GoogleTestとFakeFunctionFrameworkによるTOPPERSアプリケーションの単体テスト自動化

パイオニアシステムテクノロジー株式会社 2022年度新入社員一同

作成者:仙石 友輝

共同作業者:薄井 茜、髙橋 佑梨、村山 晴基

目次

- 著作権表記
- 参考サイト、参考資料

1. 概要
2. テストの全体像
3. テスト対象
4. テスト環境の構築
5. テストケース
   5.1. 各テストケースに共通する箇所の解説
   5.2. テストケースの解説(1)スイッチスキャンタスク
   5.3. テストケースの解説(2)LED4点滅タスク
6. 実行ファイルの生成
7. テストの自動実行
8. テスト結果
9. テストの応用
10. まとめ

著作権表記

- 本資料は、名古屋大学情報学研究科附属組込みシステム研究センター(NCES)の「組込みソフトウェア開発技術の基礎」(以降、「組込みソフトウェア開発技術の基礎」と略) を改変・再配布する形になるため、著作権を以下のとおり示します。

参考サイト、参考資料

- 組込みソフトウェア開発技術の基礎 https://www.nces.i.nagoya-u.ac.jp/NEP/materials/about.html
- GoogleTest https://github.com/google/googletest
- GoogleTest ドキュメント日本語訳 http://opencv.jp/googletestdocs/index.html
- Fake Function Framework(FFF) https://github.com/meekrosoft/fff
- モダンC言語プログラミング https://tatsu-zine.com/books/modern-cprogramming
  出版社: KADOKAWA/アスキー・メディアワークス

1. 概要(1/5)

- 目的
  - 組込みソフトウェア開発では、実機が用意されないと通信系やレジスタ・メモリ系のテストが困難です。ソースコードを改変すれば出来なくもないですが、製品に載せるソースコードに手を加えることは避けたいです（品質が担保できなくなる）。ソースファイルを疑似テスト用と本番用で入れ替えすることも考えられますが、かなり手間となります。

これらの課題を解決するために、実機不要な自動単体テスト環境を
GoogleTest ＋ FFF（Fake Function Framework）によって実現する

1. 概要(2/5)

- 狙い
  - 昨今の半導体不足や短納期開発における実機不足の改善。
  - 誤ってテストコードが本番用ソフトに混入することの防止。
  - いきなり大規模開発に単体テスト自動化を適用しようとすると失敗しがちのため、先ず小規模なカップラーメンタイマに適用することで、基本を理解する。

1. 概要 (3/5)

- GoogleTestとは
  - Google社の提供するC++用の単体テストのフレームワーク
  - C言語プログラムの単体テストにも利用可能
- GoogleTestの利点
  - C++言語の知識がなくても、容易に単体テスト環境の構築が可能
  - 関数の戻り値や変数の検証が容易
  - テストケースの追加が容易
  - テスト結果の可読性が高い
  - 日本語のドキュメントが充実
  - 非常に大きい回数の繰り返しテストが高速かつ,容易に実装が可能
  - テスト対象を改変せずに試験が可能

1. 概要(4/5)

- Fake Function Framework (FFF) とは
  - FFFは、単体テスト用のダミー関数を自動生成するツールです。FFFは、GoogleTestと組み合わせて動作します。

- Fake Function Framework (FFF) の利点
  - C++の知識が浅くても下位関数を容易に偽装でき、戻り値も指定可能
  - レジスタの偽装ができるので、実機がなくてもPC上で単体テストが可能
  - カスタムFake関数にて、独自に作成したモック関数を実行可能
    - TOPPERSのサービスコールの偽装や、コールバック関数の偽装も可能
  - GoogleTestの豊富な判定文を用いて、偽装した下位関数の呼び出し回数や下位関数に渡す引数等の検証が可能
  - GoogleTestと組み合わせて、テスト対象を改変せず、高速に試験可能

1. 概要(5/5)

- GoogleTest + FFF をTOPPERSアプリケーションの単体テストに適用するために
  - setjmp/longjmp を用いた無限ループからの脱出
    - 無限ループを含む関数の呼び出しがあると、テストを完了できない。
      - テスト実行前にsetjmpをコール。
      - TOPPERSのサービスコールを偽装。偽装したサービスコール内にlongjmp を仕込むことで無限ループからの脱出を可能とし、正しくテストが行えるようにする。
  - 再現性の低いバグを発見可能なテストケース
    - テストケースのテンプレートを作成し、数多の組み合わせを検証することで、特定の条件でのみ発現するバグを発見することが可能。
    - SW1とPUSH1の同時変化といった実機ではなかなか再現できないテストが可能。

2. テストの全体像

テストケース作成

- GoogleTest:テストケース,期待値検証マクロの提供
- FFF:関数の偽装マクロの提供

↓

テスト実施

- テスト対象:GoogleTestからコールされ、FFFで偽装された関数をコールし処理を終える
- GoogleTest:期待値を検証しながら,複数個あるテストケースを1つずつ逐次的に実行

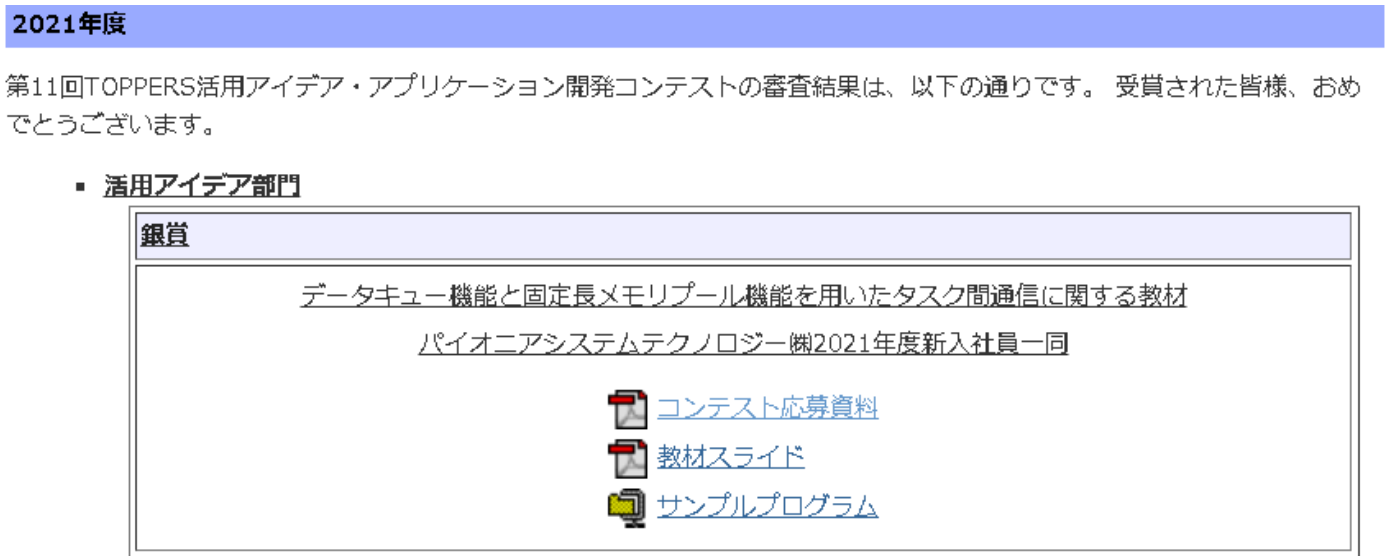
↓

テスト終了

- GoogleTest:テスト結果レポート作成

3. テスト対象(1/7)

- 今回のテスト対象はカップラーメンタイマのソースコードです。このカップラーメンタイマはTOPPERSのアプリケーションタスクとして実装されております。カップラーメンタイマのソースコードや、外観、仕様、開発環境等は、第11回TOPPERS活用アイデア・アプリケーション開発コンテストの「データキュー機能と固定長メモリプール機能を用いたタスク間通信に関する教材」を参照願います。https://www.toppers.jp/contest.html



2021年度

第11回TOPPERS活用アイデア・アプリケーション開発コンテストの審査結果は、以下の通りです。 受賞された皆様、おめでとうございます。

- **活用アイデア部門**

銀賞

データキュー機能と固定長メモリプール機能を用いたタスク間通信に関する教材

パイオニアシステムテクノロジー㈱2021年度新入社員一同

コンテスト応募資料

教材スライド

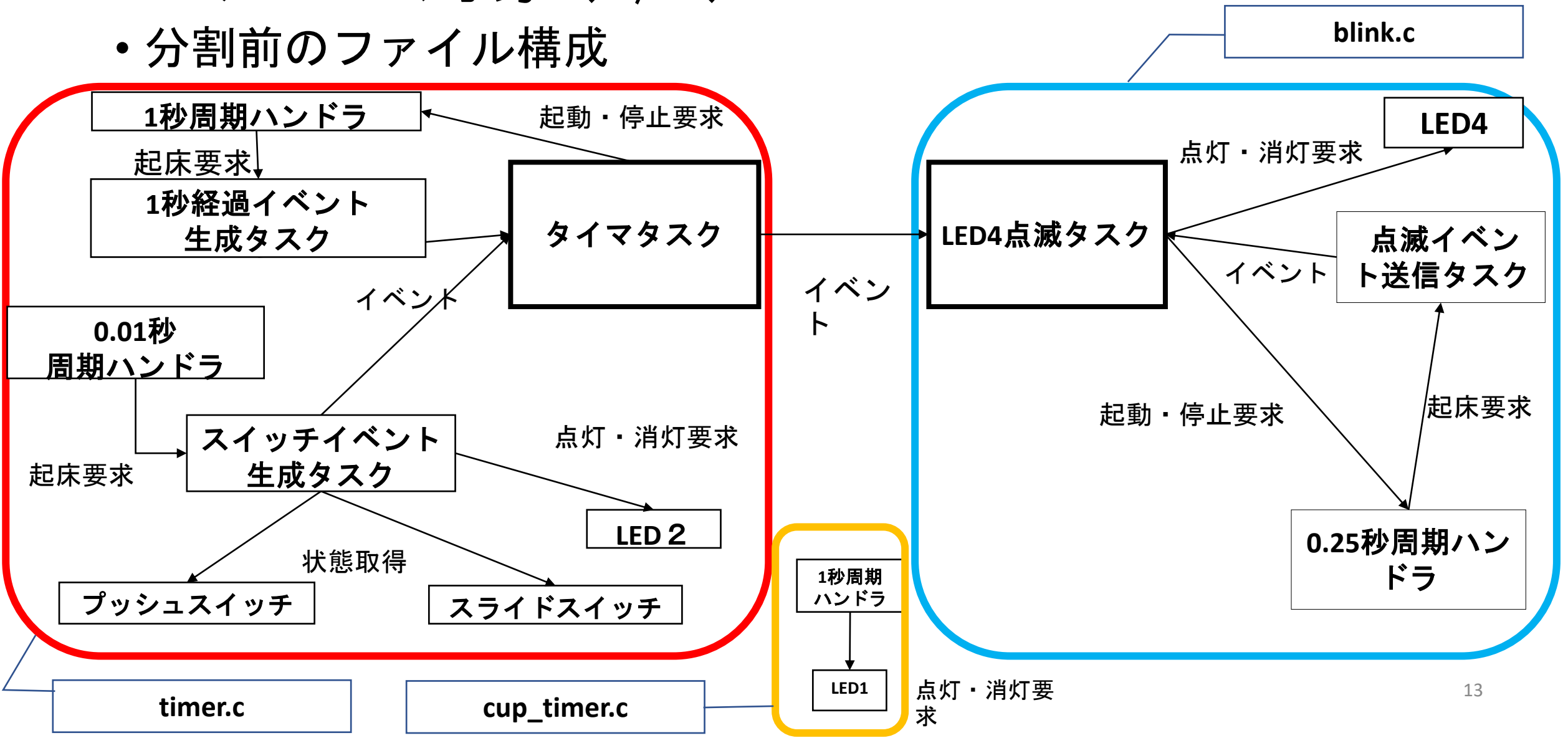
サンプルプログラム

3. テスト対象(2/7)

- ソースファイル分割
  - GoogleTestはファイル単位でテストを実行します。
  - 「第11回TOPPERSアイディア・アプリケーション開発コンテスト」にて、弊社の2021年度の新人が作成したカップラーメンタイマは、ファイル分割の粒度が粗すぎることから、テスト向きではないです。
  - そこでファイルを分割し、テストをしやすい構成にしました。

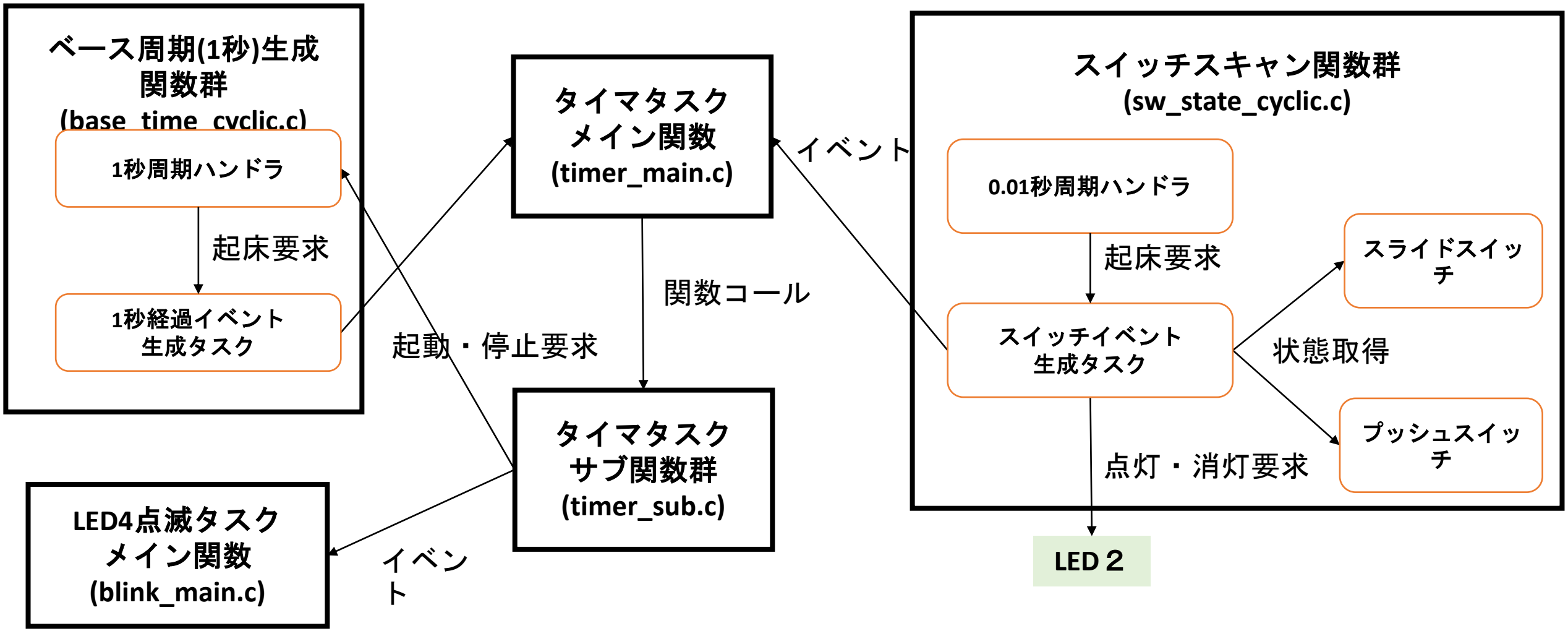
3. テスト対象(3/7)

- 分割前のファイル構成



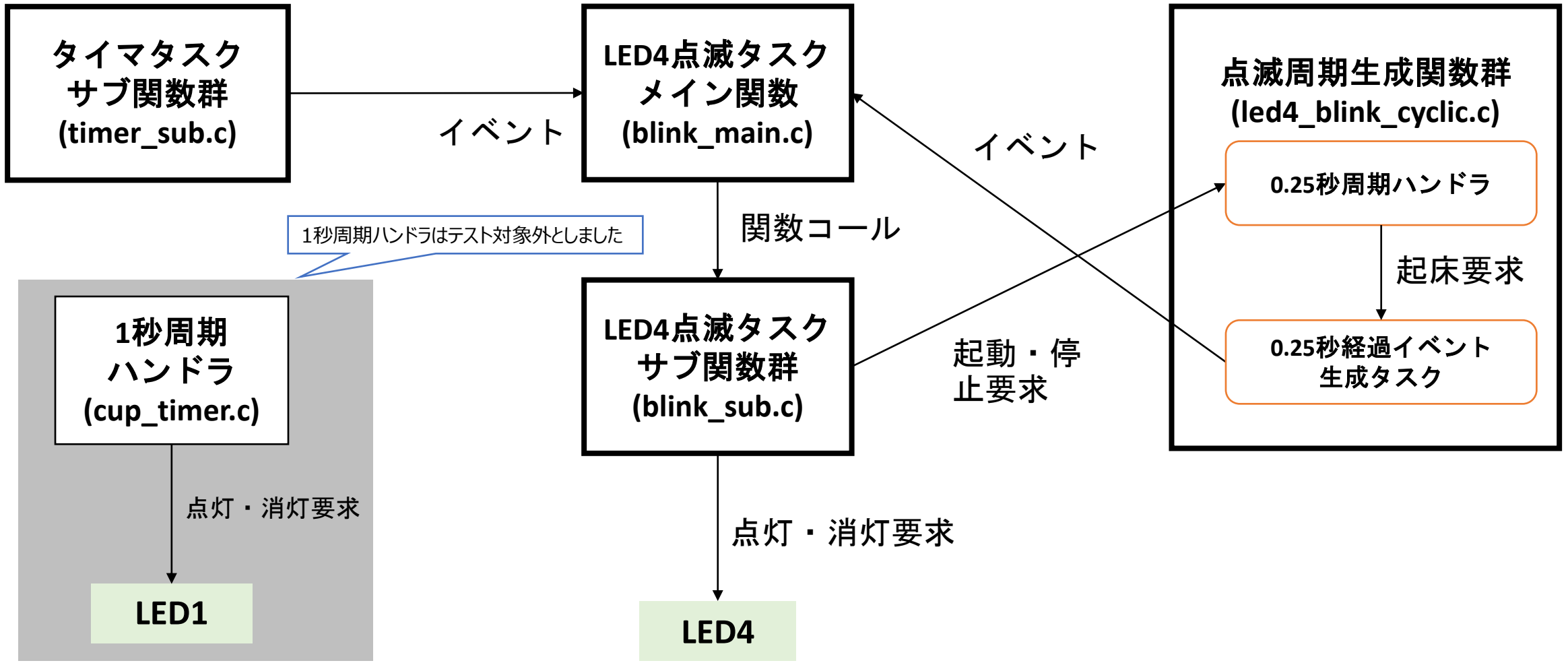
3. テスト対象(4/7)

・分割後のファイル構成（計時関連）



3. テスト対象(5/7)

- 分割後のファイル構成（LED4関連）



3. テスト対象(6/7)

- 次のフォルダにファイル分割後のソースファイル一式があります。
  - sample¥RX63N¥cup_timer10 -> NCES TRAINING BOARD用
  - sample¥stm32¥cup_timer10 -> STM32F401 Nucleo用
- テスト対象のファイルは次の7ファイルになります。

¥CUP_TIMER10
- blink_main.c
- blink_sub.c
- base_time_cyclic.c
- led4_blink_cyclic.c
- sw_state_cyclic.c
- timer_main.c
- timer_sub.c

3.テスト対象(7/7)

- 状態遷移表とフローチャート
  - タイマタスクとLED4点滅タスクの状態遷移表、及び各モジュールのフローチャートは下記エクセルを参照願います。

フローチャート

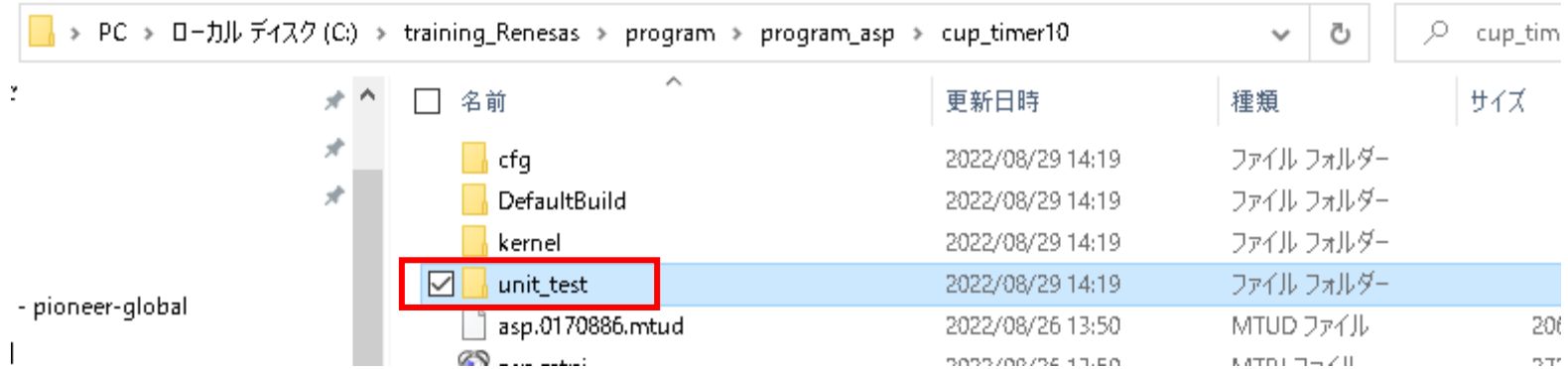
- ビルドと動作確認
  - cup_timer10ディレクトリを¥program¥program_asp¥にコピーしてください。
  - 「組込みソフトウェア開発技術の基礎」の12.ITRONプログラミング実習編を参考にcup_timer10をビルドし、マイコンボードに書き込んでください。12.ITRONプログラミング実習編「カップラーメンタイマの実装」と同じ動作となることを確認してください。

4. テスト環境の構築(1/8)

- この単体テスト環境は、以下の環境で動作確認しております。
  - Windows10 Pro 64ビット
  - Pleiades All in One Eclipse リリース2021 Ultimate Windows x64 Ultimate Full Edition　https://mergedoc.osdn.jp/#pleiades.html
    - 上記サイトから、Eclipse Ultimateをダウンロードしてインストールしてください(この単体テスト環境はCDTとPythonを使用するため、Ultimate版をインストールしております)。
    - Pythonとgccの環境があれば他の環境でも動作すると推測します。

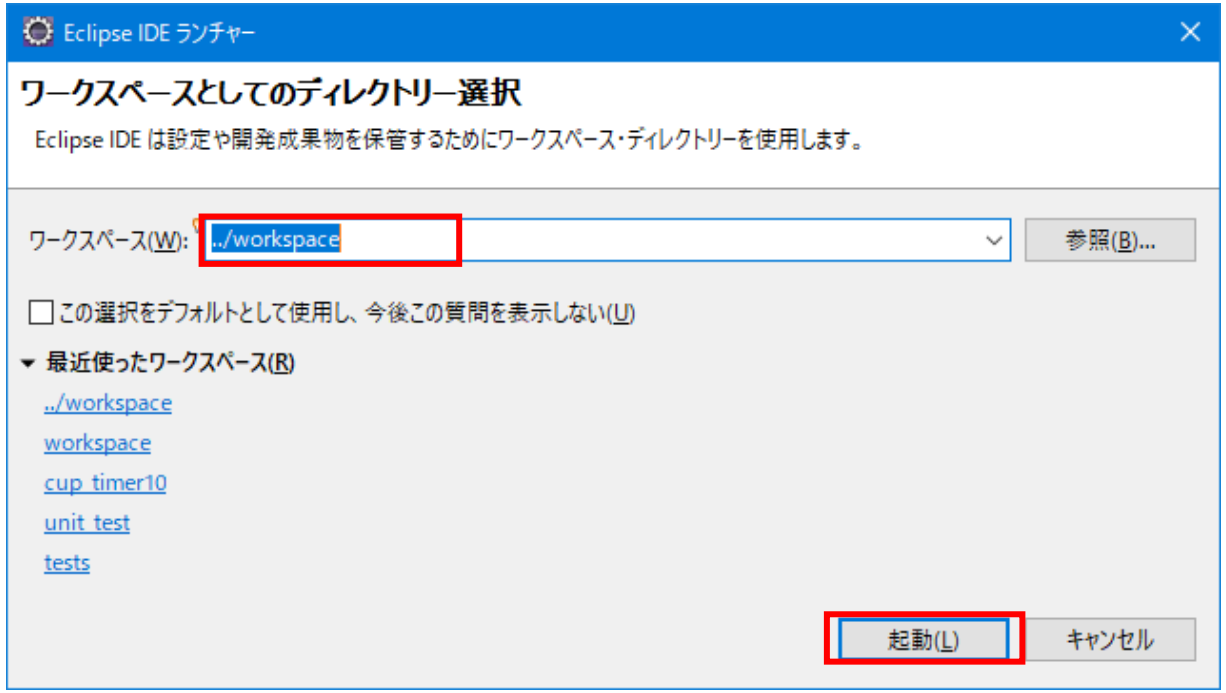
4. テスト環境の構築(2/8)

- unit_testディレクトリを¥program¥program_asp¥cup_timer10にコピーしてください。



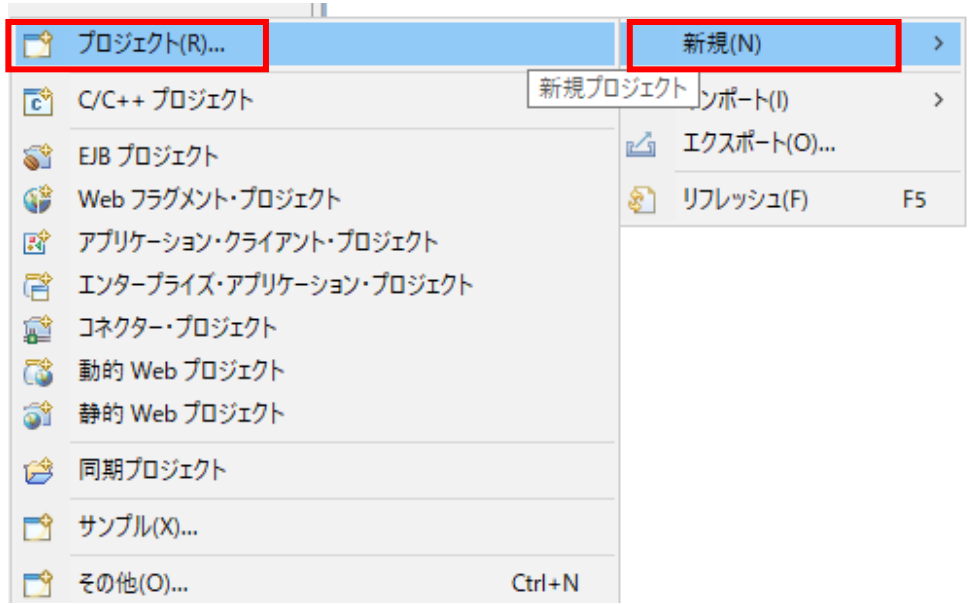
4. テスト環境の構築(3/8)

- Eclipseを起動
- デフォルトの../workspaceを選択し、起動



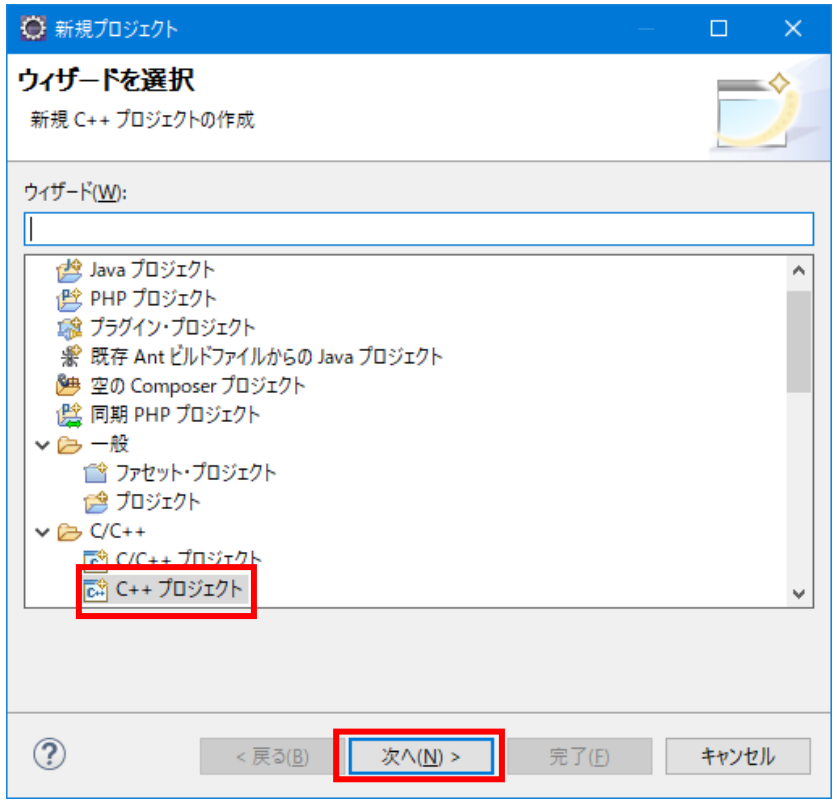
4. テスト環境の構築(4/8)

- プロジェクトエクスプローラ -> 新規 ->プロジェクトを選択



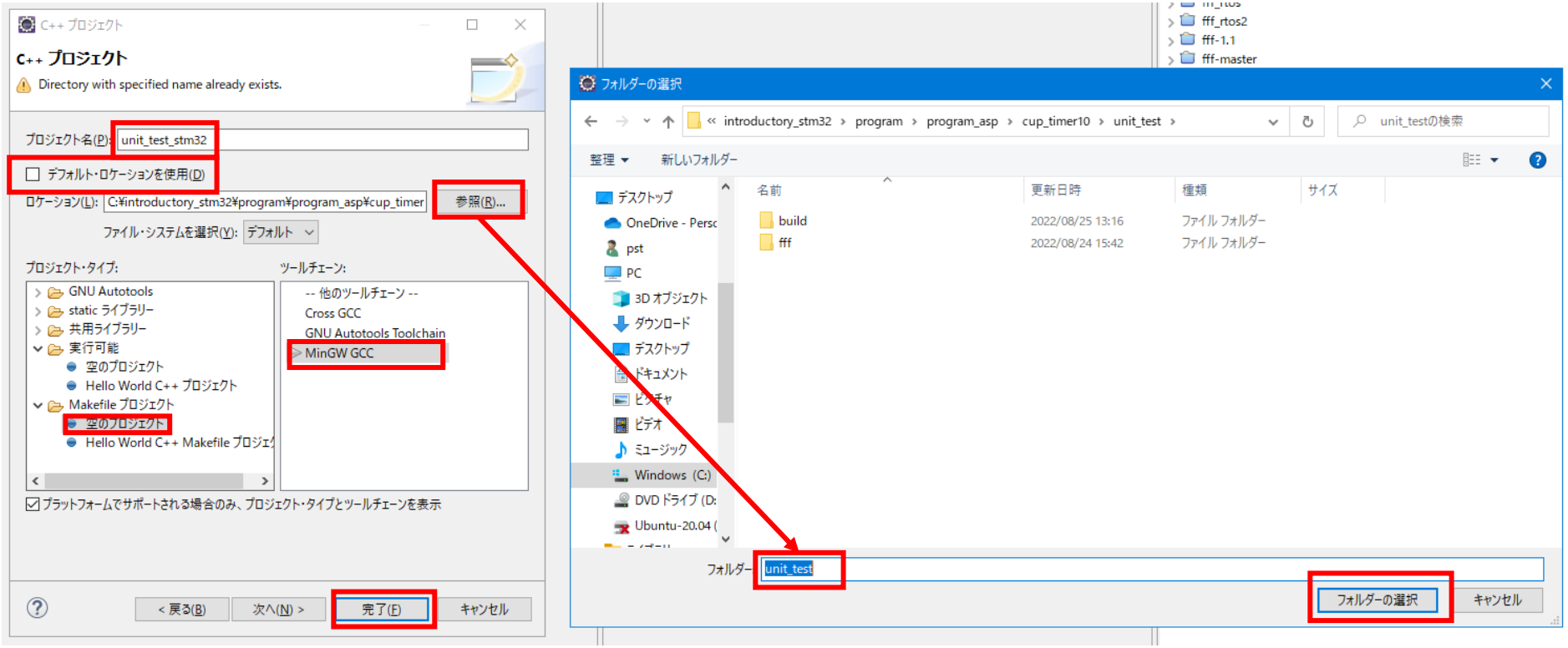
4. テスト環境の構築(5/8)

- C++プロジェクトを選択して次へ



4. テスト環境の構築(6/8)

- プロジェクト名に任意の文字列(下図ではunit_test_stm32)を入力
- デフォルト・ロケーション使用のチェックを外し、ロケーションにcuptimer10¥unit_testを指定
- Makefileプロジェクト -> 空のプロジェクト, ツールチェーン -> MinGW GCC を選択し、完了

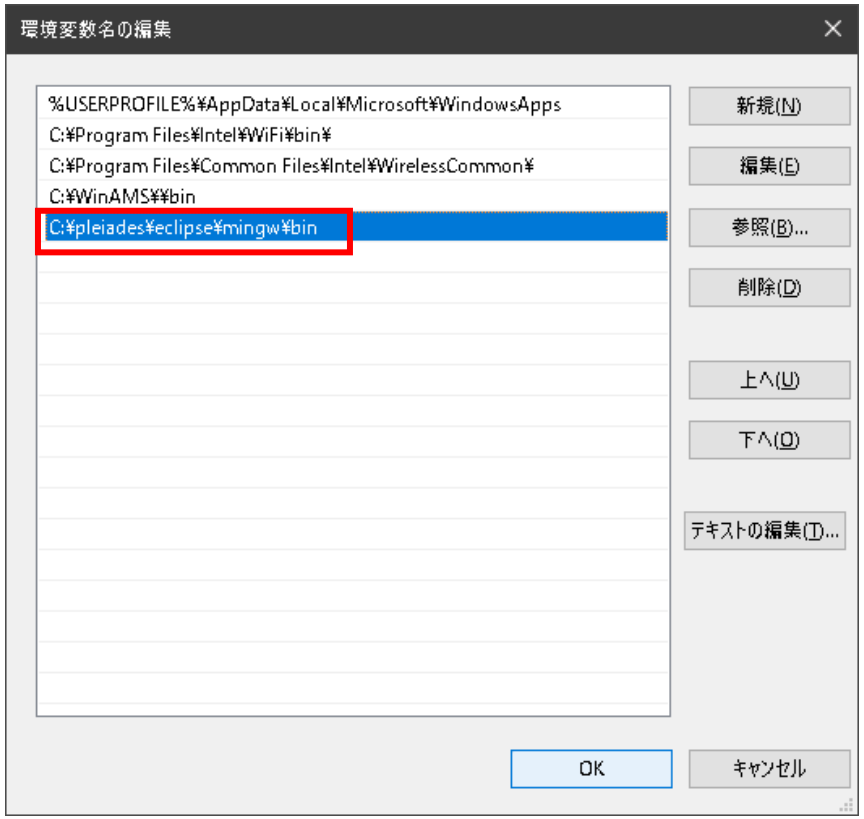


4. テスト環境の構築(7/8)

- ユーザ環境変数のPathに次のディレクトリを追加してください。

C:¥pleiades¥eclipse¥mingw¥bin

Eclipseをインストールしたディレクトリを指定



4. テスト環境の構築(8/8)

unit_test¥build¥run_all.batの下から2行目を次のように変更してください。

C:¥pleiades¥python¥3¥python.exe main.py

@pause

5. テストケース

- テストケース数と解説の範囲
  - 下表の通り40のテストケースを作成しました。全てのテストケースを説明すると膨大になるため、各テストケースに共通する箇所と、テストケースを2つ抜粋して解説します。

テスト対象のCファイル名	テストケースのファイル名	テストケースの数
timer_main.c	testTimerMain.cpp	6
timer_sub.c	testTimerSub.cpp	9
base_time_cyclic.c	testBaseTimeCyclic.cpp	2
sw_state_cyclic.c	testSwStateCyclic.cpp	6
blink_main.c	testBlinkMain.cpp	7
blink_sub.c	testBlinkSub.cpp	8
led4_blink_cyclic.c	testLed4BlinkCyclic.cpp	2
テストケースの合計		40

5.1. 各テストケースに共通する箇所の解説(1/11)

- テストケース作成の最初の作業として、GoogleTest を使用するためのヘッダgtest.hと、FFFを使用するためのヘッダFFF.hをincludeします。
- FFFを使うためにはDEFINE_FFF_GLOBALSの記述が必要となります。
- テスト対象をextern “C”で丸ごとincludeします。こうすることで、テスト対象のstatic変数やstatic関数の参照・更新が可能となります。

```
19 #include "fff/gtest/gtest.h"↓
20 #include "fff/fff.h"↓
21 DEFINE_FFF_GLOBALS;↓
22 ↓
23 void* gp_blk = (void*)NULL;››    ›    // 送信するメモリプールのアドレスの確認用↓
24 ↓
25 long loop_num;›  ›  ›  ›  ›  ›  ›    // 無限ループ実行回数↓
26 jmp_buf buf; ›  ›  ›  ›  ›  ›  ›    // 無限ループ脱出用のスタック退避領域↓
27 ↓
28 extern pthread_mutex_t mutex;›  ›  ›    // テストケースを逐次実行するための保険↓
29 ↓
30 extern "C" {↓
31 ›    // テスト対象のCソースファイルを丸ごとインクルードすることにより、↓
32 ›    // static関数やstatic変数の参照・更新が可能となる。↓
33 ›    #include "../sw_state_cyclic.c"↓
34 ↓
35 }↓
```

5.1.各テストケースに共通する箇所の解説(2/11)

- 関数の実体があるものはテスト対象のCソースファイルに記述された関数のみです。実体のない関数は、FFFで偽装関数宣言します。

  戻り値がvoidの場合:FAKE_VOID_FUNC(偽装する関数名,引数の型);

  戻り値がある場合:FAKE_VALUE_FUNC(戻り値の型,偽装する関数名,引数の型);

```
40 /* サービスコールの偽装宣言 */↓
41 /* タスク付属同期機能 */↓
42 FAKE_VALUE_FUNC(ER, slp_tsk);↓
43 FAKE_VALUE_FUNC(ER, iwup_tsk, ID);↓
44 /* 同期・通信機能 */↓
45 FAKE_VALUE_FUNC(ER, psnd_dtq , ID , intptr_t);↓
46 /* メモリプール管理機能 */↓
47 FAKE_VALUE_FUNC(ER, get_mpf, ID , void**);↓
48 ↓
49 /* ライブラリの偽装宣言 */↓
50 FAKE_VOID_FUNC(t_perror ,uint_t ,  const char* , int_t , const char* ,ER );
51 FAKE_VOID_FUNC(set_led2_state, unsigned char);↓
52 ↓
53 FAKE_VALUE_FUNC(unsigned char, get_sw1_state);↓
54 FAKE_VALUE_FUNC(unsigned char, get_push1_state);↓
```

5.1. 各テストケースに共通する箇所の解説(3/11)

・FFFで偽装する関数は、テストケース実行前に初期化する。

偽装関数の初期化リストFFF_FAKE_LISTの定義:FAKE(初期化する偽装関数名)

```
112 #define FFF_FAKES_LIST(FAKE)    \
113     FAKE(slp_tsk)               \
114     FAKE(iwup_tsk)              \
115     FAKE(get_mpf)               \
116     FAKE(get_sw1_state)         \
117     FAKE(get_push1_state)       \
118     FAKE(set_led2_state)        \
119     FAKE(slp_tsk)               \
120     FAKE(psnd_dtq)
```

5.1. 各テストケースに共通する箇所の解説(4/11)

- テストケースの初期化と後処理

  SetUpはTEST_Fで記述された各テストケースの実行前の処理、TearDownが各テストケースの実行後の処理となります。

  テスト対象、及びテストケースはグローバル変数を使用しています。

  よって、マルチスレッド環境下で複数のテストケースが同時実行された場合、グローバル変数のアクセスでテストが破綻する恐れがあります。

  複数のテストケースを同時実行することは無いとは思いますが、 保険としてmutexにてテストケースを逐次実行するようにしました。

```
129 class testSwStateCyclic: public testing::Test {
130     void SetUp() {
131         pthread_mutex_lock(&mutex);
132         /* Register resets */
133         FFF_FAKES_LIST(RESET_FAKE);
134         /* reset common FFF internal structures */
135         FFF_RESET_HISTORY();
136     }
137     void TearDown() {
138         pthread_mutex_unlock(&mutex);
139     }
140 };
```

5.1. 各テストケースに共通する箇所の解説(5/11)

- テストケースの追加の方法

TEST_F(テストグループ名,テスト名称)でテストケースを追加していきます。

TEST_FはGoogleTestのマクロで、複数のテストケースで同じデータ設定を使う際に使用します。

EXPECT_EQは GoogleTestの判定文で、1つ目の引数に期待値、2つ目の引数に検証対象を記述します。主な判定文を次ページ以降に記載します。

```
119 // 静的API( ATT_INI) に登録した初期化ルーチンのテスト↓
120 TEST_F(testTimerMain, test_timer_task_init) {↓
121 ›   memset(&gtm_ctl, 0xFF, sizeof(TTM_CTL));›   // 予めメモリを汚しておく↓
122 ↓
123 ›   timer_task_init(0);››   ›   ›   ›   ›   // テスト対象をコール↓
124 ↓
125 ›   // 初期化後の期待値を確認↓
126 ›   EXPECT_EQ(0, gtm_ctl.ctl_flg);↓
127 ›   EXPECT_EQ(0, gtm_ctl.timer);↓
128 ›   EXPECT_EQ(ST_TIMER_OFF, gtm_ctl.nowsts);↓
129 ›   EXPECT_EQ(ST_TIMER_OFF, gtm_ctl.nextsts);↓
130 }↓
```

5.1.各テストケースに共通する箇所の解説(6/11)

・GoogleTestの判定文について

GoogleTestでは、判定文を生成するマクロを用いてテストの成功/失敗を判定します。本資料では主な判定文としてTRUE/FALSEの判定と2値の判定のみ掲載しますが、他にも文字列の比較等の判定文があります。詳細はGoogle Test ドキュメント日本語訳http://opencv.jp/googletestdocs/index.htmlを参照願います。

・判定文の使い分け方

判定文は、ASSERT_*とEXPECT_*の2種類に大別されます。

ASSERT_*は、判定条件が失敗の時、実行中のテストケースを中断し、**次のテストケースを実行**します。

5.1.各テストケースに共通する箇所の解説(7/11)

EXPECT_*は、判定条件が失敗の時でもテストを中断せず、次のテストを実行します。

通常は1度のテストで複数の失敗を検出できるEXPECT_*が便利です。

・ASSERT_*が役に立つケース

①プロジェクトの規定上、1つでもエラーを検出したテストケースはテストのやり直しになる。

②テスト対象のアルゴリズムの性質上、エラー検出後のテストは無意味なものとなる。

①②のどちらもテスト時間が短ければEXPECT*でも支障ないですが、長時間かかる多数のテストケースを一晩中実行する際等に差が出てきます。

5.1. 各テストケースに共通する箇所の解説(8/11)

- TRUE/FALSEの判定
  - ASSERT/EXPECT_期待値(引数)の書式で、期待値と引数が異なるとエラーとなります。

判定	判定	判定条件
ASSERT_TRUE(condition);	EXPECT_TRUE(condition);	condition が true
ASSERT_FALSE(condition);	EXPECT_FALSE(condition);	condition が false

5.1. 各テストケースに共通する箇所の解説(9/11)

- 2つの値の判定
  - 1つ目の引数に期待値、2つ目の引数に検証対象を記述します。

判定	判定	判定条件
ASSERT_EQ(expected, actual);	EXPECT_EQ(expected, actual);	expected == actual
ASSERT_NE(val1, val2);	EXPECT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPECT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPECT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPECT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

5.1. 各テストケースに共通する箇所の解説(10/11)

- FFFの主な機能とGoogleTestの判定文の組み合わせによる検証
  - テスト対象の関数を実行すると、下記の様な値がFFFによって記録されます。
    - 渡された引数の確認(Nは引数順): <偽装した関数名>_fake.argN_val
    - 戻り値の操作:<偽装した関数名>_fake.return_val
    - 呼び出された回数の確認:<偽装した関数名>_fake.call_count
    - 関数の呼び出し履歴(Nは呼び出し順):fff.call_history[N]
    - 複数回呼び出された際の引数の確認 (Nは引数順,Mは呼び出し順):

      <偽装した関数名>_fake.argN_history[M]
  - テストの基本的な流れは、テスト対象関数実行後にGoogleTestのASSERT/EXPECT判定文にて、FFFで記録された上記の値を検証していくことになります。

```
120    // slp_tskがコールされていることを確認↓
121    EXPECT_EQ(fff.call_history[0],(void*)slp_tsk);↓
122  ↓
123    // get_mpfがコールされていることを確認↓
124    EXPECT_EQ(fff.call_history[1],(void*)get_mpf);↓
125  ↓
126    // get_mpfの第1引数が正しいことを確認↓
127    EXPECT_EQ(EVENT_MPF,get_mpf_fake.arg0_history[0]);↓
```

5.1. 各テストケースに共通する箇所の解説(11/11)

- カスタムFake関数について
  - 偽装関数の初期化リストFFF_FAKE_LISTに登録するだけで前ページのFFFの機能を利用することができますが、偽装関数内に任意のロジックを実装したい場合、カスタムFake関数を使います。
  - カスタムFake関数:<偽装した関数名>_fake.custom_fake

```
45 // カスタムFake関数の実装↓
46 ER get_mpf_custom_fake(ID mpfid, void **p_blk)↓
47 {↓
48 ›   *p_blk = malloc(sizeof(64));›   ›   ›   ›   // コンフィギュレータで指定する固定長メモリプールのサイズ
49 ↓
50 ›   gp_blk = *p_blk;›   ›   ›   ›   ›   ›   ›   // 送信するアドレスの検証用に保持↓
51 ↓
52 ›   return(E_OK);↓
53 }↓
```

固定長メモリブロックの獲得を擬似的に実装する
get_mpf_custom_fake関数を作成

```
107 // LED4点滅周期割込みハンドラから起床される点滅周期タスクのテスト↓
108 TEST_F(testLed4BlinkCyclic, testLed4BlinkTimeTask)↓
109 {↓
110 ›   get_mpf_fake.custom_fake = get_mpf_custom_fake;›   // get_mpfをカスタムフェイク関数に設定↓
111 ›   psnd_dtq_fake.custom_fake = psnd_dtq_custom_fake;›   // psnd_dtqをカスタムフェイク関数に設定↓
112 ↓
113 ›   if(0 == setjmp(buf)){↓
114 ›   ›   led4_blink_time_task(0);›   ›   ›   ›   ›   ›   // blink点滅周期タスクのメインルーチン(無限ループ)へ突入
115 ›   }↓
```

テストケース内で、テスト対象のled4_blink_time_task関数実行前にget_mpf_custom_fake関数をカスタムFake関数に設定

5.2. テストケースの解説(1)スイッチスキャンタスク(1/4)

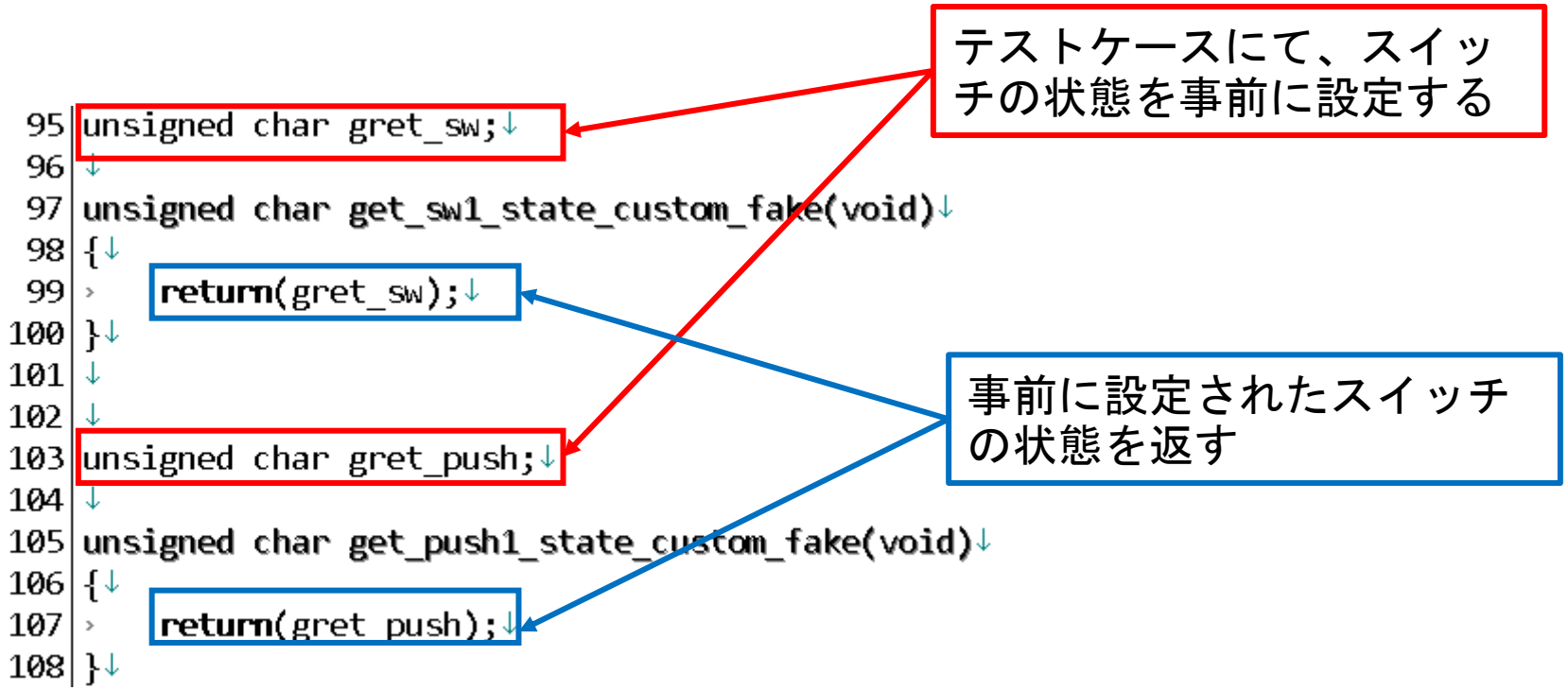
- テスト対象関数 sw_scan_task (関数の詳細は「状態遷移表とフローチャート」を参照願います)

```
44 void↓
45 sw_scan_task(intptr_t exinf){↓
46 ›   unsigned char sw;↓
47 ›   T_EVENT *psnd_data;↓
48 ↓
49 ›   for(;;){↓
50 ›   ›   SVC_PERROR(slp_tsk());          /* ハンドラから起床されるまでスリープ */↓
51 ↓
52 ›   ›   sw = get_sw1_state();↓
53 ›   ›   if (sw1 != sw) {↓
54 ›   ›   ›   /* 固定長メモリブロックの獲得 */↓
55 ›   ›   ›   SVC_PERROR(get_mpf(EVENT_MPF, (void**)&psnd_data));↓
56 ›   ›   ›   if (sw == ON) {↓
57 ›   ›   ›   ›   psnd_data -> event = EV_TIMER_SW1_ON;↓
58 ›   ›   ›   } else {↓
59 ›   ›   ›   ›   psnd_data -> event = EV_TIMER_SW1_OFF;↓
60 ›   ›   ›   }↓
61 ›   ›   ›   DEBUG_SYSLOG(LOG_NOTICE, " snd_dtq = 0x%p", psnd_data);↓
62         ›   SVC_PERROR(psnd_dtq(TIMER_DTQ, (intptr_t)psnd_data));↓
63 ›   ›   ›   sw1 = sw;↓
64 ›   ›   }↓
65 ↓
66 ›   ›   sw = get_push1_state();↓
67 ›   ›   if (sw8 != sw) {↓
```

10ms周期ハンドラから起床されると、SW1とPUSH1の状態をスキャンした後、無限ループによって再びスリープする。

5.2. テストケースの解説(1)スイッチスキャンタスク(2/4)

- デバイスドライバを偽装して、SW1 とPUSH1の状態を操る



5.2. テストケースの解説(1)スイッチスキャンタスク(3/4)

- SW1 とPUSH1の同時押下のテストケース

```
290 TEST_F(testSwStateCyclic, testSwScanTaskDouble)/*Sw1onPushon*/↓
291 {↓
292 ›   get_mpf_fake.custom_fake = get_mpf_custom_fake; ›   // get_mpfをカスタムフェイク関数に設定↓
293 ›   psnd_dtq_fake.custom_fake = psnd_dtq_custom_fake;›  // psnd_dtqをカスタムフェイク関数に設定↓
294 ›   get_sw1_state_fake.custom_fake = get_sw1_state_custom_fake;↓
295 ›   get_push1_state_fake.custom_fake = get_push1_state_custom_fake;↓
296 ›   slp_tsk_fake.custom_fake = slp_tsk_custom_fake;↓
297 ↓
298 ›   sw1 = OFF;↓
299 ›   sw8 = OFF;↓
300 ›   gret_sw = ON;↓
301 ›   gret_push = ON;↓
302 ›   g1_event = 0;↓
303 ↓
304 ›   if(0 == setjmp(buf)){↓
305 ›   ›   sw_scan_task(0);↓
306 ›   }↓
307 ›   else {↓
308 ›   ›   printf( "無限ループ脱出成功 \n\n" );↓
309 ›   }↓
310 ›   EXPECT_EQ(EV_TIMER_SW1_ON, g_event[0]);/*sw1on*/↓
311 ↓
312 ›   // get_mpfがコールされていることを確認↓
313 ›   EXPECT_EQ(fff.call_history[2],(void*)get_mpf);↓
314 ›   // get_mpfの第1引数が正しいことを確認↓
315 ›   EXPECT_EQ(EVENT_MPF,get_mpf_fake.arg0_history[0]);↓
```

前ページの偽装デバイスドライバをカスタムfAKE関数に設定

sw1とPUSH1の現状態をOFFに設定

前ページの偽装されたデバイスドライバがsw1とPUSH1をONを返すように設定

setjmpライブラリ実行後、
無限ループを含むテスト対象sw_scan_taskをコール

次ページの偽装slp_tsk内のlongjmpにより無限ループから脱出する

テスト結果の検証
関数の呼び出し履歴(Nは呼び出し順):fff.call_history[N]
渡された引数の確認(Nは引数順): <関数名>_fake.argN_val

5.2. テストケースの解説(1)スイッチスキャンタスク(4/4)

- TOPPERSのサービスコール slp_tsk の偽装
  - テスト対象(無限ループ)からコールされる偽装slp_tskが2回コールされると、longjmpライブラリによって無限ループから脱出し、テストケースに戻る。結果的に、テスト対象はキースキャンを1回実行することになる。

```
57 ER slp_tsk_custom_fake(void)↓
58 {↓
59 ›   if(1 < slp_tsk_fake.call_count){↓
60 ›   ›   printf("偽装slp_tsk経由で無限ループ脱出!\n");↓
61 ›   ›   longjmp(buf,1);↓
62 ›   ›   printf( "無限ループ脱出したのでここは実行されない\n" );
63 ›   }↓
64 ›   return(E_OK);↓
65 ↓
66 }↓
```

呼び出された回数の確認:<関数名>_fake.call_count

5.3. テストケースの解説(2)LED4点滅タスク(1/5)

- テスト対象関数 blink_task (関数の詳細は「状態遷移表とフローチャート」を参照願います)
  - データキュー受信により起床するイベントドリブンなタスク
    - データキュー受信の偽装が必要となる

```
63 void
64 blink_task(intptr_t exinf)
65 {
66     const CALLBACK_BL *pFunc;
67
68     for (;;) {
69         SVC_PERROR(rcv_dtq(BLINK_DTQ, (intptr_t *)&GBL_RCVAD));
70         if(true == check_blink_event()) {
71             pFunc = &BLStateTable[GBL_NOWSTS][GBL_RCVEV];
72             if(NULL != *pFunc){
73                 (*pFunc)();    /* 現状態とイベントに対応する関数を実行 */
74             } else {
75                 DEBUG_SYSLOG(LOG_NOTICE, " ignore event. event id = %d ",GBL_RCVEV);
76             }
77             if(GBL_CTLF & DBL_STSCHG) {    /* 状態遷移要求有り？ */
78                 GBL_CTLF &= ~DBL_STSCHG;   /* 要求クリア       */
79                 GBL_NOWSTS = GBL_NEXTSTS;  /* 状態更新         */
80             }
81         }
82         DEBUG_SYSLOG(LOG_NOTICE, " rel_mpf = 0x%p", GBL_RCVAD);
83         SVC_PERROR(rel_mpf(EVENT_MPF, GBL_RCVAD));
84     }
85 }
```

5.3. テストケースの解説(2)LED4点滅タスク(2/5)

- TOPPERSサービスコール rcv_dtq の偽装
  - テスト対象(無限ループ)からコールされる偽装rcv_dtqがテストケースで設定されるloop_numを超えてコールされると、longjmpライブラリによって無限ループから脱出し、テストケースに戻る。

```
69 ER rcv_dtq_custom_fake(ID dtqid, intptr_t *p_data)↓
70 {↓
71 ›   if(loop_num < rcv_dtq_fake.call_count  )↓
72 ›   {↓
73 ›   ›   printf("偽装rcv_dtq経由で無限ループ脱出!\n");↓
74 ›   ›   longjmp(buf,1);↓
75 ↓
76 ›   ›   printf( "無限ループ脱出したのでここは実行されない\n" );↓
77 ›   }↓
78 ›   /* データキュー受信の偽装 */↓
79 ›   *p_data = (intptr_t)malloc(sizeof(T_EVENT));↓
80 ›   gp_data = *p_data;› › › › › › › › › // 解放するアドレスの検証用に保持
81 ↓
82 ›   ((T_EVENT*)*p_data) -> event = *(pScenario + (rcv_dtq_fake.call_count) - 1);↓
83 ↓
84 ›   return(E_OK);↓
85 }↓
```

呼び出された回数の確認:<関数名>_fake.call_count

固定長メモリブロックはmalloc/freeで代替する

テストケースで事前に設定したイベントを代入する

5.3. テストケースの解説(2)LED4点滅タスク(3/5)

- TOPPERSサービスコール rel_mpfの偽装
  - 解放するメモリブロックのアドレスを検証する

```
62 ER rel_mpf_custom_fake(ID mpfid, void *blk)↓
63 {↓
64 ›   EXPECT_EQ(gp_data, (intptr_t)blk);› › › › // 解放するアドレスの検証
65 ›   free(blk);↓
66 ›   return(E_OK);↓
67 }↓
```

固定長メモリブロックはmalloc/freeで代替する

5.3. テストケースの解説(2)LED4点滅タスク(4/5)

- Blinkタスクの点滅状態(消灯)での全イベント受信のテストケース(1/2)

```
294 TEST_F(testBlinkMain, test_StateBlinkLedOff)
295 {
296     int i;
297     // シナリオの定義
298     static const uint32_t scenario[] = {
299         EV_BLINK_TIMEOUT,       /* TIMEOUT通知          */
300         EV_BLINK_ACTIVE,        /* ACTIVE通知           */
301         EV_BLINK_OFF,           /* OFF通知              */
302         EV_BLINK_BLINK,         /* 0.25秒経過           */
303         EV_BLINK_NUM            /* イベント番号が不正(上限超過) */
304     };
305
306     blink_task_init(0);
307     gbl_ctl.nowsts = ST_BLINK_LED_OFF;          // 点滅状態(消灯)に書換え
308
309     loop_num = sizeof scenario / sizeof scenario[0];    // 無限ループ実行回数
310     rcv_dtq_fake.custom_fake = rcv_dtq_custom_fake;     // rcv_dtqをカスタムフェイク関数に設定
311     rel_mpf_fake.custom_fake = rel_mpf_custom_fake;     // rel_mpfをカスタムフェイク関数に設定
312
313     pScenario = &scenario[0];                   // シナリオ配列へのポインタ取得
```

この配列にEV_BLINKで定義されるメッセージを追加していきます。
要素順にデータキューにメッセージが送信されます。
配列の最大要素数はTEST_SCENARIO_MAXで調整します。

静的API（ATT_INI）により登録した初期化ルーチンは自動的に実行されないため、ここで実行する

タスクの状態を保持する変数を書き換える

5.3. テストケースの解説(2)LED4点滅タスク(5/5)

- Blinkタスクの点滅状態(消灯)での全イベント受信のテストケース(2/2)

```
315    if(0 == setjmp(buf)){
316        blink_task(0);                    // blinkタスクのメインルーチン(無限ループ)へ突入
317    }
318    else {
319        printf( "無限ループ脱出成功 \n\n" );
320    }
321
322    i = 0;
323    // TIMEOUT通知
324    EXPECT_EQ(fff.call_history[i++],(void*)rcv_dtq);EXPECT_EQ(fff.call_history[i++],(void*)rel_mpf);
325
326    // ACTIVE通知
327    EXPECT_EQ(fff.call_history[i++],(void*)rcv_dtq);EXPECT_EQ(fff.call_history[i++],(void*)rel_mpf);
328
329    // OFF通知
330    EXPECT_EQ(fff.call_history[i++],(void*)rcv_dtq);EXPECT_EQ(fff.call_history[i++],(void*)StLedOffEvOff); EXPECT_EQ(fff.call_history[i++],(void*)rel_mpf);
331
332    // 0.25秒経過
333    EXPECT_EQ(fff.call_history[i++],(void*)rcv_dtq);EXPECT_EQ(fff.call_history[i++],(void*)StLedOffEvBlink); EXPECT_EQ(fff.call_history[i++],(void*)rel_mpf);
334
335    // イベント番号が不正(上限超過)
336    EXPECT_EQ(fff.call_history[i++],(void*)rcv_dtq);EXPECT_EQ(fff.call_history[i++],(void*)rel_mpf);
337
338    // 最後、偽装したrcv_dtq内のlongjmpにより、タスクのメインルーチンを脱出
339    EXPECT_EQ(fff.call_history[i],(void*)rcv_dtq);
340}
```

setjmpライブラリ実行後、
無限ループを含むテスト対象blink_taskをコール

偽装rcv_dtq内のlongjmpにより無限ループから脱出する

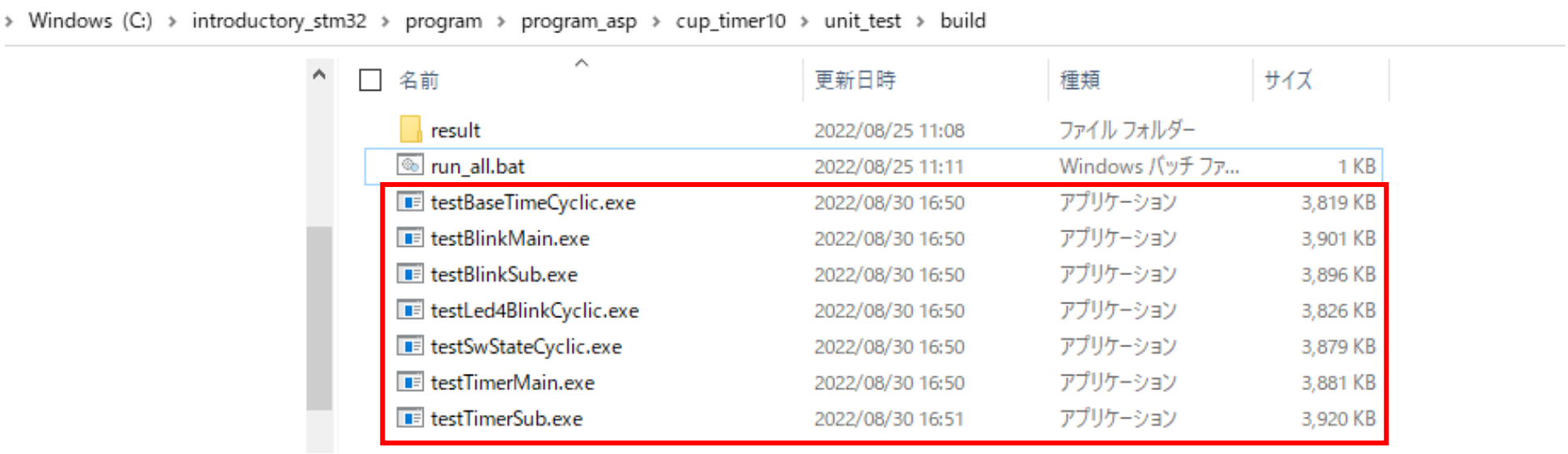
テスト結果の検証
関数の呼び出し履歴(Nは呼び出し順):fff.call_history[N]

6. 実行ファイルの生成(1/2)

- Eclipseを起動
- プロジェクトエクスプローラ -> プロジェクト(下図ではunit_test_stm32)を右クリック -> プロジェクトのビルド

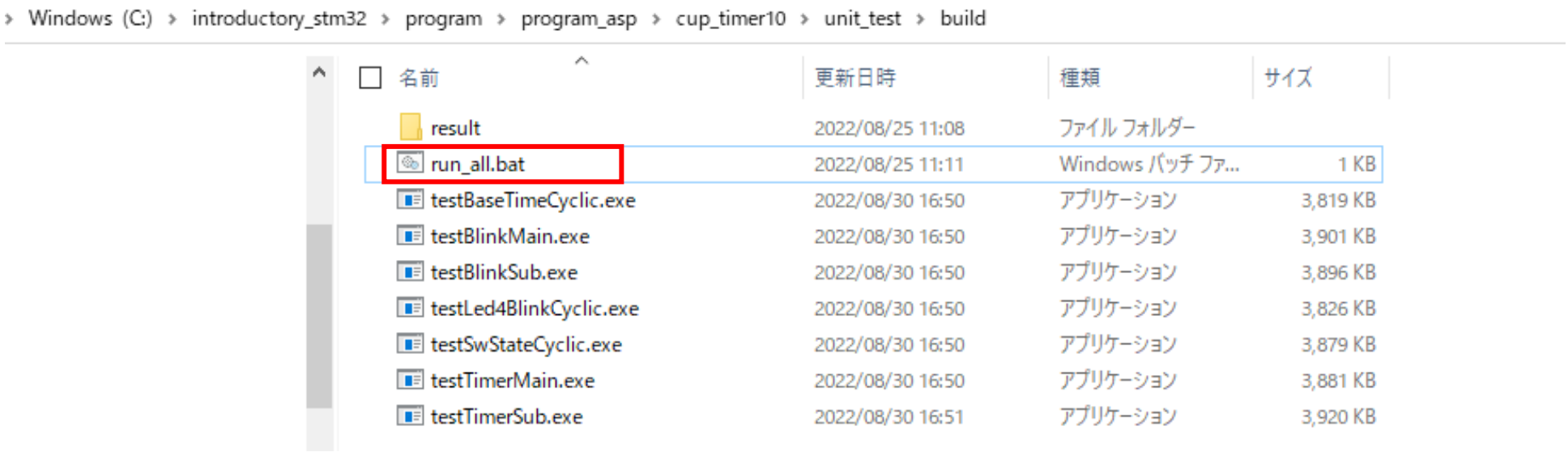
6. 実行ファイルの生成(2/2)

- ビルドが成功すると、cup_timer10¥unit_test¥build¥に、7つのexeファイルが生成されます。



7. テストの自動実行(1/2)

- Windowsのエクスプローラから、cup_timer10¥unit_test¥build¥run_all.batをダブルクリックすると、全テストケースが自動実行されます。

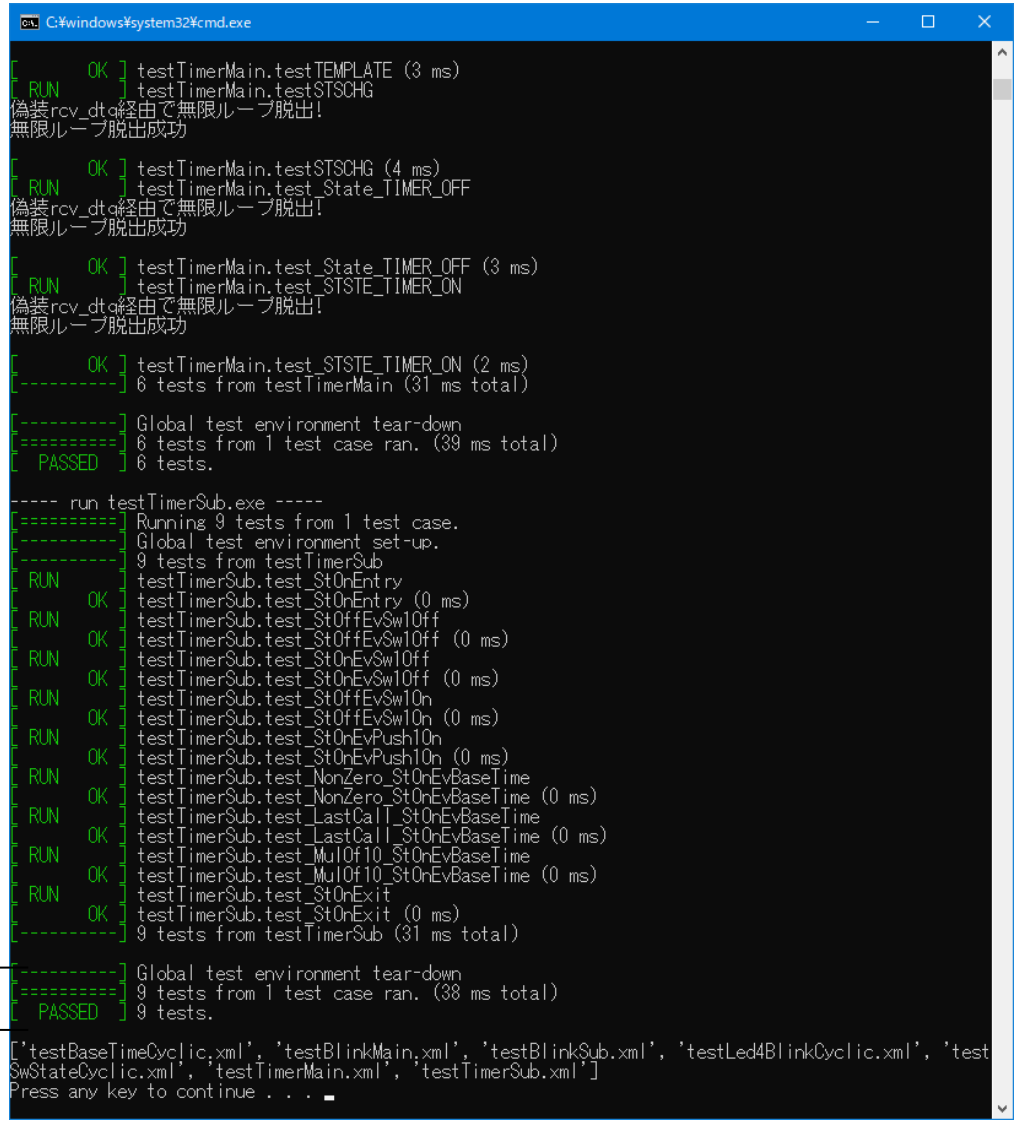


7. テストの自動実行(2/2)

- Dosプロンプトが起動し、
  テスト結果が表示されます。

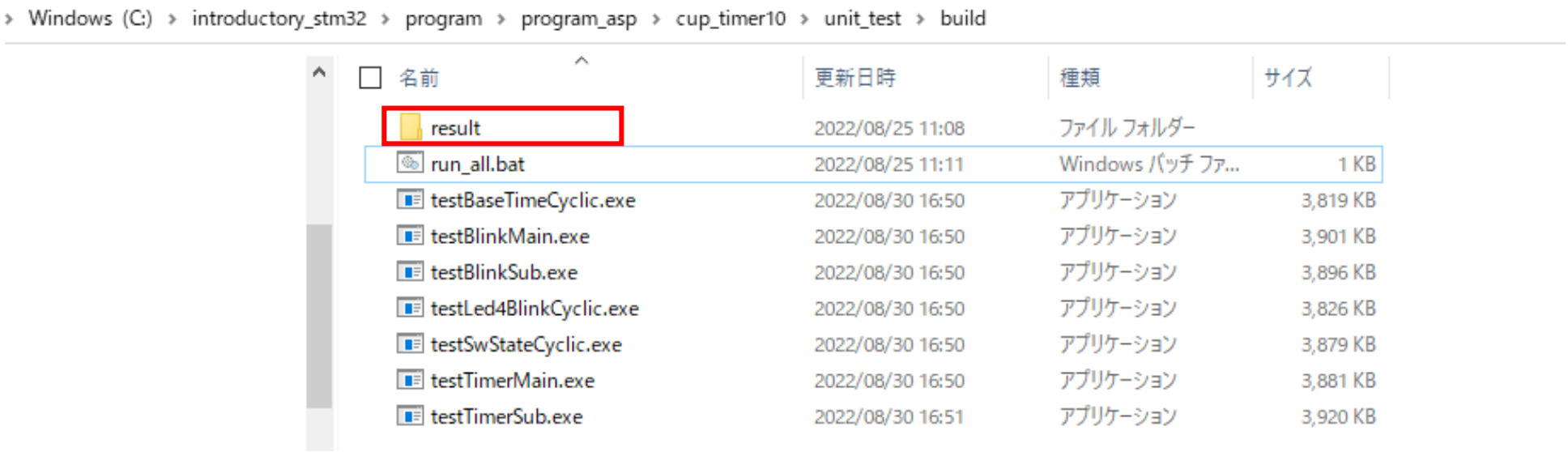
各テストケースの実行ログ

テスト全体の結果



8. テスト結果(1/10)

- テスト結果はDosプロンプトに表示されると共に、resultフォルダにxmlとcsvで出力されます。



8. テスト結果(2/10)

・タイマメイン

testTimerMain.xml

cup_timer10 > unit_test > xml_output > testTimerMain.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="6" failures="0" disabled="0" errors="0" time="0.002" name="AllTests">
  <testsuite name="test" tests="6" failures="0" disabled="0" errors="0" time="0.002">
    <testcase name="test_timer_task_init" status="run" time="0" classname="test" />
    <testcase name="testCheckTimerEvent" status="run" time="0" classname="test" />
    <testcase name="testTEMPLATE" status="run" time="0" classname="test" />
    <testcase name="testSTSCHG" status="run" time="0" classname="test" />
    <testcase name="test_State_TIMER_OFF" status="run" time="0" classname="test" />
    <testcase name="test_STSTE_TIMER_ON" status="run" time="0.001" classname="test" />
  </testsuite>
</testsuites>
```

8. テスト結果(3/10)

・タイマサブ

testTimerSub.xml

cup_timer10 > unit_test > xml_output > testTimerSub.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="9" failures="0" disabled="0" errors="0" time="0.21" name="AllTests">
  <testsuite name="test" tests="9" failures="0" disabled="0" errors="0" time="0.119">
    <testcase name="test_StOnEntry" status="run" time="0" classname="test" />
    <testcase name="test_StOffEvSw1Off" status="run" time="0" classname="test" />
    <testcase name="test_StOnEvSw1Off" status="run" time="0" classname="test" />
    <testcase name="test_StOffEvSw1On" status="run" time="0" classname="test" />
    <testcase name="test_StOnEvPush1On" status="run" time="0" classname="test" />
    <testcase name="test_NonZero_StOnEvBaseTime" status="run" time="0" classname="test" />
    <testcase name="test_LastCall_StOnEvBaseTime" status="run" time="0" classname="test" />
    <testcase name="test_MulOf10_StOnEvBaseTime" status="run" time="0" classname="test" />
    <testcase name="test_StOnExit" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>
```

8. テスト結果(4/10)

- ベースタイム周期ハンドラ

testBaseTimeCyclic.xml

cup_timer10 > unit_test > xml_output > testBaseTimeCyclic.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="2" failures="0" disabled="0" errors="0" time="0.001" name="AllTests">
  <testsuite name="test" tests="2" failures="0" disabled="0" errors="0" time="0.001">
    <testcase name="test_base_time_handler" status="run" time="0" classname="test" />
    <testcase name="test_base_time_task" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>
```

8. テスト結果(5/10)

・スイッチ

testSwStateCyclic.xml

cup_timer10 > unit_test > xml_output > testSwStateCyclic.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="6" failures="0" disabled="0" errors="0" time="0.002" name="AllTests">
  <testsuite name="test" tests="6" failures="0" disabled="0" errors="0" time="0.001">
    <testcase name="testSwScanHandler" status="run" time="0" classname="test" />
    <testcase name="testSwScanTaskSw1on" status="run" time="0" classname="test" />
    <testcase name="testSwScanTaskSw1off" status="run" time="0" classname="test" />
    <testcase name="testSwScanTaskPushon" status="run" time="0" classname="test" />
    <testcase name="testSwScanTaskPushoff" status="run" time="0" classname="test" />
    <testcase name="testSwScanTaskDouble" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>

```

8. テスト結果(6/10)

- ブリンクメイン

testBlinkMain.xml

cup_timer10 > unit_test > xml_output > testBlinkMain.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="7" failures="0" disabled="0" errors="0" time="0.003" name="AllTests">
  <testsuite name="test" tests="7" failures="0" disabled="0" errors="0" time="0.003">
    <testcase name="testBlinkTaskInit" status="run" time="0" classname="test" />
    <testcase name="testCheckBlinkEvent" status="run" time="0" classname="test" />
    <testcase name="testTEMPLATE" status="run" time="0" classname="test" />
    <testcase name="testSTSCHG" status="run" time="0" classname="test" />
    <testcase name="testStateBlinkIdle" status="run" time="0.001" classname="test" />
    <testcase name="test_StateBlinkLedOff" status="run" time="0" classname="test" />
    <testcase name="test_StateBlinkLedOn" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>
```

8. テスト結果(7/10)

- ブリンクサブ

testBlinkSub.xml

cup_timer10 > unit_test > xml_output > testBlinkSub.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="8" failures="0" disabled="0" errors="0" time="0.234" name="AllTests">
  <testsuite name="test" tests="8" failures="0" disabled="0" errors="0" time="0.154">
    <testcase name="testStIdleEvTimeOut" status="run" time="0" classname="test" />
    <testcase name="testStIdleEvAvtive" status="run" time="0" classname="test" />
    <testcase name="testStLedOffEvOff" status="run" time="0" classname="test" />
    <testcase name="testStLedOnEvOff" status="run" time="0" classname="test" />
    <testcase name="testStLedOffEvBlinkLast" status="run" time="0" classname="test" />
    <testcase name="testStLedOffEvBlinkContinue" status="run" time="0" classname="test" />
    <testcase name="testStLedOnEvBlinkLast" status="run" time="0" classname="test" />
    <testcase name="testStLedOnEvBlinkContinue" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>

```

8. テスト結果(8/10)

- LED4点滅周期ハンドラ

testLed4BlinkCyclic.xml

cup_timer10 > unit_test > xml_output > testLed4BlinkCyclic.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="2" failures="0" disabled="0" errors="0" time="0" name="AllTests">
  <testsuite name="test" tests="2" failures="0" disabled="0" errors="0" time="0">
    <testcase name="testLed4BlinkTimeHandler" status="run" time="0" classname="test" />
    <testcase name="testLed4BlinkTimeTask" status="run" time="0" classname="test" />
  </testsuite>
</testsuites>

```

8. テスト結果(9/10)

- Pythonによる XML-CSVパース
  - GoogleTestが出力したテスト結果XMLをCSV形式に成形して出力するプログラム

main.py

C: > pleiades > workspace > Python > gtestXml_parser > main.py

```python
from xml.etree import ElementTree
import codecs
import csv


def main():
    obj = ElementTree.parse('testResult.xml').getroot()

    with codecs.open('csv/0_Total Result.csv', 'w', 'utf-8') as file:
        res = csv.writer(file, delimiter=',', lineterminator='\r\n', quotechar='"')

        res.writerow(["test name", "test case count", "failue count", "error count", "execute time[ms]"])

        for val in obj.iter('testsuite'):
            res.writerow([val.get('name'), val.get('tests'), val.get('failures'), val.get('errors'), val.get('time')])

            with codecs.open((f'csv\{val.get("name")}.csv'), 'w', 'utf-8') as file:
                writer = csv.writer(file, delimiter=',', lineterminator='\r\n', quotechar='"')
                writer.writerow(["test case", "test status", "failure count", "execute time[ms]"])
                for v in val.iter('testcase'):
                    writer.writerow([v.get("name"), v.get("status"), len(list(v.iter('failure'))), v.get("time")])


if __name__ == "__main__":
    main()
```

8. テスト結果(10/10)

- テスト結果 概要出力
  (0_Total Result.csv)

	A	B	C	D	E
1	test name	test case count	failue count	error count	execute time[ms]
2	testTimerMain	6	0	0	0.002
3	testTimerSub	9	0	0	0.119
4	testBaseTimeCyclic	2	0	0	0.001
5	testSwScanCyclic	6	0	0	0.001
6	testBlinkMain	7	0	0	0.003
7	testBlinkSub	8	0	0	0.154
8	testLed4BlinkCyclic	2	0	0	0

- テストケース 詳細出力
  （testSwStateCyclic.csv）

	A	B	C	D
1	test case	test status	failure count	execute time[ms]
2	testSwScanHandler	run	0	0
3	testSwScanTaskSw1on	run	0	0
4	testSwScanTaskSw1off	run	0	0
5	testSwScanTaskPushon	run	0	0
6	testSwScanTaskPushoff	run	0	0
7	testSwScanTaskDouble	run	0	0

9.テストの応用

- 再現性の低いバグの検出

  再現性の低いバグは、タスク間メッセージの受信の順番に起因して発生することがよくあります。

  例)タスクAからaメッセージ受信(めったに受信しない)

  　　タスクBからbメッセージ受信(同上)

  　　タスクCからcメッセージ受信(同上)

  　　→この順番で受信した場合のみ不具合発生

- このような検証には,全タスクの全メッセージの組合せをテストしなければならないため,テストケースが膨大になります。
- 本書で紹介したテストケースは単純なメッセージIDの1次元配列ですが、この配列を改造することで,複数のタスクからの複数のメッセージの偽装が可能となります。夜仕掛けて朝には数兆通りのメッセージ受信順のテスト結果が出来上がっているといった運用が期待できます。

10. まとめ

- デバイス不要の単体テストが実現できた
  - 実機を使わずに、PC上で単体テストを完結できた
- テストの自動化で、網羅性・柔軟性の高いテストが容易に実現できた
  - 数多いテストケースやシナリオを容易に実行することができた
  - 実際のデバイス操作では実現しにくいケースも容易にテストすることができた
- テストケース作成の知識を得ることができた
  - テストをする際の考え方などを知ることができた