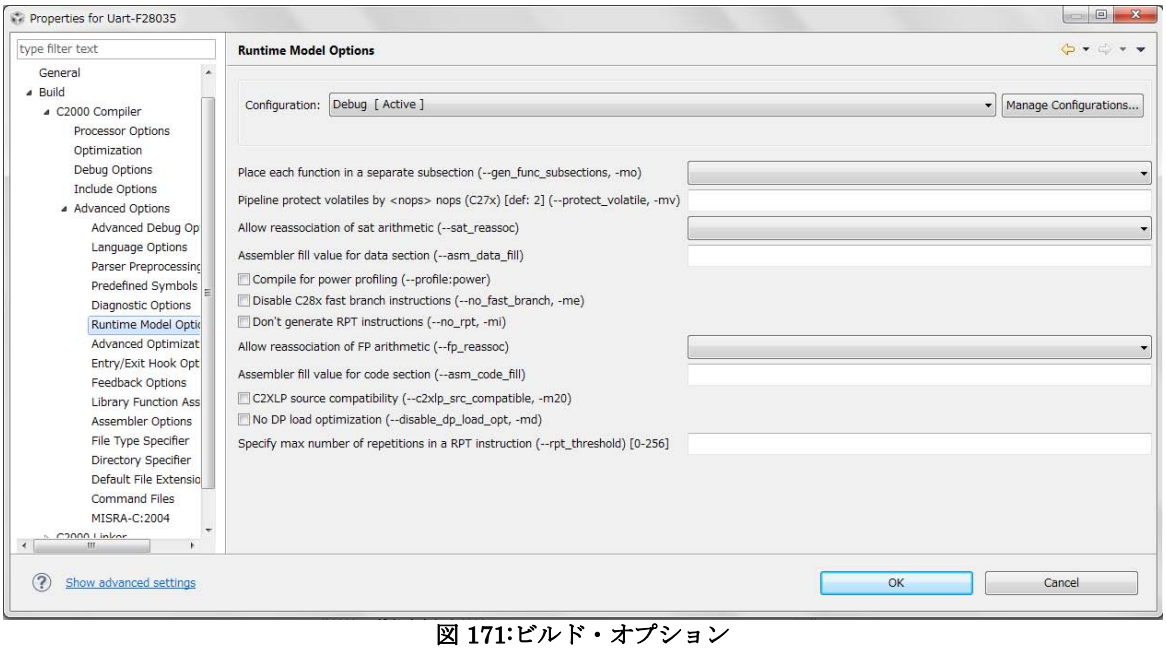


図 171:ビルド・オプション

3. --opt_for_speed, -mfオプション

まず、非常に重要な点ですが、C28xのコード生成ツールは、デフォルトでサイズ優先になっています。つまり、何も設定しないと、速度が犠牲になっている可能性があります。このオプションをつける事で速度優先になります。速度優先には、0～5までのレベルがあります。このレベルは、速度とサイズの優先度具合を示すもので、数字が大きくなる程速度優先になります。デフォルト・レベルは4です。もし、サイズ優先であれば、このオプションは選択せずに、--opt_for_space, -msオプションを選択して下さい。

4. --opt_level, -oオプション

いわゆる最適化レベルです。この設定が一番最適化に効いてきます。0～4(v6以前は0～3)のレベルがあります。数字が大きくなる程、より多くの最適化がかかります。注意点として、このレベルを上げると、シンボリック・デバッグがしづらくなるケースがあります（最適化によって、処理の順番が変わったり、不必要と判断された処理を省略したりするためです）。また、C言語上で定義されていない使用方法をとっている場合(キャストを正しくしていない等)では、最適化オプションを上げると、動作しなくなる可能性があります。

5. --symdebug:dwarf, -gオプション

デフォルトで、このオプションが選択されていると思います。シンボリック・デバッグをする場合は、必ずこのオプションがついている必要があります。一方、このオプションをつけている場合は、一部の最適化が抑制されますので、つけない方が最適化される可能性があります（シンボリック・デバッグはできません）。

6. --unified_memory, -mtオプション

デフォルトで、このオプションが選択されていると思います。もしされていない場合は、必ず選択して下さい。このオプションは、データ空間とプログラム空間が共通のメモリ構成をしているC28xデバイスに対してつけるオプションです。現在販売している全てのC28xデバイスは、この構成になっていますので、このオプションは必ずつけてください。(つけなくても、誤動作するものではありません。最適化上つけた方が望ましいという意味です)。このオプションを使う場合は、一点注意が必要です。ペリフェラル・レジスタに対してmemcpy標準関数を絶対に使用しないで下さい。memcpy標準関数をこのオプションと共に使用した場合、最も効率のよいアセンブラ命令を使用します。しかしながら、このアセンブラ命令は、プログラム・バスをデータ転送に使用するために、データがプログラム空間になければなりません。C28xのSRAM及びFlashはデータ及びプログラム空間の両方にマッピングされていますが、ペリフェラル・レジスタはデータ空間にしかマッピングされていません。そのため、ペリフェラル・レジスタに対してmemcpy標準関数を使用すると、正しく動作しなくなります。

7. --optimize_with_debug, -mn

-gオプション（シンボリック・デバッグ）をつける場合は、一部最適化項目が抑制されます。このオプションをつける事で、-gオプションをつけながらも、抑制されている最適化項目が一部解除されます。その反面、シンボリック・デバッグは若干しづらくなる可能性があります（処理順番などが変わったり、不必要と判断された処理が省略されたりするため）。しかし、ほとんどのケースでは許容範囲と思いますので、このオプションは-gオプションを使う場合は、同時につける事をお勧めします。

大変残念ながら、完璧で不具合が全くないコード生成ツールは世の中には存在しません。多少なりとも不具合は存在します。これは、大変恐縮ですが、全てのありとあらゆるケースをテストする事が実際には不可能なためです。また、C言語自体も規則があいまいな部分もある事から、処理がコンパイラまかせになるようなコードを書いてしまうケースもあります。特にそのようなケースでは、最適化オプションのレベルが上がると、正しく動作しなくなるケースもございます。生成されたコードは、必ず動作検証が必要になり、特に最適化オプション(-o)のレベルが2以上の場合には、念入りに検証して頂きますよう、お願い申し上げます。

12.4 Flash のアクセス速度

Piccoloに搭載されているFlashメモリは、残念ながら、CPUの最高動作速度では、1サイクルでアクセスする事ができません。この辺の仕組みは、このドキュメントに既に記載されていますので、詳細はそちらを参照下さい。このため、Flash上のコードは、RAM上で動作するよりも、遅くなります。高速に動作させる必要のあるコードは、FlashからRAMへコピーして、RAM上で実行するようにして下さい。このドキュメントには、その方法も掲載されています。

12.5 ハーバード・バス・アーキテクチャとメモリ・ブロック

C28x MCUは、従来DSPと呼ばれている事からもわかるように、もともとはDSPアーキテクチャをとっています。DSPアーキテクチャとして代表的なものに、ハーバード・バス・アーキテクチャ(最近は、MCUでもこのアーキテクチャをとるデバイスが増えています)があります。このアーキテクチャは、メモリ・アクセスにおける、プログラムとデータのバスを完全に分離する事で、効率よく、プログラム/データを高性能CPUに供給するためのアーキテクチャです。DSPは、もともと処理速度の優先順位が高いデバイスですので、このようなアーキテクチャが昔から採用されています。

さて、ハーバード・バス・アーキテクチャにより、メモリ空間にアクセスするバスが複数あることになります。C28x MCUでは、プログラム・フェッチ・バス、データ・リード・バス、データ・ライト・バスの３本のバスがあります(DMA搭載デバイスでは、DMA専用のバスもあります)。一方、C28x MCUを含む最近のCPUは、パイプライン動作を行っていすので、３本のバスを持つC28x MCUでは、1サイクルに３回のメモリ・アクセス（プログラム・フェッチ/データ・リード/データ・ライト）が実行される可能性があります。C28x MCUに搭載されているメモリ(RAM及びFlash)は、1サイクルに1回しか(Flashの場合は、CPU動作速度によりwaitが入ります)アクセスする事ができません(TI製のDSPでは、1サイクルに2回アクセスできるRAM(DARAM:Dual Access RAM)を搭載したデバイスもありますが、C28x MCUは、1サイクルに1回だけアクセスできるSARAM(Single Access RAM)のみ搭載されています)が、RAMもFlashも複数のブロックに分割されています。同じブロックに同時に複数のバスがアクセスした場合は、どれか1つしかアクセスできないため、どれかのバスのアクセスが待たされる事になりますが、異なるブロックであれば、同じサイクルに同時にアクセスする事が可能です。例えば、プログラムと、リードするデータと、ライトするデータが、全て別々のメモリ・ブロックにあれば、この３本は、同時にアクセスする事が可能です。データのリードとライトを別々のブロックに配置するのは、実際には少々困難ですが、少なくとも、プログラムとデータは、別のブロックに配置して下さい。安価なデバイス程、メモリ・ブロックが少なくなる傾向にありますので、プログラムとデータを別のブロックに配置できない事もあるかもしれません。そのような時は、少なくとも、高速に実行しなければならないプログラムと、データを別のブロックに配置するよう、工夫して下さい。プログラムとデータの配置を変更するためには、リンカ・コマンド・ファイルを変更します。このドキュメントには、リンカ・コマンド・ファイルの記述方法が解説されていますので、是非一読下さい。

12.6 データ配置と struct について

各メーカーの各ＣＰＵは、それぞれ工夫をこらしたアーキテクチャになっており、それぞれ得意な処理、不得意な処理があります。また同様に、コードの書き方によって、そのアーキテクチャの良い特徴を発揮できるケースと、発揮できないケースが存在します。アセンブラでコードを記述する場合は、それを意識しながら記述する事で性能を発揮する事も容易ですが、Ｃ言語で記述する場合は、この辺をあまり意識せずに書くケースが多くなります(実際に、意識してもコントロール出来ない事も多いです。)。C28xにて、Ｃ言語で簡単に制御可能な大きな最適化項目について解説します。それが、グローバル変数の配置です。C28x CPUは、もともとグローバル変数アクセスに強いアーキテクチャを持っています(RISC型アーキテクチャと違って、レジスタを通すことなくメモリ値を演算する事ができるためです)。少し工夫するだけで、その特徴をさらに生かすことができます。グローバル変数は、メモリ上に存在します。

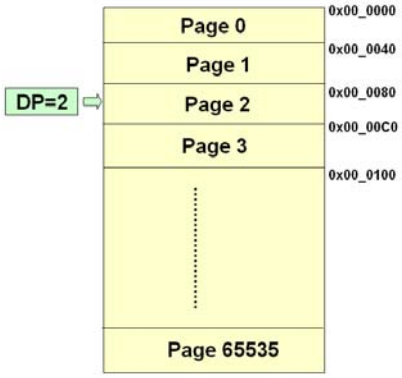


図 172:DP(データ・ページを使ったメモリ・アクセス)

C28xのメモリ・アクセスには、数種類の方法が提供されていますが、その中のひとつにDPモード・アクセス（正確には、直接アドレッシング・モード）があります。C28xのメモリ空間は、0x00_0000～0x3F_FFFF番地までありますが、これを0x40毎の0～65535(つまり16bit)のデータページに分けています。このDPモード・アクセスでは、メモリにアクセスする前に、DPレジスタにページ数を設定し、それからメモリにアクセスします。このようにすることで、アドレッシングに使うビット数が6bitになるため、命令長を短くする事ができます。例えば、abcとdefというグローバル変数の足し算を行うコードを考えます。

1.abcとdefが違うページにある場合

```
MOVW    DP, #abc    ← DPをabcがあるページに設定する
MOVL    ACC, @abc   ← ACCにabcをリード
MOVW    DP, #def    ← DPをdefがあるページに設定する
ADDL    ACC, @def   ← ACC=ACC+def
```

2.abcとdefが同じページにある場合

```
MOVW    DP, #abc    ← DPをabcがあるページに設定する
MOVL    ACC, @abc   ← ACCにabcをリード
ADDL    ACC, @def   ← ACC=ACC+def
```

abcとdefが同じページにある場合は、ＤＰを設定しなおさなくてよくなりますので、サイクルが少なくなります。このように、できるだけ同じページにグローバル変数を配置しておく事は、効率よいコードにつながります。これを効率よく行うためには、２つの事項を考える必要があります。

１．出来るだけページを設定しなおさなくても良いように、メモリ配置を考える
２．出来るだけページを設定しなおさなくても良いように、処理の順番を考える

まず、1について考えてみましょう。グローバル変数のアドレスは、ユーザーが何も指定しないと、リンカが勝手に決めてしまいます。例えば、

```
unsigned int abc;
unsigned int def;
```

```
unsigned int ghi;
```

と続けて変数を定義しても、これが同じページである保証はどこにもありません（実際には同じページになる事が多いですが）。どこに配置されるかは、リンカまかせです。さらに、変数を宣言しているファイルと、使うファイルが、別々の場合は、もっと深刻です。コンパイル作業はファイル毎に行います。変数を宣言しているファイルの場合は、どの変数が同じページかは判断できますが、変数が別ファイルにて宣言されている場合は、それら変数が同じページかどうかは、リンク時までわかりません。つまり、全て別ページにあると仮定してコンパイルされてしまい、非常に効率が悪くなります。

これらグローバル変数を確実に同じページにおさめるためには、構造体(struct)を使用するのが便利です。配列でも同じ効果がありますが、structの方が、それぞれに変数名をつけられますので、より実用的だと思います。

```
struct val{
 unsigned int abc;
 unsigned int def;
 unsigned int ghi;
};
struct val Val1;
```

のように、structで定義すると、このケースではVal1のメンバは全て必ず同じページになります。仕組みとしましては、structや配列は、可能な限り同じページになるように、コンパイラがアラインします。structの内容が1ページ(つまり0x40ワード)を超えない場合は、全てのメンバは必ず同じページ内に配置されます。1ページを超える場合は、先頭のメンバがページの先頭アドレスに配置されます。注意点としては、C28xは16bit単位でアドレスが割り振られています。32bitのデータのアドレスは偶数アラインがとられます。つまり、たとえば、

```
struct xxx{
  16bit
  32bit
  16bit
  32bit
};
```

という順番でメンバを定義した場合、16bitと次の32bitの間には1ワードの空きができます。注意下さい。32bitよりも大きいデータや、配列、structをメンバにした場合、それぞれルールがありますので、詳細はコンパイラのマニュアルを参照下さい。

さて、このように、structを使った場合は1ページにおさまる範囲のstructであれば、必ず同じページになりますので、非常に効率よくアクセスする事ができますので、グローバル変数には、是非structを活用下さい。また、変数にアクセスする順番を考え、できるだけ、DPを設定しなおさないような記述を出来る限り考えてみてください。

12.7 不要なグローバル変数

C言語の初心者が記述するコードでよく見かけるのですが、ローカル変数とグローバル変数の違いをよく理解していないケースがあります。極端な例としては、ほとんどローカル変数を使わずに、ほとんど全ての変数にグローバル変数を使っている例も見たことがあります。

グローバル変数は、あるメモリ・アドレスを占有するためメモリ容量を増やしてしまいます。一方、ローカル変数は、スタック又は、レジスタを使用するために、メモリを節約する事ができます。また、サイクル数を考えるために、以下のようなケースを考えてみます。

```
void func(void){
 ....
 a = b + c;
 d = e * f + a;
 ....
 ....
```

}
この場合、aは中間変数です。aは保存しておく必要がないため、本来はローカル変数で良いケースです。このaがグローバル変数の場合は、C言語は最適化オプションを設定していた場合でも、aへの代入を行います。これは、aが他の関数でも使用されるかもしれないからです。一方、このaがローカル変数であれば、Cコンパイラはaへの代入を行う必要がないと理解でき、最適化オプションによっては、この2行をまとめた1行として実行するかもしれません。つまり、ローカル変数にした方が、メモリも節約できるし、サイクル数も少なくすむ可能性があるわけです。これは、あくまでも一例ですが、不要なグローバル変数はできるだけ避けるべきです(グローバル変数を使うと値が残るためデバッグがしやすいという便利な面はあります)。

12.8 浮動小数点演算

稀に固定小数点デバイス(つまりFPU無のデバイス)で、浮動小数点演算を実行しているコードを記述しているユーザーを見かけます。C28x MCUでは、FPU付のデバイス(PiccoloではF2806x)とFPU無のデバイス(PiccoloではF2802x/F2803x)があります。FPU付のデバイスでは、浮動小数点演算を高速に実行できますので、浮動小数点を使う事は何の問題もありませんが、FPU無のデバイスで浮動小数点演算を実行すると、乗算１つだけでも、膨大なサイクル(数100サイクルかかります)がかかります。FPU無のデバイスでは、全く速度が気にならない部分(例えば初期化等)以外は、浮動小数点を使用するべきではありません。また、C28x MCUのFPUは、32bit単精度FPUです。32bit単精度FPUは、64bit倍精度浮動小数点演算には、全く役に立ちませんので、ご注意ください(つまり、64bit倍精度浮動小数点演算には、膨大なサイクル数がかかります)。

CLAもFPUが搭載されていますので、高速に32bit単精度小数点演算が実行できます。

12.9 飽和処理

制御アプリケーションにおいては、頻繁に飽和処理が入ります。例えば、次のような処理です。

```
if( abc < min){
  abc = min;
}else if(abc > max){
  abc = max;
}
```

abcがmin～maxを超えた場合に、飽和させる処理です。一見単純な処理ですが、分岐が起きるコードですので、結構サイクルがかかってしまいます。C28x CPUはこのような処理を行う専用命令MIN(L)/MAX(L)という強力な命令を持っています。これをＣ言語にて生成させるためには、２つの方法があります。

1.Intrinsics関数を用いる
MCUやDSPには特徴的な命令を使用するための特別な関数(Intrinsics関数)を用意している事があります。C28xも複数のIntrinsics関数を用意しています。その中で、このMAXL/MINLをサポートする関数が、

```
long dst = __IQsat(long A, long max, long min);
```

です。Aを最小値min,最大値maxの範囲に飽和させます。
例えば、

```
long a, b, c;  // 同じデータページと仮定

a = __IQsat(a, b, c);
```

と使った場合は、

```
MOVL        ACC, @_a
MINL        ACC, @_b
MAXL        ACC, @_c
```

```
MOVL        @_a, ACC
```

と極めて高効率なコードが生成されます。

2.条件演算子を使用(-o0以上で、使用する変数にvolatileがついていない事が条件です)。

```
long a, b, c;
a = ( (a<b) ? b : a);
a = ( (a>c) ? c : a);
```

といった記述方法です。これも同じように、MAXL/MINL命令を使用してくれます。

それでは、__IQsat()と条件演算子のどちらが便利でしょうか？
MIN/MAXの両方の処理をする場合は、迷わず__IQsat()が便利です。どちらか一方であれば、条件演算子が便利です。しかしながら、条件演算子は、-o0以上かつvolatileが無い場合にしかつかえませんので、その時は、１サイクルが無駄になりますが、__IQsat()をご使用下さい。if文をつかうより、はるかに効率的です。

12.10 FPU 搭載デバイス(F2806x シリーズ)について

F2806xシリーズMCUは、32bit単精度浮動小数点演算を高速に実行できるFPU(浮動小数点演算ユニット)を搭載しています。FPUを使う時の割り込み処理ルーチンにてひとつ考慮に入れて頂きたい事項があります。FPUは、既存のC28x CPUを拡張するユニットとなっていて、FPU用のレジスタが追加されています。何も工夫をしないと、レジスタが増えた分だけ、割り込み発生時のコンテキスト・セーブ(レジスタの現状保存)に時間がかかってしまいます。そこで、FPUには、High Priority割り込みモードというモードが用意されています。FPUレジスタのR0H～R7HとSTFレジスタは、シャドウ化されたレジスタになっています。このHigh Priority割り込みモードでは、シャドウ・レジスタを利用して、高速にコンテキストセーブを行えます。しかし、欠点もあります。シャドウ・レジスタは１面しかありませんので、割り込みのネストはできません。ここは注意が必要です。C言語でISRを記述する時は、

```
interrupt  void func(void){
    処理
}
```

のように、interruptキーワードをつけますが、これだけですと、High Priority割り込みモードではなく、Low Priority割り込みモードという、遅いコンテキスト・セーブ(スタックに地道のPUSHします)になります。こちらは、ネストも可能です（ソフトウェアが介在する必要があります）。

High Priority割り込みモードを利用するためには、

```
#pragma INTERRUPT(関数名, HPI);
```

を利用します。

例えば、

```
#pragma INTERRUPT(func, HPI);
interrupt  void func(void){
  処理
}
```

というように使います。これで、このISRがHPI(High Priority Interrupt)ですと宣言し、コンパイラは、High Priority割り込み用のコードを生成してくれます。

また、FPU用に高速なatan/atan2/cos/div/isqrt/sin/sincos/sqrt演算ライブラリ(fastRTSライブラリ)が用意されていますので、こちらのご利用もご検討下さい。

C:\ti\controlSUITE\libs\math\FPUfastRTS\

にあります。

13 更新履歴

文章番号	更新内容
JAJA230	初期リリース版 ● Code Composer Studio Ver4.x用
JAJA230A	● 記述間違いの更新（更新箇所が多いため、詳細は省略） ● 記述内容の更新 （更新箇所が多いため、詳細は省略） ● Code Composer Studio Ver 5.2用に更新 ● Header Filesのバージョンを変更 ● CLA、SCI、DMAを追加 ● 対応ボードにC2000 LaunchPadを追加

ご注意

日本テキサス・インスツルメンツ株式会社（以下TIJといいます）及びTexas Instruments Incorporated（TIJの親会社、以下TIJないしTexas Instruments Incorporatedを総称してTIといいます）は、その製品及びサービスを任意に修正し、改善、改良、その他の変更をし、もしくは製品の製造中止またはサービスの提供を中止する権利を留保します。従いまして、お客様は、発注される前に、関連する最新の情報を取得して頂き、その情報が現在有効かつ完全なものであるかどうかご確認下さい。全ての製品は、お客様とTIJとの間に取引契約が締結されている場合は、当該契約条件に基づき、また当該取引契約が締結されていない場合は、ご注文の受諾の際に提示されるTIJの標準販売契約約款に従って販売されます。

TIは、そのハードウェア製品が、TIの標準保証条件に従い販売時の仕様に対応した性能を有していること、またはお客様とTIJとの間で合意された保証条件に従い合意された仕様に対応した性能を有していることを保証します。検査およびその他の品質管理技法は、TIが当該保証を支援するのに必要とみなす範囲で行なわれております。各デバイスの全てのパラメーターに関する固有の検査は、政府がそれ等の実行を義務づけている場合を除き、必ずしも行なわれておりません。

TIは、製品のアプリケーションに関する支援もしくはお客様の製品の設計について責任を負うことはありません。TI製部品を使用しているお客様の製品及びそのアプリケーションについての責任はお客様にあります。TI製部品を使用したお客様の製品及びアプリケーションについて想定されうる危険を最小のものとするため、適切な設計上および操作上の安全対策は、必ずお客様にてお取り下さい。

TIは、TIの製品もしくはサービスが使用されている組み合せ、機械装置、もしくは方法に関連しているTIの特許権、著作権、回路配置利用権、その他のTIの知的財産権に基づいて何らかのライセンスを許諾するということは明示的にも黙示的にも保証も表明もしておりません。TIが第三者の製品もしくはサービスについて情報を提供することは、TIが当該製品もしくはサービスを使用することについてライセンスを与えるとか、保証もしくは是認するということを意味しません。そのような情報を使用するには第三者の特許その他の知的財産権に基づき当該第三者からライセンスを得なければならない場合もあり、またTIの特許その他の知的財産権に基づきTI からライセンスを得て頂かなければならない場合もあります。

TIのデータ・ブックもしくはデータ・シートの中にある情報を複製することは、その情報に一切の変更を加えること無く、かつその情報と結び付られた全ての保証、条件、制限及び通知と共に複製がなされる限りにおいて許されるものとします。当該情報に変更を加えて複製することは不公正で誤認を生じさせる行為です。TIは、そのような変更された情報や複製については何の義務も責任も負いません。

TIの製品もしくはサービスについてTIにより示された数値、特性、条件その他のパラメーターと異なる、あるいは、それを超えてなされた説明で当該TI製品もしくはサービスを再販売することは、当該TI製品もしくはサービスに対する全ての明示的保証、及び何らかの黙示的保証を無効にし、かつ不公正で誤認を生じさせる行為です。TIは、そのような説明については何の義務も責任もありません。

TIは、TIの製品が、安全でないことが致命的となる用途ないしアプリケーション（例えば、生命維持装置のように、TI製品に不良があった場合に、その不良により相当な確率で死傷等の重篤な事故が発生するようなもの）に使用されることを認めておりません。但し、お客様とTIの双方の権限有る役員が書面でそのような使用について明確に合意した場合は除きます。たとえTIがアプリケーションに関連した情報やサポートを提供したとしても、お客様は、そのようなアプリケーションの安全面及び規制面から見た諸問題を解決するために必要とされる専門的知識及び技術を持ち、かつ、お客様の製品について、またTI製品をそのような安全でないことが致命的となる用途に使用することについて、お客様が全ての法的責任、規制を遵守する責任、及び安全に関する要求事項を満足させる責任を負っていることを認め、かつそのことに同意します。さらに、もし万一、TIの製品がそのような安全でないことが致命的となる用途に使用されたことによって損害が発生し、TIないしその代表者がその損害を賠償した場合は、お客様がTIないしその代表者にその全額の補償をするものとします。

TI製品は、軍事的用途もしくは宇宙航空アプリケーションないし軍事的環境、航空宇宙環境にて使用されるようには設計もされていませんし、使用されることを意図されておりません。但し、当該TI製品が、軍需対応グレード品、若しくは「強化プラスティック」製品としてTIが特別に指定した製品である場合は除きます。TIが軍需対応グレード品として指定した製品のみが軍需品の仕様書に合致いたします。お客様は、TIが軍需対応グレード品として指定していない製品を、軍事的用途もしくは軍事的環境下で使用することは、もっぱらお客様の危険負担においてなされるということ、及び、お客様がもっぱら責任をもって、そのような使用に関して必要とされる全ての法的要求事項及び規制上の要求事項を満足させなければならないことを認め、かつ同意します。

TI製品は、自動車用アプリケーションないし自動車の環境において使用されるようには設計されていませんし、また使用されることを意図されておりません。但し、TIがISO/TS 16949の要求事項を満たしていると特別に指定したTI製品は除きます。お客様は、お客様が当該TI指定品以外のTI製品を自動車用アプリケーションに使用しても、TIは当該要求事項を満たしていなかったことについて、いかなる責任も負わないことを認め、かつ同意します。

弊社半導体製品の取り扱い・保管について

半導体製品は、取り扱い、保管・輸送環境、基板実装条件によっては、お客様での実装前後に破壊/劣化、または故障を起こすことがあります。

弊社半導体製品のお取り扱い、ご使用にあたっては下記の点を遵守して下さい。

1. 静電気
   - 素手で半導体製品単体を触らないこと。どうしても触る必要がある場合は、リストストラップ等で人体からアースをとり、導電性手袋等をして取り扱うこと。
   - 弊社出荷梱包単位（外装から取り出された内装及び個装）又は製品単品で取り扱いを行う場合は、接地された導電性のテーブル上で（導電性マットにアースをとったもの等）、アースをした作業者が行うこと。また、コンテナ等も、導電性のものを使うこと。
   - マウンタやはんだ付け設備等、半導体の実装に関わる全ての装置類は、静電気の帯電を防止する措置を施すこと。
   - 前記のリストストラップ・導電性手袋・テーブル表面及び実装装置類の接地等の静電気帯電防止措置は、常に管理されその機能が確認されていること。
2. 温・湿度環境
   - 温度：0～40℃、相対湿度：40～85%で保管・輸送及び取り扱いを行うこと。（但し、結露しないこと。）
   - 直射日光があたる状態で保管・輸送しないこと。
3. 防湿梱包
   - 防湿梱包品は、開封後は個別推奨保管環境及び期間に従い基板実装すること。
4. 機械的衝撃
   - 梱包品（外装、内装、個装）及び製品単品を落下させたり、衝撃を与えないこと。
5. 熱衝撃
   - はんだ付け時は、最低限260℃以上の高温状態に、10秒以上さらさないこと。（個別推奨条件がある時はそれに従うこと。）
6. 汚染
   - はんだ付け性を損なう、又はアルミ配線腐食の原因となるような汚染物質（硫黄、塩素等ハロゲン）のある環境で保管・輸送しないこと。
   - はんだ付け後は十分にフラックスの洗浄を行うこと。（不純物含有率が一定以下に保証された無洗浄タイプのフラックスは除く。）

以上

2001.11