06
28
2019

RXマイコンで、Unityを使ってハードウェアの動作確認をしてみた

RXマイコン テスト

前回記事「RXマイコンで、Unityを使って単体テストをやってみた」のおまけとして、ハードウェアの動作確認のテストを書いてみました。

- Unityを使った単体テスト環境のセットアップについては、こちら。
- プロジェクト一式は、こちら(rx231_unit_test_3.zip - Google ドライブ)
  ▶ インポート方法

環境

- Board: Target Board for RX231
  - マルツで3000円くらいで購入できる
  - エミュレータ機能内蔵(E2Liteとして認識)。USB接続だけで電源供給・デバック可能
- Device: RX231(R5F52318ADFP)
- IDE: e2Stdio V7.4.0
- Compiler: CC-RX V3.01.00
- Unit Test Framework: Unity (https://github.com/ThrowTheSwitch/Unity)

お題

ターゲットボードには、下記仕様のLEDとSWがあるので、これらのIOチェックを作ります。

port	name	dir	L	H
PD6	LED0	out	点灯	消灯
PD7	LED1	out	〃	〃
PB1	SW1	in	押す	離す

ターゲットボード環境でのテスト

1. テストファイルを用意する

以前の記事で、下記のテストファイルを `/srt/test/` 以下に用意しています。

- Test0.c - テストグループの定義、および、各テストケースを記述
- AllTests.c - テストで実行するテストグループを記述

このうち、 `Test0.c` を修正してテストを用意します。

ハードウェアの動作確認では、人がLEDの点灯状態をチェックしたり、SWを操作する必要があります。このため、チェック結果を入力させたり、操作メッセージを表示する補助関数を用意します。

- MANUAL_CHECK(question)
  確認メッセージを表示し、チェック結果のキー入力 `y(es)` もしくは `n(o)` を受け取ります。その結果によって、テストをPASS/FAILさせます。FAIL時に行番号がわかるようにマクロで記述しています。

```
#define MANUAL_CHECK(question)  _ManualCheck(question, __LINE__)
static void _ManualCheck(const char *question, int line)
{
    printf("\nManual Check> %s -> press <y/n>: ", question);
    if ( getchar() == 'y' ) {
        // 手動判定:PASS
    } else {
        // 手動判定:FAIL
        char msg[100];
        sprintf(msg, "Manual Check(line:%d)", line);
        TEST_FAIL_MESSAGE(msg);
    }
}
```

- ManualOperation(const char *operation)
  操作メッセージを表示し、キー入力 `y(es)` を受けとると、処理を継続します。

```
static void ManualOperation(const char *operation){
    printf("\nManual Operation> %s -> press <y>: ", operation);
    getchar();
}
```

これらを利用すると、テストコードは下記になります。

/src/test/Test0.c

```
#include "../unity/unity_fixture.h"
#include <stdio.h>
#include "../io.h"

// 各テストケースの前に実行する共通処理(初期化)
TEST_SETUP(Test0)
{
    LED0 = LED_OFF;
    LED1 = LED_OFF;
}

// テストケース(LED0 の IOチェック)
TEST(Test0, Led0_IoCheck)
{
    LED0 = LED_ON;
    MANUAL_CHECK("Is LED0 on?");
    LED0 = LED_OFF;
    MANUAL_CHECK("Is LED0 off?");
}
```

```
// テストケース(LED1 の IOチェック)
TEST(Test0, Led1_IoCheck)
{
    LED1 = LED_ON;
    MANUAL_CHECK("Is LED1 on?");
    LED1 = LED_OFF;
    MANUAL_CHECK("Is LED1 off?");
}

// テストケース(SW1 の IOチェック)
TEST(Test0, SW1_IoCheck)
{
    ManualOperation("push sw1.");
    TEST_ASSERT_EQUAL(SW1_PUSH, SW1);
    ManualOperation("release sw1.");
    TEST_ASSERT_EQUAL(SW1_RELEASE, SW1);
}

// テストグループで、実行するテストケースを列挙する
TEST_GROUP_RUNNER(Test0)
{
    RUN_TEST_CASE(Test0, Led0_IoCheck);
    RUN_TEST_CASE(Test0, Led1_IoCheck);
    RUN_TEST_CASE(Test0, SW1_IoCheck);
}
```

/src/io.h

```
#define LED_ON          (0)
#define LED_OFF         (1)
#define SW1_PUSH        (0)
#define SW1_RELEASE     (1)

/* Switches */
#define SW1             (PORTB.PIDR.BIT.B1)

/* LED port settings */
#define LED0            (PORTD.PODR.BIT.B6)
#define LED1            (PORTD.PODR.BIT.B7)
```

2. ターゲットボード環境でのテスト実行

- ターゲットボードをPCに接続しておく
- `ビルド構成` として `HardwareDebug(Debug on hardware)` を選択し、ビルド
- `デバックの構成` として `～ HardwareDebug` を選択し、デバック開始＆プログラム実行
- `Renesas Debug Virtual Console` の表示に従い、SW操作とキーボードを押下し、テストを進める。
- 最終的に、下記表示となればＯＫ。

Renesas Debug Virtual Console(OK時)

```
Unity test run 1 of 1
TEST(Test0, Led0_IoCheck)
Manual Check> Is LED0 on? -> press <y/n>: y
Manual Check> Is LED0 off? -> press <y/n>: y PASS
TEST(Test0, Led1_IoCheck)
Manual Check> Is LED1 on? -> press <y/n>: y
Manual Check> Is LED1 off? -> press <y/n>: y PASS
```

```
TEST(Test0, SW1_IoCheck)
Manual Operation> push sw1. -> press <y>: y
Manual Operation> release sw1. -> press <y>: y PASS

-----------------------
3 Tests 0 Failures 0 Ignored
OK
```

ちなみに、NG時は下記です。手動判定でのNG行番号は `Manual Check(line:XX)` と表示されます。

Renesas Debug Virtual Console(NG例)

```
Unity test run 1 of 1
TEST(Test0, Led0_IoCheck)
Manual Check> Is LED0 on? -> press <y/n>: n../src/test/Test0.c:19::FAIL: Manual Check(lin
TEST(Test0, Led1_IoCheck)
Manual Check> Is LED1 on? -> press <y/n>: n../src/test/Test0.c:19::FAIL: Manual Check(lin
TEST(Test0, SW1_IoCheck)
Manual Operation> push sw1. -> press <y>: y../src/test/Test0.c:69::FAIL: Expected 0 Was 1

-----------------------
3 Tests 3 Failures 0 Ignored
FAIL
```

最後に

今回は、テストフレームワークをハードウェアの動作確認として使ってみました。
ハードウェアの動作確認というと、PCからターゲットボードにコマンドを投げて確認するとか、デバックポートでCUIを作ったりとか、になりそうですが、既存のIDE環境で手をかけずに作れるのはメリットと思いました。また、項目が増えてきた時、確認項目(テストケース)をテストフレームワークの管理方法で整理しておくと、保守しやすいかも知れないです。

sonoka_gi 1年前

関連記事

2020-07-12
Processingで、学戦都市アスタリスクのクレジット表示を真似してみた
アニメ「学戦都市アスタリスク」の２期オープニングのクレジッ...

2019-06-26
RXマイコンで、Unityによる単体テストをやってみた
Qiitaの「TDDによるマイコンのLチカ開発」を、ルネサス製のRXマ...

2019-06-24
RXマイコンで、Unityによる単体テスト環境を作ってみた（後編）
Qiitaの記事「TDDによるマイコンのLチカ開発」を、ルネサス製の...

コメントを書く

« PROCESSINGで、学戦都市アスタリスクのク... RXマイコンで、UNITYによる単体テストをや... »

プロフィール

sonoka_gi

電子工作やプログラミングなど、やってみたことのメモ

読者になる 1

検索

記事を検索

リンク

はてなブログ
ブログをはじめる
週刊はてなブログ
はてなブログPro

最新記事

Logicool製Webカメラ C922n のカメラ設定
C言語で、符号やサイズが異なる場合のキャスト動作を確認してみた
Processingで、学戦都市アスタリスクのクレジット表示を真似してみた
RXマイコンで、Unityを使ってハードウェアの動作確認をしてみた
RXマイコンで、Unityによる単体テストをやってみた

月別アーカイブ

▼ 2020 (3)
2020 / 12 (1)
2020 / 10 (1)
2020 / 7 (1)
▶ 2019 (4)
▶ 2018 (2)