統合開発環境 CS+

Python で CS+のデバッグを自動化入門(チュートリアル)

要旨

本アプリケーションノートは、統合開発環境 CS+に搭載されている Python 機能について、RH850 を使用して入門レベルの使用方法を紹介するものです。

本アプリケーションノートは、2021 年 6 月に実施したセミナー資料をアプリケーションノートとして編集し直したものです。

外部リンクなどは最新のものではありませんので、ご注意ください。

本アプリケーションノートには以下の２つのサンプルプロジェクトが付属します。

RH850_F1L_GoWaitBreakSetRegister

（フォルダ：「デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行う」）

RH850_F1L_ExcelReadWrite

（フォルダ：

「Excel ファイルのテストデータを使用したファジングテストを自動的に実行」）

上記のサンプルプロジェクトは、CS+のサンプルプロジェクト RH850_F1L_Tutorial_Basic_Operation をベースに作成したものです。

付属のサンプルプロジェクトはセミナー開催時の CS+ for CC V8.06.00 で動作を確認していますが、最新の CS+ for CC をご使用する事を推奨いたします。CS+ for CC 関連のドキュメントの参照は、現在の最新版の CS+ for CC V8.08.00 を記載しています。最新版の CS+ for CC は以下より入手可能です。

https://www.renesas.com/jp/ja/software-tool/cs

本アプリケーションノートの内容とソフトウェアは Python 機能の応用例を説明するもので、その内容を保障するものではありません。

動作確認デバイス

RH850/F1L

関連ドキュメント (現在最新のバージョンを記載)

CS+ V8.08.00 統合開発環境 ユーザーズマニュアル RH850 デバッグ・ツール編 (renesas.com)

CS+ V8.08.00 統合開発環境 ユーザーズマニュアル Python コンソール編 Rev.1.00 (renesas.com)

目次

1. はじめに 3
1.1 本資料の流れ 3

2. Python とは？ 簡単な紹介 3
2.1 Python とは? 簡単な紹介 3
2.2 コンパイラ言語とインタープリタ言語の違い 4
2.3 CS+での Python の役割 4

3. Python コンソール 5
3.1 Python コンソールパネルの準備 5
3.2 Python コマンドを使用した基本的なデバッガ操作 7
3.3 スクリプトを準備することで効率的に操作可能 10

4. Python スクリプトを使用した自動デバッグ手法 11
4.1 スクリプト 1 11
4.2 スクリプト 2 21

5. CS+と Python スクリプトのコマンドライン操作 34

6. 次のステップのために 36
6.1 サンプル・スクリプト 36
6.2 かふぇルネ 36

Appendix 37
CS+に実装された Python について 37
.NET Framework について 37
.NET アセンブリについて 37
一つ目のプロジェクトの LED 動作を Excel で確認できるプロジェクト 38
CI/CD について 38
Python スクリプトのデバッグに使えるヒント 39

1. はじめに

1.1 本資料の流れ

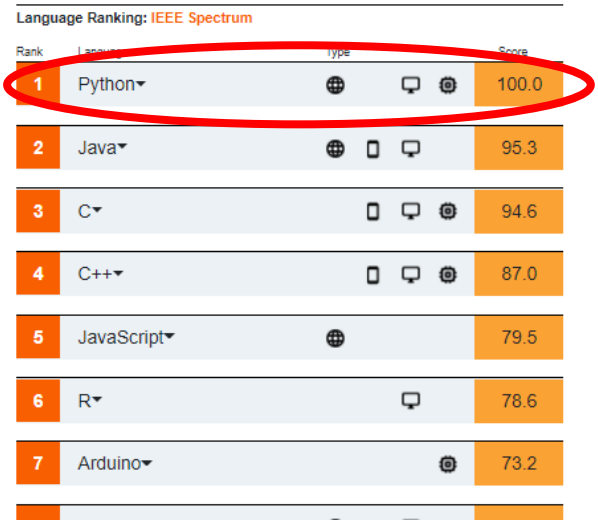
CS+に搭載された Python をデバッグに使用して、デバッグの効率化、自動化に役立つヒントを提供いたします。

- Python の簡単なご紹介
- CS+での Python の役割
- Python 関数を使用した基本的なデバッグ操作
- 2 つのスクリプトを使用したデバッグ操作

2. Python とは？ 簡単な紹介

2.1 Python とは? 簡単な紹介

Python は汎用的なプログラミング言語の利用度調査などでは、常に高い位置を占めています。



左図は 2020 年の IEEE による調査結果です。

IEEE *The 2020 Top Programming Languages* より

Python はシステム管理やツール・アプリケーション開発・科学技術計算・Web システムなどで広く利用されています。

特徴として、

- Python はインタープリタ言語（逐次翻訳）で、実行性能よりも使いやすさを優先した構造
- C 言語などのコンパイラ言語(一括翻訳)と比べ実行速度は劣り、メモリー使用量も多い

採用例として：Instagram は実行性能では劣る Python を採用

⇒その理由として、

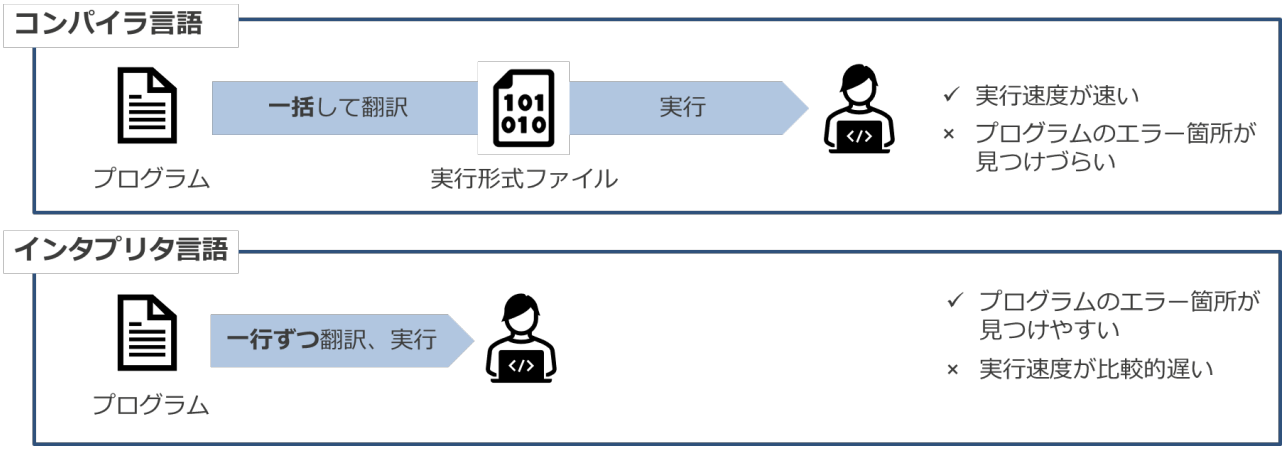
> Instagramのボトルネックは
> 開発速度
> です。コードの実行速度ではありません。

@ プログラミング言語 Python の紹介 - python.jp

と述べています。

2.2　コンパイラ言語とインタープリタ言語の違い

コンパイラとインタープリタの違いを簡単に確認します。プログラミング言語をコンピュータ上で実際に実行させるためには、機械語に翻訳（コンパイル）する必要があります。プログラムを一括して機械語に翻訳コンパイルする処理系を持つ言語をコンパイラ言語と言い、通訳のように一行ずつ逐次翻訳コンパイルする処理系を持つ言語がインタープリタ言語です。



そのため、コンパイラでは一括して翻訳された実行形式ファイルがコンピュータでそのまま実行されるので、実行速度が速い利点がありますが、反面、プログラムのエラー箇所が見つけにくい、修正ごとにビルドしなおす必要など、デバッグに時間が掛かり、開発に時間を要する傾向があります。

インタープリタでは、逆に実際の実行は逐次翻訳なので実効速度が遅くなりますが、比較的にデバッグはやり易いため開発速度は速くなりえる特徴があります。

2.3　CS+での Python の役割

CS+での Python の役割は、例えるなら、Excel のマクロを記述する VBA に相当します。

すなわち、定型作業の自動化、作業自体の高速化になります￥。

CS+の Python の特徴として、（詳細は Appendix）

1. IronPython(.NET Framework 上で動作する Python、Python2.7 互換)を搭載
   - 制御文等も含む
   - Excel など外部アプリと連携（.NET アセンブリを Python にインポート可能）
   - デバッガなど一部機能に制限があります。
2. CS+の機能を独自実装した関数

   CS+上で使うために、CS+の機能を関数として独自実装しています。
   - プロジェクト操作用：project.Close, project.Open, project.GetFunctionList 等
   - ビルド・ツール用：build.All, build.ChangeBuildMode, build.Clean 等
   - デバッグ・ツール用：debugger.ActionEvent.Delete, debugger.ActionEvent.Set 等
   - 他（基本操作、共通）：ClearConsole, Save, common.GetOutputPanel 等

合計 160

3. Python コンソール

Python コンソールは Windows のコマンドプロンプトに相当し、Python 関数や制御文，および CS+ Python 関数を入力，実行します。また，関数の実行結果やエラーの表示も行います。

Python コンソールは Python コンソールパネル内にあります。

3.1 Python コンソールパネルの準備

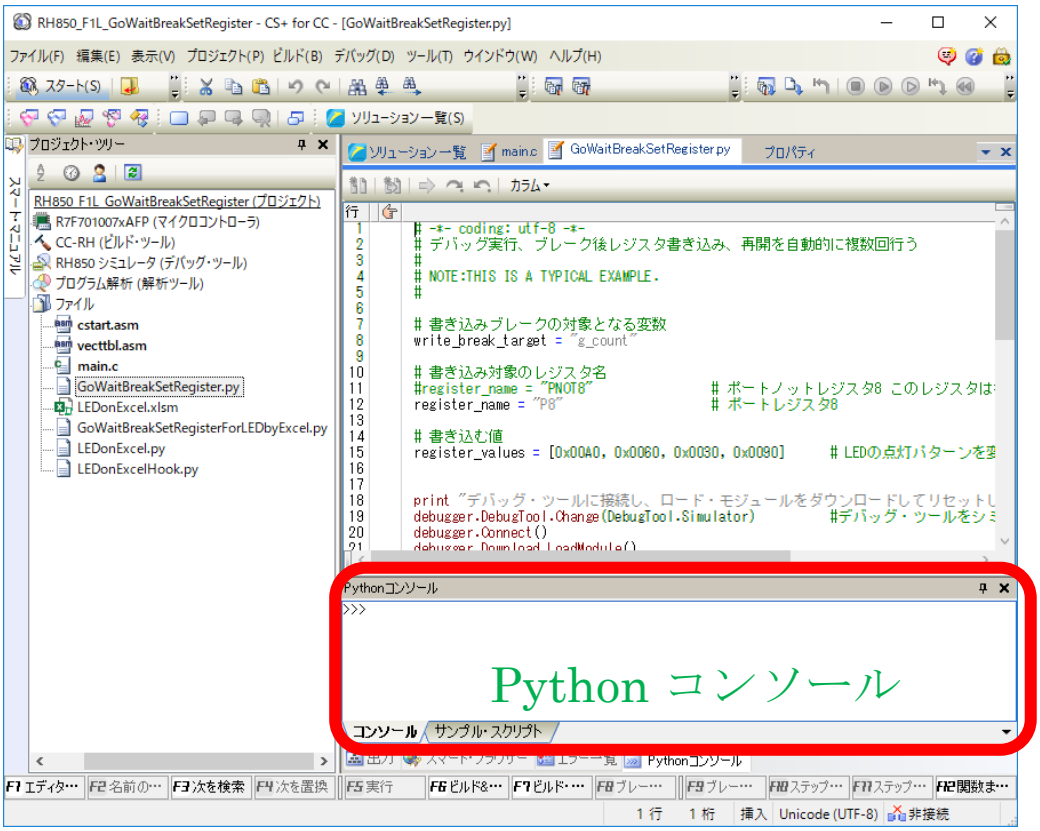
Python コンソールパネルは、Python のコマンドを実行させ、実行結果を表示させる「コンソール」と、サンプル・スクリプトを表示し，スクリプトをプロジェクトに登録する「サンプル・スクリプト」からなるパネルです。

Python コンソールパネルを表示させます。

手順:

ⅰ． プロジェクトを開く

ⅱ． Python コンソールパネルの表示



手順ⅰ　プロジェクトを開く

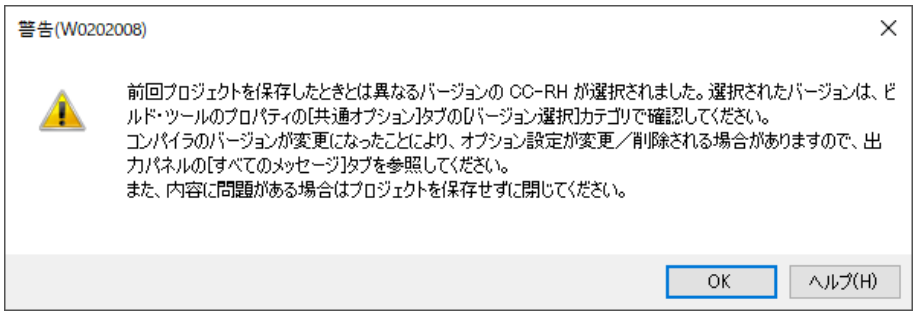
プロジェクト RH850_F1L_GoWaitBreakSetRegister を使用します。

デスクトップに付属のフォルダ「デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行う」を置き、その下にある

RH850_F1L_GoWaitBreakSetRegister フォルダ内の

RH850_F1L_GoWaitBreakSetRegister.mtpj をダブルクリックしてプロジェクトを開きます。

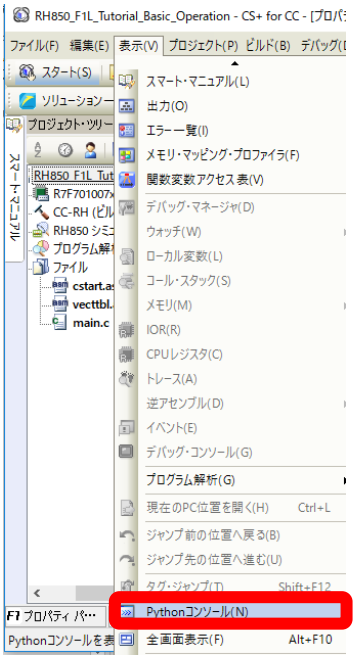
注：付属のサンプルプロジェクトは CS+V8.06.00 で動作を確認しています。これ以降の CS+をご使用で、同梱の C コンパイラ CC-RH がバージョンアップされている場合は、以下の警告が表示されます。問題はありませんので、OK を教えてください。



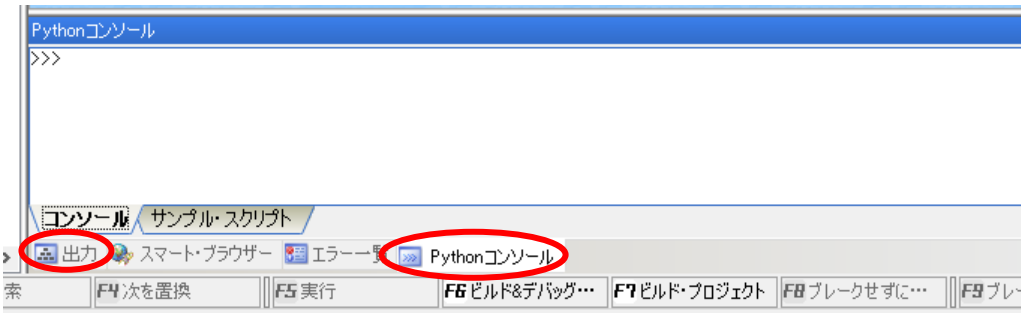
手順 ii　Python コンソールパネルの表示

プロジェクトが開いたら、Python コンソールパネルを表示させます。

(1) [表示]メニューの[Python コンソール]を選択します。



(2) Python コンソールパネルが表示されます。



出力パネルに被って同じ位置に Python コンソールが開きます。

3.2　Python コマンドを使用した基本的なデバッガ操作

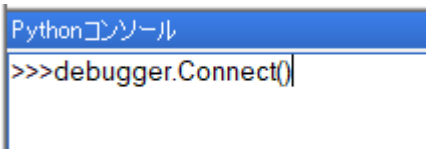
ここでは、以下の基本的なデバッガの操作を Python コンソール から直接、関数(コマンド)を使って実行してみます。

デバッガ接続 ⇒ プログラムダウンロード ⇒ ブレークポイント設定 ⇒ 実行 ⇒ デバッガ切断

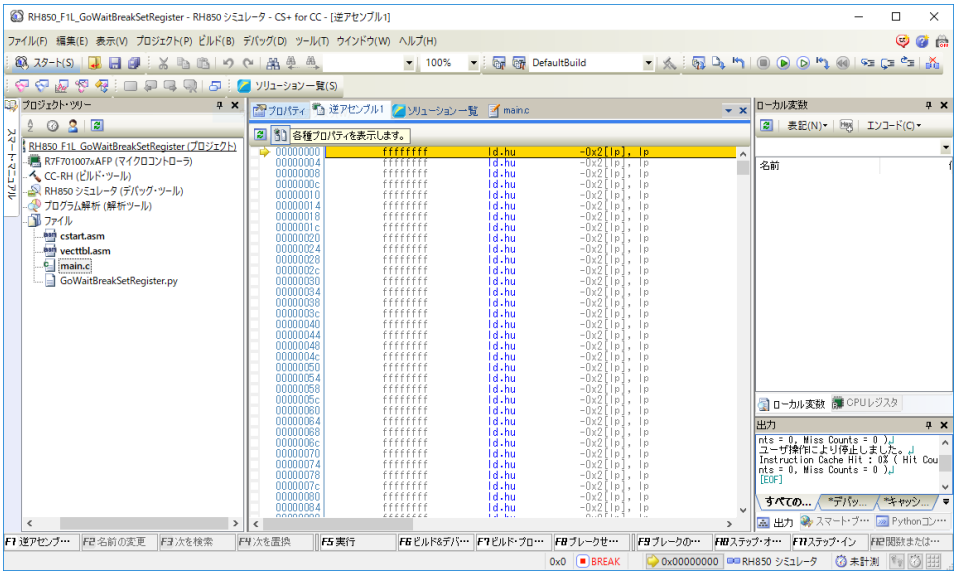
1. デバッガ接続

Python コンソールに以下の関数(コマンド)を打ち込み Enter キーを押します。

debugger.Connect()



デバッグ・レイアウトが開きます。



逆アセンブルの表示にコードが何もない事を確認してください。プログラムはまだダウンロードされていません。

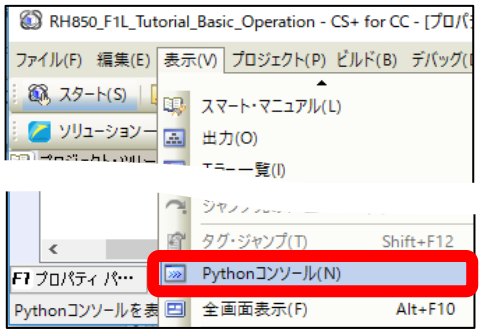
Python コンソールパネルが表示されていません。デバッグ時には、レイアウトがデバッグ専用になるため別に設定する必要があります。改めて Python コンソールを表示させます。

・Python コンソールの表示

プログラムダウンロードの前にデバッグ時のレイアウトを設定します。

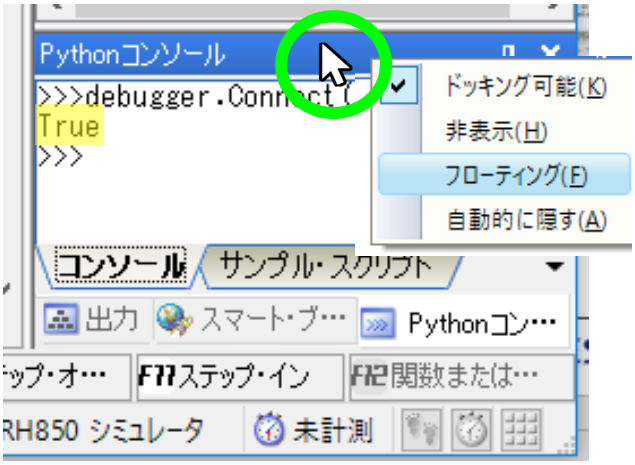
Python コンソールを開きます。

②　[表示]メニューの[Python コンソール]を選択します。



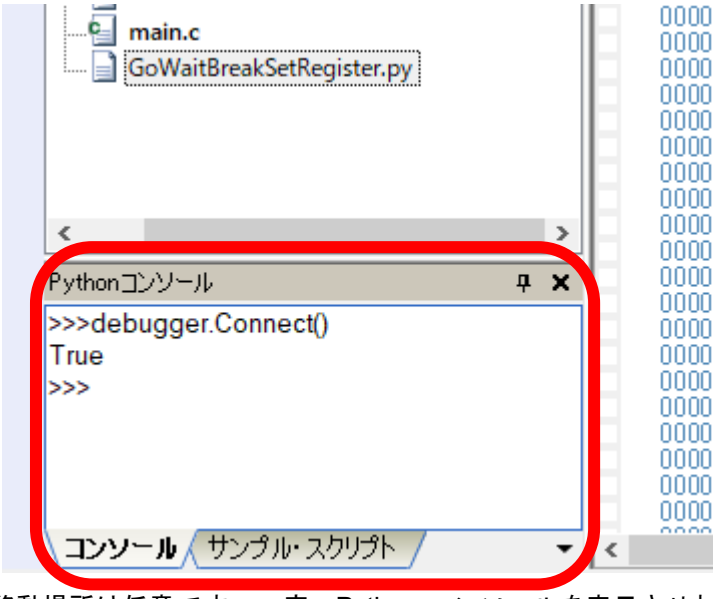
② Python コンソールが開くと先に入力した関数 debugger.Connect()に対して、True が返されいて、正常に実行された事が確認できます。

タイトル・バー上で右クリックし、フローティングを選択します。



※ デバッグ中は、Python コンソールが出力パネルと重なると表示が出力パネル優先になります。Python コンソールを常に確認したい場合は移動します。移動のため、一旦フローティングにしますが、フローティングのままではスクリプト実行中、Python コンソールが CS+に隠れて出力を実行中に確認できない可能性があるため出力パネルと重ならない位置に移動、 固定します。

③ タイトル・バーをつかみプロジェクト・ツリー下に移動します。



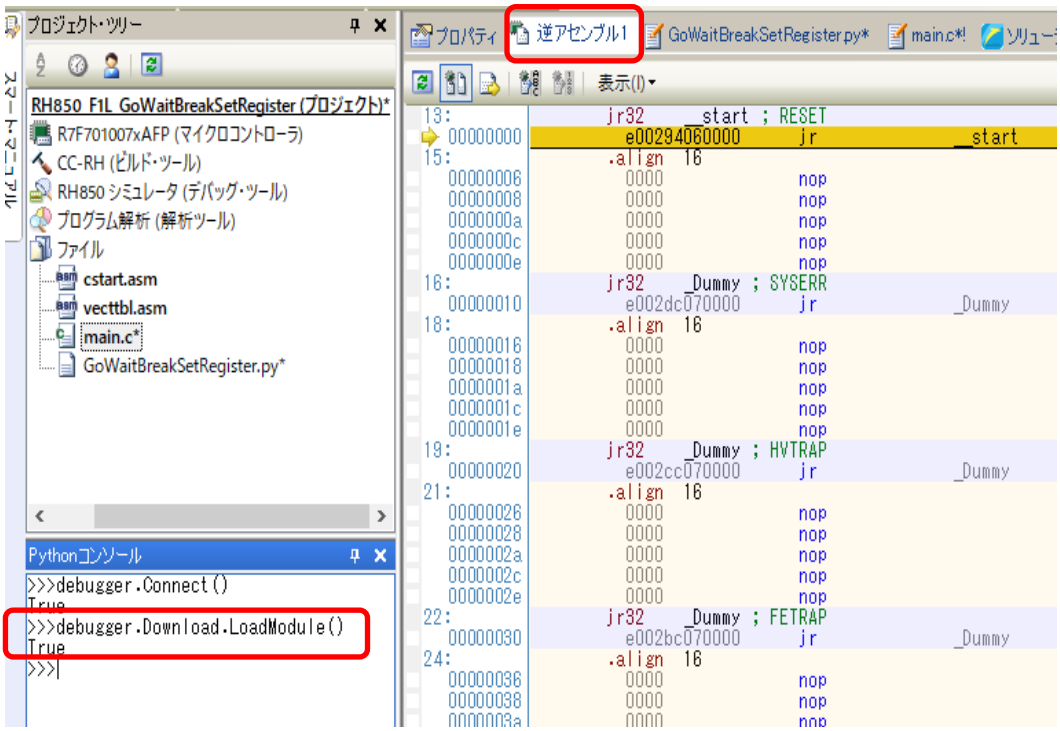
移動場所は任意です。一度、Python コンソールを表示させれば、プロジェクト保存でレイアウトは保存されます。

2. プログラムのダウンロード

Python コンソールに以下を打ち込みます。

```
debugger.Download.LoadModule()
```

プログラムがダウンロードされます。Python コンソールに True が返され、逆アセンブルにコードがロードされているのが確認できます。



3. ブレークポイントの設定

前回の評価で設定したブレークポイントが残っている可能性があるため、念のため、ブレークポイントをいったん削除します。

debugger.Breakpoint.Delete()

ブレークポイントを main.c の 46 行目アドレス 0x77a に設定します。

以下を打ち込みブレークポイントを設定します。

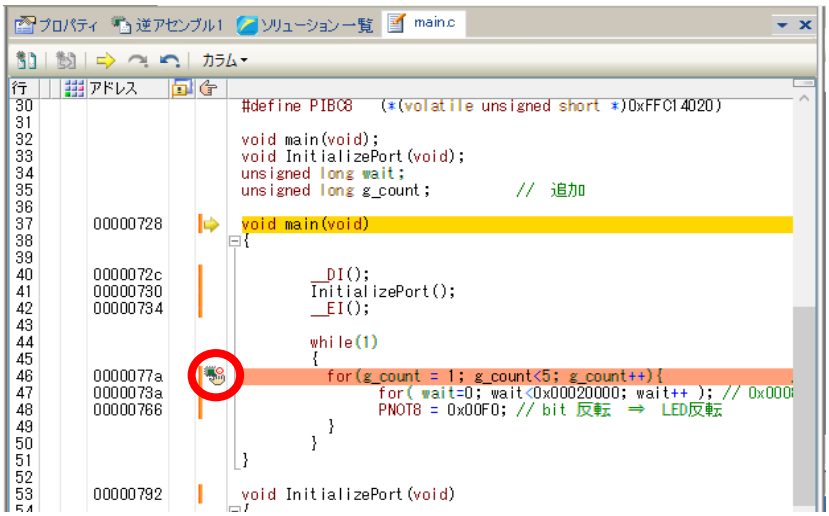
bp = BreakCondition()

bp.Address = “0x77a"

debugger.Breakpoint.Set(bp)

（これらの記述については、別途スクリプトの解説で説明します。）

main.c を確認すると、イベントマークが表示され、ブレークポイントがアドレス 0x77a に設定されたのが確認できます。（赤丸部分）。

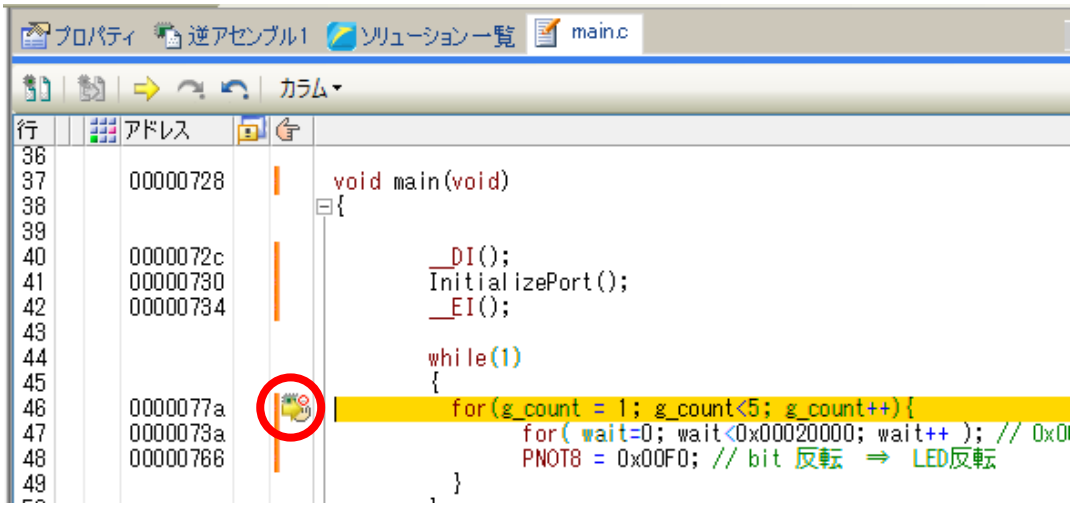


4. 実行

ブレークポイントまで実行します。以下を入力します。カッコ内の GoOption.WaitBreak は引数で、次のブレークポイントまで実行を意味します。

```
debugger.Go(GoOption.WaitBreak)
```

プログラムカウンタ（PC）の黄色い右矢印がブレークポイントに表示されプログラムが停止したことを示します。



5. デバッガ切断

最後に以下を投入して、デバッガから切断、終了します。

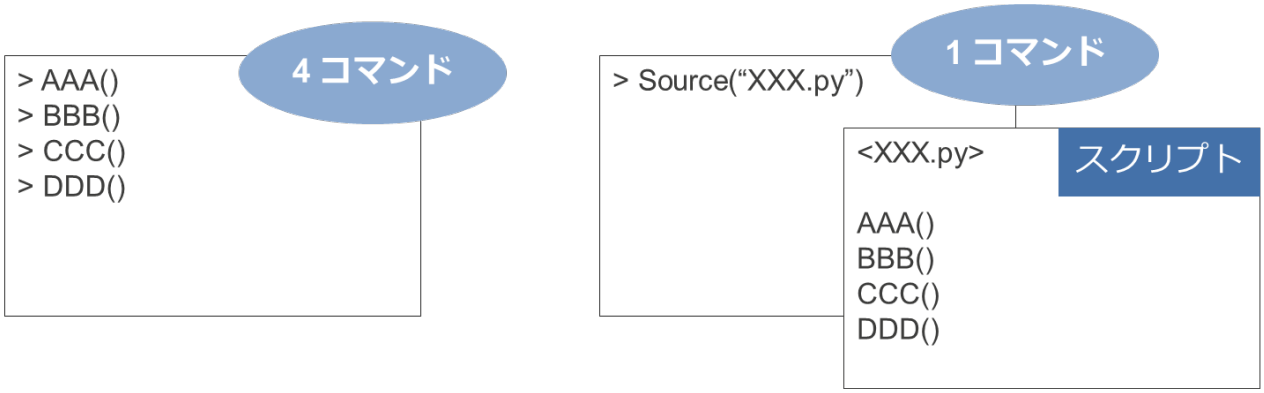
```
debugger.Disconnect()
```

3.3 スクリプトを準備することで効率的に操作可能

操作を行う度に関数、コマンドを入力するのは手間がかかります。また、制御文が使えないので、高度な制御ができません。そこで、実行したい事を、ファイルに記述してプログラムとして実行できるようにします。

関数や制御文などを事前にスクリプトファイルに記述することで 1 つのコマンド入力で、複数の操作が可能です。このファイルを、Python スクリプトと呼び、それを実行することで 1 つのコマンド入力で、複数の操作が可能です。

イメージ）



4. Python スクリプトを使用した自動デバッグ手法

最初に、本資料で扱う２つのスクリプトの簡単な紹介です。

スクリプト１：

「デバッグ実行、ブレーク後レジスタ書き込み、再開」を自動的に複数回行うスクリプトです。

⇒プログラムの変数を任意に書き換えることで単体テストの実施に便利なスクリプトです。

スクリプト２：

「Excel ファイルのテストデータを使用したファジングテスト」を自動的に実行するスクリプトです。

⇒プログラムの変数を Excel ファイルのデータで書き換えて実行し、演算結果を同じ Excel ファイルに出力しデータ処理ができるファジングテストに便利なスクリプトです。

ファジングテストとはランダムなデータを入力しプログラムの挙動を確認するテスト手法の１つです。

4.1 スクリプト１

以下、スクリプト１の概要です。

デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行うスクリプトです。

このスクリプトの目的は、

LED の点灯パターンを複数の条件で確認する事です。

スクリプトが実行する動作は以下になります。

1) デバッグ・ツール（E2 エミュレータ、またはシミュレータ）に接続
2) ブレークポイントを設定、実行
3) ブレーク後テストしたい LED の点滅パターンをレジスタ書き込み
4) 実行再開
5) ３～４を指定回実行（初期設定を含め５回）
6) デバッグ・ツールから切断

点灯パターンはプログラム上では固定しています。それが初期設定です。

そのパターンを Python でレジスタを 4 回書き換えて点灯パターンを確認します。

スクリプトの実行する動作は、まず、デバッグ・ツールに接続し、ブレークポイントを設定します。

ブレーク発生で Python によりレジスタを書き換え、再開します。

その後、ブレーク発生毎、書換えを 4 回繰り返した後、デバッグ・ツールから切断します。

※使用機器：E2 エミュレータ／シミュレータ

評価ボード TESSERA 製 EB-850/F1L-176-S (CPU ボード)

1. 1 つ目のプロジェクトの準備

以下は、1 つ目のスクリプトの事前準備です。

手順：（手順ⅰ、ⅲは 3.1 に記載の手順と重複しますので図は省略します。）

ⅰ. プロジェクトを開く

ⅱ. デバッグ・ツールの設定

ⅲ. Python コンソールの表示と配置変更

ⅳ. スクリプトによるデバッグ実行のための準備（デバッグレイアウトの設定）

手順ⅰ プロジェクトを開く

同梱のプロジェクトを開らきます。

RH850_F1L_GoWaitBreakSetRegister プロジェクトを使用します。フォルダ「デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行う」にパッケージされています。

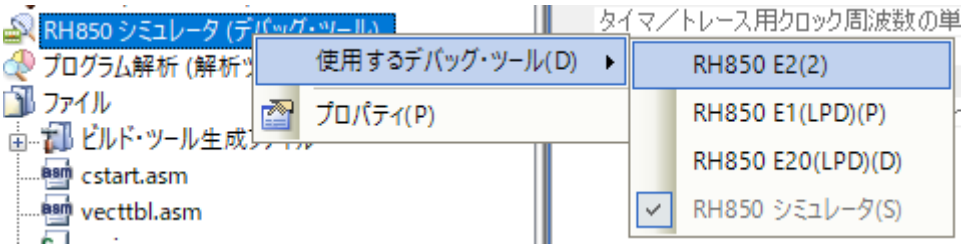
解凍場所は任意ですが、ここではデスクトップとします。解凍した「デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行う」フォルダより

→「RH850_F1L_GoWaitBreakSetRegister」フォルダ

→「RH850_F1L_GoWaitBreakSetRegister.mtpj」をダブルクリックしてプロジェクトを開きます。

手順ⅱ デバッグ・ツールの設定

デバッグ・ツールを E2 エミュレータ、またはシミュレータに設定します。



E2 エミュレータの場合は、評価ボードに接続します。また、メイン・クロックの設定、電源供給など、プロパティで接続条件を確認してください。

（シミュレータの場合、LED 動作は確認できませんが、別途 LED 動作を Excel で確認できるプロジェクト(Appendix)を用意してます。）

手順ⅲ Python コンソールの表示と配置変更

(1) [表示]メニューの[Python コンソール]を選択します。

(2) Python コンソールパネルが表示されます。

手順ⅳ スクリプトによるデバッグ実行のための準備（デバッグレイアウトの設定）

Python スクリプト実行中に、Python コンソールの表示をリアルタイムで確認したい場合の設定です。デバッグ実行時にはレイアウトが専用になるため、スクリプト実行前に事前に設定しておく必要があります。

同時に、スクリプトをデバッグする際の変数やレジスタ値を確認する準備(ウォッチドッグの設定)も行います。

(1) デバッグ・ツールに接続します。

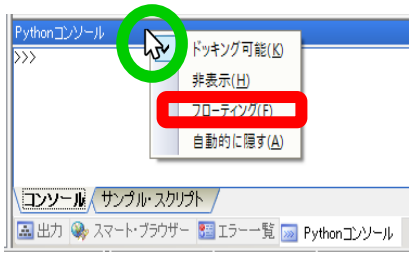
CS+の右上の  （デバッグ・ツールへプログラムをダウンロードします）ボタンを押すと、デバッグ・ツールへ接続されプログラムがダウンロードされます。

(2) Python コンソールを表示、任意の位置に移動配置します。

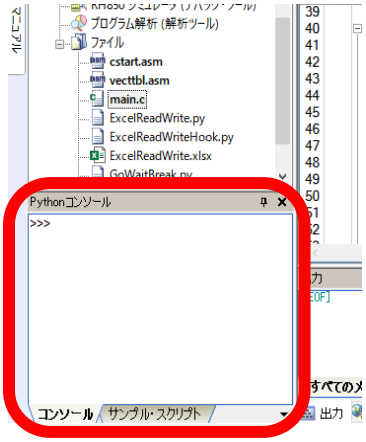
スクリプト実行中に Python の出力を確認するためです。(本手順は 3.2 と一部重複します。)

① [表示]メニューの[Python コンソール]を選択します。

② Python コンソールのタイトル・バー上で**右クリック**し、**フローティング**を選択します。



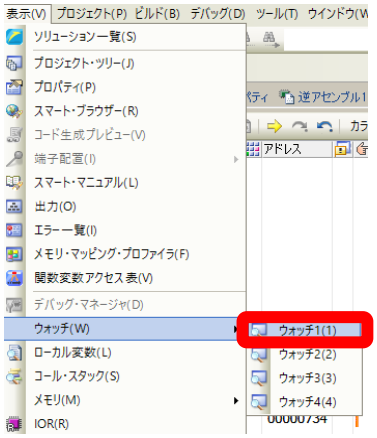
③ タイトル・バーをつかみプロジェクト・ツリー下に移動します。



(3) 変数の値を実行中に GUI 上で観察するためウォッチパネルを表示し、ウォッチ式を登録します。

ここで観察する変数は、wait, g_count, P8 です。

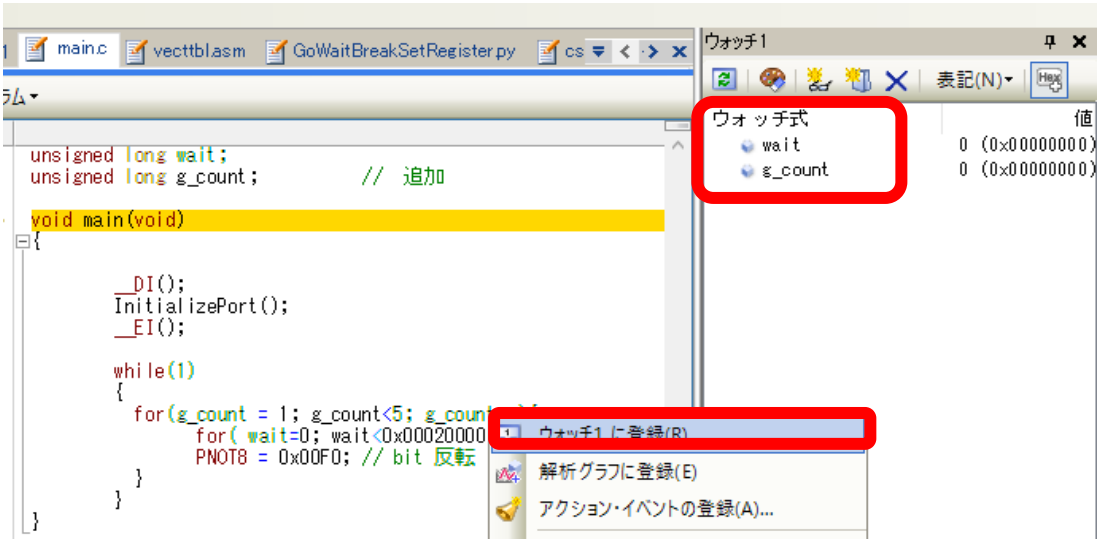
① [表示]メニューの[ウォッチ] ⇒ [ウォッチ 1] を選択します



② ウォッチ式を登録します。

エディタパネルの main.c の記述にある wait, g_count, P8 のそれぞれの変数上で右クリックし、表示されたコンテキストメニューの一番上にある[ウォッチ 1 に登録]を選択します。

ウォッチパネルに登録されます。



(4) デバッグ・ツールから切断します。

登録が終わりましたら、CS+の右上の切断ボタン  を押してデバッグ・ツールから切断します。

これで準備完了です。

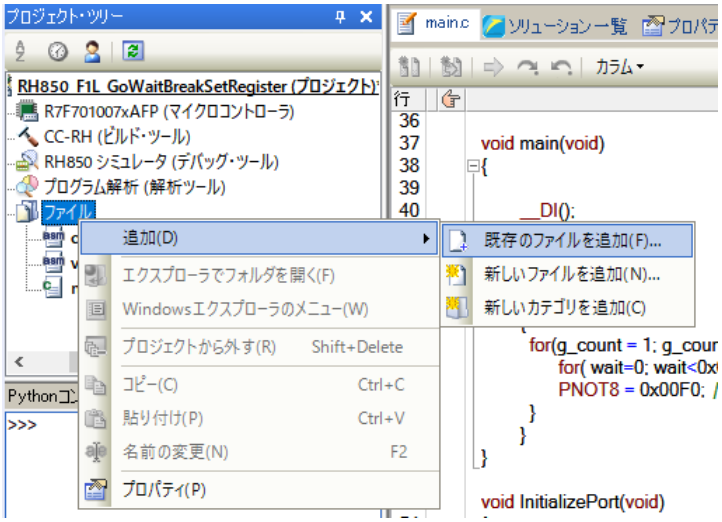
2. スクリプト 1 の実行

スクリプトの解説の前に、スクリプトの動作を確認します。

手順 i. スクリプトの登録

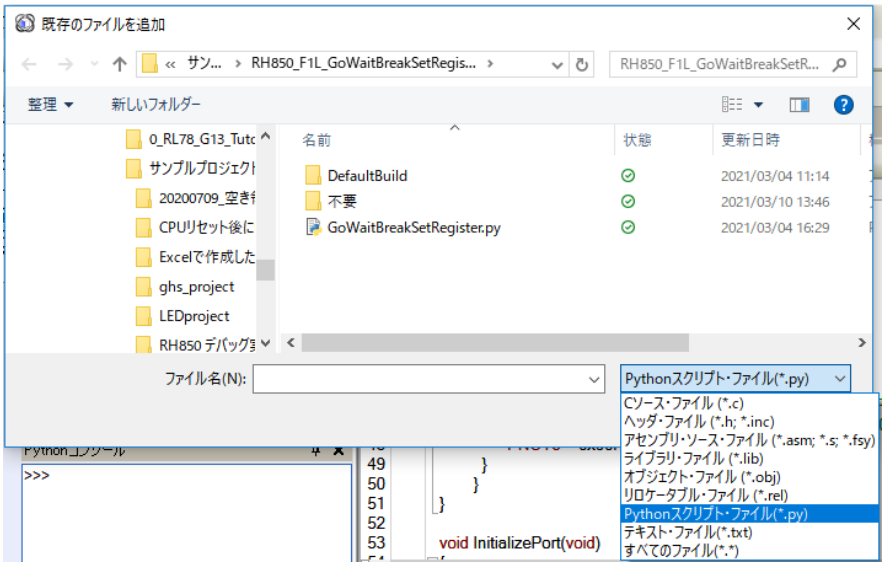
（プロジェクトでは既に登録済みですが、未登録の場合の操作です。）

(1) プロジェクト・ツリーのファイル上で右クリックして、コンテキストメニューから、[追加] ⇒ [既存のファイルを追加]を選択します。

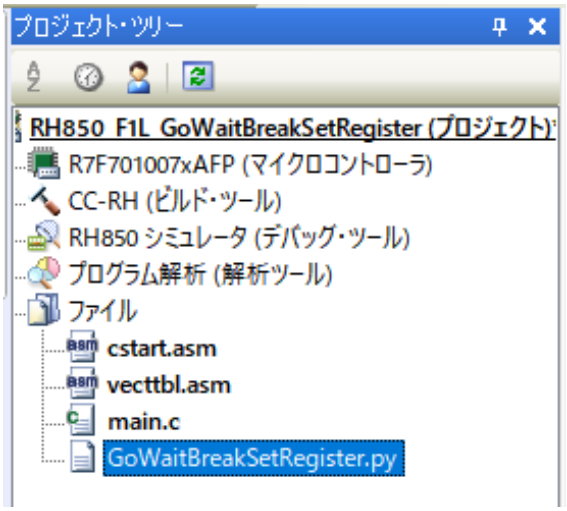


(2) 「既存のファイルを追加」ダイアログが開くので、ファイルの種類を Python スクリプトファイル(*.py)を選択します。

GoWaitBreakSetRegister.py を選択します。

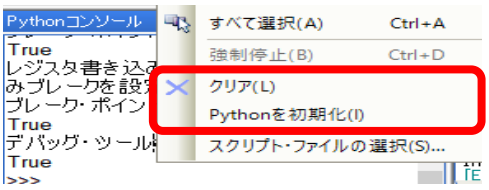


(3) スクリプトファイル GoWaitBreakSetRegister.py が登録されます。



(4)　スクリプト実行前に、Python コンソールの初期化と画面のクリアを行います。
Python コンソール上で右クリック

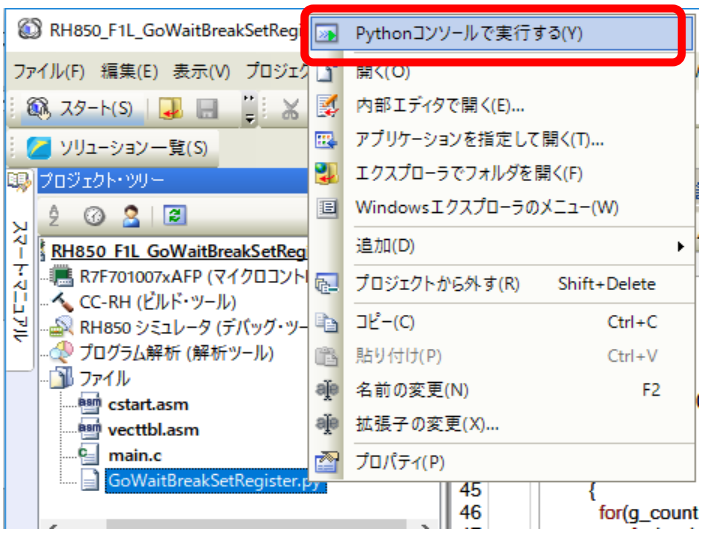
（この作業は 2 回目以降、コンソールに前の出力が残ってた場合、あるいは、スクリプトの実行でエラーが発生した場合などメモリーをクリアするために実施します。）



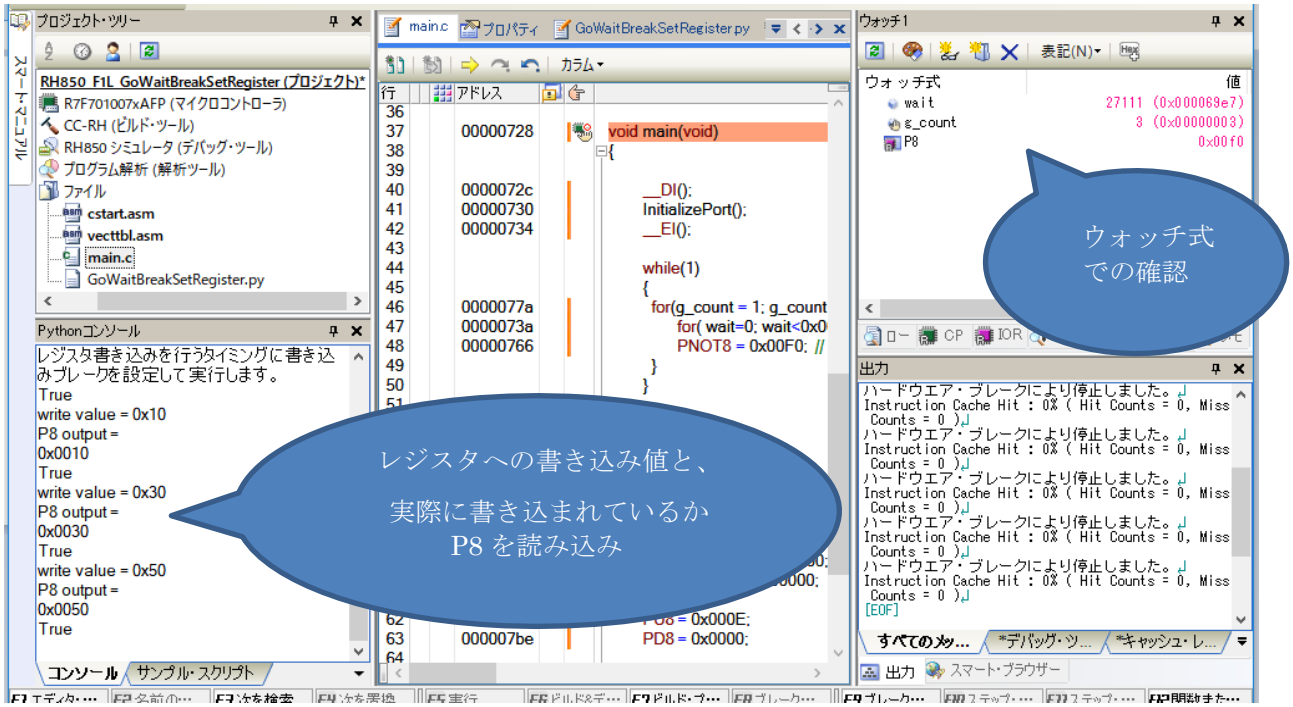
(5)　GoWaitBreakSetRegister.py を実行します。

プロジェクト・ツリーの GoWaitBreakSetRegister.py 上で右クリックしコンテキストの一番上の、[Python コンソールで実行する]を選択。

スクリプトが実行され、ボードの LED 点灯パターンが変って行く事が確認できます。



実行中のキャプチャです。



Python コンソールには、レジスタへの書き込み値と、その値が実際に書き込まれているのかを P8 を読み込んで表示させています。

ウォッチパネルには、登録した wait, g_count, P8 のそれぞれの変数がリアルタイムで表示されます。

3.　プログラムとスクリプトの解説

ここは、プログラムとスクリプトの解説を行います。まず、プログラムからです。

プログラムの解説

（概要：ポート 8 に接続した LED を点滅させる。）

main.c

```
//ポートレジスタ 8
#define P8 (*(volatile unsigned short *)0xFFC10020)

main(){
while(1)
 {
  for(g_count= 1; g_count<21; g_count++){
   以下を 20 回繰り返し
  for( wait=0; wait<0x00020000; wait++ );
  P8 = P8 ^ 0x00F0;
 }
}
```

ポート 8 の bit4-7 に LED を接続

・レジスタ定義：
- P8：ポートレジスタ 8　出力
　　LED 点灯パターンを書込み
　　ブレーク時スクリプトで書換え

・for(g_count=1, ...)　：一つの LED 点灯パターンの点灯を 10 回点滅させるため、g_count= 21 でブレークを設定

・for(wait=0; ...)：点滅のタイマーとして使用

・P8 レジスタ bit4-7 を反転
　　bit　反転　⇒　LED 反転

このプログラムはポート 8 の bit4 から 7 に接続した LED を点滅させます。ポート関連のレジスタとして、ポートの出力を設定するポートレジスタ 8、P8 レジスタに LED の点灯パターンを書込みます。ブレーク発生時に、スクリプトで書き換えて点灯パターンを変えます。プログラムでの設定値は 0xF0 で全点灯です。

LED の点滅は変数 wait をカウントしている for 文をタイマーとして使用し、P8 レジスタのビット 4-7 を bit 反転すると、P8 の出力が反転し、LED を点滅させます。

g_count の for 文は、一つの LED 点灯パターンを何回点滅させるかを設定します。ここでは、10 回点滅させるため、g_count は 20 回カウントした後、 21 に達したところでブレークが発生するように、スクリプトでブレークポイントを設定します。なお、ここで、g_count を 1 から始めていますが、0 から始めると Python コンソールに出力する読込み値が反転データではなく、書き込み値と同じ値になることを確認できます。

ブレークが発生したら、スクリプトで P8 レジスタを書き換えて点灯パターンを変更します。

スクリプトの解説

GoWaitBreakSetRegister.py です。

このスクリプトは、デバッガ接続 ⇒ ブレーク設定 ⇒ 実行５回 ⇒ 切断　を行います。

解説は、(1) 実行関連の制御、(2) ブレーク設定、(3) 繰り返し処理 に分けて説明します。

なお、説明のため関連コードをまとめてますので、実際のコードと流れが異なります。

(1) 実行関連の制御

debugger.Connect()	接続
debugger.Download.LoadModule()	プログラムダウンロード
debugger.Reset()	リセット
debugger.Disconnect()	切断

(2) ブレーク設定

bp =BreakCondition()	インスタンスを生成
bp.Address=“g_count“	ブレークを設定するアドレスを指定（今回は“g_count”）
bp.BreakType=BreakType.Write	ブレークのタイプを書き込みブレークに設定
bp.Data=21	データのブレーク条件を設定する数値を指定
debugger.Breakpoint.Set(bp)	ブレークポイントをセット
debugger.Go(GoOption.WaitBreak)	プログラムが停止するまでスクリプト待機

BreakCondition はブレーク条件をまとめたクラスです。変数 bp をインスタンスとします。

BreakCondition クラスには変数が、Address、BreakType、Data、AccessSize の 4 つがありますが、AccessSize を除いた 3 つを設定します。

プログラムで説明しましたが、LED 点灯パターンを書き換えるため LED が 10 回点滅後、ブレークするように設定します。そのために、変数 g_count に 21 が書き込まれた時に、ブレークするように書き込みブレークを設定します。ブレークさせる Address は、変数 g_count を設定します。

設定した条件により、debugger.Breakpoint.Set(bp)でブレークポイントをセットします。

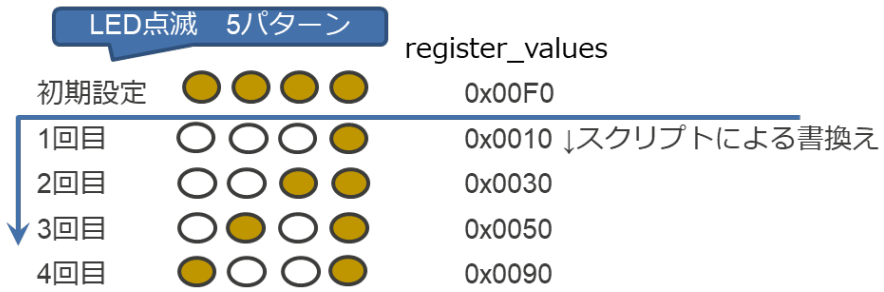
(3) 繰り返し処理（レジスタ書き込み、再開、モニター出力)

繰り返し処理の部分の説明です。

ここでは、ブレーク後に、レジスタの書き込み、実行の再開、モニターへのレジスタ値の出力を繰り返しています。

事前に使用する変数を 2 つ設定します。register_name に書き込み対象の P8 を設定します。それから、配列として register_values に LED の点灯パターンを 4 つ設定します。

```
#設定：
# 書き込み対象のレジスタ名
register_name="P8"                    # P8 :ポートレジスタ 8 出力ポート
# 書き込む値(LED の点灯パターンを変える)
register_values = [0x0010, 0x0030, 0x0050, 0x0090]
```



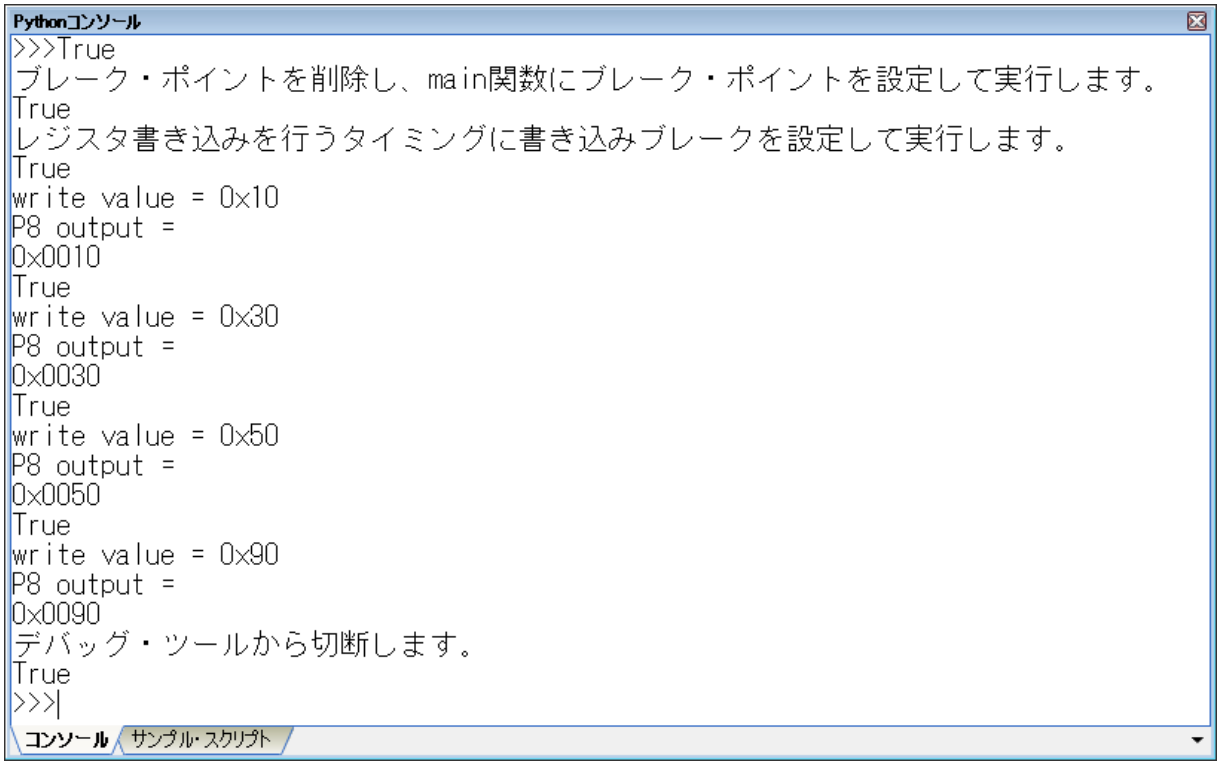
左の図にあるように、プログラムの初期設定の値、0x00F0 は初期値でプログラム内に設定されています。その後、4 つのパターンをブレーク発生毎に書き換えます。

```
以下を繰り返し
# 配列 register_values の先頭データから
  変数 value に読み込む
for value in register_values:                          #レジスタ書き込み
    debugger.Register.SetValue(register_name, value)   # ”P8”に配列 register_values を順次書込む
    debugger.Go(GoOption.WaitBreak)                    #再開(次のブレーク待ち)
    # Python コンソールにモニター出力
    output ="write value = 0x%x"%value                 # 書き込む値
    print(output)
    #P8 を読み込み書き込まれたことを確認
    print "P8 output = “
    debugger.Register.GetValue(register_name)          #参照するレジスタ名は直接指定も可能
```

書き換えの繰り返し処理は for 文ループで配列 register_values から、一つずつ変数 value に取り込み、debugger.Register.SetValue 関数に設定、レジスタ値を書き換え後、再実行して次のブレークを待ちます。

それから、ループ内で print 文を使って Python コンソールに書き込み値とレジスタ値を出力して、モニターしています。debugger.Register.GetValue はレジスタ値を参照し、print 文なしで python コンソールに値を返します。

Python コンソール上でのモニター出力結果が以下になります。書き込み値と、P8 レジスタから読み込んだ値を、出力させたものです。



この出力結果は、プログラムのところで触れました、for 文の中の g_count の始まりを 0 に設定したものです。書き込み値と読込み値が同じ値になります。g_count の始まりを 1 にした場合、反転値になります。

以上が、スクリプト 1 の解説です。

4.2 スクリプト 2

以下、スクリプト 2 の概要です。

Excel ファイルのテストデータを使用したファジングテストを自動的に実行します。ファジングテストとはランダムなデータを入力しプログラムの挙動を確認するテスト手法の 1 つです。

このスクリプトの目的は、以下になります。

・プログラム内の変数 test_count にテストデータ（ランダムデータ）を入力してファジングテストを行う。

・テストデータの設定と記録を Excel で行う。

・上記を連続してテストする。

実行する動作は以下になります。

1) ブレーク発生で Excel から**テストデータを読み込む**

2) 変数 test_count を 1)のテストデータで書き変え

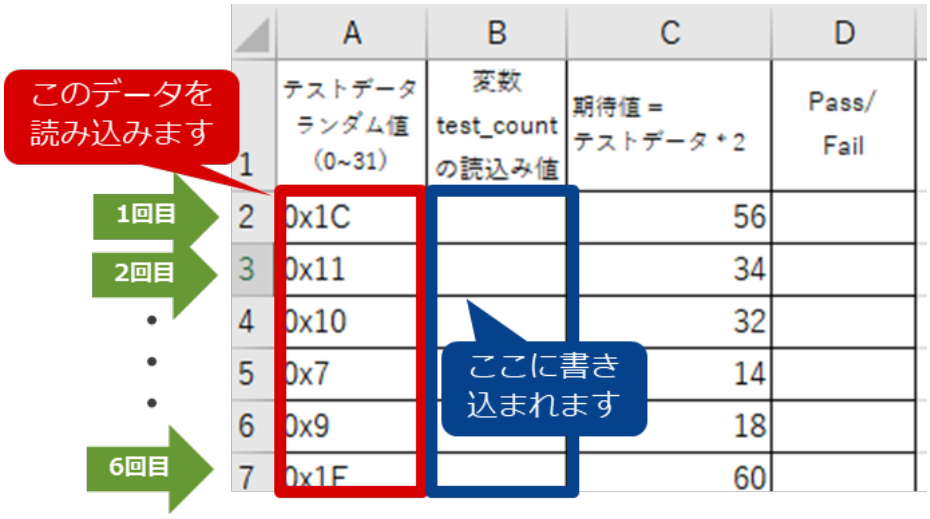
3) 変数 test_count の演算（プログラム内）

4) 次のブレークで変数 test_count を読み込み、**Excel に書き込む**

5) 1～4 を指定回（16 回）実行

＊上記を自動実行させる。(自動実行はスクリプト 1 の応用です）

下図はデータの設定と記録のためのエクセルです。



	A	B	C	D
1	テストデータ ランダム値 (0~31)	変数 test_count の読込み値	期待値＝ テストデータ＊2	Pass/ Fail
2	0x1C		56	
3	0x11		34	
4	0x10		32	
5	0x7		14	
6	0x9		18	
7	0x1F		60	

ここで、テストに使用するランダムデータを Excel の A 列から読み込み、処理(test_count *= 2)の結果を Excel の B 列に書き込み、事前に計算した期待値 C 列と比較、結果の判定を D 列に表示させます。

これを、Excel に記述されたデータ数繰り返すスクリプトです。このスクリプトと、プロジェクト 1 の自動実行スクリプトを組み合わせます。

このプロジェクトは MCU の周辺機能は使用しませんので、デバッグ・ツールは RH850 シミュレータで動作します。

2 つ目のプロジェクトのファイル構成

プロジェクト 2 はファイル構成が少し複雑なのでそれを説明します。

プログラム：

- main.c ：

  main()にテストしたい変数 test_count とその処理 test_count*= 2 を記載してます。

スクリプト：（3 つあります）

- ExcelReadWrite.py ：

  エクセルの設定、操作用のユーザー関数(open, close, read, write) の定義、Hook 関数の登録などを行います。デバッグ前に一回実行してメモリーに取り込まれます。

- ExcelReadWriteHook.py ：

  ブレーク時に呼び出される Hook 関数を記載します。ExcelReadWrite.py 実行時に登録され一緒にメモリーに取り込まれます。Hook 関数は特定イベントで実行され、名前が決まっています。例えば、AfterCpuStop()はブレーク後に実行、AfterCpuReset()は CPU リセット後に実行などです。

- AutoExecute.py ：

  自動実行スクリプトです。スクリプト 1 を応用したものです。ブレークポイント設定、デバッグ・ツール接続、切断などを実施します。ExcelReadWrite.py は手動デバッグでも、動作します。

エクセルファイル：

- ExcelReadWrite.xlsm ：

  データ処理用です。変数 test_count に書込むためのランダムなテストデータ作成と処理後の test_count 読込みと記録、期待値との比較・判定ができるように事前に作成しています。

1．2 つ目のプロジェクトの準備

以下は、2 つ目のスクリプトの事前準備です。

手順：（手順ⅰ、ii は 3.1 に記載の手順と重複しますので図は省略します。）

ⅰ． プロジェクトを開く

ii． Python コンソールの表示と配置変更

iii． スクリプトによるデバッグ実行のための準備（デバッグレイアウトの設定）

手順ⅰ． プロジェクトを開く

同梱のプロジェクトを開らきます。

RH850_F1L_ExcelReadWrite プロジェクトを使用します。フォルダ「Excel ファイルのテストデータを使用したファジングテストを自動的に実行」にパッケージされています。

解凍場所は任意ですが、ここではデスクトップとします。解凍した「Excel ファイルのテストデータを使用したファジングテストを自動的に実行」フォルダより

→「RH850_F1L_ExcelReadWrite」フォルダ

→「RH850_F1L_ExcelReadWrite.mtpj」をダブルクリックしてプロジェクトを開きます。

手順 ii Python コンソールの表示と配置変更

(1) [表示]メニューの[Python コンソール]を選択します。

(2) Python コンソールパネルが表示されます。

手順 iii スクリプトによるデバッグ実行のための準備（デバッグレイアウトの設定）

Python スクリプト実行中に、Python コンソールの表示をリアルタイムで確認したい場合の設定です。デバッグ実行時にはレイアウトが専用になるため、スクリプト実行前に事前に設定しておく必要があります。

同時に、スクリプトをデバッグする際の変数やレジスタ値を確認する準備(ウォッチドッグの設定)も行います。

(1) デバッグ・ツールに接続します。

CS+の右上の  （デバッグ・ツールへプログラムをダウンロードします）ボタンを押すと、デバッグ・ツールへ接続されプログラムがダウンロードされます。

(2) Python コンソールを表示、任意の位置に移動配置します。

スクリプト実行中に Python の出力を確認するためです。(本手順は 3.2 と一部重複します。)

① [表示]メニューの[Python コンソール]を選択します。

② Python コンソールのタイトル・バー上で右クリックし、フローティングを選択します。

③ タイトル・バーをつかみプロジェクト・ツリー下に移動します。

(3) 変数の値を実行中に GUI 上で観察するためウォッチパネルを表示し、ウォッチ式を登録します。

ここで観察する変数は、wait, g_count, test_count です。

① [表示]メニューの[ウォッチ] ⇒ [ウォッチ 1] を選択します

② ウォッチ式を登録します。

エディタパネルの main.c の記述にある wait, g_count, test_count のそれぞれの変数上で右クリックし、表示されたコンテキストメニューの一番上にある[ウォッチ 1 に登録]を選択します。

ウォッチパネルに登録されます。

(4) デバッグ・ツールから切断します。

登録が終わりましたら、CS+の右上の切断ボタン  を押してデバッグ・ツールから切断します。

これで準備完了です。

2.. スクリプト２の実行

スクリプトの解説の前に、スクリプトの動作を確認していただきます。

スクリプト２は３つのスクリプトがあります。

手順ｉ．スクリプトの登録

（プロジェクトでは既に登録済みですが、未登録の場合の操作です。）

(1) プロジェクト・ツリーのファイル上で右クリックして、コンテキストメニューから、[追加] ⇒ [既存のファイルを追加]を選択します。

(2) 既存のファイルを追加ダイアログが開きます。

今回は、Excel ファイルも登録しますので、ファイルの種類を「すべてのファイル(*.*)」を選択します。

・ExcelReadWrite.py ⇐ エクセルの設定など

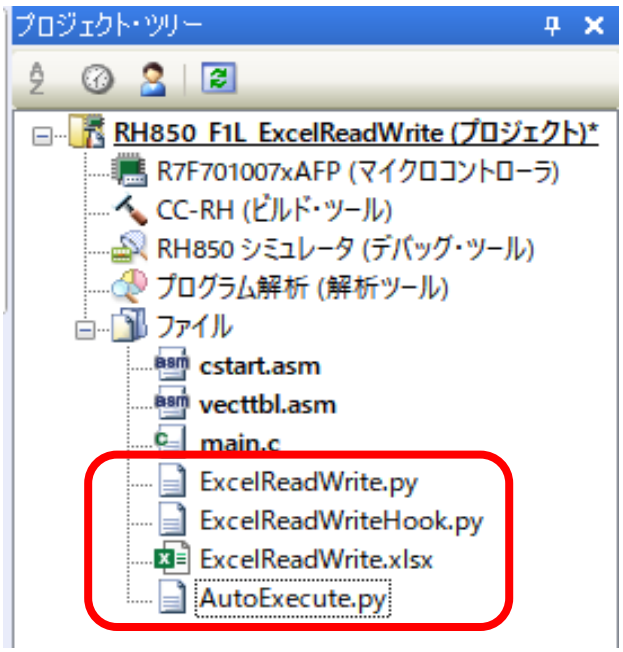
・ExcelReadWriteHook.py ⇐ ブレーク時に呼び出される。

・ExcelReadWrite.xlsx ⇐ データ処理用 Excel ファイル

・AutoExecute.py ⇐ 自動実行スクリプトスクリプト１の応用

以上の４つのファイルを選択します。

(3) 以下の４つファイルを登録します。



(4) Excel ファイル ExcelReadWrite.xlsx を初期化します。

プロジェクト・ツリーExcelReadWrite.xlsx をダブルクリックし、ExcelReadWrite.xlsx ファイルを開きます。

	A	B	C	D	E	F
1	テストデータ ランダム値 (0~31)	変数 test_count の読込み値	期待値 = テストデータ * 2	Pass/ Fail		ランダム発生 (0~31)
2	0x1C		56			0xC
3	0x11		34			0x12
4	0x10		32			0x1E
14	0x1E					
15	0x1B		54			0x1C
16	0xE		28			0x1D

B 列の読込み値に既に値がある場合、クリアします。

新しいランダムデータを使用したいときは、F 列の F2：F16 をコピーして、A 列の A2：A16 に貼り付けます。「値の貼り付け」で貼り付けてください。

※ 同梱のエクセルファイル ExcelReadWrite.xlsm は B 列のクリアと、ランダムデータのコピー、更新のマクロを組み込んでいます。

(5) スクリプト実行前に、Python コンソールの初期化と画面のクリアを行います。

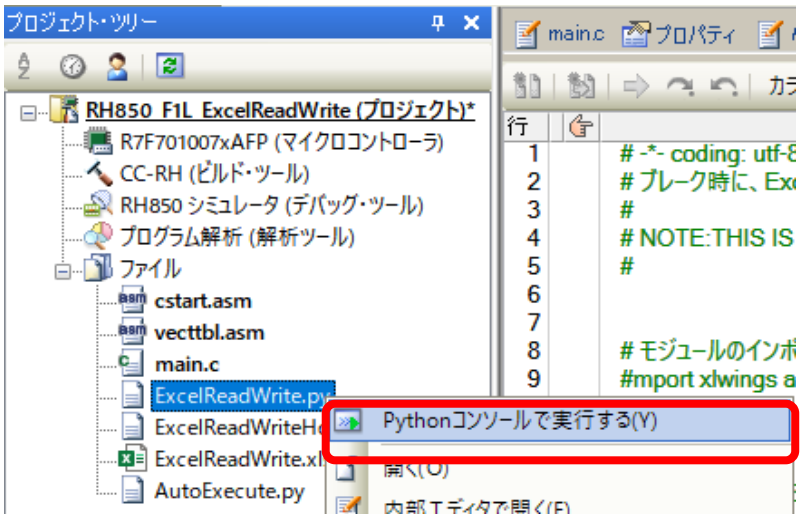
Python コンソール上で右クリック

（必要に応じて実施します）

(6) スクリプトファイル ExcelReadWrite.py を実行します。

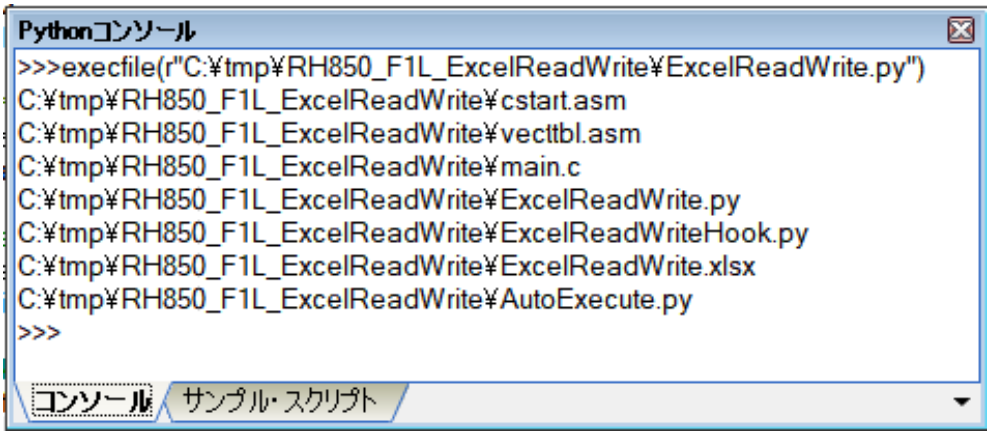
このスクリプトは Excel の設定、ユーザー定義関数を Python に取り込みます。

プロジェクト・ツリーの ExcelReadWrite.py 上で右クリックし、コンテキストメニューの一番上の Python コンソールで実行する(Y)を選択。スクリプトが実行されます。



(7) ExcelReadWrite.py の実行結果

下図は、ExcelReadWrite.py でプロジェクト・フォルダ内の Excel ファイルを検索した事を表示しています。Excel ファイルの Path を取得するためです。

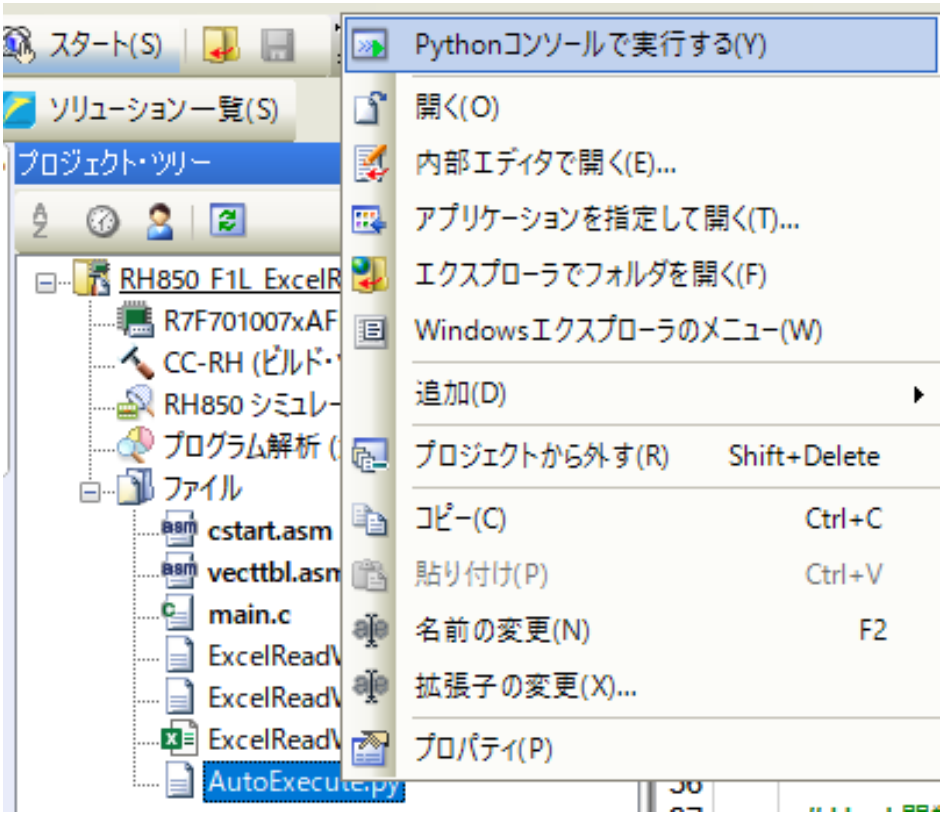


表示されるのはこれだけですが、さらに Excel の設定やユーザー定義関数を Python に取込んでいます。

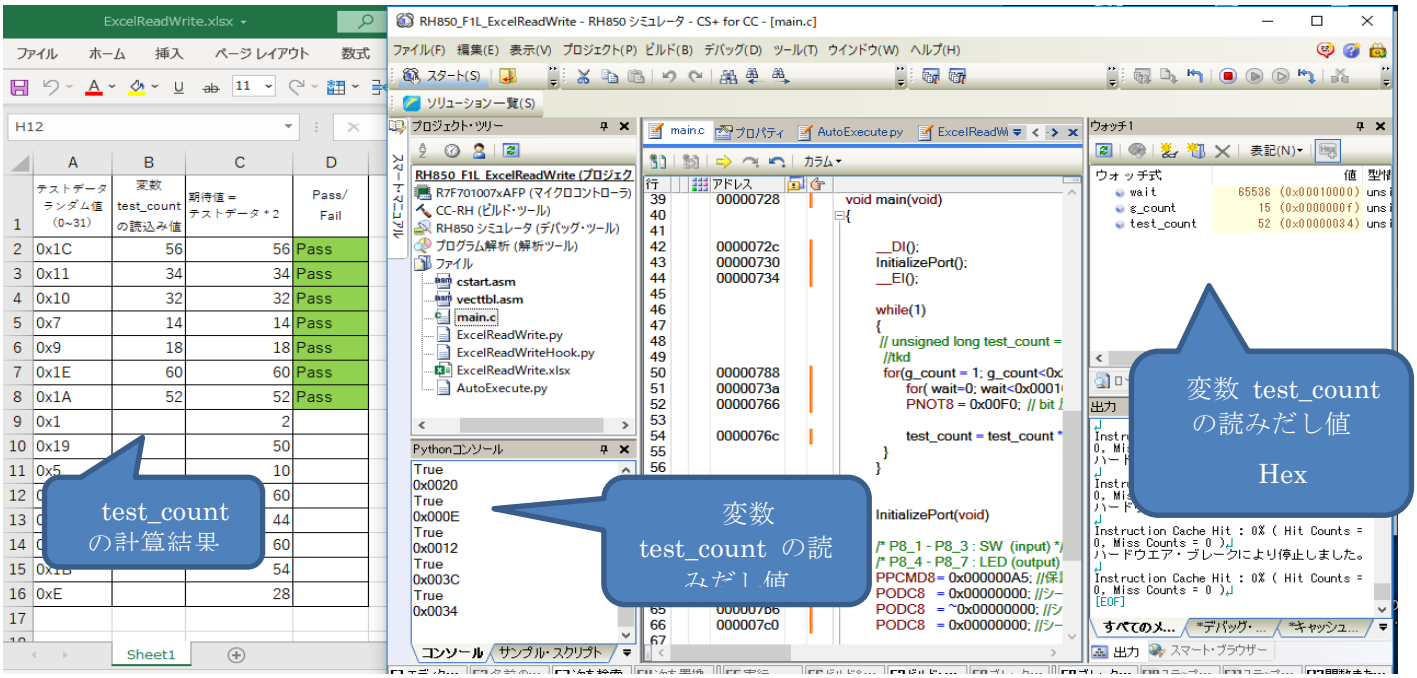
(8) スクリプトファイル AutoExecute.py を実行します。

このスクリプトはデバッグを自動実行します。デバッグ・ツールへの接続、ブレークポイントの設定などです。

プロジェクト・ツリーの AutoExecute.py 上で右クリックし、コンテキストメニューの一番上の Python コンソールで実行する(Y)を選択。スクリプトが実行されます。



(9) AutoExecute.py の実行中のキャプチャ



ブレーク発生でエクセルファイルが開かれ、A 列にあるランダムデータを先頭から読み込み、変数 test_count に書き込みます。次のブレーク発生で処理、つまり 2 倍された test_count を読出し、エクセルの B 列に書き込みます。エクセルでは期待値 C 列と比較し、D 列に判定を表示します。これを、A 列のランダムデータがなくなるまで繰り返し、エクセルをクローズ、デバッガを切断します。

CS+の Python コンソールには、test_count を読出し値が表示されます。

ウォッチパネルでは、変数の動きが確認できます。

3. プログラムとスクリプトの解説

ここでは、プログラムとスクリプトの解説を行います。まず、プログラムからです。

プログラムの解説

（概要：読み書き対象の変数 test_count と式 test_count*= 2 を追加。）

main.c

```
unsigned long   test_count=1;
main(){:
while(1)
 {
   for(g_count= 1; g_count<0x20; g_count++){
      for( wait=0; wait<0x00020000; wait++ );
      PNOT8 = 0x00F0;
      test_count*= 2;
 }
}
```

・変数 test_count 定義

・PNOT8 への書き込みでブレークを設定

・ test_count を 2 倍する処理

このプロジェクトは変数 test_count のファジングテストを目的としていますので、プログラムの説明はその部分に限ります。まず、変数 test_count を定義し、テストしたい処理、すなわち test_count を 2 倍する処理を追加しています。

スクリプトで test_count に書き込み／読み込みを実行するためにブレークポイントでプログラムをいったん停止する必要があります。デバッグ・ツール接続時に設定しますが、ここでは PNOT8 への書き込み時にブレークするように設定します。手動でも可能ですが、今回は自動実行スクリプトの AutoExecute.py で設定します。

スクリプトの解説

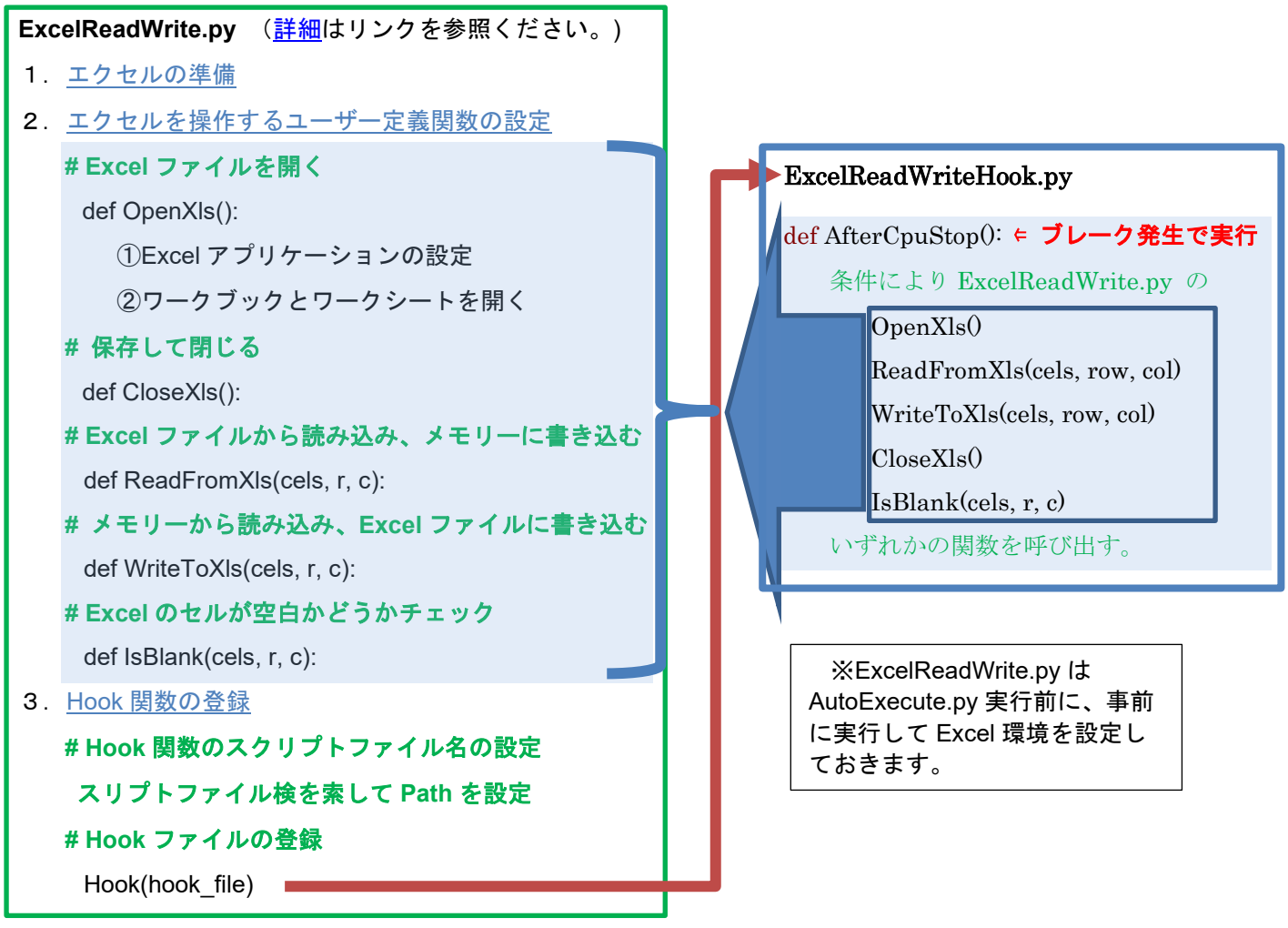
スクリプトは 3 つありますが、最初に ExcelReadWrite.py と ExcelReadWriteHook.py について説明します。この 2 つのスクリプトの組み合わせで Excel を制御しています。

エクセル設定 ⇒ Hook 関数の登録 ⇒ イベント発生で Hook 関数実行 を行います。

ExcelReadWrite.py は、エクセルの準備、エクセルを操作するユーザー定義関数の設定、それからブレーク時に発動する Hook 関数の登録を行います。ユーザー定義関数には、エクセルを開く関数、保存して閉じる関数、エクセルへ読み書きする関数が定義されています。フック関数には ExcelReadWriteHook.py ファイルを登録します。

ブレーク発生で ExcelReadWriteHook.py に移動し、その時の状態、条件に合わせて、ExcelReadWrite.py にあるユーザー定義関数を呼び出します。

ExcelReadWrite.py と ExcelReadWriteHook.py の関係



AutoExecute.py について

AutoExecute.py は自動実行をコントロールします。

スクリプト 1 の GoWaitBreakSetRegister.py を応用したものです。

- 実行関連の制御
- ブレークポイントの設定
- 繰返し処理

からなります。

実行関連の制御とブレークポイントの設定については、スクリプト 1 の解説を参照してください。ここで、ブレークポイントの対象は

PNOT8

に設定しています。

繰り返し処理は For 文に range 関数を使用しています。Range 関数は、中に入る数字、すなわち 0x21 の場合、0 から 0x20 までの要素を指定、配列表現の代わりになります。

繰り返し処理

ブレーク回数のチェック、再開、モニター出力について繰り返します。

```
以下を繰り返し
# break_count 変数に range 関数で 0 から 0x20 までの要素を指定
  range(0x21) は [ 0x0, 0x1, 0x2, … , 0x19, 0x20] と同等
for break_count in range(0x21):
    #再開(次のブレーク待ち)
    debugger.Go(GoOption.WaitBreak)
    # Python コンソールにモニター出力
    print("実行回数= 0x%x"%break_count)
```

※Excel のテストデータは 16 (0x10)個用意しています。

ブレーク回数は処理対象の変数 test_count への書き込みと、処理後の読込みの 2 回のブレークが必要です。そのため、break_count を 0x10 * 2 + 1 =0x21 としています。

フローチャート（プログラムとスクリプトの関係）

次のページにユーザー定義関数の呼び出しの関連をフローチャートで示します。また、AutoExecute.py の関わりも記載してます。

AutoExecute.py は実行すると、デバッグ・ツールを接続、プログラムをダウンロード、ブレークポイントを設定して、プログラムを起動します。

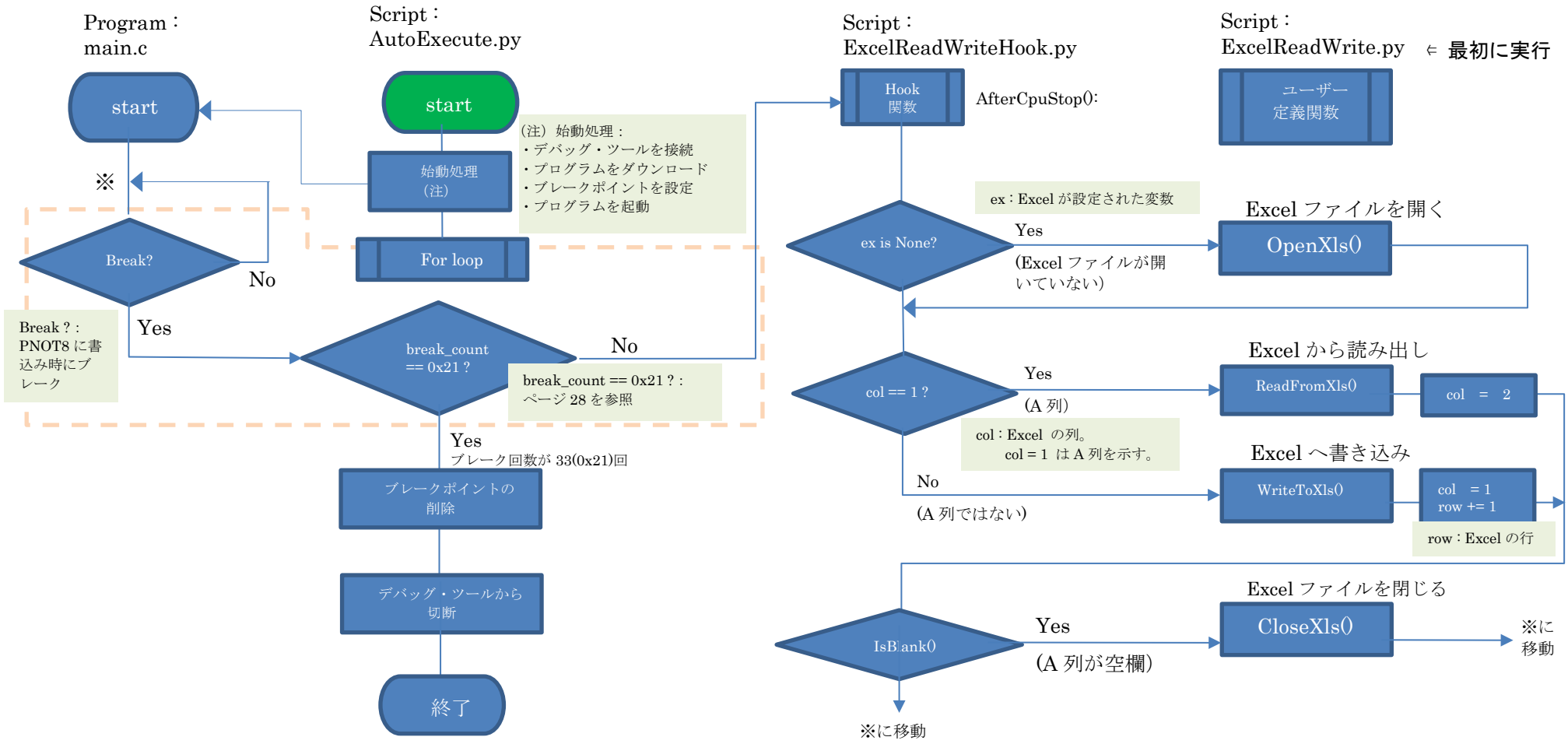
フローチャートでは、まとめて始動処理としています。

そのあと、For loop があり、ここで、ブレークした回数をカウントし、プログラムを再実行します。

ブレーク回数が 0x21、すなわち 33 回で For loop を抜け、ブレークポイントを削除、デバッグ・ツールから切断、終了します。

ループ回数は少し多めに設定しています。

プログラムとスクリプトの関係フローチャート



[スクリプトのデバッグに使えるヒント](#)を Appendix に記載しました。

スクリプトの詳細

各スクリプトの詳細を示します。

ExcelReadWrite.py

ExcelReadWrite.py の詳細１：エクセルの準備の部分

ExcelReadWrite.py にあるエクセルの準備に関係する部分をまとめています。

左側は、Import 文で、OS の操作をするためのモジュールと、.NET Framework で使用されるライブラリモジュールをインポートします。さらに、Excel が使えるように登録します。

右側は、Excel ファイルの関連変数を定義、設定しています。Excel のファイル名、ワークブック、シート、セル、行、列、それぞれに設定します。

```
#Python 上で OS の操作を実現するためのモジュール
Import os

# .NET Framework モジュールのインポート
Import clr

# .以下 NET アセンブリを読み込み、Python にインポート
clr.AddReference("office")

clr.AddReference("Microsoft.Office.Interop.Excel")

from Microsoft.Office.Coreimport*

from Microsoft.Office.Interop.Excelimport*

from System.Runtime.InteropServicesimport*
```

```
# Excel ファイル関連変数の定義、設定

#変数 ex にエクセルを設定
ex    = ApplicationClass()

#ワークブック
wb    = ex.Workbooks.Open(xls_file)

#ワークシート
wss    = wb.Worksheets

#sheet1
ws    = wss[1]

#セル
cels    = ws.Cells

#行の初期設定
row    = 2

#列の初期設定
col    = 1

# Excel ファイル名
xls_file_name    = "ExcelReadWrite.xlsm"
```

（次ページへ）

ExcelReadWrite.py

ExcelReadWrite.py の詳細２：エクセルを操作するユーザー定義関数の設定部分

ExcelReadWrite.py にある Excel を操作するユーザー定義関数をそれぞれまとめています。

Excel ファイルを開 OpenXls()、保存して閉じる CloseXls()、 Excel から読出しメモリー(変数)に書込む ReadFromXls()、 メモリー(変数)から読込み、Excel に書込む WriteToXls() があります。

 # Excel ファイルを開く関数

```
def OpenXls():
    # Excel ファイルを検索
    xls_file=""
    for path inproject.File.Information():
        if os.path.basename(path) ==xls_file_name:
            xls_file=path
    # Excel のワークブックとワークシートを開く
    wb=ex.Workbooks.Open(xls_file)
    wss=wb.Worksheets
    ws=wss[1]
    cels=ws.Cells
```

 # Excel ファイルを保存して閉じる関数

```
def CloseXls():
    #  ワークブック(ファイル)保存
    wb.Save()
    #メモリーの開放
    Marshal.ReleaseComObject(cels)
    Marshal.ReleaseComObject(ws)
    Marshal.ReleaseComObject(wss)
    #  ワークブック(ファイル)を閉じる
    wb.Close(False)
    #メモリーの開放
    Marshal.ReleaseComObject(wb)
    #  エクセルを終了
    ex.Quit.()
    #メモリーの開放
    Marshal.ReleaseComObject(ex)
```

```
# 書き込み対象アドレス#test_count の設定
write_address = “test_count”   # 実アドレス 0xfede0000 でも可
```

 # Excel から読出し、メモリー(変数)に書込む関数

```
def ReadFromXls(cels, r, c):
    cell = cels[r, c]   # [ r：行、c：列 ]でセルを特定
    # v にセル値を読み込み
    v = cell.Value[XlRangeValueDataType.xlRangeValueDefault]
    Marshal.ReleaseComObject(cell)   # メモリーの開放
    #  メモリーへの書き込み
    debugger.Memory.Write(write_address, int(v, 0),MemoryOption.HalfWord)
```

```
# 読み込み対象アドレス #test_count の設定
read_address = “test_count”   # 実アドレス 0xfede0000 でも可
```

 # メモリー(変数)から読込み、Excel に書込む関数

```
def WriteToXls(cels, r, c):
    cell = cels[r, c]   # [ r：行、c：列 ]でセルを特定
    # v にメモリー(変数)値を読み込み
    v = debugger.Memory.Read(read_address, MemoryOption.HalfWord)
    cell.Value = v   # エクセルセルに書き込み
    Marshal.ReleaseComObject(cell)   # メモリーの開放
```

ExcelReadWrite.py & ExcelReadWriteHook.py

ExcelReadWrite.py の詳細３：HOOK 関数関連部分 & ExcelReadWriteHook.py の詳細

ここでは、Hook 関数に関連するものをまとめています。

ExcelReadWrite.py で Hook 関数の登録を行っています。Hook 関数の登録は、フック関数を記述するスクリプトファイル名 ExcelReadWriteHook.py で指定します。

Hook 関数 AfterCpuStop()は、プログラムのブレークで実行される指定された関数です。

Excel ファイル関連変数の状態により、ユーザー定義関数が選択され実行されます。

- Excel ファイルを開 OpenXls()
- 保存して閉じる CloseXls()
- Excel から読出しメモリー(変数)に書込む ReadFromXls()
- メモリー(変数)から読込み、Excel に書込む WriteToXls()

があります。

ExcelReadWrite.py

```
# Hook 関数のスクリプトファイル名
hook_file_name = "ExcelReadWriteHook.py"

# Hook 関数のスクリプトファイルを検索
hook_file = ""
for path in project.File.Information():
    if os.path.basename(path) == hook_file_name:
        hook_file = path

# Hook 関数の登録
Hook(hook_file)   #パスを含めたファイル名を指定
```

ExcelReadWriteHook.py

```
def AfterCpuStop(): ⇐ ブレーク後実行される指定された関数
    #  条件分け
    if row == 0:    # 行の初期値 2 が設定されてない場合
        return    # 何もしない
    if ex is None:     # Excel が開いていない場合
        OpenXls()      # Excel を開く
    if col == 1: # 1 列目(テストデータ)の場合
        # 1 列目のセルから読込み、メモリーに書込む
        ReadFromXls(cels, row, col)
        col = 2 # 2 列目(結果の書き込み列)に設定
    else:     # 2 列目(結果の書き込み列)の場合
        #  メモリーから読込み、2 列目のセルに書込む
        WriteToXls(cels, row, col)
        col = 1    # 1 列目(テストデータ)に戻す
        row += 1    #  次のテストデータの行に移る
    if IsBlank(cels, row, 1): # 1 列目のセルにデータがない場合
        #読み書きを終了し、Excel を閉じる
        CloseXls()
        row = 0
        col = 0
```

5. CS+と Python スクリプトのコマンドライン操作

CS+と Python スクリプトをコマンドラインから実行する方法をご紹介します。

CS+ Python スクリプトは，コマンドライン（Windows のコマンドプロンプト）から起動したり、CS+のメイン・ウインドウ(GUI)を表示せずに実行させるなど、コマンド入力方式で操作することができます。

検査現場などで、CS+(GUI)を立ち上げることなく、連続でデバイスをテストするなど(CI/CD)の用途に最適です。

以下の書式にて Windows のコマンドプロンプトからコマンドを実行できます。

書式： CS+をコマンドラインで操作する | V8.08.00 (renesas.com)

CubeSuite+.exe /ps [スクリプトファイル名] [プロジェクト・ファイル名]

例：(プロジェクト 2 を例として)

CubeSuite+.exe /ps AutoExecuteWithExcelpy.py RH850_F1L_ExcelReadWrite.mtpj

(CS+を表示させる場合)

CubeSuite**W**+.exe /ps AutoExecuteWithExcelpy.py RH850_F1L_ExcelReadWrite.mtpj

ここでは新しく AutoExecuteWithExcelpy.py と言うスクリプトを使用します。

これまで、プロジェクト 2 は、3 つのスクリプトの構成で説明してきました。

ExcelReadWrite.py
　　ExcelReadWriteHook.py

AutoExecute.py

スクリプトの実行は、 ExcelReadWrite.py、 AutoExecute.py をそれぞれ分けて実行するようにしていました。スクリプトを分けるメリットは、それぞれ別にスクリプトをデバッグできることにあります。また、順番に実行することで目的の処理を行うことができると言う面白い特徴を知ってもらうためにスクリプトを分けてご紹介しました。

ここで、特にコマンドラインから、2 度もスクリプトを分けて実行するのは不便です。

AutoExecuteWithExcelpy.py は、 AutoExecute.py の先頭に ExcelReadWrite.py を以下のコードを追加して呼び出し、実行させるスクリプトです。

Source("ExcelReadWrite.py")

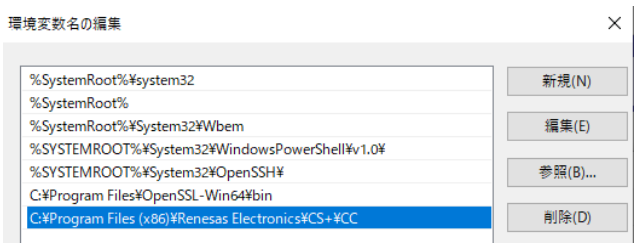
実際の操作

実際の操作方法は 2 つありますが、共通の事前準備があります。

0．Windows 環境変数 path に CS+の実行ファイルのフォルダを登録（事前準備）

Windows10 のシステムのプロパティのダイアログボックスにある詳細設定タブの「環境変数(N)」より、環境変数ダイアログボックスを開きます。システム環境変数の Path を選んで「編集」を押し表示される「環境変数名の編集」ダイアログボックスの「新規」を押すと、Path を入力できます。

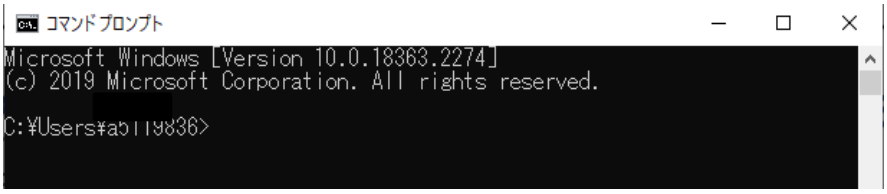
通常は” C:¥Program Files (x86)¥Renesas Electronics¥CS+¥CC”となります。



1．方法① コマンドプロンプトからの実行

一つ目は、コマンドプロンプトから実行する方法です。

1．コマンドプロンプトを開きます



2．プロジェクトのフォルダに移動

3．コマンドの投入、実行

上矢印↑キーで、コマンド再投入が可能です。

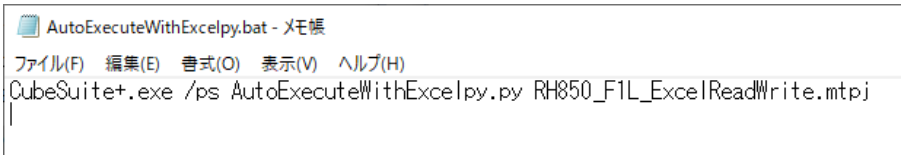
なお、上記の 1，2 の操作は、プロジェクトを開いているエクスプローラのアドレスバーに cmd を打ち込むと同時に可能です。

2．方法② バッチファイルからの実行

二つ目は、バッチファイルからの実行する方法です。

1．バッチファイルの作成

コマンドを記載して、拡張子.bat でプロジェクトのフォルダに保存します。



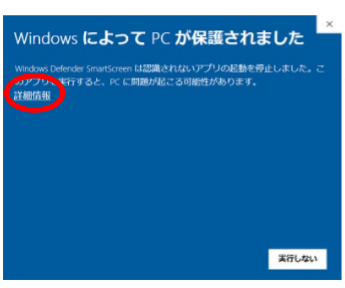
2．バッチファイルをダブルクリックで実行

コマンドプロンプトからバッチファイルの実行も可能です。

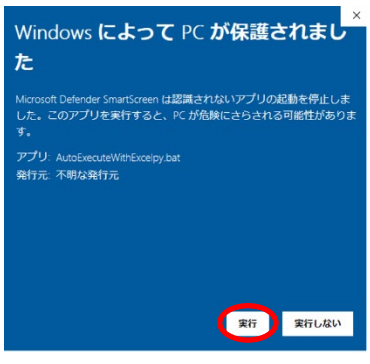
コマンドプロンプトが立ち上がり、そこに、CS+の Python コンソールに表示されていた情報が表示されます。しばらくして、Excel が開き、テストが実施されます。

注：下記のメッセージが表示される場合があります。その場合、詳細情報を押し、次の画面で「実行」を押してください。

「WindowsによってPCが保護されました」と表示される



Windows10で、ダウンロードしたアプリケーションやインストーラーを実行しようとした時に「**Windowsによって PC が保護されました**」「Windows Defender SmartScreen は認識されないアプリの起動を停止しました。」と表示されることがあります。



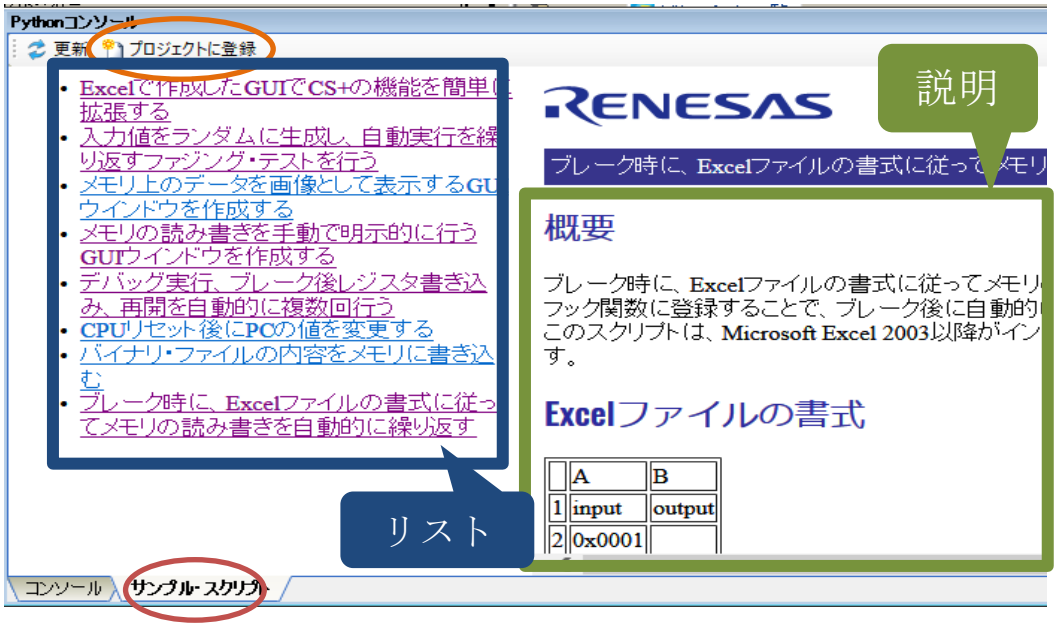
6. 次のステップのために

一通りご紹介してきましたが、最後に、次のステップに進むためのサポートになる情報をご紹介します。サンプル・スクリプトと「かふぇルネ」です。

6.1 サンプル・スクリプト

参考スクリプトとして 9 つ(CS+ V8.08)のスクリプトを用意してます。

1. Python コンソールのタブの右の**「サンプル・スクリプト」タブ**をクリックします。左にリスト、右に説明があります。



2. 任意のサンプル・スクリプトをリストから選んで、左上の「プロジェクトに登録」クリックすると、スクリプトファイル.py や、必要な Excel などのファイルがプロジェクトに登録されます。

6.2 かふぇルネ

もう一つ有用なのはルネサスの開発ツールフォーラム「かふぇルネ」です。

かふぇルネで Python を検索していただくと、様々な Python の情報、使用例、Q&A を見ることができます。また、フォーラムメンバに相談も可能です。

Search -Renesas Rulz-Japan

Appendix

CS+に実装された Python について

IronPython(.NET Framework 上で動作する Python、Python2.7 互換)の実行環境です。

- オプション指定が必要な機能に制限があるため、一部機能(pdb： デバッガ機能など)は使用できません。
- Python 言語以外で記述されたライブラリは使用できません。
- 現行の IronPython が Python2.7 相当のため、それ以降に追加された機能は使用できなません。
- For 文、If 文などの制御文を使いスクリプトでのプログラム作成が可能
- 参照するレジスタ名は定義せずに直接指定可能
- IronPython の言語仕様については，以下の URL を参照してください。

  http://ironpython.net/

  ユーザーズマニュアル：Python コンソール編 (CS+ V8.08.00)

戻る

.NET Framework について

特徴：

- Windows に入っている、プログラムを動かしたり作ったりするときに使える便利な部品
- Windows 用のプログラムを使う人は標準で使用できるモジュール
- どの言語からでも呼び出せます。

以下、ご参考となる解説サイトです。（外部サイトですので、リンクが切れる可能性があります。）

- .NET Framework ドキュメント| Microsoft Docs（マイクロソフト公式サイト）
- .NetFramework(ドットネット)を理解する！初心者でも分かる特徴、C#.NET、VB.NET、ASP.NET との違いや関係性について簡単に解説！｜案件評判(anken-hyouban.com)
- .NET とは何か？：基礎解説：.NET 初心者のための.NET 入門【2011 年版】-@IT (atmarkit.co.jp)

戻る

.NET アセンブリについて

.NET Framework で使用されるライブラリのことを指す

・ファイルの拡張子は*.EXE/*.DLL となる

すなわち、

Windows に入っているプログラムを動かしたり作ったりするときに使える部品（環境）。

その.NET アセンブリを読み込み、Python にインポートする。

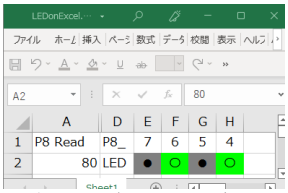
例：

```
Import clr          ⇐.NET Framework の「共通言語ランタイム(CLR：Common Language
                        Runtime)」と呼ばれるプログラム実行エンジンモジュールのインポート
clr.AddReference("office")
clr.AddReference("Microsoft.Office.Interop.Excel")
from Microsoft.Office.Core import*
from Microsoft.Office.Interop.Excel import*
from System.Runtime.InteropServices import*
```

戻る

一つ目のプロジェクトの LED 動作を Excel で確認できるプロジェクト

一つ目のプロジェクトで評価ボードがない場合、シミュレータで動作を確認できますが、LED 動作は見ることができません。そのため Excel で LED 動作を確認できるプロジェクトを用意しました。



「デバッグ実行、ブレーク後レジスタ書き込み、再開を自動的に複数回行う」フォルダの「LED_by_Excel_Pproject」フォルダの「LED_by_Excel.mtpj」をダブルクリックしてプロジェクトを開きます。「LEDonExcel.py」を最初に実行した後で、「GoWaitBreakSetRegister2.py」を実行します。

Excel の制御に関するスクリプトの解説は、スクリプト 2 の解説をご参照ください。

戻る

CI/CD について

「CI」とは「Continuous Integration（継続的インテグレーション）」の略で、ソフトウェア開発におけるビルドやテストを自動化し、継続的に行うアプローチのことです。

「CD」とは「Continuous Delivery（継続的デリバリー）」の略で、CI によってテストされたコードのマージや、本番環境向けのビルドの作成を自動的に行い、本番環境にデプロイが可能な状態を整えるプロセスのことです。

また「CD」は「Continuous Deployment（継続的デプロイ）」の略として使われることもあります。継続的デプロイは継続的デリバリーとよく似た概念ですが、継続的デリバリーがデプロイ可能な状態を準備するだけなのに対し、継続的デプロイは実際に本番環境へのデプロイまでを行う点が異なります。

このようなビルド、テスト、デプロイを自動化し、継続的に行う手法のことを、あわせて「CI/CD」と呼びます。

参考サイト：CI/CD とは| ニフクラ(nifcloud.com)

戻る

Python スクリプトのデバッグに使えるヒント

- デバッグに使えるコマンド

CS+の Python は、pdb をサポートしていないため、デバッグに使えるコマンドを記載します。

メッセージ出力

・print ：print(変数)、print “メッセージ” または print (“メッセージ”)

一時停止

・sleep ：import time
time_duration = 60
time.sleep(time_duration)

・pause ：import os
os.system("pause")

強制終了

・exit ：import sys
sys.exit()

実行中のスクリプトの強制終了は、python コンソール上で右クリックから強制終了を選択、あるいは、ctrl + D で強制終了できます。

- スクリプトの構成

スクリプトは分割して作成し、個別に実行させることができます。スクリプト 2 では、エクセルの制御関連のスクリプトと、自動実行させるスクリプトを分けて、個別に実行させるようにしています。これのメリットは、デバッグが楽になることです。また、実際の実行時には一方のスクリプトから他方のスクリプトを Source 関数で呼び出して、一括して実行することも可能です。

Source 関数を使用するときの注意：

Source 関数で呼び出すスクリプトファイルを指定しますが、注意が必要です。CS+のプロジェクト・ツリーや Python コンソールからスクリプトを実行する場合、次のエラーが発生する可能性があります。例えば、Source(“sample.py”)とした場合、

「ファイル 'C:¥Program Files (x86)¥Renesas Electronics¥CS+¥CC¥sample.py' が見つかりませんでした。」のエラーが発生します。これは、カレント・ディレクトリ(フォルダ)が CS+の実行ファイル CubeSuiteW+.exe のディレクトリ(フォルダ)にあるためです。これを避けるには、スクリプトファイルを絶対パス指定する事が必要ですが、プロジェクトは移動やコピーをして使う可能性があるため、以下の方法で回避します。

```
# 呼び出すスクリプトファイル名
Source_file_name = "sample.py"
# スクリプトファイルを検索
Source_file = ""
for path in project.File.Information():
    if os.path.basename(path) == Source_file_name:
        Source_file = path
Source(Source_file)
```

なお、プロジェクト・フォルダに置いた、バッチファイルから実行する場合は

Source(“sample.py”) として問題ありません。また、Hook 関数も同様です。

戻る

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2022.09.15	全頁	初版作成

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策

CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。

2. 電源投入時の処置

電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。

3. 電源オフ時における入力信号

当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。

4. 未使用端子の処理

未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。

5. クロックについて

リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。

6. 入力端子の印加波形

入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。

7. リザーブアドレス（予約領域）のアクセス禁止

リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。

8. 製品間の相違について

型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。

2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。

3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。

4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。

5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。

6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等

高品質水準： 輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。

7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。

8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。

9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。

10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。

11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。

12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。

13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。

14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.5.0-1 2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。