Technical Note

TecStar

Silicon Labs 社 EFM32
クイックスタートガイド

2016 年 2 月

株式会社マクニカ
テクスター カンパニー

TecStar
Company

Silicon Labs 社 EFM32

クイックスタートガイド

目次

1 はじめに

この資料は、Silicon Laboratories（以下、Silicon Labs）社製 MCU　EFM32 ファミリの開発環境について簡易にまとめたものです。内容に誤りがないよう注意は払っておりますが、もし Silicon Labs 社が提供するドキュメント等と差異がございましたら、メーカー提供のものを優先してご参照ください。

また、Silicon Labs 社の　ナレッジベース（FAQ）やコミュニティフォーラム（ユーザ同士で問題解決。Silicon Labs のエンジニアも頻繁にコメントしています）には、本資料で取り上げていない様々な情報が記載されております。

製品をご使用頂く過程で疑問や課題が生じることもあると思いますが、他のユーザが既に解決方法を見つけている場合も多々ございます。非常に有益ですので、ぜひご活用下さい。

◆　アクセス方法

Simplicity Studio から

Web Site から

http://community.silabs.com/t5/Forum/ct-p/Forum

◆　使用方法

Community か Knowledge Base を選択

2 開発環境のご紹介

EFM32 の開発環境について、ハードウェアとソフトウェアに分けてご紹介します。

2-1 ハードウェア

EFM32 の開発環境としては、Starter Kit と Development Kit を用意しております。Starter Kit には周辺装置やセンサが実装されていますので、簡単に EFM32 の機能を確認できるようになっています。Development Kit は拡張性が高く、より高度な検証を行えるようになっています。

2-1-1 EFM32 Starter Kit

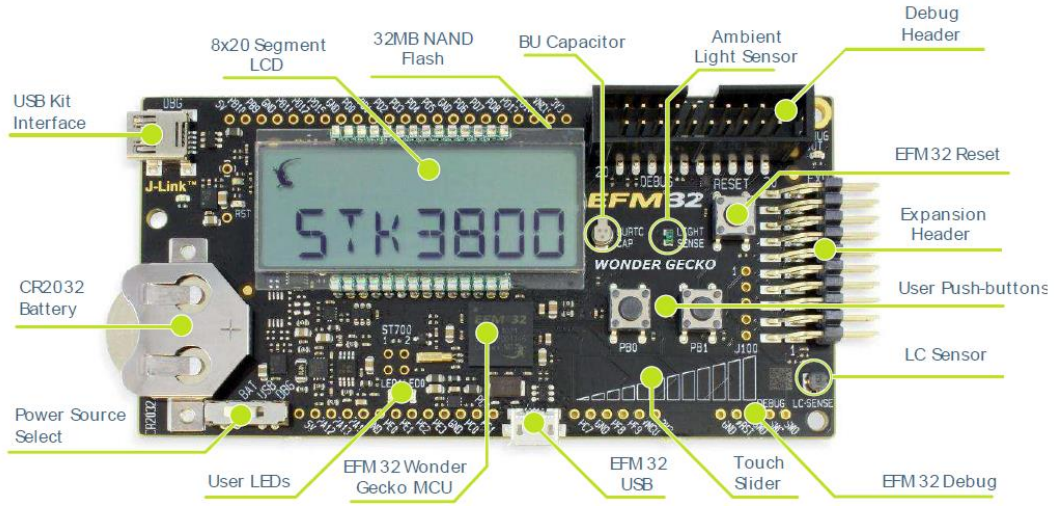
Starter Kit は、各ファミリに 1 種ずつ用意されています。同一ファミリであっても、USB 有無、LCD コントローラ有無などで幾つもの製品型番が用意されていますが、Starter Kit にはフルセットの MCU が実装されていますので、これを用いて設計を進めて頂くことが可能です。

Starter Kit には、下記が同梱されています。

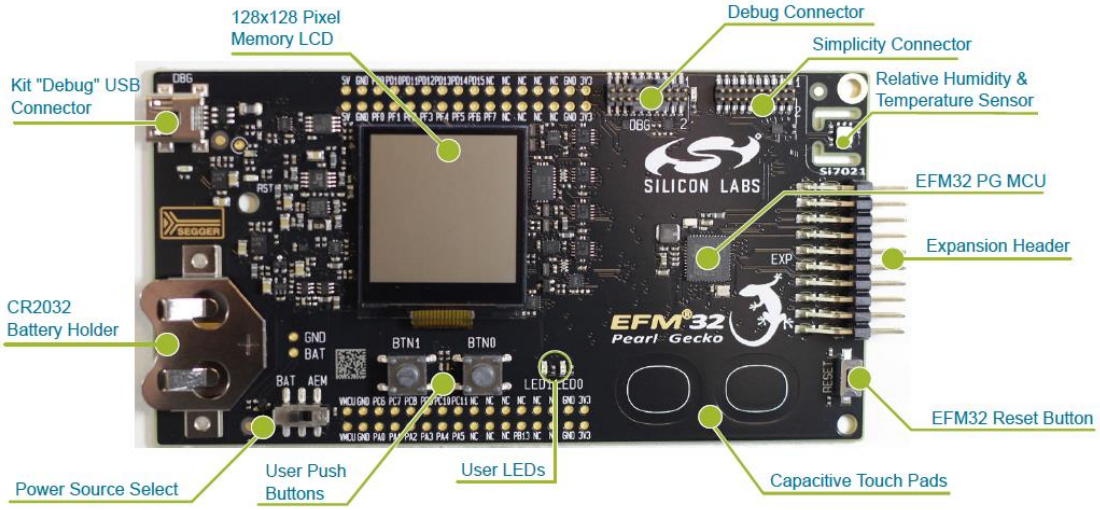
・Starter Kit
・IAR Embedded Workbench 評価版
・ケーブル各種

ファミリ名	CPU Core	Starter Kit	実装されている型番
Wonder Gecko	Cortex M4F	EFM32WG-STK3800	EFM32WG990F256
Pearl Gecko	Cortex M4	SLSTK3401A（JG と共通）	EFM32PG1B200F256
Giant Gecko	Cortex M3	EFM32GG-STK3700	EFM32GG990F1024
Leopard Gecko	Cortex M3	EFM32LG-STK3600	EFM32LG990F256
Jade Gecko	Cortex M3	SLSTK3401A（PG と共通）	EFM32PG1B200F256
Gecko	Cortex M3	EFM32-G8XX-STK	EFM32G290F128
Tiny Gecko	Cortex M3	EFM32TG-STK3300	EFM32TG840F32
Happy Gecko	Cortex M0+	SLSTK3400A	EFM32HG322F64
Zero Gecko	Cortex M0+	EFM32ZG-STK3200	EFM32ZG222F32

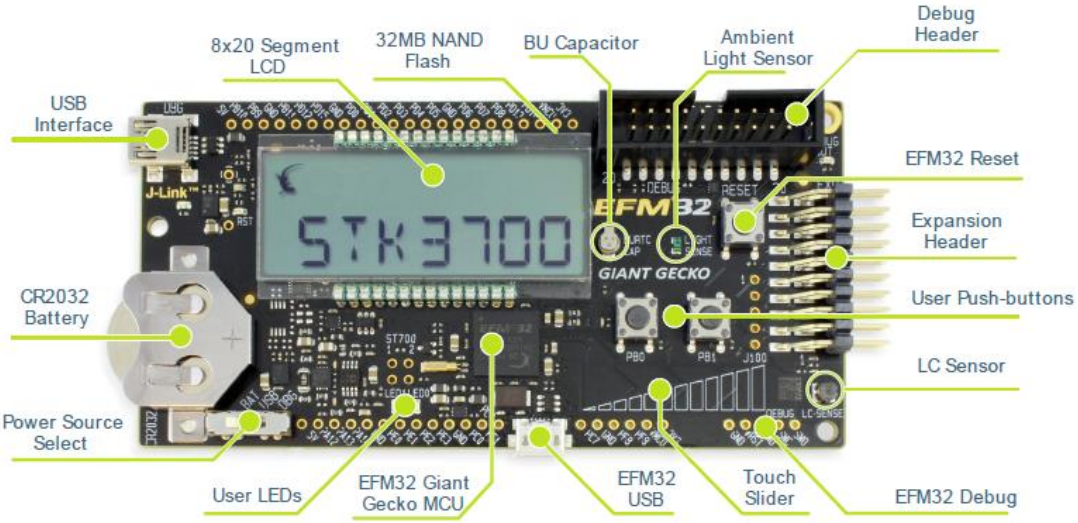
◆ Wonder Gecko : EFM32WG-STK3800



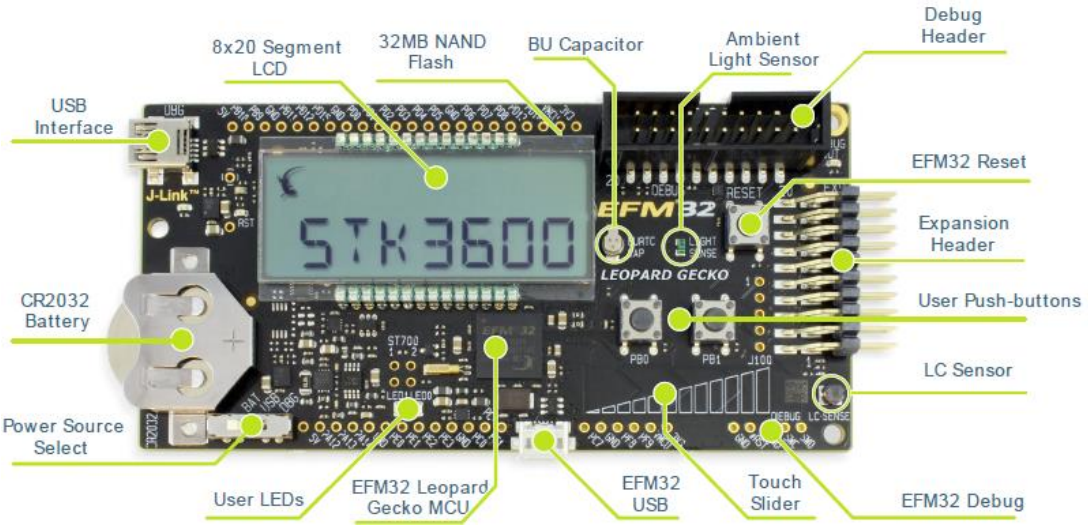
◆ Pearl Gecko / Jade Gecko : SLSTK3400A



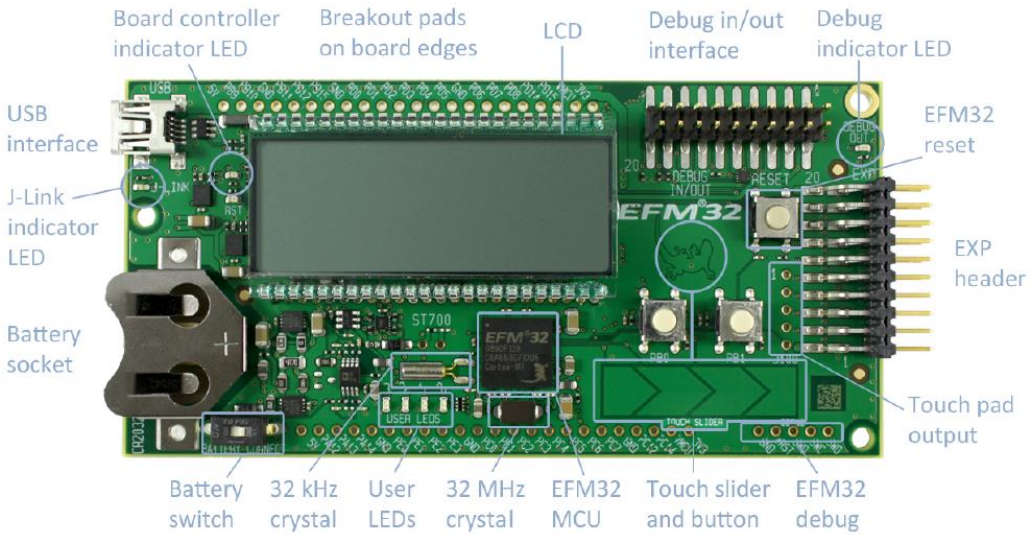
◆ Giant Gecko : EFM32GG-STK3700



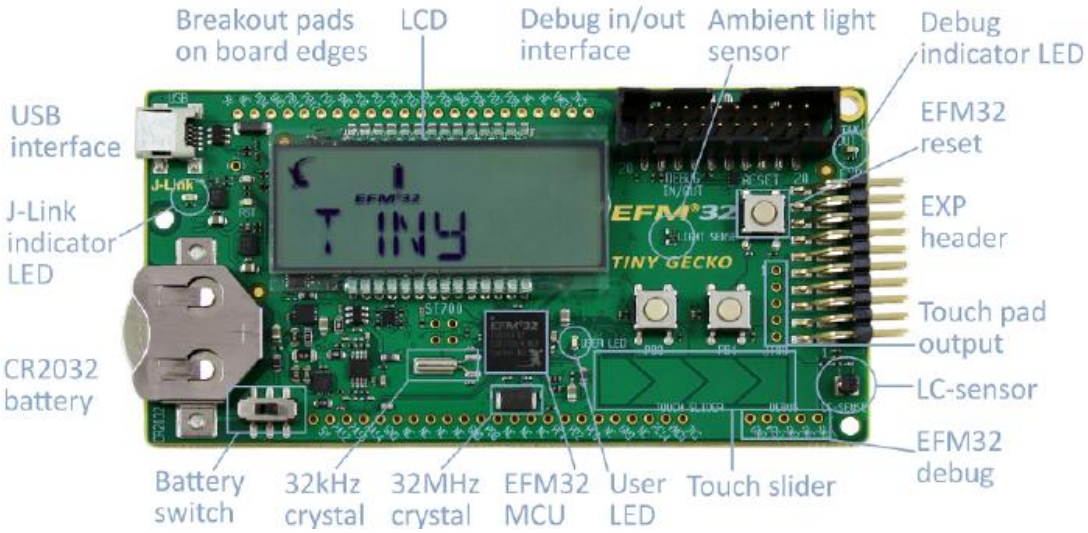
◆ Leopard Gecko : EFM32LG-STK3600



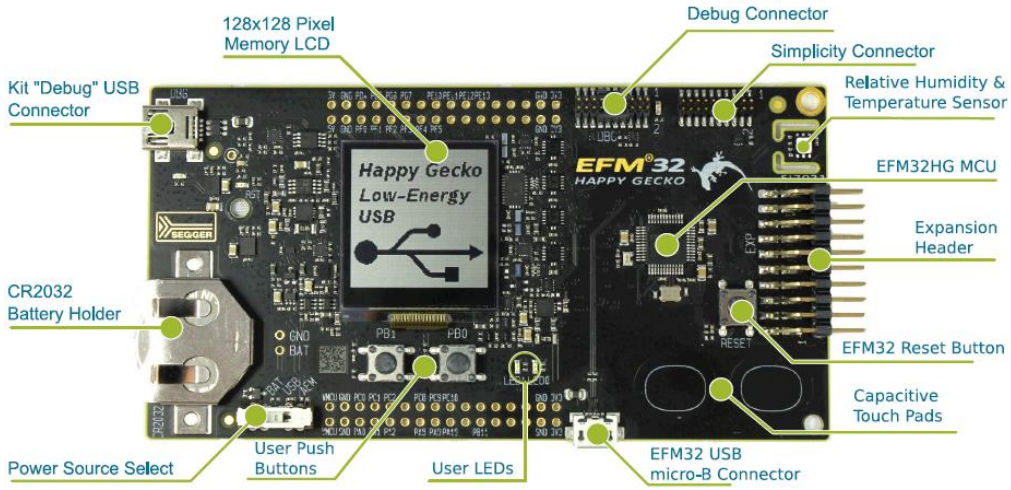
◆ Gecko : EFM32-G8XX-STK



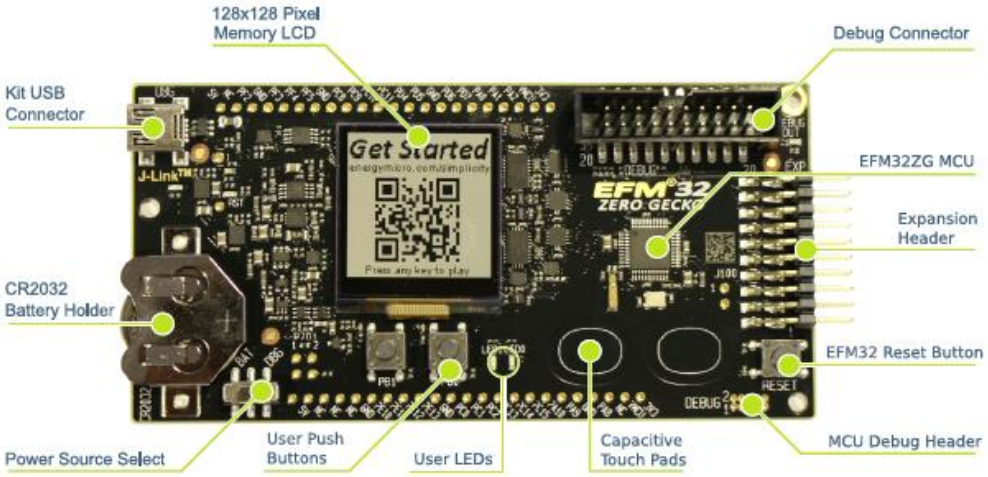
◆ Tiny Gecko : EFM32TG-STK3300



◆ Happy Gecko : SLSTK3400A



◆ Zero Gecko : EFM32ZG-STK3200



2-1-2 EFM32 Development Kit

Development Kit は 3 枚の基板から構成されています。EFM32 が実装された MCU プラグイン・ボードと、色々な部品を実装できるプロトタイピング・ボードの 2 つを、マザーボードに挿入して使用します。

Development Kit には、下記が同梱されています。

・EFM32 Development Kit マザーボード

・EFM32 MCU プラグイン・ボード

・EXP32 プロトタイピング・ボード

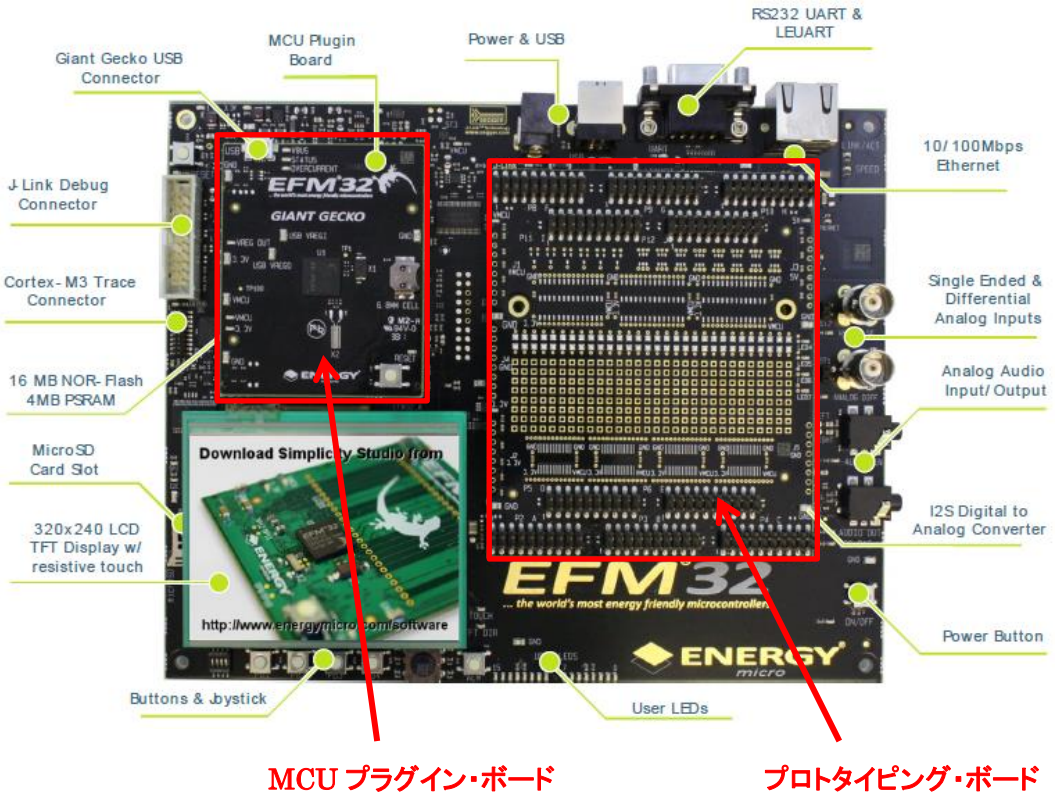
・IAR Embedded Workbench 評価版

・Atollic TrueSTUDIO

・ケーブル各種

ファミリ名	CPU Core	Starter Kit	実装されている型番
Wonder Gecko	Cortex M4F	EFM32WG-DK3850	EFM32WG990F256
Giant Gecko	Cortex M3	EFM32GG-DK3750	EFM32GG990F1024
Leopard Gecko	Cortex M3	EFM32LG-DK3650	EFM32LG990F256
Gecko	Cortex M3	EFM32G-DK3550	EFM32G890F128

◆ Giant Gecko : EFM32GG-DK3750



2-2 ソフトウェア

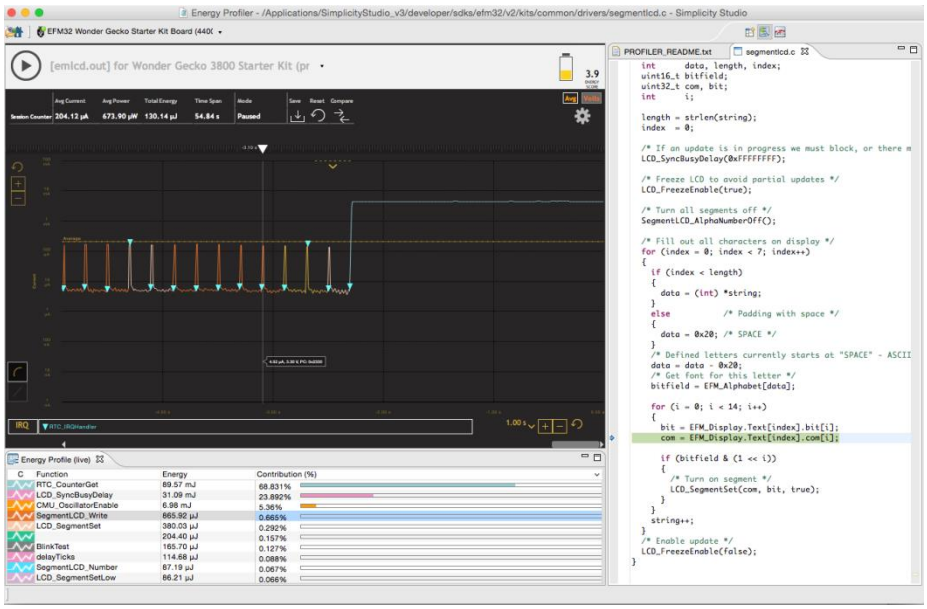
EFM32 の開発環境である Simplicity Studio を使用して設計を行うことになります。

2-2-1 Simplicity Studio

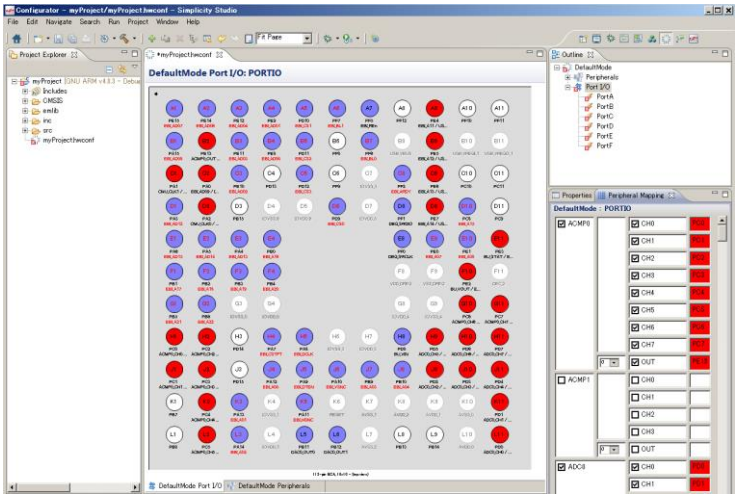
Simplicity Studio は、EFM32 をターゲットとしたコンパイル・デバッグ・プログラミングを 1 つのプラットで提供することができるソフトウェアです。統合開発環境（IDE）を中心に、非常に便利なツール群が充実しています。同社製の 8bit MCU や無線 MCU も同一プラットフォームで開発が可能です。

ツール名	機能の概要
Simplicity IDE	統合開発環境（IDE）。無償の GCC コンパイラを搭載
Energy Profiler	実機の消費電流値を測定することが可能
Configurator	ピン設定やペリフェラル設定を簡単に行うことができる
energy Aware Battery	消費電流値のシミュレータ機能。バッテリ寿命も簡単に算出
Flash Programmer	フラッシュ ROM のライト／イレース
Kit Manager	キット情報、デバッグ対象の選択

◆ 消費電流が実測できます（Energy Profiler）



◆ ピン設定やペリフェラル設定を簡単に行えます（Configurator）



◆ バッテリ寿命のシミュレーションが行えます（energy Aware Battery）

3 各種ドキュメント・サンプルコードの入手先

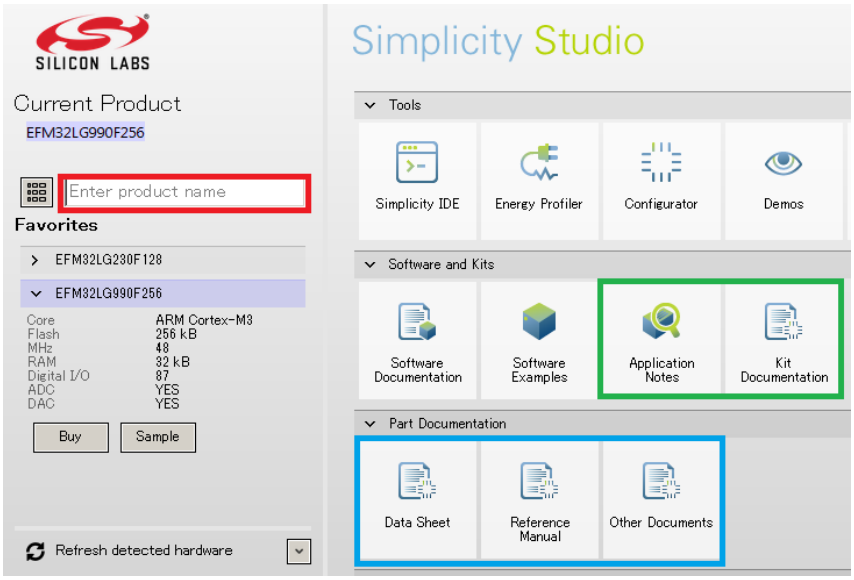
EFM32 の最新ドキュメント・サンプルコードの入手方法について紹介します。

3-1 EFM32 のドキュメント

EFM32 のドキュメントは、Simplicity Studio を経由してご入手頂けます。Simplicity Studio の入手先やインストール方法につきましては、「4.ソフトウェア・インストール」をご参照ください。

Simplicity Studio を起動し、Product の欄（下図の赤枠）に使用する製品型番を入力すると、青枠・緑枠から下記情報をご入手頂けるようになります。

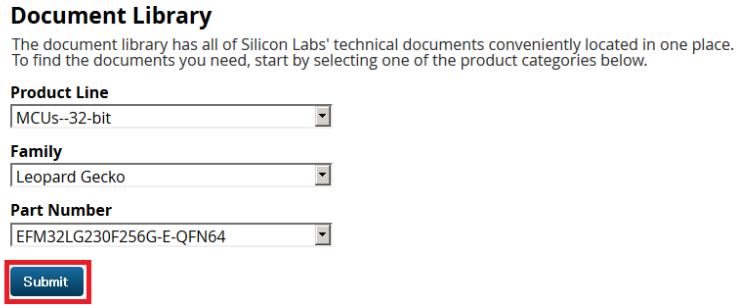
- Datasheet: データシート。スペック、ピン配置情報など
- Reference Manual: リファレンス・マニュアル。ペリフェラルの使用方法などの解説
- Other Documents : エラッタ（バグ情報）、Cortex-M ガイド
- Application Notes: アプリケーションノート、それに付随するサンプルコード
- Kit Documentation: Starter Kit のユーザガイド、回路・レイアウト情報



また、ドキュメントは Silicon Labs 社の Web Site からもご入手可能です。

http://www.silabs.com/support/pages/document-library.aspx

製品型番を入力すると、関連ドキュメントがリストアップされます。



3-2 Starter Kit と Development Kit のユーザガイド

以下に Starter Kit および Development Kit のユーザガイドのリンクを掲載します。Happy Gecko、Pearl Gecko、Jade Gecko については Quick Start Guide へのリンクです。ユーザガイドは Simplicity Studio をインストールしてからご入手ください。

◆ Starter Kit のユーザガイド

Wonder: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32wg-stk3800-ug.pdf

Pearl: http://www.silabs.com/Support%20Documents/TechnicalDocs/QSG118.pdf

Giant: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32gg-stk3700-ug.pdf

Leopard: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32lg-stk3600-ug.pdf

Jade: http://www.silabs.com/Support%20Documents/TechnicalDocs/QSG118.pdf

Gecko: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32-stk-documentation.zip

Tiny: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32-stk-documentation.zip

Happy: https://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32HG-SLSTK3400A-QuickStartGuide.pdf

Zero: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32zg-stk3200-ug.pdf

◆ Development Kit のユーザガイド

Wonder: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32gg-dk3750-ug.pdf

Giant: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32gg-dk3750-ug.pdf

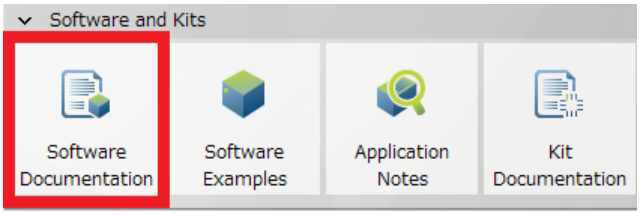
Leopard: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32lg-dk3650-ug.pdf

Gecko: http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32g-dk3550-ug.pdf

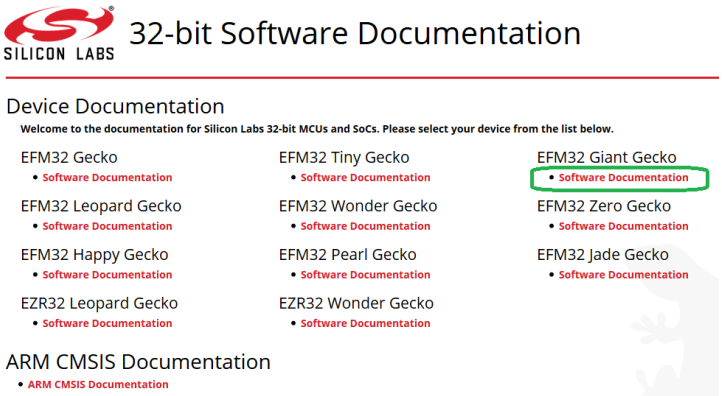
3-3 EFM32 の API 情報

ペリフェラルを制御するためライブラリ（API）が用意されており、それを使用することでソフト設計を円滑に進めて頂くことが可能です。

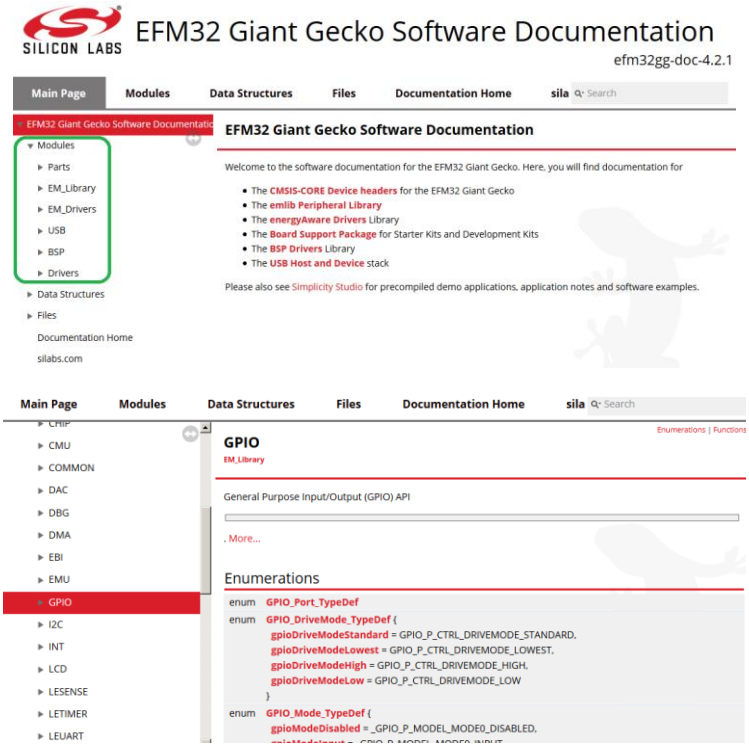
Simplicity Studio を起動し、Product の欄に使用する製品型番を入力して、Software Documentation をクリックします。（インターネットに接続した状態で行ってください。）



Help が起動し、製品ファミリー覧が表示されますので、使用するファミリを選択します。



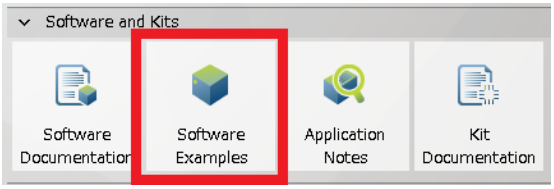
ウィンドウ左で閲覧したいライブラリや API を選択すると、右側に情報が表示されます。



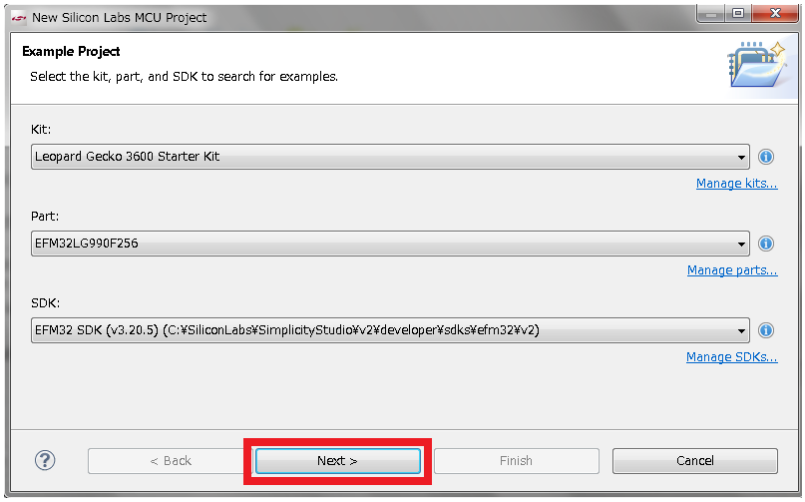
3-4 EFM32 のサンプルコード

EFM32 のサンプルコードは、Simplicity Studio を経由してご入手頂けます。Simplicity Studio の入手先やインストール方法につきましては、「4.ソフトウェア・インストール」をご参照ください。

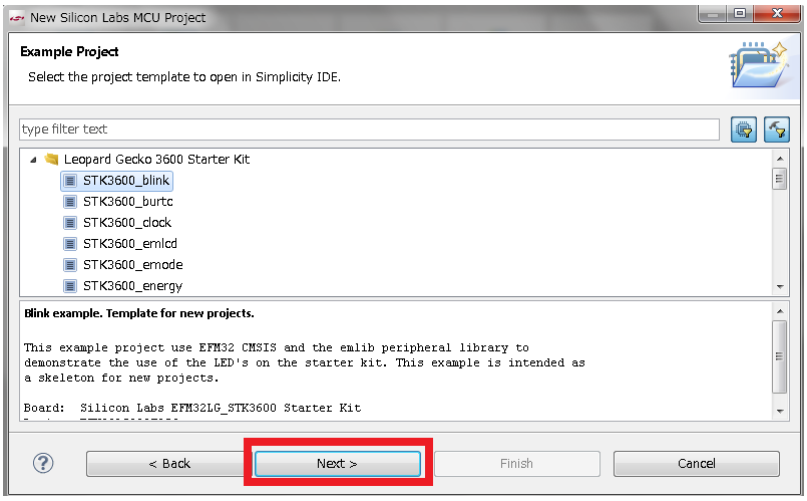
Simplicity Studio を起動し、Product の欄に使用する製品型番を入力して、Software Examples をクリックします。



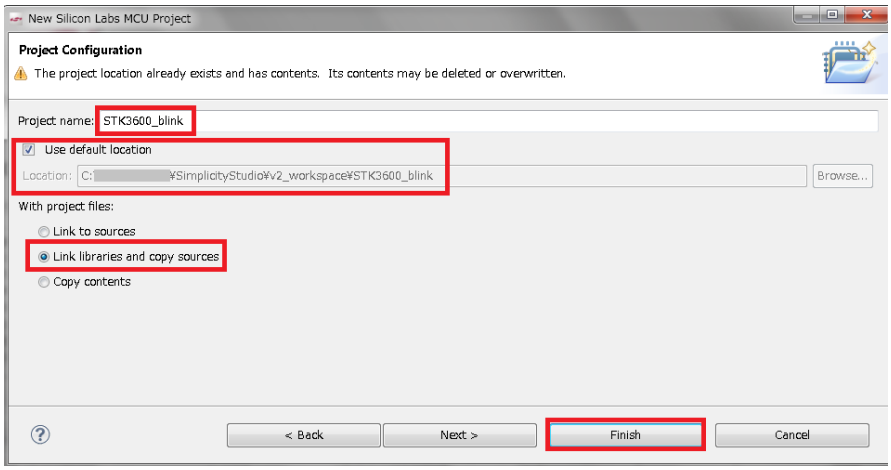
Example Project ウィンドウにて、Kit で使用する Starter Kit が選択されていることを確認し、Next ボタンをクリックします。



生成したいサンプルコードを選択し、Next をクリックします。



プロジェクト名を入力し、サンプルコードの生成フォルダを確認して、Finish ボタンをクリックします。プロジェクトとサンプルコードが生成されます。



生成が完了すると、プロジェクトが自動でロードされ、コンパイルが行える状態になります。

4 ソフトウェア・インストール

EFM32 の評価に必要なソフトウェアをインストールします。

4-1 Simplicity Studio のインストール

Simplicity Studio を下記アドレスからダウンロードし、install-studio.exe を実行してインストールを行います。インストール path に全角文字（日本語）が入らないようにご注意ください。

http://www.silabs.com/products/mcu/Pages/simplicity-studio.aspx

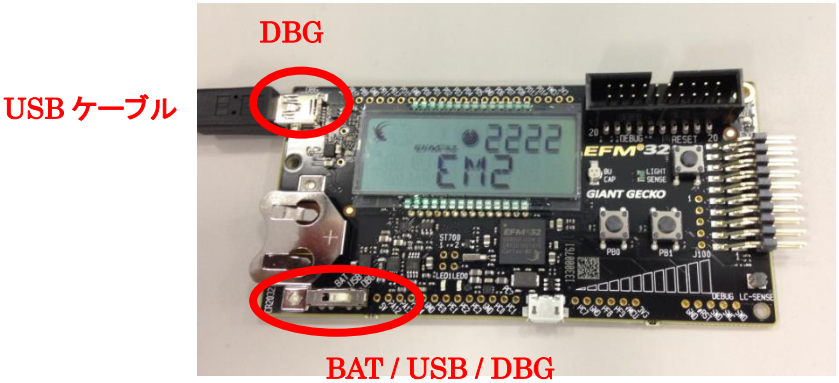
5 ハードウェア・セットアップ

EFM32 の評価に必要なハードウェアの設定を行います。

5-1 Starter Kit のセットアップ

以下の手順で設定していきます。

1. BAT, USB, DBG の中から、基板に給電する方法を選びます。スイッチを DBG に切り替えます。
2. DBG と PC を USB ケーブルで接続します



6 使用方法

Starter Kit と Simplicity Studio を使用した評価手順をご紹介します。ここでは STK3600（Leopard Gecko）を使用しておりますが、他の Starter Kit でも手順は同じです。

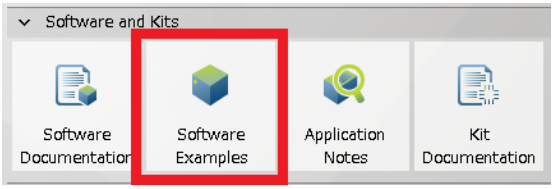
なお、各ツールから Simplicity Studio の Top 画面に戻るには、画面右上のアイコンを使用します。

6-1 サンプルコードを動かしてみる

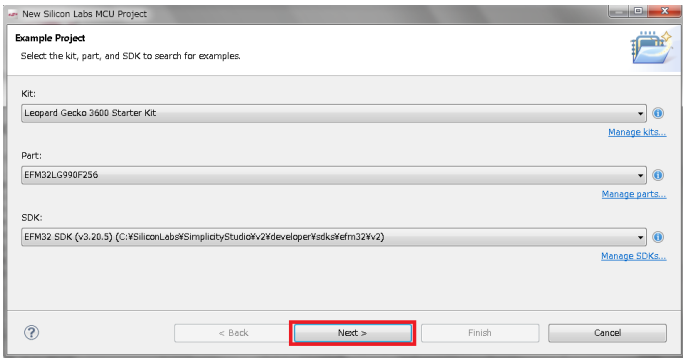
Starter Kit 上の LED を点滅させるサンプルコードを、ダウンロードして動作を見てみます。

Starter Kit を PC に接続すると、Simplicity Studio が Starter Kit を自動認識します。うまく認識してくれない場合には、Refresh detected hardware アイコンを押してみてください。

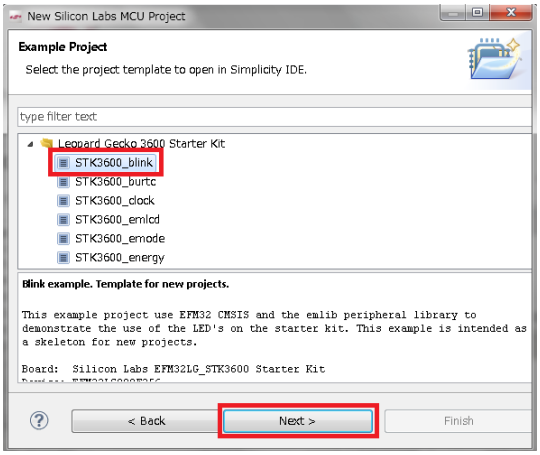
Software Examples を選択します。



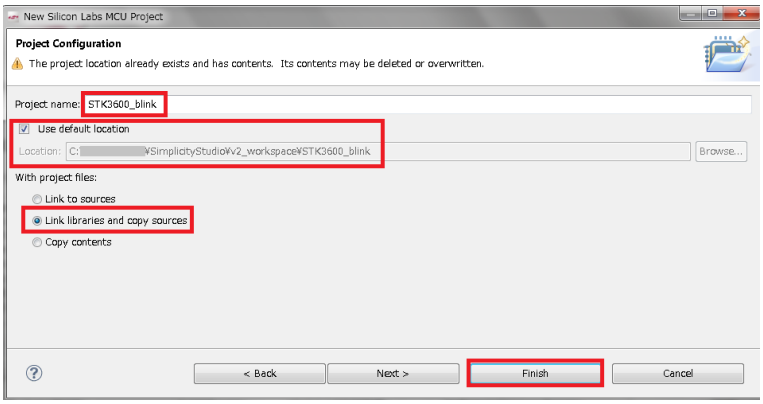
接続した Starter Kit に合せて、Kit, Part, SDK が自動で選ばれますので、Next をクリックします。



Example Project で STK3600_blink（STKxxxx_blink）を選択し、Next をクリックします。

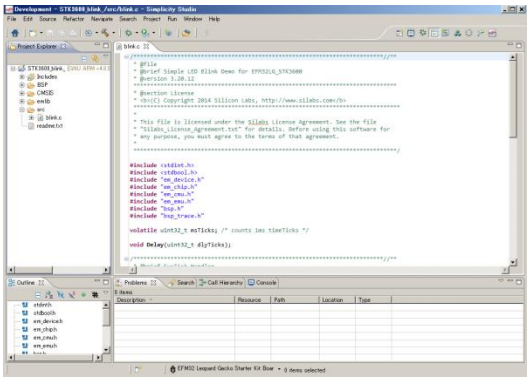


プロジェクト名を入力し、作業フォルダを指定します。With project files では、サンプルコードをローカルにコピーして使うかどうかを指定します。指定が終わったら、Finish をクリックします。

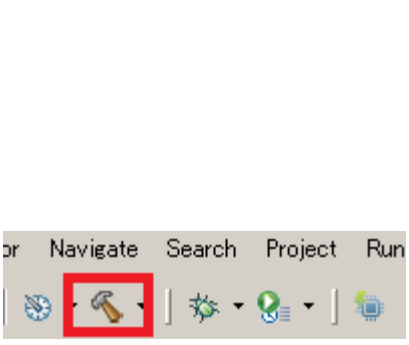


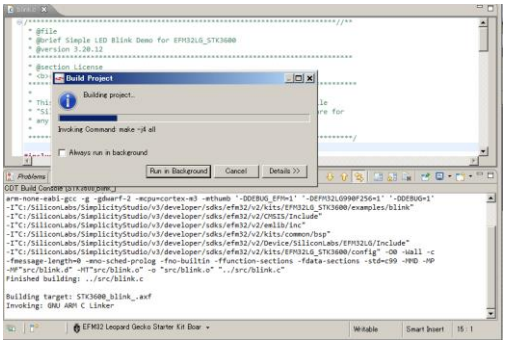
With project files	内容
Link to sources	ライブラリもソースも、オリジナルのものを使う。ライブラリもソースも修正しない人向け。
Link libraries and copy sources	ライブラリはオリジナルのものを参照し、ソースコードはローカルにコピーして使う。
Copy contents	ライブラリもソースも、ローカルにコピーして使う。ライブラリを修正する可能性がある人向け。

サンプルコードの準備が整うと、Simplicity IDE が起動します。Simplicity IDE の使い方については「6-2 デバッグ機能を使ってみる」で詳しく紹介します。

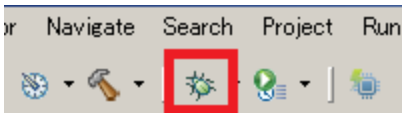


この Simplicity IDE を使用して、サンプルコードをビルドし、Starter Kit にダウンロードします。まずはトンカチのアイコン（Build）をクリックします。コンパイラが走り、サンプルコードがビルドされます。

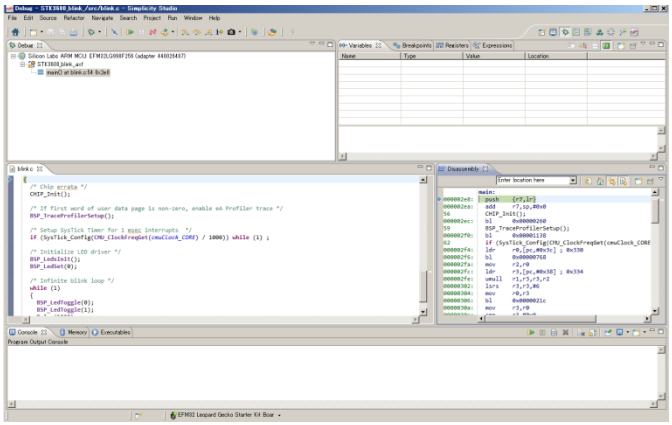




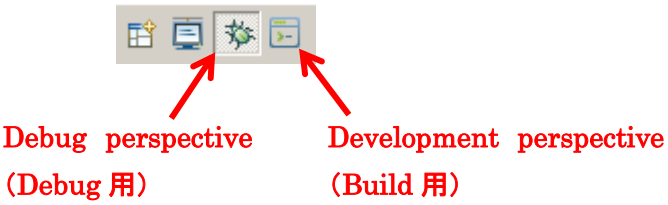
ビルドが完了したら、次に虫のアイコン（Debug）をクリックし、Starter Kit にダウンロードします。



ダウンロードが完了すると、デバッグ用の画面に切り替わります。

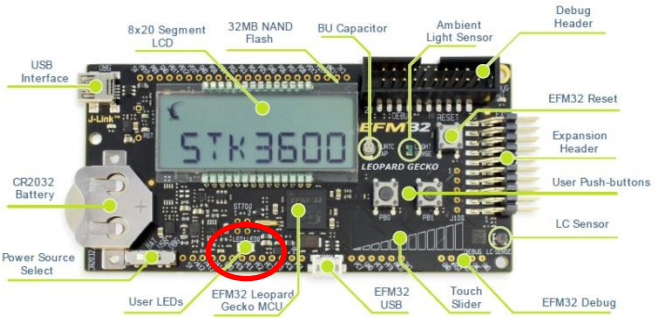


ビルド用の画面と、デバッグ用の画面の切り替えは、ウィンドウ右上のアイコンで行います。



サンプルコードを実行します。下図の実行のアイコン（Resume）をクリックしてください。

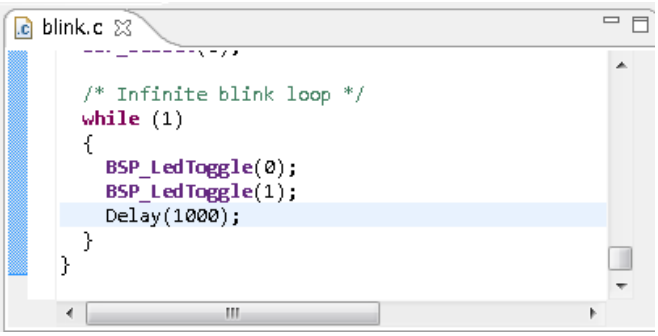
Starter Kit 上の LED が、ゆっくりと点滅しているのが確認できます。STK3600 の場合には、LCD 下の LED0 と LED1 が点滅します。



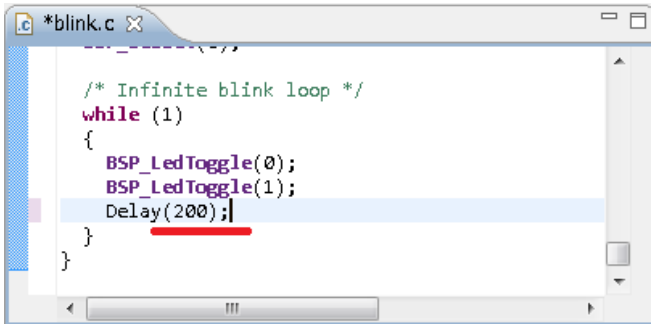
◆ LED の点滅スピードを変更してみましょう。

Development perspective アイコンをクリックして、ビルド用の画面に切り替えます。

画面中央に、コード（blink.c）が表示されていると思います。blink.c の最後あたりに、Delay(1000)という記述が見つかります。この数値が点滅時間を決定しています。



値を 200 に変更してみます。



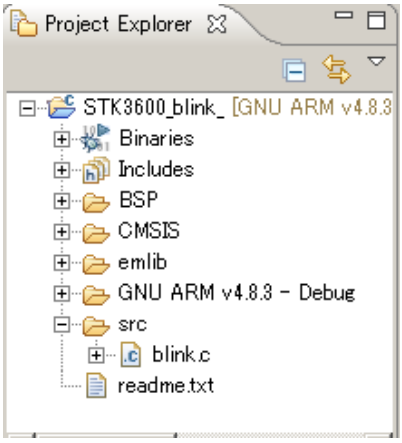
あとは、ビルドして、ダウンロードして、実行します。

先ほどと同じ手順で、トンカチのアイコン（Build）⇒虫のアイコン（Debug）⇒実行のアイコン（Resume）の順にクリックします。

Starter Kit の LED の点滅が、先ほどよりも早くなったことを確認できるかと思います。

◆ サンプルコードのツリー構造

ビルド用の画面（Development Perspective）を開くと、Project Explorer というウィンドウがあり、サンプルコードで使用されているライブラリが表示されています。



- BSP: Starter Kit 上の、MCU 外のペリフェラル（LCD、LED など）の定義
- CMSIS: 標準 Cortex-M と、ユーザが選択した製品（Tiny, Leopard, Giant…）との差分の定義
- emlib: MCU 内のペリフェラルの定義
- src: サンプルコード

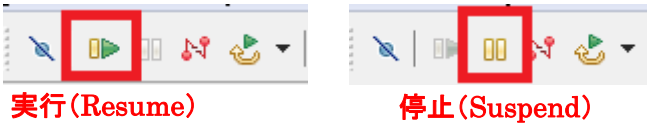
MCU 内外のペリフェラルのライブラリが用意されていますので、簡単に評価・設計を行える環境が整っています。

6-2 デバッグ機能を使ってみる（Simplicity IDE）

ソフトウェア・デバッグの際に使用する、ブレークポイント、ステップ実行などの機能は、Simplicity IDE が提供します。「6-1 サンプルコードを動かしてみる」でも紹介しましたが、Simplicity IDE には、ビルド用の画面と、デバッグ用の画面が用意されています。ソフトウェア・デバッグは、デバッグ用の画面で行います。

◆ コードの実行・停止

コードの実行には Resume ボタン、停止には Suspend ボタンを使用します。



◆ ハードウェア・リセット

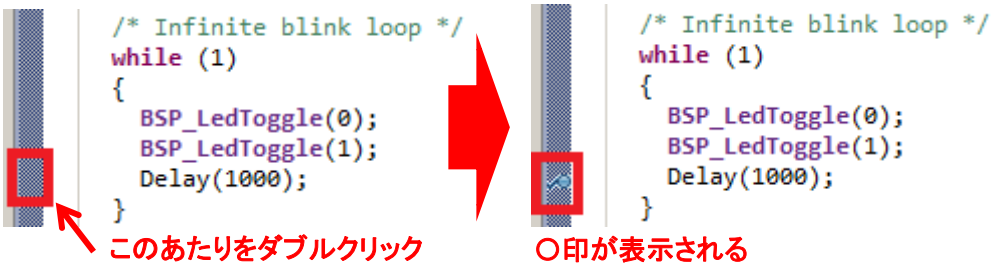
MCU にハードウェア・リセットをかけます。

◆ デバッグ経路の切断

デバッグモード経路を切断して、デバッグ用の画面を終了します。ビルド用の画面に切り替わります。

◆ ブレークポイント

ブレークポイントを設定するには、停止させたい行の左横をダブルクリックします。設定されると、水色の小さな〇印が表示されます。再度ダブルクリックすれば解除されます。



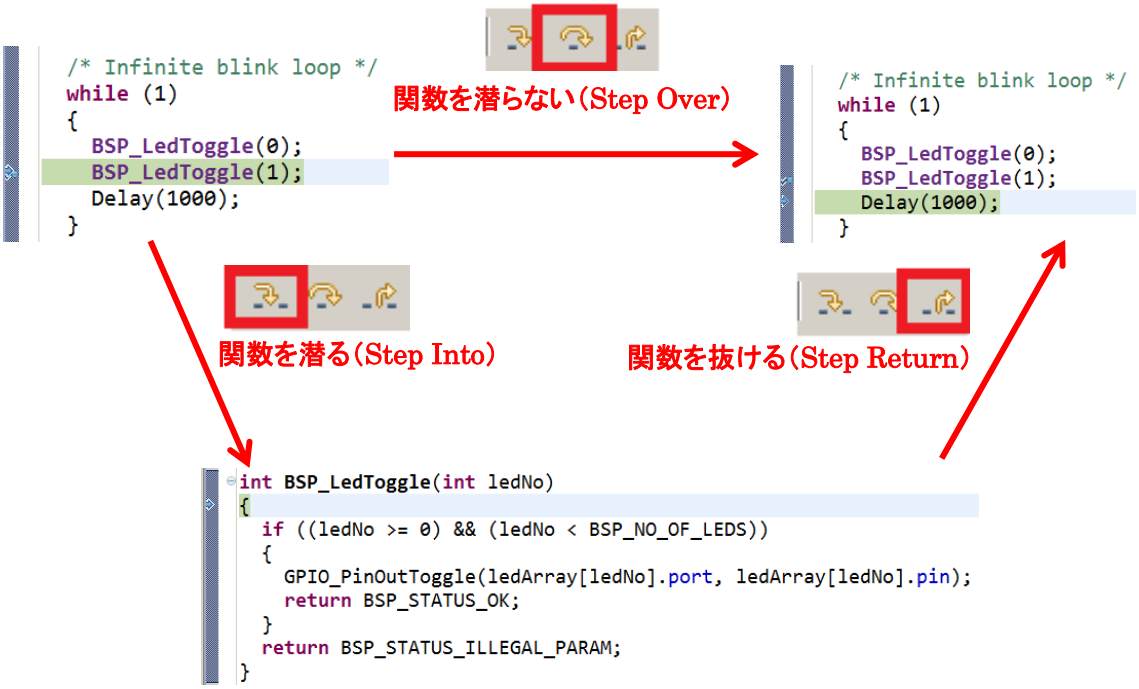
なお、ハードウェア・ブレークポイントの数は CPU Core ごとに決まっており、以下の通りです。

CPU Core	ハードウェア・ブレークポイントの数
Cortex M0+	4
Cortex M3	6
Cortex M4F	6

◆　ステップ実行

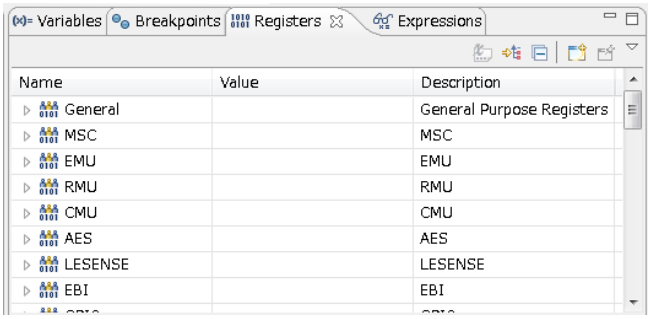
各種ステップ実行に対応しています。

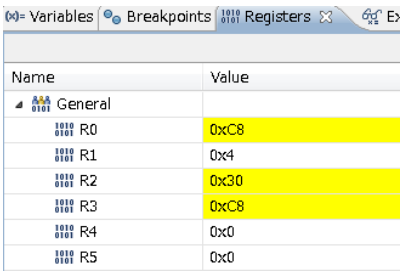
実機で実際に動作を見て頂くのが、判りやすいです。



◆　レジスタ値の閲覧・変更

レジスタ・変数の閲覧や変更は、下のウィンドウ（Register ウィンドウなど）で行うことができます。前回の停止から、値が変化した場合には黄色で表示されます。



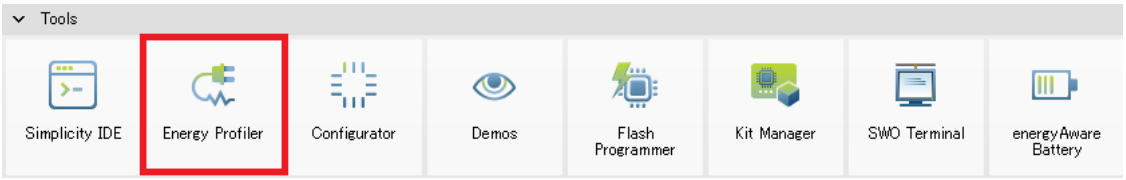


6-3 消費電流を測定してみる （Energy Profiler）

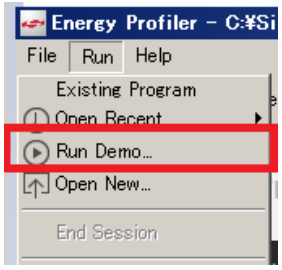
Starter Kit には電流センサが搭載されており、消費電流測定ツール（Energy Profiler）と組み合わせることで nA レベルでの電流測定が可能です。Starter Kit には LCD など外部部品も実装されていますが、MCU 単体の消費電流が測定できるように配慮されています。

ここではサンプルコードを使用して、消費電流測定ツール（Energy Profiler）の使用方法をご紹介します。

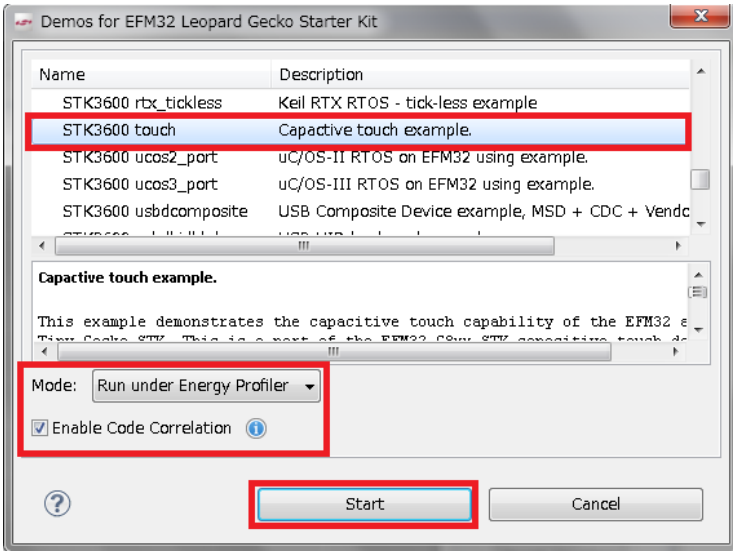
Energy Profiler を起動します。



Run メニューから、Run Demo を選択します。

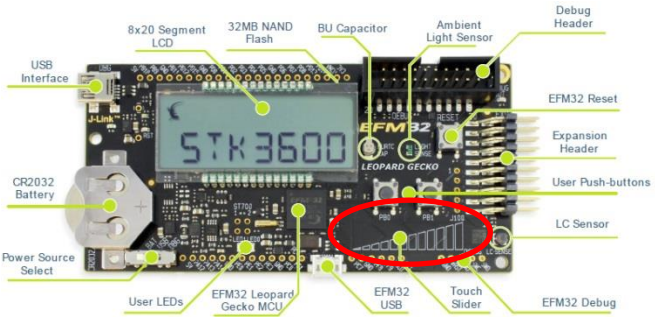


STK3600 touch（STKxxxx touch）を選択します。Mode は Run under Energy Profiler に設定し、Enable Code Correlation にチェックが付いていることを確認してください。そして Start をクリックします。



このサンプルコードは、通常時は Idle（EM2）で待機しており、Starter Kit 上のタッチ・スライダーに触れると通常動作（EM0）に移行する、というモード遷移を含んだものです。EM0⇔EM2 の遷移で消費電流が大きく変化しますので、消費電流の観点では非常に判りやすいサンプルコードになっています。

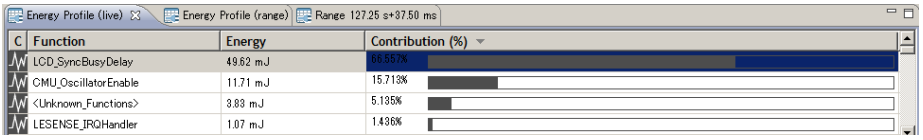
Starter Kit 上のタッチ・スライダーに触ってみます。



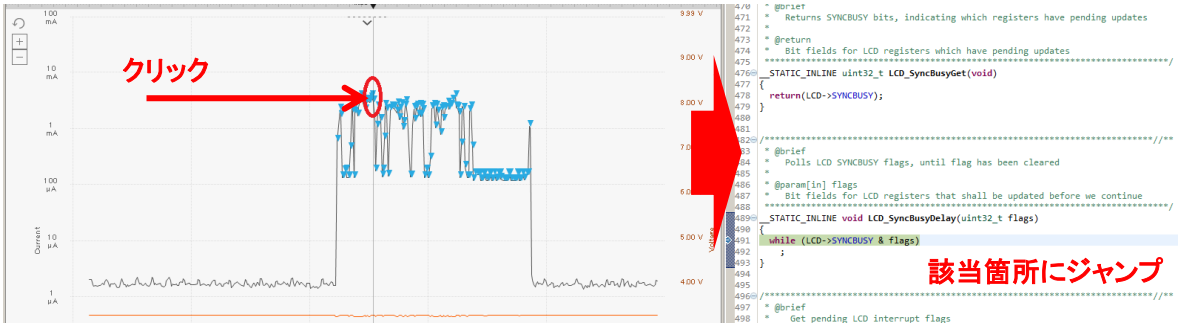
そうすると、Energy Profiler 上の消費電流値が増減するのが判ります。

初期状態では、背景は黒、フォントは小になっていますので、見づらい場合には歯車アイコンにて設定変更してください。
（左図では背景色を Light に変更）

画面下には、関数単位での消費電流値が表示されます。消費電流の最適化を行いたい場合に、どの関数に手を加えるのが効率的か？の手がかりとなります。



なお、消費電流波形（Profiling）の上でクリックをすると、その瞬間に実行されているコード部分にジャンプすることができます。消費電流が大きく変化した箇所を特定したい場合などに役立つ機能です。

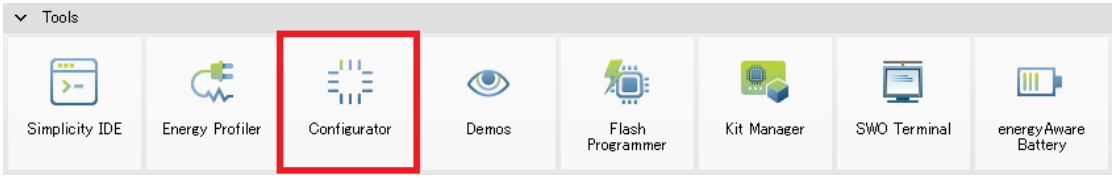


注意： Cortex-M0+（Zero Gecko）は、関数単位でのプロファイリング機能、ソースコードとの連動機能をサポートしていません。デバッグ時にだけ M3 を使用する、という方法もございます。

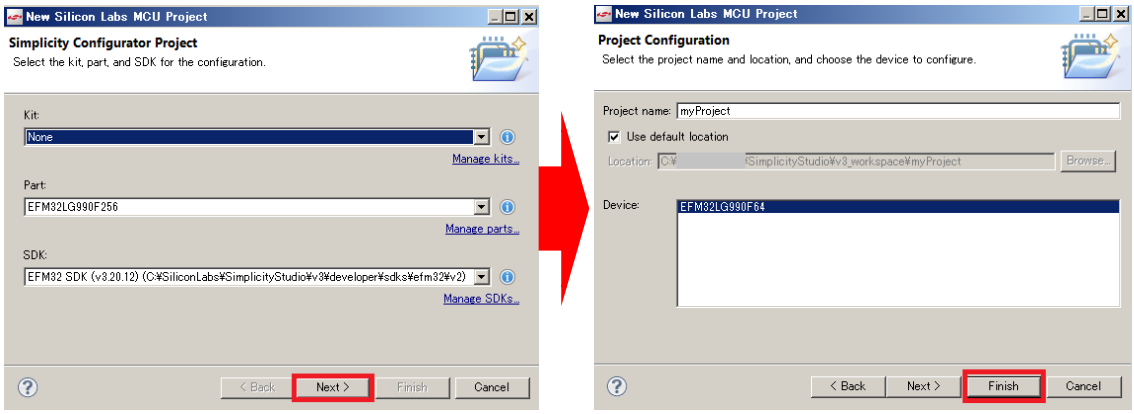
6-4 ピン設定やペリフェラル設定をしてみる （Configurator）

Simplicity Studio では、ピン設定やペリフェラル設定を直感的に行うことができるツールを用意しています。ここでは設定ツール（Configurator）の使用方法をご紹介します。

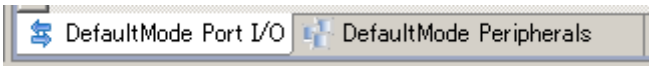
Configurator をクリックします。



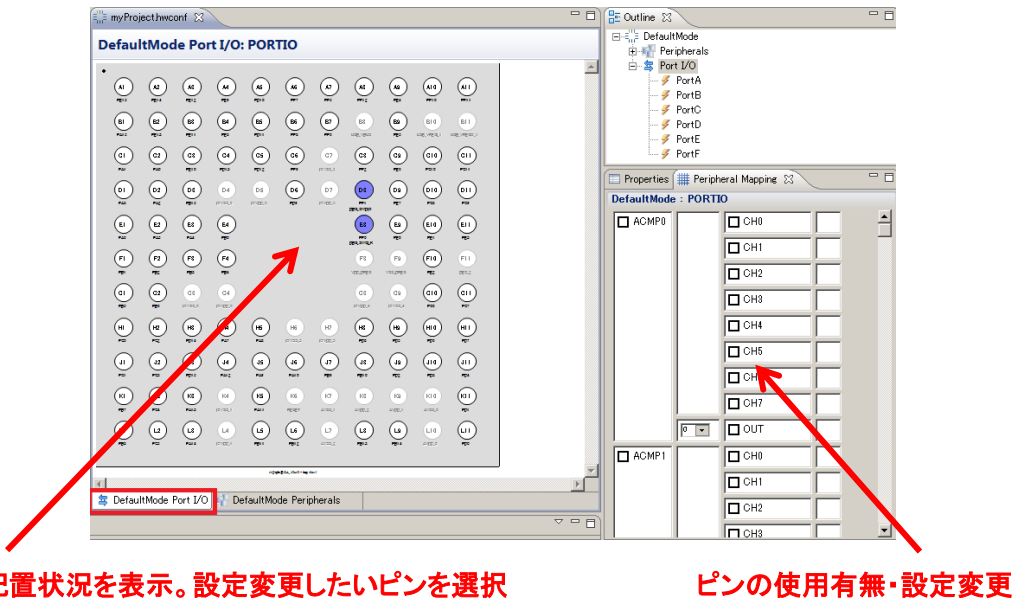
Configurator を使用するために、プロジェクトの作成が求められます。Kit で None を選択し、Part で使用する MCU を選択し、Project name にプロジェクト名を入力してプロジェクトを作成します。



Configurator が起動します。画面中央に 2 つのタブがあり、Default Mode Port I/O でピン設定を、Default Mode Peripherals でペリフェラル設定を行うことができます。

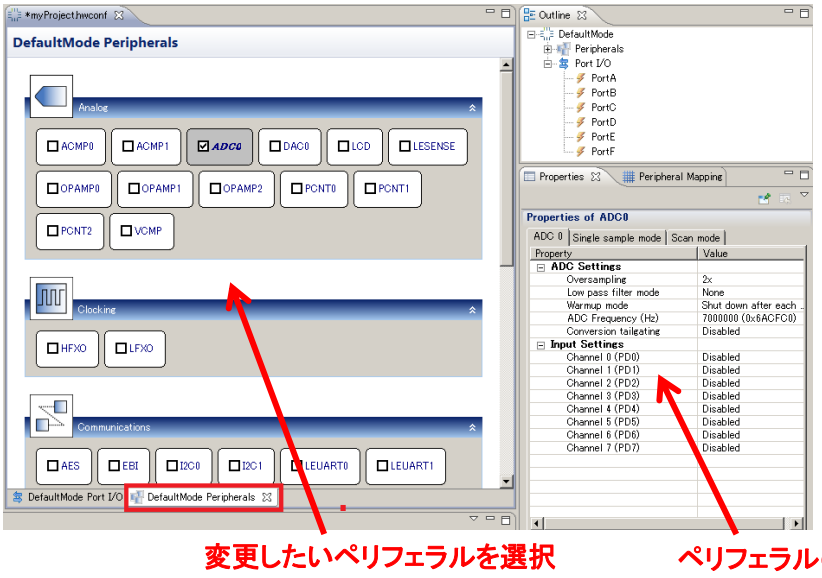


◆ Default Mode Port I/O



ピン配置状況を表示。設定変更したいピンを選択　　ピンの使用有無・設定変更

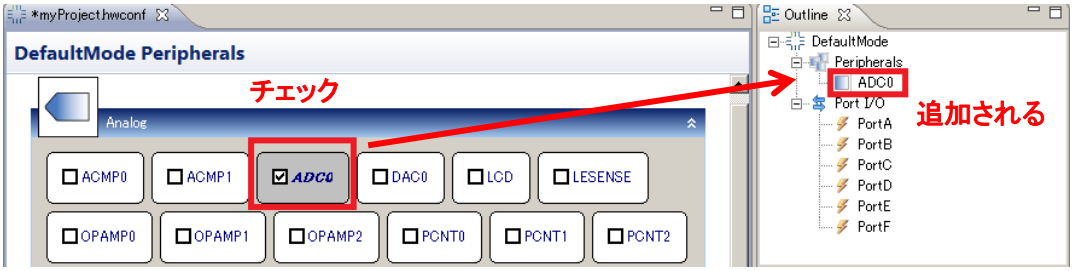
◆ Default Mode Peripheral



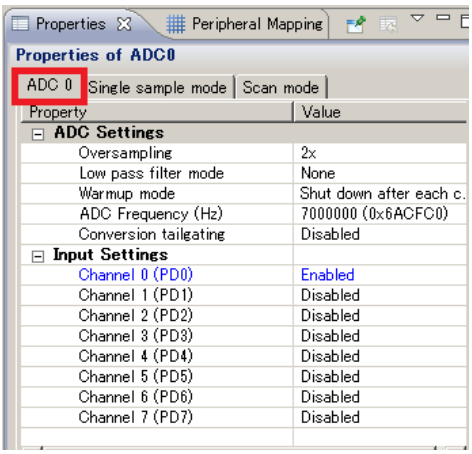
◆ ADC0 を設定してみましょう

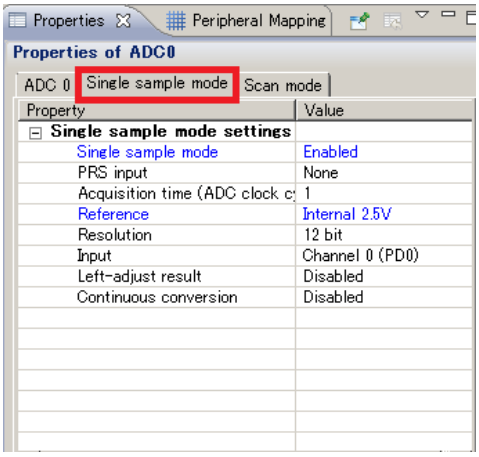
ADC0 を有効にし、ADC0 の動作モードを設定し、ピン設定を行う、という流れになります。

ADC0 を有効にするには、Default Mode Peripherals に切り替えて、ADC0 にチェックを入れます。Outline ウィンドウに ADC0 が追加されます。



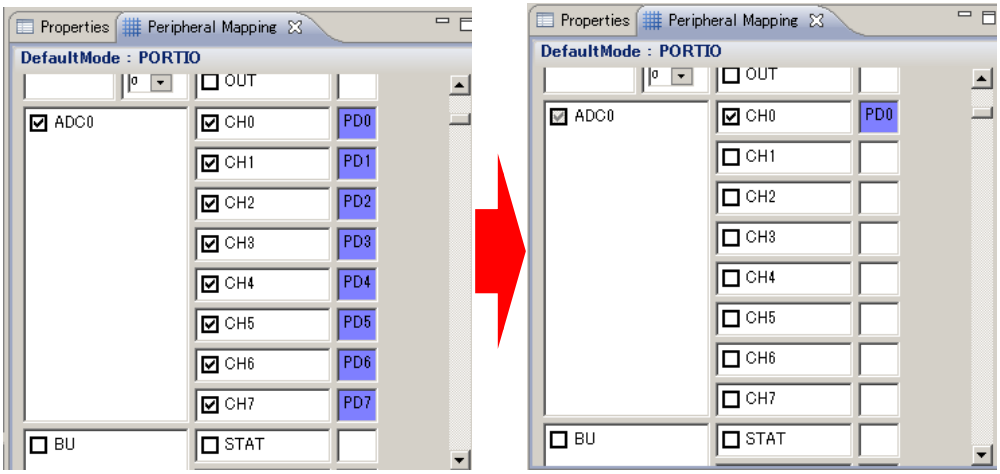
ADC0 の設定を変更するには、Outline に表示された ADC0 を選択します。Properties ウィンドウに ADC0 の設定画面が表示されますので、動作設定を行います。Single sample mode にして、リファレンス電圧は 2.5V に、ADC の入力には Channel0（L11 ピン）を使用する設定にしてみます。



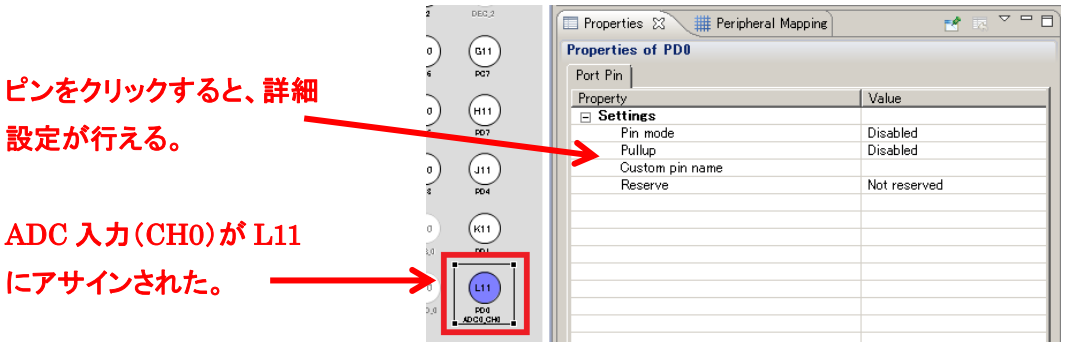


続いてピン設定を行います。Default Mode Port I/O ウィンドウに切り替え、Outline に表示された Port I/O を選択します。Peripheral Mapping ウィンドウにペリフェラル一覧が表示されますので、そこから

ADC0 を探して、ピン設定を ADC0 の入力設定に合わせます。(つまり CH0 だけを有効にする)



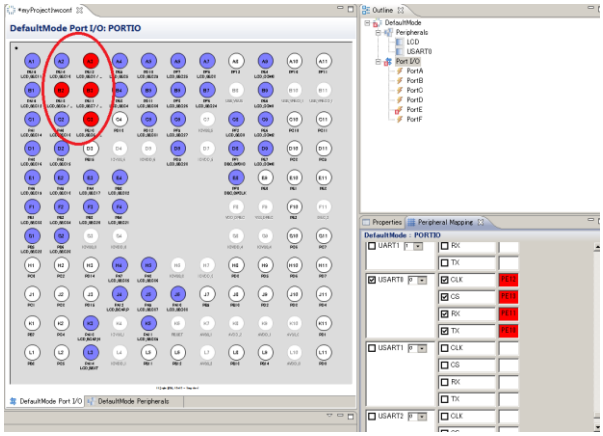
設定が完了すると、Default Mode Port I/O ウィンドウに、ADC0 入力(L11 ピン)の設定が反映されました。使用中のピンは紫色になります。ピンをクリックすると、そのピンの設定(入出力、プルアップの有無など)も設定が行えます。



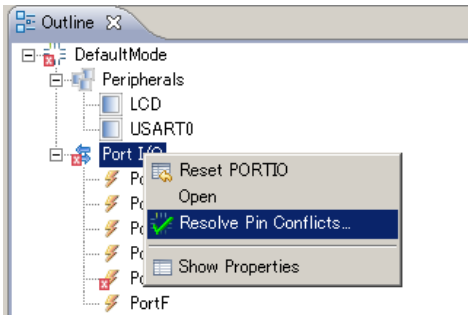
◆ ピン配置の矛盾を解決(Resolve Pin Conflicts)

ペリフェラルごとに使用可能なピンはある程度決まっています。使用するペリフェラルが増えてくると、ピン配置が競合してしまい、うまくピン配置が行えない場合が出てきます。それを解消してくれるのが Resolve Pin Conflicts 機能です。競合が生じないピン配置の組み合わせを探してくれます。

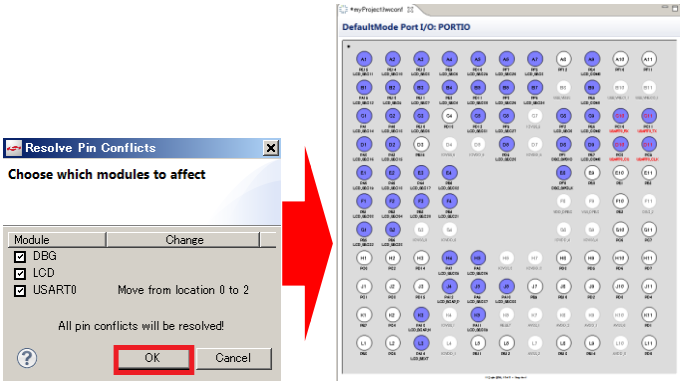
機能紹介のために、ピン配置が競合させてみます。LCD と USART0 をイネーブルにしてみてください。ピン 4 本分の設定が競合してしまい、ペリフェラル設定とピン設定が、競合状況を示す赤色になります。



このような場合、Resolve Pin Conflict 機能を使います。Outline の Port IO を選択して（或いは Default Mode I/O ウィンドウの上で）右クリックし、Resolve Pin Conflicts を選択します。

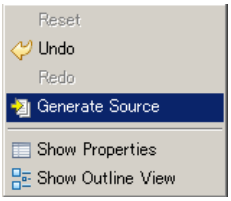


解決案を提示してくれますので、OK をクリックすれば問題が生じないピン配置に割り当てなおしてくれます。競合を示す赤色のピンがなくなり、問題が解消されました。



◆　コードに反映させる（Generate Source）

設定が完了したら、Default Mode I/O ウィンドウか、Default Mode Peripheral ウィンドウの上で右クリックし、Generate Source を選択します。ペリフェラル設定、ピン設定を含んだ C コードが生成されます。



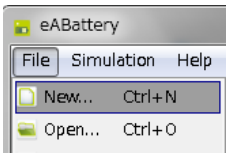
6-5 電池寿命の見積もりをしてみる （energy Aware Battery）

バッテリー・アプリケーションにおいて、電池寿命の見積もりは非常に重要です。しかし MCU には幾つもの消費電力モードがあり、多数のペリフェラルがあり、電池寿命の見積もりを複雑にしています。Simplicity Studio には、電池寿命を簡単に見積もることが可能なツールを用意しています。ここでは、電池寿命の見積もりツール（energy Aware Battery）の使用方法を紹介します。

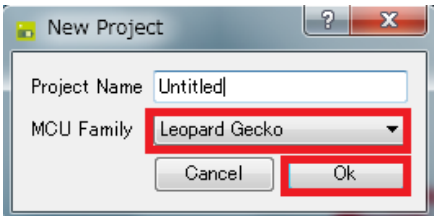
電池寿命を見積もるにあたって、MCU の動作を特定する必要があります。ここでは以下の動作を例に取って、電池寿命を見積もっていきます。通常動作（EM0）で 10ms の作業をし、Idle（EM2）に移行して 1s 待機し、という動作を繰り返し行う、という設定です。

energy Aware Battery をクリックします。

File メニューから New を選択します。



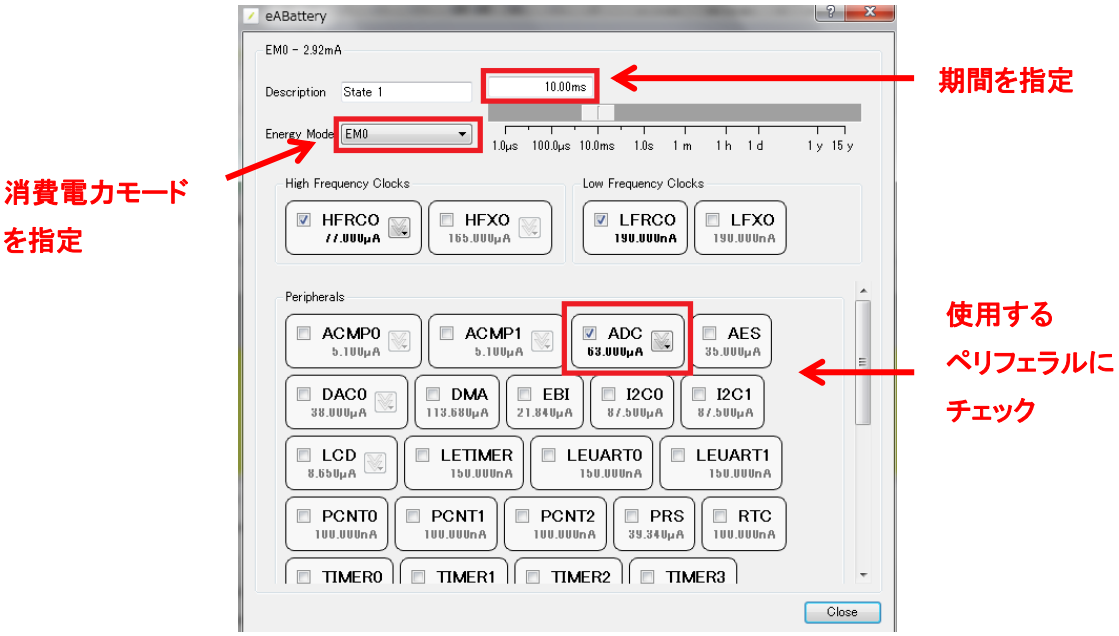
シミュレーションを行いたいファミリ名を選択し、OK をクリックします。



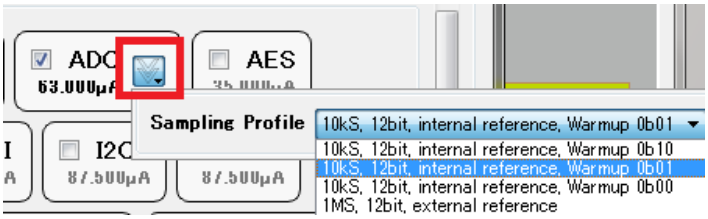
◆ EM0 の期間（ADC 有効、10ms）を登録しましょう

Add State をクリックします。

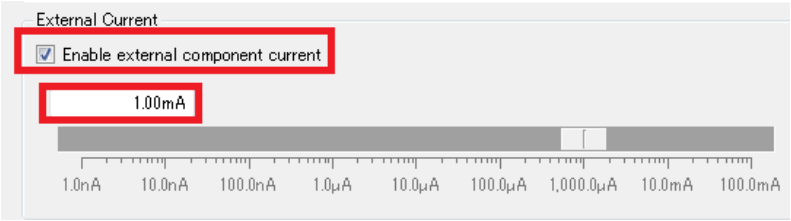
Energy Mode を EM0 に設定し、期間を 10ms とします。ペリフェラルは ADC を有効にしますので、ADC にチェックを付けます。設定が終わったら、Close をクリックします。



なお、ADC の横のボタンをクリックすると、動作モードをより詳細に指定することが可能です。



また、MCU だけではなく、外部部品での消費電流も含めてシミュレーションを行いたい場合には、External Current（ペリフェラルを下にスクロールしていくとあります）に数値を入力してください。



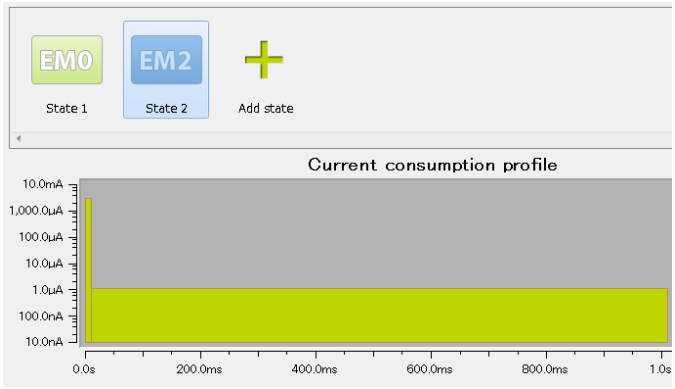
◆　EM2 の期間（RTC 有効、1s）を登録しましょう

EM0 の時と同じ作業を、EM2 についても行います。

Add State をクリックします。

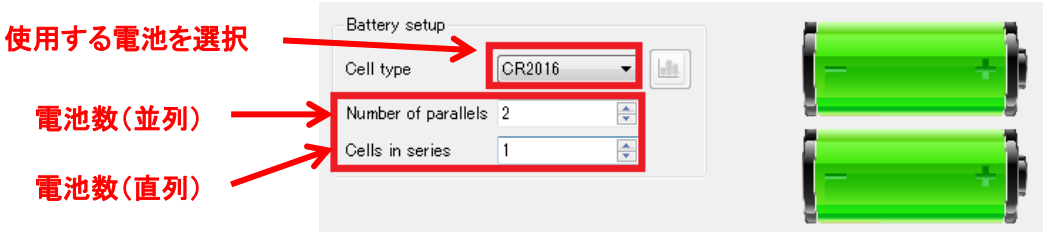
Energy Mode を EM2 に設定し、期間を 1s とします。ペリフェラルは RTC を有効にしますので、RTC にチェックを付けます。設定が終わったら、Close をクリックします。

登録が完了すると、次図のように表示されます。

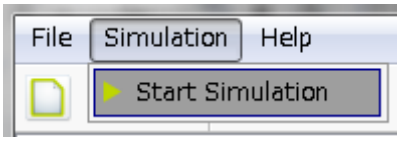


◆　シミュレーションを行いましょう

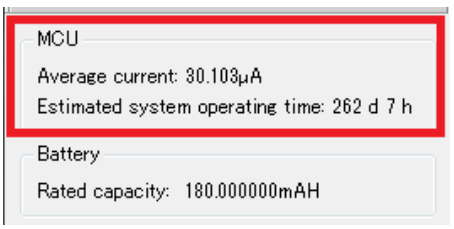
使用する電池を決めます。今回は、CR2016 を並列に 2 個繋いだ場合を考えます。Cell type で CR2016 を選択し、Number of parallels（並列の電池数）を 2、Cells in series（直列の電池数）を 1 にします。

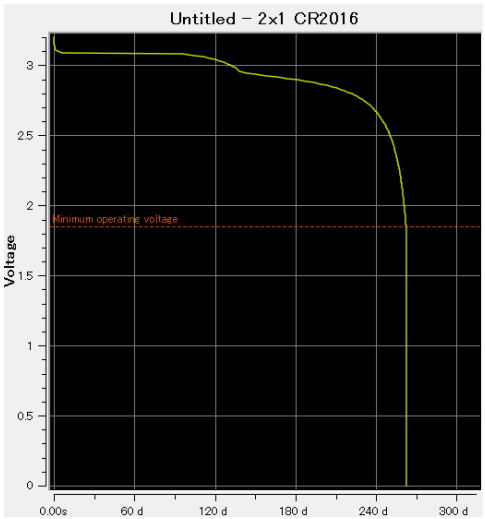


Simulation メニューから、Start Simulation を選択します。



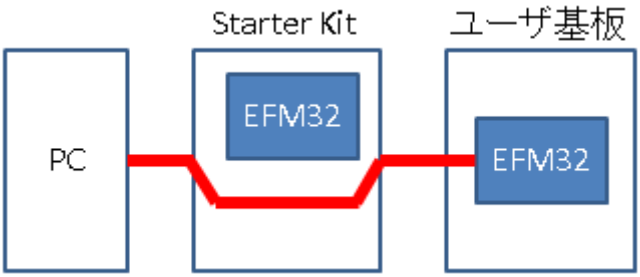
シミュレーションが実行され、この使用条件の場合には電池寿命が 260 日程度であることが算出できました。





6-6 ユーザ基板のプログラミング・デバッグを行ってみる

Starter Kit を使用することで、ユーザ基板上の EFM32 に対して、プログラミング或いはデバッグを行うことが可能です。その手順について紹介します。



6-6-1 参考資料

・AN0042: EFM32 Debug and Trace

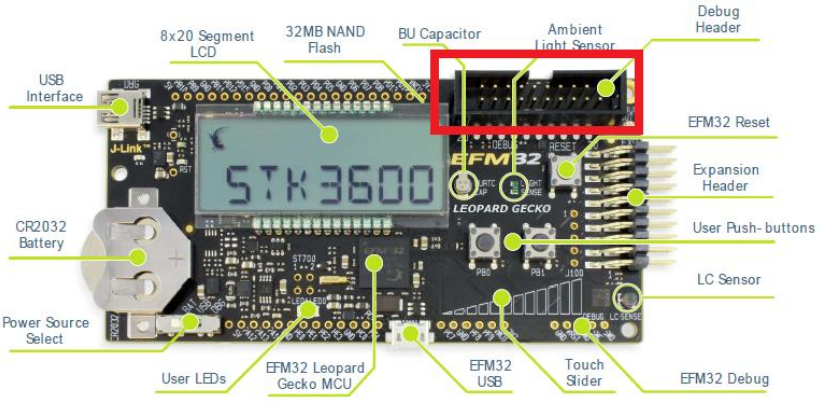
・AN0062: Programming Internal Flash Over the Serial Wire Debug Interface

・各ファミリの Starter Kit ユーザガイド

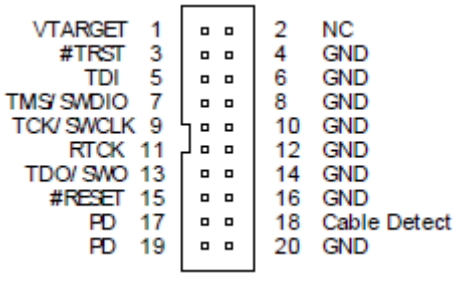
http://community.silabs.com/t5/32-bit-MCU-Knowledge-Base/Using-an-EFM32-Starter-Kit-as-an-external-debugger/ta-p/124194

6-6-2 ハードウェア接続

Starter Kit の Debug Connector を介して、ユーザ基板に接続します。Debug Connector は、Starter Kit の右上にある 20 ピンのコネクタです。



Debug Connector のピン配置情報は、Starter Kit ユーザガイドに記載されています。下図は、EFM32LG-STK3600 ユーザガイドからの抜粋です。このうち、SWCLK、SWDIO、SWO、#RESET、VTARGET、GND の計 6 ピンを使用します。



SWCLK	EFM32（ユーザ基板上）の該当ピンに接続してください。直結で結構です。
SWDIO	EFM32（ユーザ基板上）の該当ピンに接続してください。直結で結構です。
SWO	EFM32（ユーザ基板上）の該当ピンに接続してください。直結で結構です。
#RESET	EFM32（ユーザ基板上）の該当ピンに接続してください。直結で結構です。
VTARGET	EFM32（ユーザ基板上）への供給電源に接続してください。Starter Kit とユーザ基板の信号レベルを合せるために使用します。接続し忘れると、Starter Kit からユーザ基板を認識できませんので、ご注意ください。
GND	Starter Kit の GND と、ユーザ基板の GND を接続してください。

6-6-3 ハードウェア接続 （例）

例として、EFM32LG990 の場合を取り上げて説明します。

まず、データシートの Alternate Functionality Pinout で、SWCLK、SWDIO、SWO がどの Pin Name（ピン名称）に割り当てられているかを確認します。下図のとおり、SWCLK は PF0、SWDIO は PF1 に割り当てられています。SWO は PF2/PD1/PD2 のいずれかに割り当てが可能となっていますが、デフォルトは Location0 の PF2 になっています。

Alternate	LOCATION							
Functionality	0	1	2	3	4	5	6	Description
DBG_SWCLK	PF0	PF0	PF0	PF0				Debug-interface Serial Wire clock input. Note that this function is enabled to pin out of reset, and has a built-in pull down.
DBG_SWDIO	PF1	PF1	PF1	PF1				Debug-interface Serial Wire data input / output. Note that this function is enabled to pin out of reset, and has a built-in pull up.
DBG_SWO	PF2		PD1	PD2				Debug-interface Serial Wire viewer Output. Note that this function is not enabled after reset, and must be enabled by software to be used.

次に、データシートの Pinout で、どの Pin Name（ピン名称）が、どの Pin Number（ピン番号）に割り振られているかを確認します。

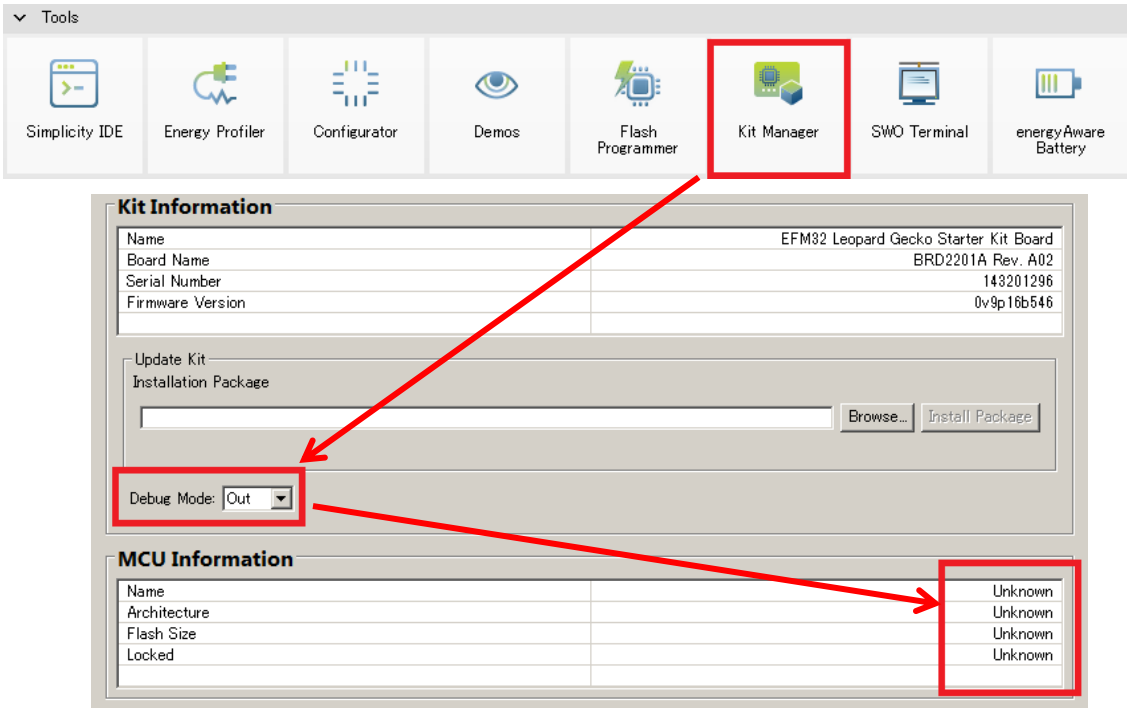
PF0（SWCLK）は E8 ピン、PF1（SWDIO）は D8 ピン、PF2（SWO#0）は C8 ピン、#RESET は K6 ピンに割り当てられていることが判ります。

BGA112 Pin# and Name		Pin Alternate Functionality / Description				
Pin #	Pin Name	Analog	EBI	Timers	Communication	Other
E8	PF0			TIM0_CC0 #5 LETIM0_OUT0 #2	US1_CLK #2 LEU0_TX #3 I2C0_SDA #5	DBG_SWCLK #0/1/2/3
D8	PF1			TIM0_CC1 #5 LETIM0_OUT1 #2	US1_CS #2 LEU0_RX #3 I2C0_SCL #5	DBG_SWDIO #0/1/2/3 GPIO_EM4WU3
C8	PF2	LCD_SEG0	EBI_ARDY #0/1/2	TIM0_CC2 #5	LEU0_TX #4	ACMP1_O #0 DBG_SWO #0 GPIO_EM4WU4
K6	RESETn	Reset input, active low. To apply an external reset source to this pin, it is required to only drive this pin low during reset, and let the internal pull-up ensure that reset is released.				

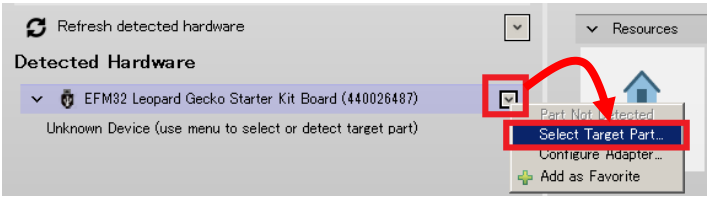
上記を見ると、デバッグピンには、TIM_CC0#5 や US1_CLK#2 など、他の機能を割り当てることが可能になっていることが判ります。デフォルトでは、デバッグ用のピンに設定されていますので、ユーザプログラムから設定変更を行うことになります。しかし、特別な理由がない限り、他の機能は割り当てずに、デバッグ専用ピンとしてご使用になることを強くお勧めします。いったん設定変更を行うと、デバッグ回路にアクセスできなくなりますので、プログラムの書き替えも行えなくなる可能性があります。

6-6-4 デバッグ対象の切り替え （Kit Manager）

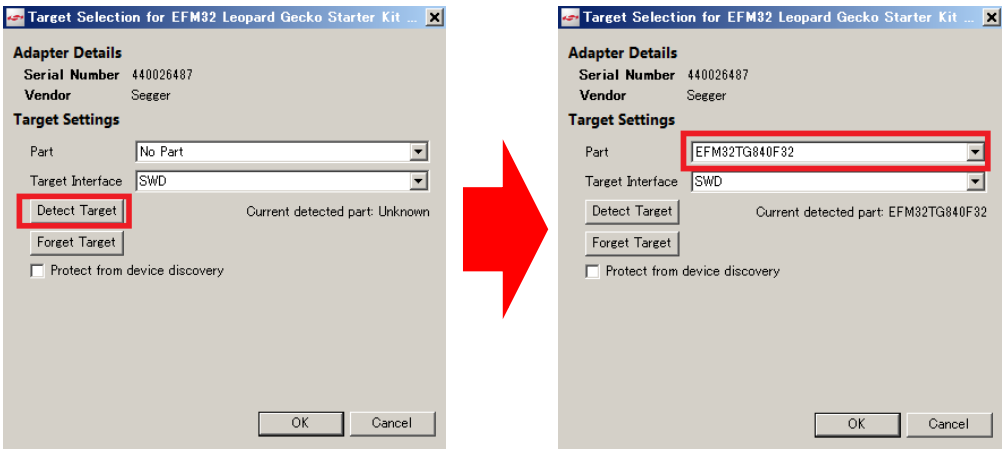
Kit Manager の Debug Mode 選択メニューにて、Out を選択してください。MCU Information が Unknown に変わります。



次に、Simplicity Studio の Top 画面に戻り、画面左下の Detected Hardware に表示されたスターターキットの横のボタンを押し、Select Target Part を選択します。



Target Selection ウィンドウが表示されますので、Detect Target ボタンを押します。ユーザ基板上の MCU が認識されました。あとは、Starter Kit 上の EFM32 と同様に使用できます。

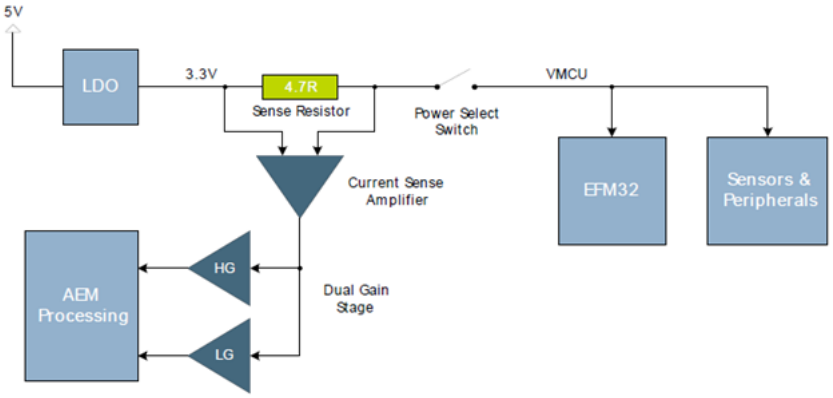


6-7 ユーザ基板の消費電流を測定する

ユーザ基板の消費電流を測定するためには、次の 2 つの処理を行う必要があります。

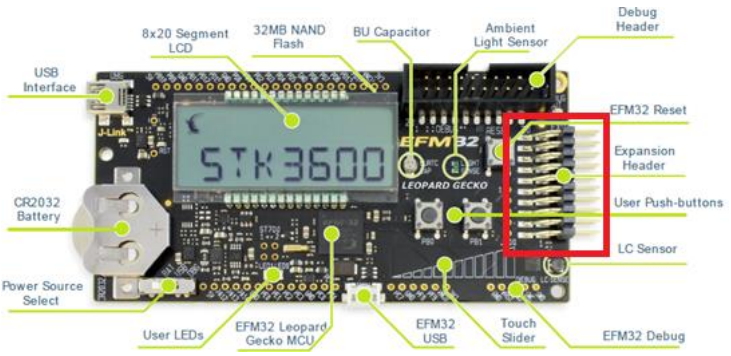
- ユーザ基板には、Starter Kit の VMCU ピンから給電を行う
- Starter Kit 上の部品（EFM32, その他周辺部品）の消費電流を極限まで下げる

Starter Kit には、電流モニタ用の回路が実装されています。下図は STK3600（Leopard Gecko）のユーザガイドからの抜粋ですが、Debug 用 USB から 5V 給電を受け、LDO で 3.3V を作り、それを VMCU として Starter Kit 上の EFM32 やセンサ、ペリフェラルに給電しています。Energy Profiler は VMCU の消費電流をモニタするようになっています。



6-7-1 VMCU をどこから取るか

VMCU は Starter Kit の Expansion Header から出ています。Expansion Header は、Starter Kit の右側にある 20 ピンのコネクタです。



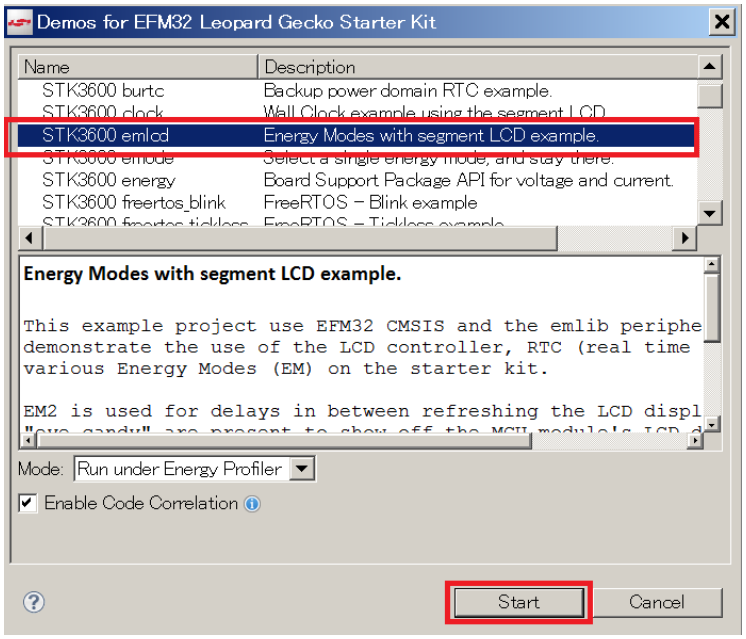
Expansion Header のピン配置情報は、Starter Kit ユーザガイドに記載されています。下図は、EFM32LG-STK3600 ユーザガイドからの抜粋です。2 ピンに VMCU が配置されています。

GND	1	2	VMCU
PC0	3	4	PD0
PC3	5	6	PD1
PC4	7	8	PD2
PC5	9	10	PD3
PB11	11	12	PD4
PB12	13	14	PD5
PC6	15	16	PD6
PD7	17	18	5V
GND	19	20	3V3

6-7-2 Starter Kit 上の部品の消費電流を極限まで下げる

Starter Kit 上の EFM32 は、EM4（消費電流がもっとも低い電力モード）に移行することで消費電流を極限まで下げることができます。また EFM32 が EM4 にいる間は、センサやペリフェラルはほとんど電流を消費しません。ですから EFM32 を EM4 にすれば良い事になりますが、電力モードを切り替えるデモコードがありますので、それを利用します。デモコードのダウンロード手順は以下の通りです。

1. Starter Kit を PC に接続し、Detect Connected Device で Starter Kit を認識させます。ユーザ基板は接続しない状態で行ってください。もし Starter Kit を認識しない場合には、Kit Manager の Debug Mode が MCU になっているか確認してください。
2. Energy Profiler を起動します。
3. Run メニューから Profile Demo を選択します。実行可能なデモの一覧が表示されますので、STK3600 emlcd を選択し、Start を押します。Energy Profiler に電流波形が表示されればダウンロード完了です。



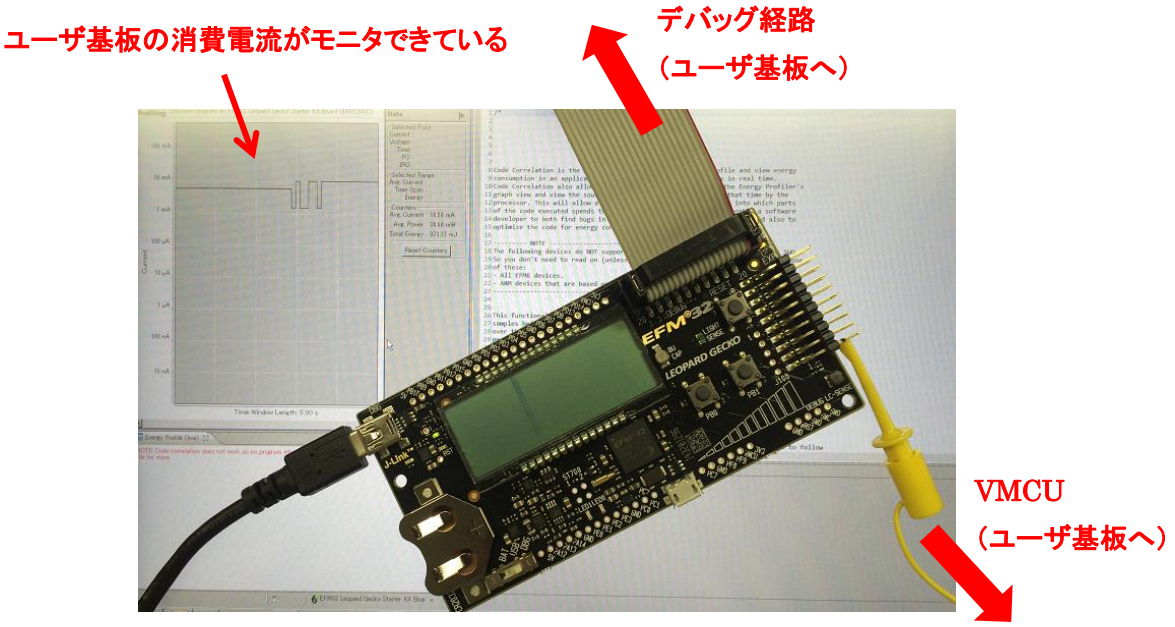
このデモは、ボタンを押すと電力モードを順々に切り替えていきます。STK3600 の場合は、PB0 を押すと EM0⇒EM1⇒EM2⇒EM3 の順に切り替えていき、PB1 を押すと EM0⇒EM1⇒EM2⇒EM4 の順に切り替えていきます。今回は EM4 にしたいですので PB1 を押すことになります。LCD に動作説明が表示されますので、そちらも参考にしてください。

試しに PB1 を押して EM4 にしましょう。消費電流が極限まで下げられたことが確認できます。（下図で、測定結果がマイナスになっているのは読み取り誤差です）

6-7-3 ユーザ基板の消費電流の測定手順

手順は以下の通りです。

1. 「6-7-1 VMCU をどこから取るか」を参考に、VMCU をユーザ基板に接続します。
2. 「6-6 ユーザ基板のプログラミング・デバッグを行ってみる」を参考に、デバッグ経路をユーザ基板に接続し、Kit Manager の Debug Mode を Out に変更します。
3. 「6-7-2 Starter Kit 上の部品の消費電流を極限まで下げる」を参考に、PB1 を押して Starter Kit 上の EFM32 を EM4 に移行させます。
4. （必要であれば、ユーザ基板のプログラムの書き換えなどを行います。）
5. Energy Profiler を起動し、Run メニューから Profile Running Program を選択します。ユーザ基板の消費電流がモニタできます。



7 ソフトウェア設計

ソフトウェア設計に役立つ情報をご紹介します。

7-1 Cortex-M を初めて使う方に （ARM 社ドキュメント）

ARM 社が、Cortex-M そのものに関する日本語ドキュメントを提供しております。ARM 社の Web Site からご入手頂くことができます。

◆ Cortex-M4 テクニカル・リファレンスマニュアル
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439cj/DDI0439CJ_cortex_m4_r0p1_trm_ja-JP.pdf

◆ Cortex-M3 テクニカル・リファレンスマニュアル
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337ij/DDI0337IJ_cortexm3_r2p1_trm_japanese.pdf

◆ Cortex-M0+ テクニカル・リファレンスマニュアル
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0484cj/DDI0484CJ_cortex_m0p_r0p0_trm_JP.pdf

◆ Cortex-M0+ デバイス・ジェネリックユーザ・ガイド
http://infocenter.arm.com/help/topic/com.arm.doc.dui0662bj/DUI0662BJ_cortex_m0p_r0p1_dgug_JP.pdf

また、役立ちそうな日本語アプリケーションノートもご紹介します。

◆ Cortex-M3 組み込みソフトウェア開発（アプリケーションノート 179）
http://infocenter.arm.com/help/topic/com.arm.doc.dai0179bj/DAI0179BJ.pdf

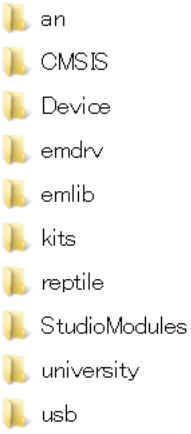
◆ PIC マイコンから Cortex-M3 への置き換え（アプリケーションノート 234）
http://infocenter.arm.com/help/topic/com.arm.doc.dai0234a/DAI0234A_migrating_from_pic_to_m3.pdf

◆ 8051 マイコンから Cortex-M への置き換え（アプリケーションノート 237）
http://infocenter.arm.com/help/topic/com.arm.doc.dai0237a/DAI0237A_migrating_from_8051_to_Cortex_M.pdf

7-2 開発用のソースコードについて

ソースコード一式は、下記フォルダにインストールされています。

¥¥SiliconLabs¥SimplicityStudio¥v2¥developer¥sdks¥efm32¥v2

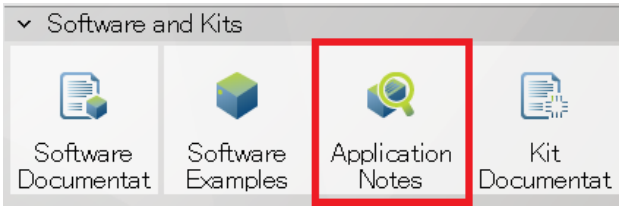


◆ アプリケーションノート（¥an フォルダ）

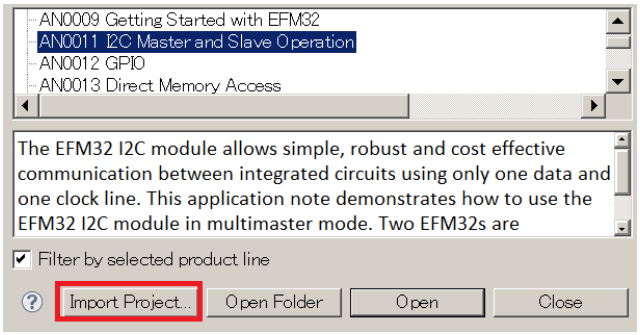
EFM32 を使用した応用例（各種ペリフェラル、割り込み、Bootloader 等）を提供しています。

用意されたソースコードを、簡単に Simplicity Studio から呼び出す事ができますので、その手順を紹介します。

Simplicity Studio を起動し、Product の欄に使用する製品型番を入力して、Application Notes をクリックします。



使用したいアプリケーションノートを選択し、Import Project ボタンを押すと、Simplicity Studio 用のプロジェクトが生成されます。



◆ EFM32 ライブラリ（¥emlib フォルダ）

EFM32 で共通に持つ各種機能の API を提供しています。UART, I2C, タイマ, RTC, GPIO などが用意されています。ヘッダおよびソースコードは、それぞれ下記フォルダに保存されています。

ヘッダ： ¥emlib¥inc

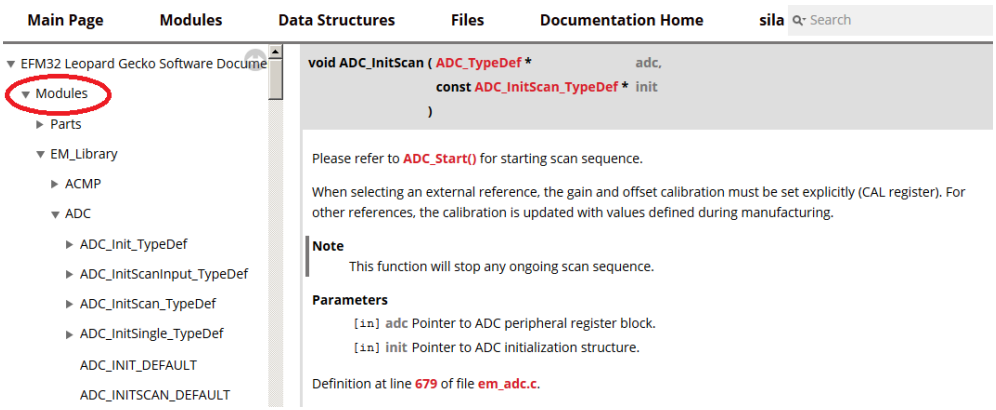
ソースコード： ¥emlib¥src

◆ EFM32 ドライバ（¥emdrv フォルダ）

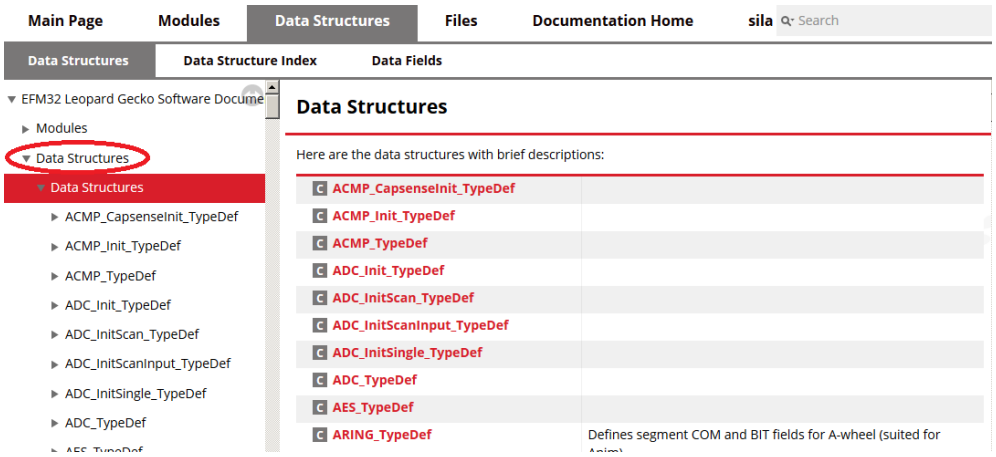
EFM32 ライブラリ以外の機能の API を提供しています。内蔵 Flash、スリープモード、SPI などが用意されています。

API の詳細情報については、Software Documentation（3-3 EFM32 の API 情報）から入手頂けます。
（オンラインでのみご使用頂けます）

EM_Library にて、API について確認頂けます。各関数の引数が一覧できます。



Data Structures にて、データ構造について確認頂けます。



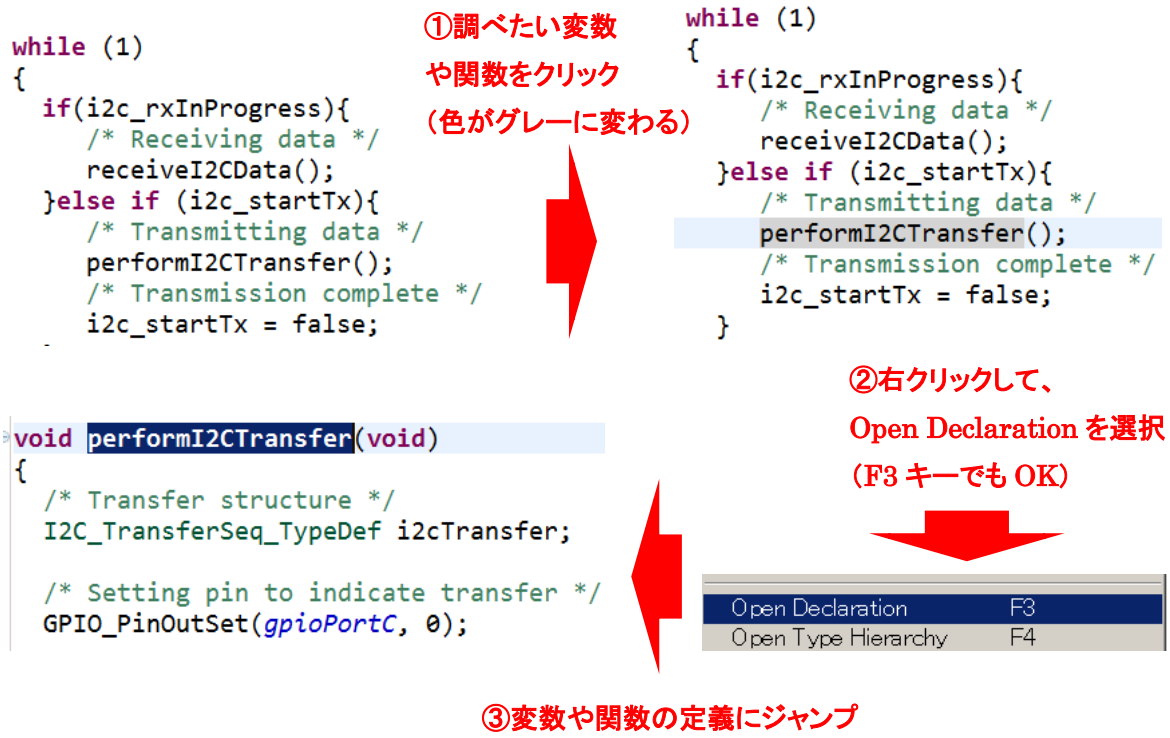
◆　キット別サンプルアプリケーション（¥kits フォルダ）

EFM32 ライブラリ/ドライバを使った簡易アプリを提供しています。

7-3 ソースコードの追い方

Simplicity IDE でソースコードを追うための方法を紹介します。

◆ 変数や関数を定義している記述を探す



7-4 割り込みハンドラ

EFM32 の割り込みコントローラは、Cortex-M 系コアに内包される NVIC いうブロックに含まれており、市場にある多くの Cortex-M コア搭載品と動作に差がありません。

EFM32 のリファレンスマニュアルの NVIC の関連事項のほか、ARM 社のドキュメント内の NVIC の記述が参照頂けます。

参考） 日本語の Web サイトとしては、APS ACADEMY の Cortex-M 入門編 第 13 回で、非常に判りやすく説明されています。「APS 例外/割り込み処理」で検索してみてください。

7-5 ピン設定、ペリフェラル設定の流れ

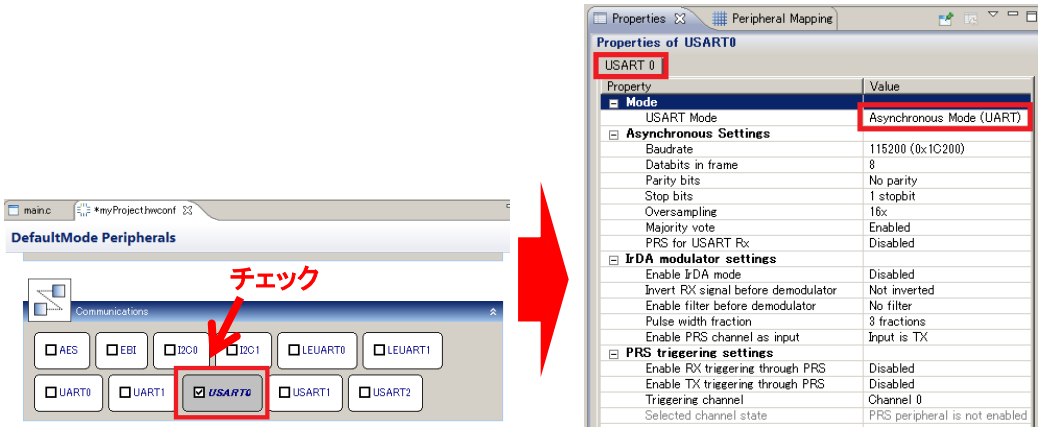
ピン設定は、Configurator を使用して行うのが簡単です。1 つのピンを複数のペリフェラルで共有している場合も多いですので、競合が生じないような設定を、直感的に行うことができます。ソースコードも自動生成されます。

ペリフェラル設定は、リファレンスマニュアルとアプリケーションノート（およびソースコード）を参考に、行って頂くことになります。

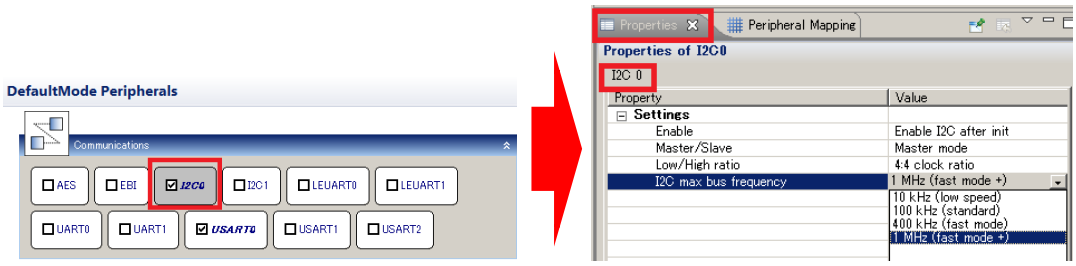
7-6 ピン設定

EFM32LG990 を使用し、①USART0 (Asynchronous mode=UART)、②I2C0、③GPIO1 本を使用した場合の設定手順をご紹介します。もちろんゼロからコードを書くこともできますが、Configurator を使用するのが簡単です。

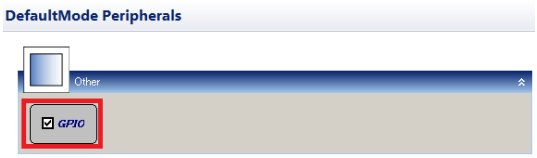
1. Configurator をクリックし、「6-4 ピン設定やペリフェラル設定してみる」と同じ手順でプロジェクトを作成します。
2. Configurator が起動したら、Default Mode Peripherals でペリフェラル設定をしていきます。まず①USART0 の設定を行います。USART0 にチェックを入れ、Outline に表示された USART0 を選択し、Properties ウィンドウにて USART0 の設定を行います。ここでボーレートなどの設定を行うことができます。



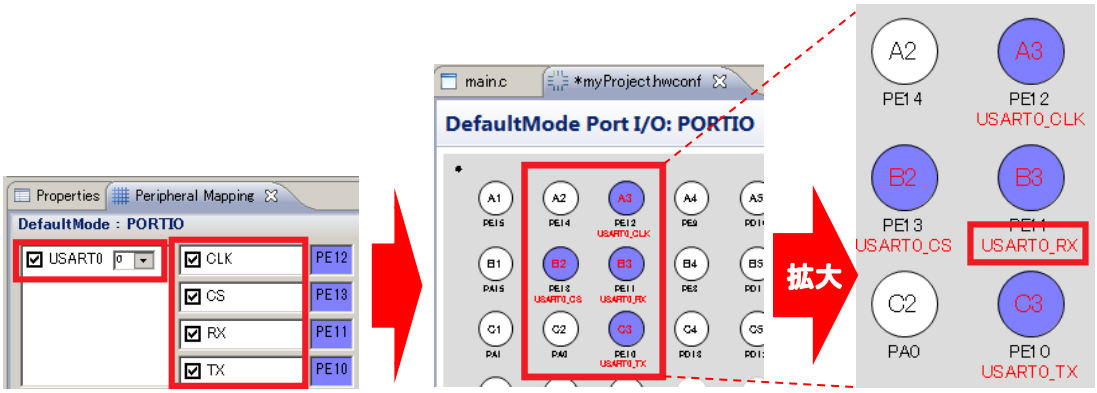
3. ②I2C0 の設定を行います。Default Mode Peripherals で I2C0 にチェックを入れ、Outline に表示された I2C0 を選択し、Properties ウィンドウにて I2C0 の設定を行います。



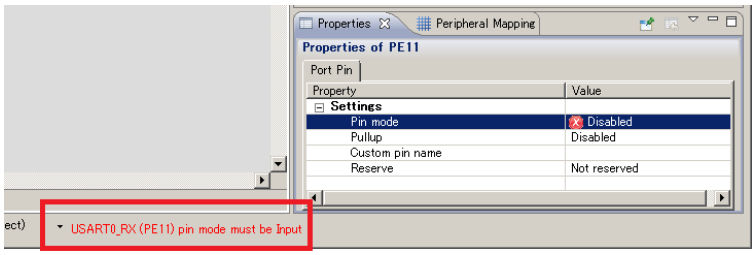
4. ③GPIO の設定を行います。Default Mode Peripherals で GPIO にチェックを入れます。



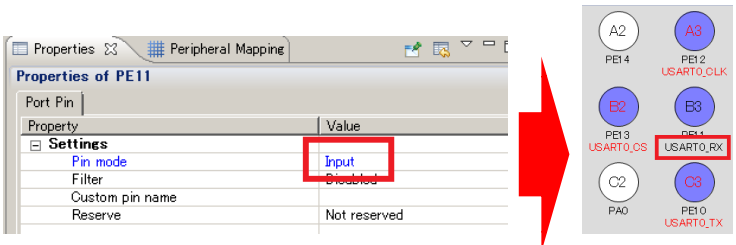
5. Default Mode Port IO に移動して、①USART0 からピン設定を行っていきます。Outline で Port IO を選択し、Peripheral Mapping ウィンドウで USART0 にチェックし、使用するピン(CLK, CS, RX, TX)にチェックします。すると、Default Mode Port IO にピン配置が反映されます。次図の場合では、パッケージ左上の 4 本ピンに USART0 ピンが割り振られました。よく見ると、ピン下の信号名が赤字で表示されています。これはピン設定がまだ完了していないことを示しています。



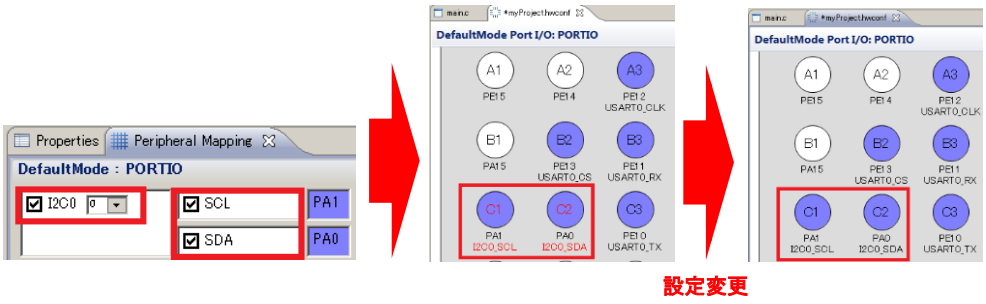
6. ピン設定を行っていきます。例として USART0_RX ピンをダブルクリックします。Property ウィンドウにピンの設定画面が表示されますが、Pin mode に  の表示があります。これが修正すべき項目になります。Pin mode を選択して黒く反転させると、画面下に赤字で、どのように設定変更すれば良いか？のアドバイスが表示されます。



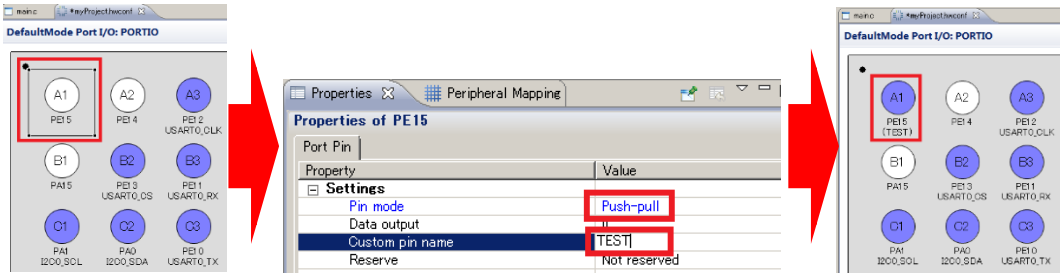
USART0_RX の Pin mode を Input にするように指示されていますので、その通りに設定すると、Default Mode Port IO の信号名（USART0_RX）が赤字から黒字に変わりました。同じ要領で、他の 3 つのピンの Pin mode を push-pull に変更します。



7. ②I2C0 のピン設定を行います。Outline で Port IO を選択し、Peripheral Mapping ウィンドウで I2C0 にチェックし、使用するピン（SCL, SDA）にチェックします。Default Mode Port IO のピン下の信号名を見ると赤字で表示されていますので、まだ設定が完了していません。USART0 と同じ要領で、2 つのピンの Pin mode を Wired-and pullup filter に変更します。

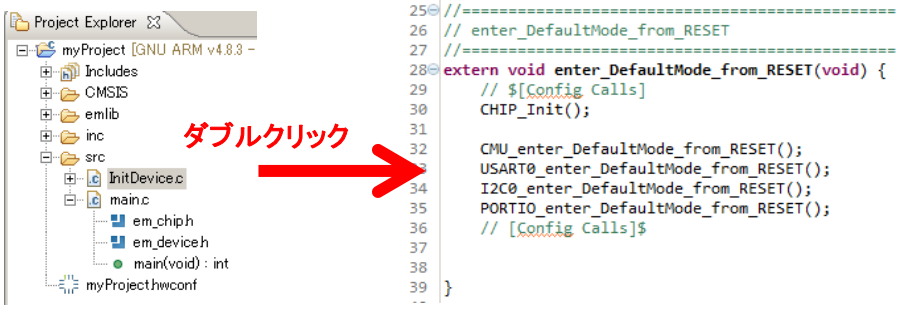


8. ③GPIO のピン設定を行います。今回は、A1(PE15)に push pull 出力を割り当ててみます。Default Mode Port IO にて、A1(PE15)をダブルクリックします。Property ウィンドウにピンの設定画面で Pin mode を Push-pull に変更し、Custom pin name に信号名（下図では TEST）を入力します。Default Mode Port IO に反映されました。



9. 設定が完了したら、Default Mode I/O ウィンドウか、Default Mode Peripheral ウィンドウの上で右クリックし、Generate Source を選択します。ペリフェラル設定、ピン設定を含んだ C コードが生成されます。

左下図はプロジェクトツリーですが、InitDevice.c に行った設定が反映されています。Enter_DefaultMode_from_RESET という関数の中で、USART0, I2C0, PortI/O の初期化関数を呼んでいます。それぞれの初期化関数も参考のために掲載します。




```
25 //=============================================
26 // enter_DefaultMode_from_RESET
27 //=============================================
28 extern void enter_DefaultMode_from_RESET(void) {
29     // $[Config Calls]
30     CHIP_Init();
31
32     CMU_enter_DefaultMode_from_RESET();
33     USART0_enter_DefaultMode_from_RESET();
34     I2C0_enter_DefaultMode_from_RESET();
35     PORTIO_enter_DefaultMode_from_RESET();
36     // [Config Calls]$
37
38
39 }
```

```
221 //=============================================================================
222 // USART0_enter_DefaultMode_from_RESET
223 //=============================================================================
224 extern void USART0_enter_DefaultMode_from_RESET(void) {
225     // $[USART_InitAsync]
226     USART_InitAsync_TypeDef initasync = USART_INITASYNC_DEFAULT;
227
228     initasync.baudrate              = 115200;
229     initasync.databits              = usartDatabits8;
230     initasync.parity                = usartNoParity;
231     initasync.stopbits              = usartStopbits1;
232     initasync.oversampling          = usartOVS16;
233     #if defined( USART_INPUT_RXPRS ) && defined( USART_CTRL_MVDIS )
234     initasync.mvdis                 = 0;
235     initasync.prsRxEnable           = 0;
236     initasync.prsRxCh               = 0;
237     #endif
238
239     USART_InitAsync(USART0, &initasync);
240     // [USART_InitAsync]$
241
242     // $[USART_InitSync]
243     // [USART_InitSync]$
244
245     // $[USART_InitPrsTrigger]
246     USART_PrsTriggerInit_TypeDef initprs = USART_INITPRSTRIGGER_DEFAULT;
247
248     initprs.rxTriggerEnable         = 0;
249     initprs.txTriggerEnable         = 0;
250     initprs.prsTriggerChannel       = usartPrsTriggerCh0;
251
252     USART_InitPrsTrigger(USART0, &initprs);
253     // [USART_InitPrsTrigger]$
254
255
256 }
```

```
362 //==========================================================================
363 // I2C0_enter_DefaultMode_from_RESET
364 //==========================================================================
365 extern void I2C0_enter_DefaultMode_from_RESET(void) {
366     // $[I2C0 initialization]
367     I2C_Init_TypeDef init = I2C_INIT_DEFAULT;
368 
369     init.enable                   = 1;
370     init.master                   = 1;
371     init.freq                     = I2C_FREQ_STANDARD_MAX;
372     init.clhr                     = i2cClockHLRStandard;
373     I2C_Init(I2C0, &init);
374     // [I2C0 initialization]$
375 
376 
377 }
```

```
540 extern void PORTIO_enter_DefaultMode_from_RESET(void) {
541 
542     // $[Port A Configuration]
543 
544     /* Pin PA0 is configured to Open-drain with pull-up and filter */
545     GPIO->P[0].MODEL = (GPIO->P[0].MODEL & ~_GPIO_P_MODEL_MODE0_MASK) | GPIO_P_MODEL_MODE0_WIREDANDPULLUPFILTER;
546 
547     /* Pin PA1 is configured to Open-drain with pull-up and filter */
548     GPIO->P[0].MODEL = (GPIO->P[0].MODEL & ~_GPIO_P_MODEL_MODE1_MASK) | GPIO_P_MODEL_MODE1_WIREDANDPULLUPFILTER;
549     // [Port A Configuration]$
550 
551 
552     // $[Port B Configuration]
553     // [Port B Configuration]$
554 
555 
556     // $[Port C Configuration]
557     // [Port C Configuration]$
558 
559 
560     // $[Port D Configuration]
561     // [Port D Configuration]$
562 
563 
564     // $[Port E Configuration]
565 
566     /* Pin PE10 is configured to Push-pull */
567     GPIO->P[4].MODEH = (GPIO->P[4].MODEH & ~_GPIO_P_MODEH_MODE10_MASK) | GPIO_P_MODEH_MODE10_PUSHPULL;
568 
569     /* Pin PE11 is configured to Input enabled */
570     GPIO->P[4].MODEH = (GPIO->P[4].MODEH & ~_GPIO_P_MODEH_MODE11_MASK) | GPIO_P_MODEH_MODE11_INPUT;
571 
572     /* Pin PE12 is configured to Push-pull */
573     GPIO->P[4].MODEH = (GPIO->P[4].MODEH & ~_GPIO_P_MODEH_MODE12_MASK) | GPIO_P_MODEH_MODE12_PUSHPULL;
574 
575     /* Pin PE13 is configured to Push-pull */
576     GPIO->P[4].MODEH = (GPIO->P[4].MODEH & ~_GPIO_P_MODEH_MODE13_MASK) | GPIO_P_MODEH_MODE13_PUSHPULL;
577 
578     /* Pin PE15 is configured to Push-pull */
579     GPIO->P[4].MODEH = (GPIO->P[4].MODEH & ~_GPIO_P_MODEH_MODE15_MASK) | GPIO_P_MODEH_MODE15_PUSHPULL;
580     // [Port E Configuration]$
581 
582 
583     // $[Port F Configuration]
584     // [Port F Configuration]$
585 
586 
587     // $[Route Configuration]
```

```
588 
589     /* Enable signals SCL, SDA */
590     I2C0->ROUTE |= I2C_ROUTE_SCLPEN | I2C_ROUTE_SDAPEN;
591 
592     /* Module PCNT0 is configured to location 1 */
593     PCNT0->ROUTE = (PCNT0->ROUTE & ~_PCNT_ROUTE_LOCATION_MASK) | PCNT_ROUTE_LOCATION_LOC1;
594 
595     /* Enable signals CLK, CS, RX, TX */
596     USART0->ROUTE |= USART_ROUTE_CLKPEN | USART_ROUTE_CSPEN | USART_ROUTE_RXPEN |
597         USART_ROUTE_TXPEN;
598     // [Route Configuration]$
599 
600 
601 }
```

Configurator が生成されたコードをベースに、リファレンスマニュアルとアプリケーションノート（およびサンプルコード）を参考に、必要に応じて追記・修正を行ってください。

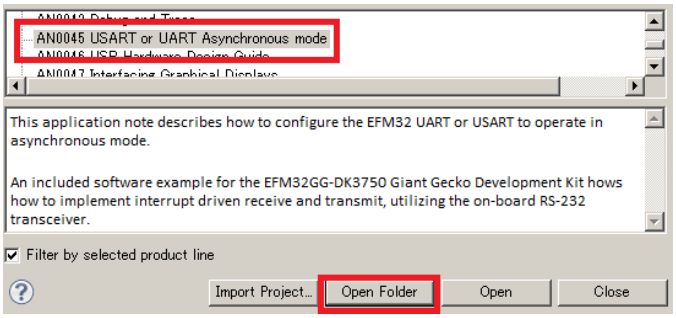
7-7 ペリフェラル設定

ペリフェラル設定の流れをご紹介します。

7-7-1 USART（Asynchronous mode）

◆ 参考にするソースコード

Application Notes ⇒ AN0045 で入手できます。Import Project をすると Simplicity IDE からソースコードが開けますので、ソースコードを追うのが楽になります。



◆ ヘッダファイル

"em_gpio.h" と"em_usart.h"を include してください。

◆ 初期化

以下は、初期化の例です。

void usart1_init(void) **任意の関数名**

```
{
	USART_InitAsync_TypeDef uart_init = {
		usartEnable,	// Enable RX/TX when init completed
		0,		// Use current configured reference clock for configuring baudrate
		115200,		// 115200 bits/s
		usartOVS16,	// 16x oversampling
		usartDatabits8, // 8 databits
		usartNoParity,	// No parity
		usartStopbits1, // 1 stopbit
		false,		// Do not disable majority vote
		false,		// Not USART PRS input mode
		usartPrsRxCh0	// PRS channel 0
	};
	// Using "TX" and "RX"
	GPIO_PinModeSet(gpioPortD, 7, gpioModePushPull, 1);
	GPIO_PinModeSet(gpioPortD, 6, gpioModeInput,	0);
	// Configure USART for basic async operation
	uart_init.enable = usartDisable;
	USART_InitAsync(USART1, &uart_init);
	// Prepare UART Rx and Tx interrupts
	USART_IntClear(USART1, _USART_IFC_MASK);
		USART_IntEnable(USART1, USART_IEN_RXDATAV);
		NVIC_ClearPendingIRQ(USART1_RX_IRQn);
		NVIC_ClearPendingIRQ(USART1_TX_IRQn);
		NVIC_EnableIRQ(USART1_RX_IRQn);
		NVIC_EnableIRQ(USART1_TX_IRQn);
	// Enable pins at default location
	USART1 ->ROUTE = USART_ROUTE_RXPEN | USART_ROUTE_TXPEN |
USART_ROUTE_LOCATION_LOC3;
	// Finally enable it
	USART_Enable(USART1, usartEnable);
}
```

◆　割り込み処理

割り込みハンドラは“system_efm32lg.h”（EFM32LG の場合）でプロトタイプ宣言されています。

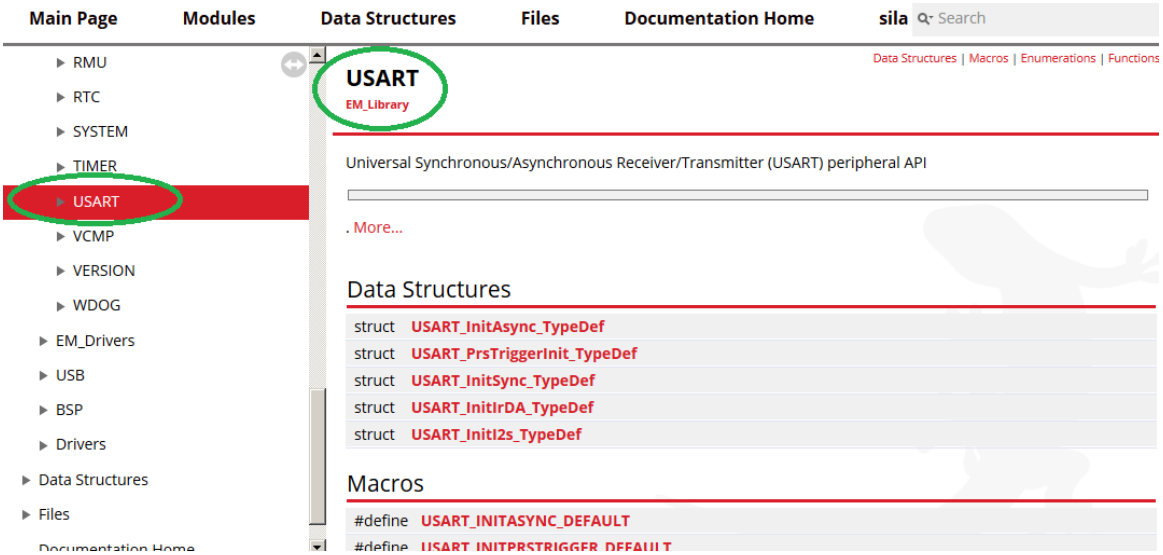
¥¥SiliconLabs¥SimplicityStudio¥v2¥developer¥sdks¥efm32¥v2¥Device¥SiliconLabs¥EFM32LG¥Include

以下は、割り込み処理の例です。

```
void USART1_RX_IRQHandler(void)        UART1 RX の割り込みハンドラ
{
    uint8_t          rxdata;

    // Checking that RX-flag is set
    if (USART1->STATUS & USART_STATUS_RXDATAV) {
        rxdata = USART1->RXDATA;

        //  独自の処理

        // Disable Rx interrupt
        USART1->IEN &= ~USART_IEN_RXDATAV;
        NVIC_ClearPendingIRQ(USART1_RX_IRQn);
        NVIC_DisableIRQ(USART1_RX_IRQn);
    }
}
void USART1_TX_IRQHandler(void)        UART1 TX の割り込みハンドラ
{
    // Checking that the USART is waiting for data
    if (USART1->STATUS & USART_STATUS_TXBL) {
        // Transmitting the next byte
        USART1->TXDATA = “user data”

        //  独自の処理

        // Disable Tx interrupt
        USART1->IEN &= ~USART_IEN_TXBL;
        NVIC_DisableIRQ(USART1_TX_IRQn);
        NVIC_ClearPendingIRQ(USART1_TX_IRQn);
    }
}
```

◆　制御方法

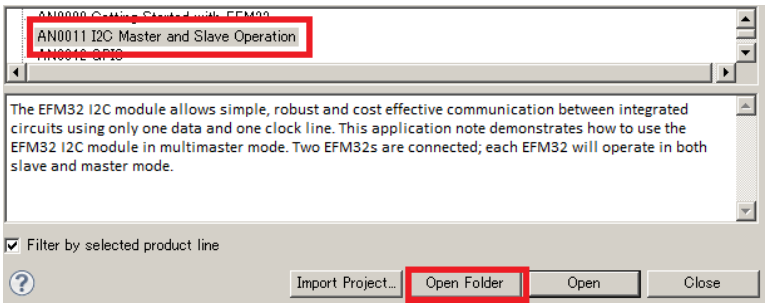
“em_usart.h”の API を使用して制御します。7-2「開発用のソースコードについて」をご参照ください。



7-7-2 I2C

◆ 参考にするソースコード

Application Notes ⇒ AN0011 で入手できます。Import Project をすると Simplicity IDE からソースコードが開けますので、ソースコードを追うのが楽になります。



◆ ヘッダファイル

"em_gpio.h" と"em_i2c.h"を include してください。

◆ 初期化

以下は、初期化の例です。

void i2c0_init(void) **任意の関数名**

```
{
    // default settings is 100KHz
    I2C_Init_TypeDef i2cInit = {
        true,                       // Enable when init done
        true,                       // Set to master mode
        0,                          // Use currently configured reference clock
        I2C_FREQ_STANDARD_MAX, // Set to standard rate assuring being within I2C spec
        i2cClockHLRStandard,        // Set to use 4:4 low/high duty cycle
    };

    // Using "SDA" and "SCL" pin
    GPIO_PinModeSet(gpioPortE, 12, gpioModeWiredAndPullUpFilter, 1);
    GPIO_PinModeSet(gpioPortE, 13, gpioModeWiredAndPullUpFilter, 1);

    // Initializing the I2C
    I2C_Init(I2C0, &i2cInit);

    // Clear and enable interrupt
    NVIC_ClearPendingIRQ(I2C0_IRQn);
    NVIC_EnableIRQ(I2C0_IRQn);

    I2C_Enable(I2C0, 1);
}
```

◆ 割り込み処理

割り込みハンドラは“system_efm32lg.h”（EFM32LG の場合）でプロトタイプ宣言されています。

以下は、割り込み処理の例です。

void I2C0_IRQHandler (void) **割り込みハンドラ**

```
{
    // 独自の処理

    NVIC_DisableIRQ(I2C0_IRQn);
}
```

◆ 制御方法

“em_i2c.h”の API を使用して制御します。7-2「開発用のソースコードについて」をご参照ください。

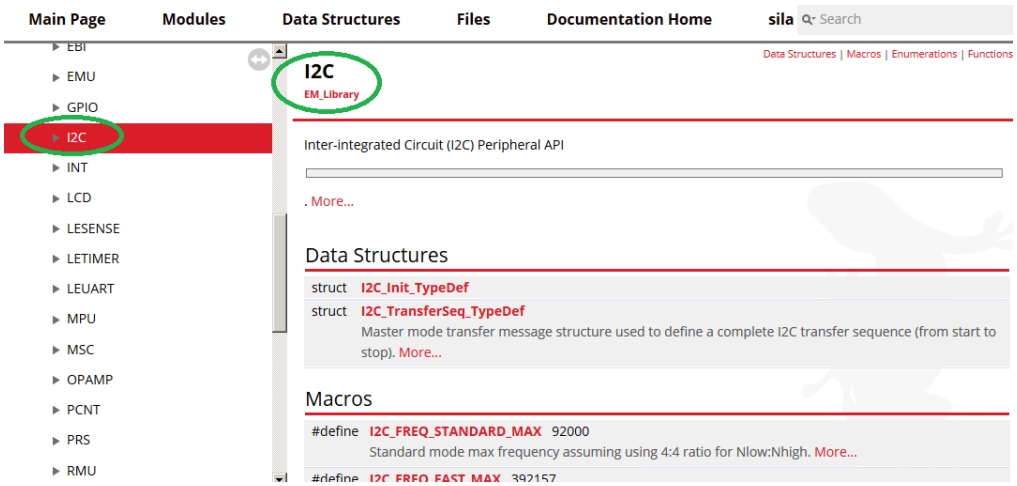
以下は、I2C ライト、I2C リードの例になります。

int i2c_write(I2C_TypeDef *i2c, uint8_t addr, uint8_t wval) I2C ライトの制御例

```
{
	uint8_t wdata[1];
	I2C_TransferSeq_TypeDef seq;
	I2C_TransferReturn_TypeDef I2C_Status;
	seq.addr = addr;
	seq.flags = I2C_FLAG_WRITE_WRITE;
	// write buffer setting
	wdata[0] = (uint8_t) wval;
	seq.buf[0].data = wdata;
	seq.buf[0].len   = 1;
	// Do a polled transfer
	I2C_Status = I2C_TransferInit(i2c, &seq);
	while (I2C_Status == i2cTransferInProgress) {
		I2C_Status = I2C_Transfer(i2c);
	}
	return(I2C_Status);
}
```

int i2c_read(I2C_TypeDef *i2c, uint8_t addr, uint8_t *p_rval) I2C リードの制御例

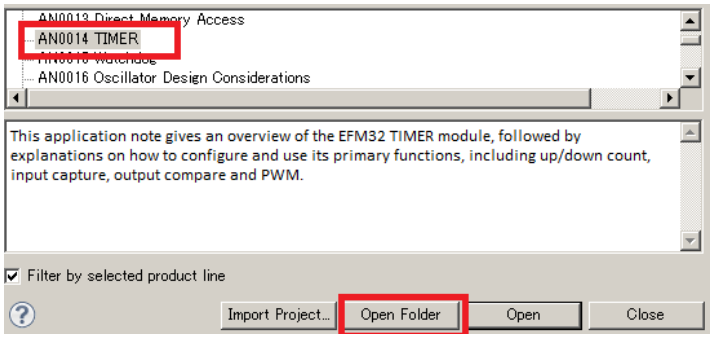
```
{
	uint8_t rdata[1] = {0};
	I2C_TransferSeq_TypeDef seq;
	I2C_TransferReturn_TypeDef I2C_Status;
	seq.addr = addr;
	seq.flags = I2C_FLAG_READ;

	// register setting
	seq.buf[0].data = rdata;
	seq.buf[0].len   = 1;

	// Do a polled transfer
	I2C_Status = I2C_TransferInit(i2c, &seq);
	while (I2C_Status == i2cTransferInProgress) {
		I2C_Status = I2C_Transfer(i2c);
	}
	*p_rval = rdata[0];

	return (I2C_Status);
}
```



7-7-3 タイマ

◆ 参考にするソースコード

Application Notes ⇒ AN0014 で入手できます。Import Project をすると Simplicity IDE からソースコードが開けますので、ソースコードを追うのが楽になります。



◆ ヘッダファイル

"em_timer.h"を include してください。

◆ 初期化

以下は、初期化の例です。

```
static void start_timer1(void)        任意の関数名
{
      // Enable clock for TIMER1 module
      CMU_ClockEnable(cmuClock_TIMER1, true);

      // Select TIMER1 parameters
      TIMER_Init_TypeDef timerInit = TIMER_INIT_DEFAULT;

      // Enable overflow interrupt
      TIMER_IntEnable(TIMER1, TIMER_IF_OF);

      // Enable TIMER1 interrupt vector in NVIC
      NVIC_EnableIRQ(TIMER1_IRQn);

      // Set TIMER1 Top value
      TIMER_TopSet(TIMER1, CMU_ClockFreqGet(cmuClock_HFPER) / 1000);

      // Configure TIMER1
      TIMER_Init(TIMER1, &timerInit);
}
```

初期設定

```
{
.enable         = true,
.debugRun       = false,
.prescale       = timerPrescale1024,
.clkSel         = timerClkSelHFPerClk,
.fallAction = timerInputActionNone,
.riseAction = timerInputActionNone,
.mode           = timerModeUp,
.dmaClrAct      = false,
.quadModeX4 = false,
.oneShot        = false,
.sync           = false,
};
```

◆ 割り込み処理

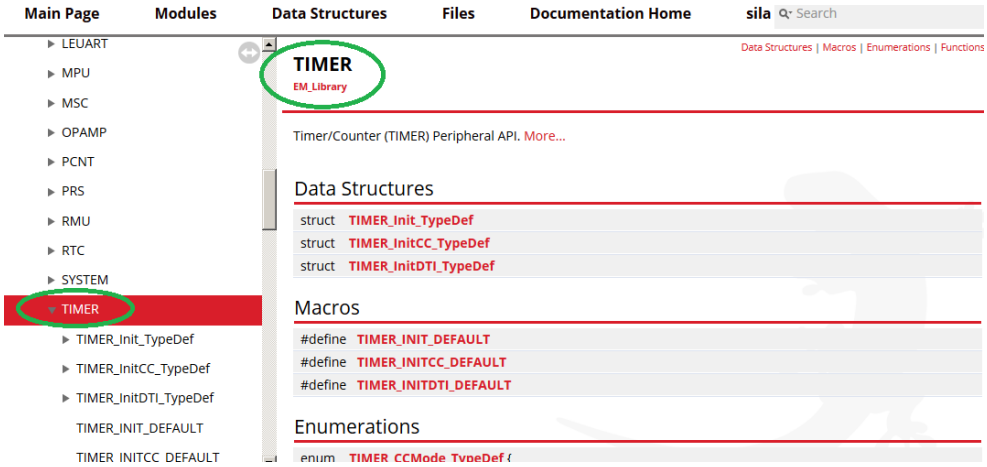
割り込みハンドラは“system_efm32lg.h”（EFM32LG の場合）でプロトタイプ宣言されています。

以下は、割り込み処理の例です。

```
void TIMER1_IRQHandler(void)      割り込みハンドラ
{
      // 独自の処理

      // Clear flag for TIMER1 overflow interrupt
      TIMER_IntClear(TIMER1, TIMER_IF_OF);
}
```

◆ 制御方法

“em_timer.h”の API を使用して制御します。7-2「開発用のソースコードについて」をご参照ください。

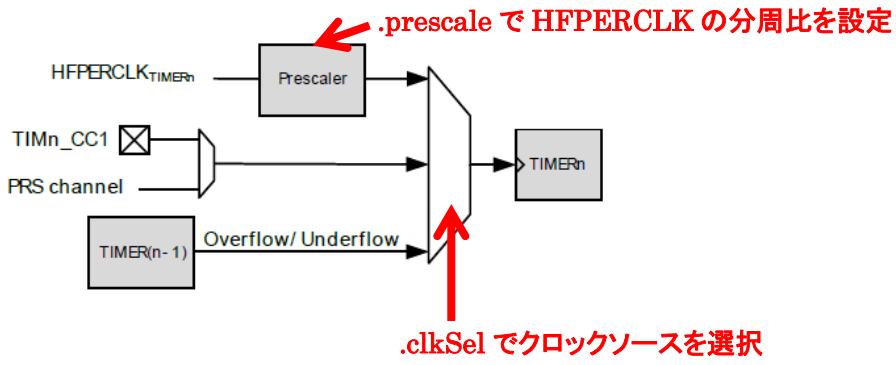


7-7-4 タイマ （X 秒タイマの作り方）

TIMER_Init_TypeDef 型のメンバーである“.prescale”と“.clkSel”と、“TIMER_TopSet()”の値を変えて任意の時間のタイマを作ります。

AN0014 の“main_timer_up_count.c”では 2 秒タイマを作っており、これを例に説明します。

タイマへ供給されるクロックは、下図のような構成になっています。（AN0014 Fig.2.1）



“main_timer_up_count.c”では、“.prescale”と“.clkSel”は以下の設定になっています。

```
TIMER_Init_TypeDef timerInit =
{
  .enable       = true,
  .debugRun     = true,
  .prescale     = timerPrescale1024,
  .clkSel       = timerClkSelHFPerClk,
  .fallAction   = timerInputActionNone,
  .riseAction   = timerInputActionNone,
  .mode         = timerModeUp,
  .dmaClrAct    = false,
  .quadModeX4   = false,
  .oneShot      = false,
  .sync         = false,
};
```

.prescale で 1024 分の 1 を選択

.clkSel で HFPERCLK（14MHz）を選択

つまりタイマへの供給クロックは、14 MHz（周波数） ÷ 1024（prescale） ＝ 13671 Hz となります。

2 秒タイマを作るには、13671 Hz × 2 ＝ 27342 のカウントが必要です。“TIMER_TopSet()”を使って、タイマの上限値を 27342 に設定します。

```
#define TOP 27342

/* Set TIMER Top value */
TIMER_TopSet(TIMER0, TOP);
```

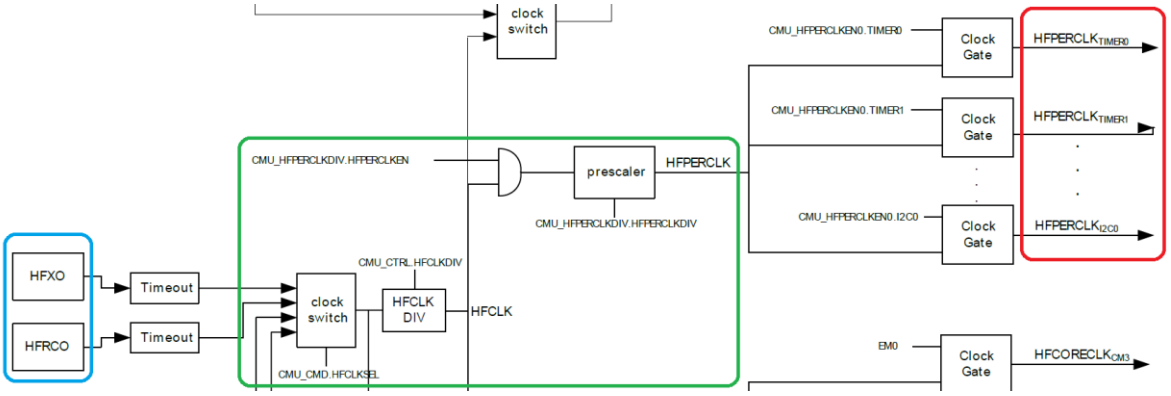
変数 TOP を 27342 に設定

TIMER_TopSet()で 上限を TOP（27342）に変更

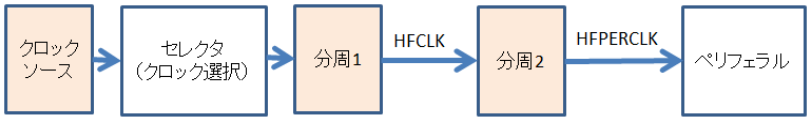
7-7-5 CMU （ペリフェラル・クロックの周波数）

ペリフェラルへの供給クロックの周波数を、どのように設定するか紹介します。

リファレンスマニュアルの CMU - Clock Management Unit の章に、CMU Overview という図があります。この図で、EFM32 のクロックツリーを確認する事ができます。下図は CMU Overview の一部抜粋になりますが、元になるクロックソース（下図の青枠）が、セレクタや分周回路（下図の緑枠）を経て、ペリフェラル・クロック（下図の赤枠）になります。



シンプルな図にすると下図になります。色づけした箇所が、周波数に影響するブロックです。



クロックソースとして、HFRCO（内蔵の高速クロック）を選択した場合を例に説明します。

◆ HFRCO の周波数設定 （クロックソース）

HFRCO の周波数は、CMU_HFRCOCTRL の BAND ビットで設定されています。

下図は Leopard Gecko のリファレンスマニュアルからの抜粋ですが、1（実際には 1.2）, 7（実際には 6.6）, 11, 14, 21, 28MHz から選択ができます。デフォルトでは 14MHz に設定されています。

11.5.4 CMU_HFRCOCTRL - HFRCO Control Register

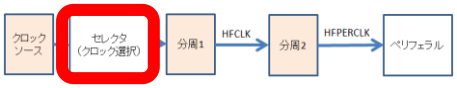
Offset	Bit Position																															
0x00C	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																	0x00					0x3			0x80							
Access																	RW					RW			RW							
Name																	SUDELAY					BAND			TUNING							

Bit	Name	Reset	Access	Description
10:8	BAND	0x3	RW	**HFRCO Band Select**

Write this field to set the frequency band in which the HFRCO is to operate. When changing this setting there will be no glitches on the HFRCO output, hence it is safe to change this setting even while the system is running on the HFRCO. To ensure an accurate frequency, the HFTUNING value should also be written when changing the frequency band. The calibrated tuning value for the different bands can be read from the Device Information page.

Value	Mode	Description
0	1MHZ	1 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.
1	7MHZ	7 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.
2	11MHZ	11 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.
3	14MHZ	14 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.
4	21MHZ	21 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.
5	28MHZ	28 MHz band. NOTE: Also set the TUNING value (bits 7:0) when changing band.

◆　クロックソースの選択　（セレクタ）



クロックソースは、CMU_CMD の HFCLKSEL で設定します。

11.5.10 CMU_CMD - Command Register

Offset	Bit Position																															
0x024	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																										0x0		0	0	0x0		
Access																										W1		W1	W1	W1		
Name																										USBCCLKSEL		CALSTOP	CALSTART	HFCLKSEL		

2:0	HFCLKSEL	0x0	W1	**HFCLK Select**

Selects the clock source for HFCLK. Note that selecting an oscillator that is disabled will cause the system clock to stop. Check the status register and confirm that oscillator is ready before switching.

Value	Mode	Description
1	HFRCO	Select HFRCO as HFCLK.
2	HFXO	Select HFXO as HFCLK.
3	LFRCO	Select LFRCO as HFCLK.
4	LFXO	Select LFXO as HFCLK.

◆　HFCLK の決定　（分周 1）

クロックソース（HFRCO）を分周して HFCLK にします。何分の 1 にするかは、CMU_CTRL の HFCLKDIV で決定します。

HFCLK は、クロックソース（HFRCO）の 1/ (HFCLKDIV+1) なります。1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8 から選択できます。HFCLKDIV = 0 であれば HFCLK = HFRCO、HFCLKDIV = 7 であれば HFCLK = HFRCO *（1/8）になります。

この HFCLK は、CPU コアの動作周波数にも利用します。

11.5.1 CMU_CTRL - CMU Control Register

Offset	Bit Position																															
0x000	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset		0		0			0x0			0x0			0x3		0	0x0			1	0x0		0x3			0	0x1			0x3		0x0	
Access		RW		RW			RW			RW			RW		RW	RW			RW	RW		RW			RW	RW			RW		RW	
Name		HFLE		DBGCLK			CLKOUTSEL1			CLKOUTSEL0			LFXOTIMEOUT		LFXOBUFCUR	HFCLKDIV			LFXOBOOST	LFXOMODE		HFXOTIMEOUT			HFXOGLITCHDETEN	HFXOBUFCUR			HFXOBOOST		HFXOMODE	

16:14	HFCLKDIV	0x0	RW	**HFCLK Division**

Use to divide HFCLK frequency by (HFCLKDIV + 1).

◆　HFPERCLK の決定　（分周 2）

HFCLK を分周して、HFPERCLK にします。何分の 1 にするかは、CMU_HFPERCLKDIV の HFPERCLVDIV ビットで決定します。

分周は、1/1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256, 1/512 から選択できます。1/512 を選択した場合には、HFPERCLK = HFCLK * (1/512) になります。

11.5.3 CMU_HFPERCLKDIV - High Frequency Peripheral Clock Division Register

Offset	Bit Position																															
0x008	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																								1					0x0			
Access																								RW					RW			
Name																								HFPERCLKEN					HFPERCLKDIV			

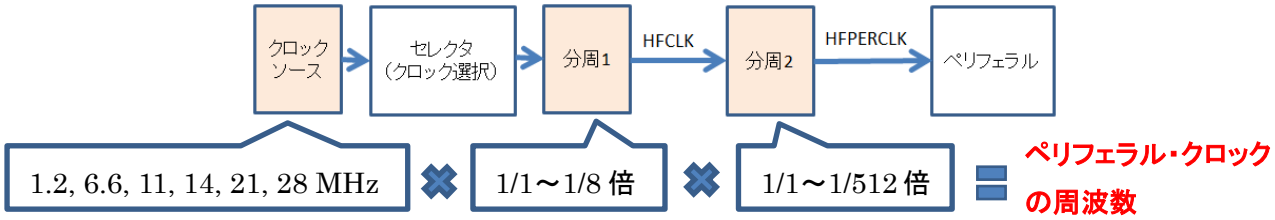
3:0 HFPERCLKDIV 0x0 RW **HFPERCLK Divider**

Specifies the clock divider for the HFPERCLK.

Value	Mode	Description
0	HFCLK	HFPERCLK = HFCLK.
1	HFCLK2	HFPERCLK = HFCLK/2.
2	HFCLK4	HFPERCLK = HFCLK/4.
3	HFCLK8	HFPERCLK = HFCLK/8.
4	HFCLK16	HFPERCLK = HFCLK/16.
5	HFCLK32	HFPERCLK = HFCLK/32.
6	HFCLK64	HFPERCLK = HFCLK/64.
7	HFCLK128	HFPERCLK = HFCLK/128.
8	HFCLK256	HFPERCLK = HFCLK/256.
9	HFCLK512	HFPERCLK = HFCLK/512.

◆ まとめ

以上をまとめますと、ペリフェラルに供給されるクロックの周波数は、下記の組み合わせになります。



HFRCO を分周し、通常ペリフェラルで使用した場合の事例を紹介しましたが、HFRCO ではなく HFXO（外付け高速クロック）/LFRCO（内蔵の低速クロック）/LFXO（外付け低速クロック）を使用した場合や、通常ペリフェラルではなく Low Energy ペリフェラルを使用した場合にも、CMU Overview のクロックツリーから読み解いていくという考え方は共通です。

改版履歴

Version	改定日	改定内容
1.0	2014 年 07 月	・新規作成
1.13	2015 年 06 月	・マクニカオンラインで公開
1.14	2015 年 08 月	・誤植訂正（6-2 章 M0+の breakpoint 数）、4-1, 6-3, 6-6-4, 6-7-3 章追記
1.15	2016 年 02 月	・Pearl Gecko/Jade Gecko 追加。最新の Simplicity Studio に合わせて説明を一部変更。

参考文献

- Silicon Labs 社 各種ドキュメント
- Silicon Labs 社 ナレッジベース、コミュニティフォーラム

本社
〒222-8561 横浜市港北区新横浜 1-6-3 TEL 045-470-9841 FAX 045-470-9844