SWEST25 2023/9/1 セッションS2

【テストからより良い組込みソフトウェア開発を考える】

パーソルクロステクノロジー株式会社
第1技術開発本部　第4設計部　設計2課　阿部耕二

目次

自己紹介

- 名前: 阿部　耕二（あべ　こうじ）
- 所属: パーソルクロステクノロジー株式会社
  第1技術開発本部 第4設計部 設計2課
- 医療機器の組込みソフトウェア開発。C言語。
- 趣味: 宇宙開発（リーマンサットプロジェクト広報メンバー）
- LAPRASポートフォリオ: https://lapras.com/public/k-abe
- Twitter: @juraruming

TDDを学ぼうと思った背景

- 昨年度、医療機器のバージョンアップで新機能の追加のたびに既存機能が動かなくなるデグレードが多く発生した。
  繰り返されるデグレードの課題解決の案として、テスト駆動開発(TDD)の学習をはじめた。
- 学んだこと、実験したことを共有します。何かお役にたてばと思います。

理想のテスト体制について考える

現在のテスト体制

- 実機を使ったテストがテストのメインになっている。
  テストに工数がかかる。
  機材の準備、テスト環境の構築

-> なにか良い手はないか?と思っていた🤔

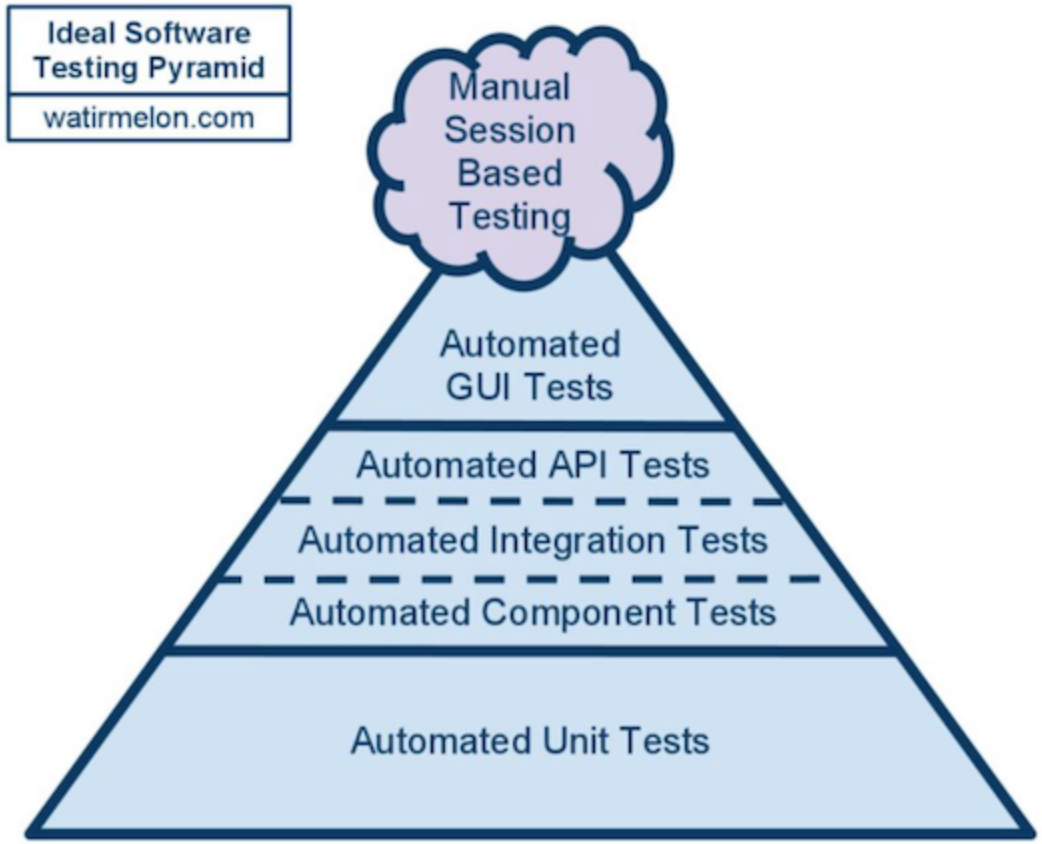
- こちらの動画を見てテスト体制の理想、進む方向が明確になった。

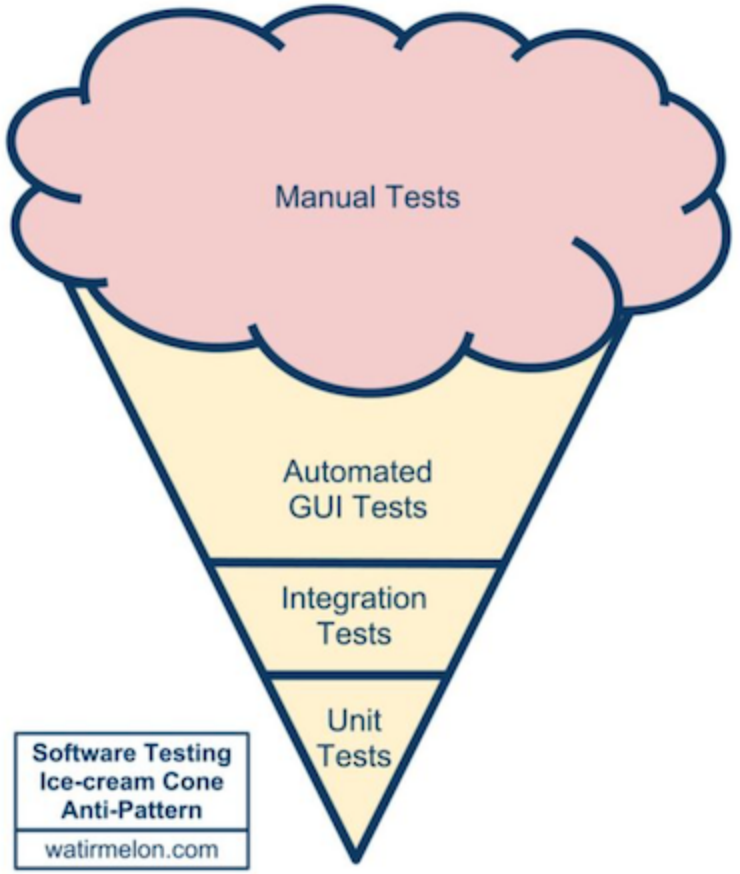
参考資料1:
「サバンナ便り～自動テストに関する連載で得られた知見のまとめ～」t wada（和田 卓人）

- 理想のテスト体制で目指すところは**ピラミッド型**。
  アイスクリームコーン型のテスト体制から脱却しピラミッド型へ近づきたい!!!

テストピラミッドとアイスクリームコーンアンチパターン

参考資料2より引用

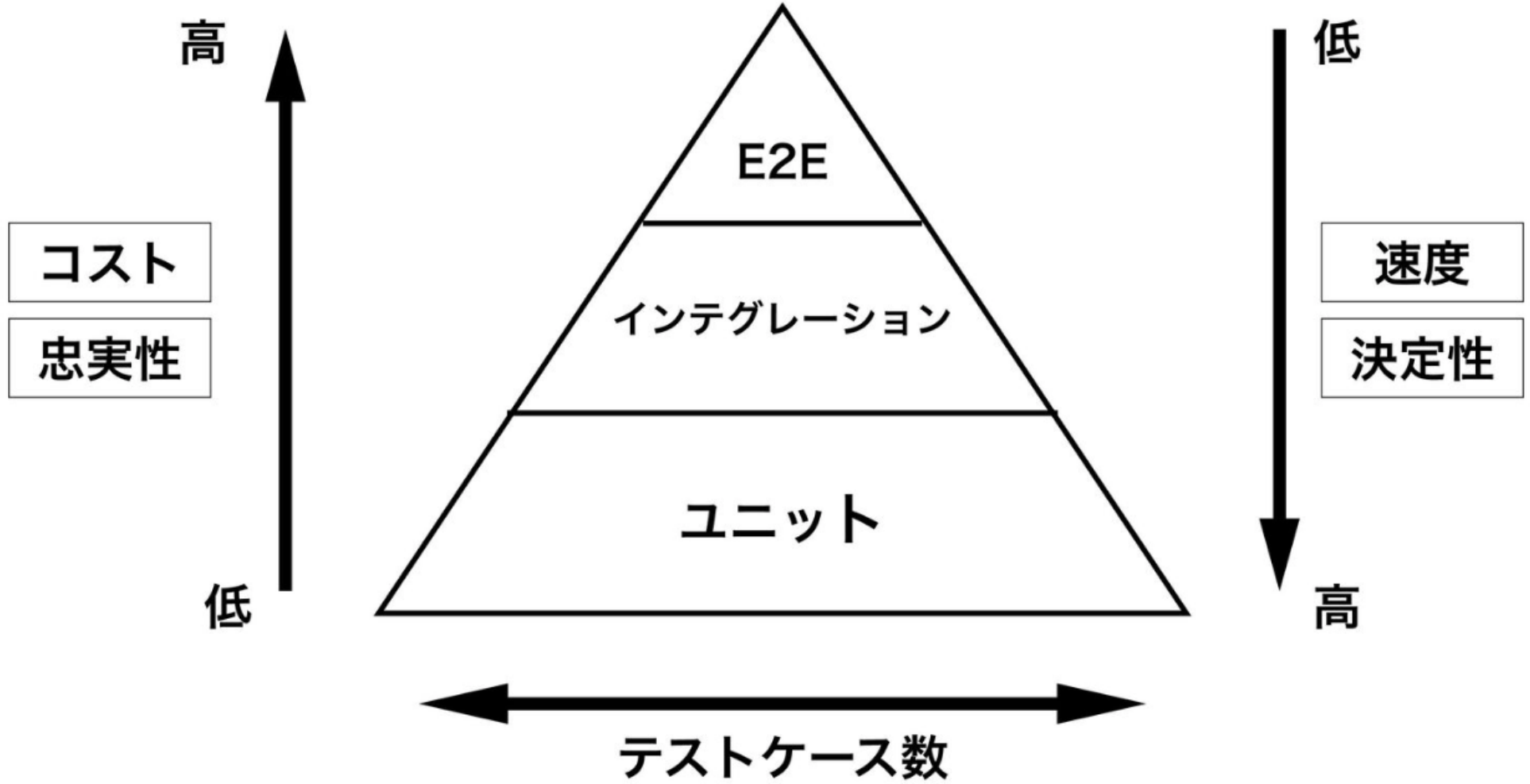




https://watirmelon.blog/2012/01/31/introducing-the-software-testing-ice-cream-cone/

テストピラミッド

参考資料2より引用。開発者のマシンで行うテストはコスト・忠実性が低い。速度が早いという特徴がある。



組込みでTDDするうえでの課題

- クロス環境
  ホスト環境、ターゲット環境
  なにをホストでテストし、なにをターゲットでテストするか?
- ホスト環境とターゲット環境の違い
  コンパイラ、コンパイラの制約、不具合
  リンクするライブラリ、ライブラリの制約、不具合
  アーキテクチャ（型のサイズ、エンディアン）

- テスト環境の制約（テストフレームワーク）
  ターゲット環境ではマイコンリソースにより使えるテストフレームワークが限られる。

マイコンではつぎの選択肢になることが多い感触

- Unity
- CppUTest

- テスト環境の制約（テスト実行環境）

TDDは

- RED⇒GREEN⇒リファクタリング
  のサイクルを早く・リズムよく回すことが望ましい。

ターゲット環境（マイコン）はホスト環境に比べてマシンパワーが貧弱。

- テストコードのダウンロードに時間がかかる
- FlashROMの書き換え回数が気になる

などなど、組込みソフトウェア開発特有の課題がつきまとう。

そんな課題がある組込みソフトウェア開発でTDDをするには?

- なにをホスト環境でやり、なにをターゲット環境でやるのか**テスト戦略・組織の方針**が重要と感じた。
- ターゲット環境でリズムよくTDDをするための方法として、RAMでテストコードをダウンロード・テストする、などがある。

組込みでTDDする際のTips

TDDの学び、実験で得たTipsを共有します。

1. ターゲットの統合開発環境でTDDする例
2. NTShellでテスト環境を整える
3. テストを学びのツールとして使う

1. ターゲットの統合開発環境でTDDする例

STM32マイコンの統合開発環境(以降STM32CubeIDE)にC/C++のテストフレームワーク(CppUTest)を環境構築した例

STM32CubeIDEにCppUTestを環境構築し、STM32マイコンでTDDする

- 評価ボード: STMicroelectronics NUCLEO-F446RE
- CppUTestはライブラリ化しリンクした(v3.8にて確認。最新はv4.0)
- ITMでSTM32CubeIDEのSWV ITM Data Consoleにテスト結果を表示する
- RAMでテストを実行する

STM32マイコンの統合開発環境(以降STM32CubeIDE)にC/C++のテストフレームワーク(CppUTest)を環境構築した例

- 評価ボード NUCLEO-F446RE ハードウェアスペック

- STM32F446RET6 64ピン
- Arm Cortex-M4コア 180MHz
- フラッシュ: 512Kbyte
- SRAM: 128Kbyte
- Arm Mbed対応

STM32マイコンの統合開発環境(以降STM32CubeIDE)にC/C++のテストフレームワーク(CppUTest)を環境構築した例

- 確認できたこと

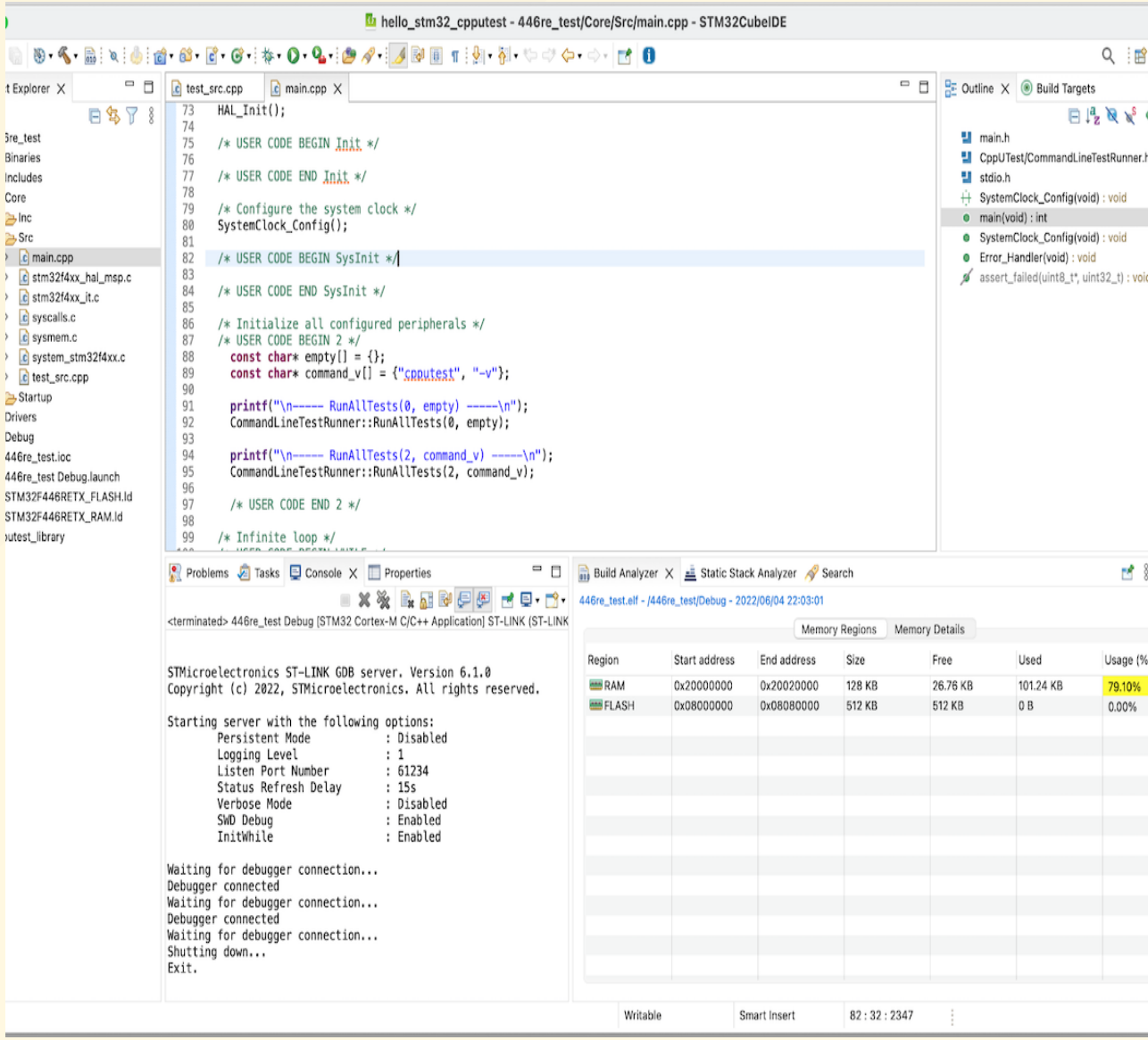
- STM32CubeIDEの中でTDDができた。
  CppUTestのテスト失敗・成功が期待とおりに動作していることを確認した。
- ITMでテスト結果を表示することができた。
  STM32CubeIDE以外にシリアルコンソールなどのツールも不要。

他のSTMicroelectronics評価ボードでも同様にTDD環境を構築可能なはず（未確認）。

STM32マイコンの統合開発環境(以降STM32CubeIDE)にC/C++のテストフレームワーク(CppUTest)を環境構築した例

- 参考）メモリ使用量

- RAM使用量は79.10%となった(128KBのうち101.24KBを使用)。



2. NTShellでテスト環境を整える

前述の**【1. STM32マイコンの統合開発環境(以降STM32CubeIDE)にC/C++のテストフレームワーク(CppUTest)を環境構築した例】**ではテストを任意のタイミングで実行することができなかった。

そこでシリアルコンソールからのコマンドをトリガにして、軽量シェル（NTShell）経由でテストを実行できる環境を作った。

STM32CubeIDEにCppUTestを環境構築し、STM32マイコンでTDDする(2) ～シリアル通信でCppUTestを実行する～

- NTShellとは?

組込み向けの軽量なシェル。
Natural Tiny Shell (NT-Shell)

NTShellを使うことで容易にシェル、シェル経由で実行するコマンドを実現できる。
今回は**cpputest**というコマンド名でCppUTestを使ったテスト機能を組み込んだ。

```
>help
help    :This is a description text string for help command.
info    :This is a description text string for info command.
read_userbutton :User button B1 reads.
write_led       :Write to LED LD2.
cpputest        :Exec CppUTest.
```

- 使用方法

シリアルコンソールに接続し、コマンド名称をタイプし、エンターキーを押下する。

```
>cpputest
..
../Core/Src/test_src.cpp:10: error: Failure in TEST(FirstTestGroup, FirstTest)
        FAIL: FirstTestGroup, FirstTest


.
Errors (1 failures, 3 tests, 3 ran, 3 checks, 0 ignored, 0 filtered out, 0 ms)
```

test_src.cpp

```
TEST(FirstTestGroup, FirstTest)
{
    FAIL("FAIL: FirstTestGroup, FirstTest\n");
}

TEST(FirstTestGroup, SecondTest)
{
    CHECK_EQUAL_ZERO(0);
}

TEST(FirstTestGroup, IntSize)
{
    LONGS_EQUAL(4, sizeof(int));
}
```

- 使用方法2

CppUTestのオプションをそのまま使える。下記は【-v】オプションを指定し、テストの詳細を表示している。
どのテストを実行しているか明確になった。

```
>cpputest -v
TEST(FirstTestGroup, IntSize) - 0 ms
TEST(FirstTestGroup, SecondTest) - 0 ms
TEST(FirstTestGroup, FirstTest)
../Core/Src/test_src.cpp:10: error: Failure in TEST(FirstTestGroup, FirstTest)
        FAIL: FirstTestGroup, FirstTest


 - 0 ms

Errors (1 failures, 3 tests, 3 ran, 3 checks, 0 ignored, 0 filtered out, 0 ms)
```

- 使用方法3

CppUTestのオプションの一部を紹介する。

- cpputest -v -r3
  テストをn回繰り返すオプションは-rn。実行回数で挙動が変わること・または変わらないことをテストする時などに使えそう。
- cpputest -gFirstTestGroup -nFirstTest
  -gxxxにはテストグループ名、-nxxxにはテスト名を指定する。指定テストグループの指定テストを実行する。テストグループの任意のテストだけを実行したいときに使えそう。

3. テストを学びのツールとして使う

もしTDDをプロダクトコードのテストのみに使っていたらもったいない。
つぎの用途にTDDを使った事例を紹介する。

1. プログラミング言語の学習としてTDDを使う
2. ライブラリの挙動を確認する

1. プログラミング言語の学習としてTDDを使う

- C++のデフォルト引数の動作を学習するテスト
  プログラミング言語の学習にTDDを使うのも有効かと思う。
  C++のデフォルト引数の動作を学習するテスト

デフォルト引数の学習用コードの定義

```
#ifndef __DEFAULT_ARGUMENT_H
#define __DEFAULT_ARGUMENT_H

int defargtest_add(int a = 1, int b = 2);

#endif
```

```
// デフォルト引数を使わない
TEST(CppLerningTestGroup, defaultArgumentNone)
{
        LONGS_EQUAL(7, defargtest_add(3, 4));
}

// 第1引数のみデフォルト引数 -> コンパイルエラー
//TEST(CppLerningTestGroup, defaultArgument_arg1)
//{
//      LONGS_EQUAL(4, defargtest_add(, 3));
//}

// 第2引数のみデフォルト引数
TEST(CppLerningTestGroup, defaultArgument_arg2)
{
        LONGS_EQUAL(2, defargtest_add(0));
}

// 第1, 2引数ともにデフォルト引数
TEST(CppLerningTestGroup, defaultArgument_arg1_2)
{
        LONGS_EQUAL(3, defargtest_add());
}
```

1. プログラミング言語の学習としてTDDを使う

今回は例としてC++のデフォルト引数の動きを確認するテストだった。プログラミング言語の学習をするテストや組込みソフトウェアの特徴的な動作をテストを書いて確認することは学習としても良いと思う。

- ポインタのアドレス演算
- CPUのint型のサイズ
- エンディアンの確認
- 初期値付き変数コピー、0クリア領域の0クリア
- その他

2. ライブラリの挙動を確認する
   ライブラリで動作に確認が持てない場合などはテストで確認する。
   今回は2038年問題の対応状況を確認する目的のテストを書き、確認した。

2038年問題確認用テストコード

【テスト実施の背景】
以前、仕事でファイルシステムのミドルウェアが組み込まれたソフトウェアを確認した時に2038年問題に対応していなかった。ドキュメントを見ると2038年問題対応のマクロがデフォルト無効でビルドされていた。STM32CubeIDEの対応はどうなっているか気になったのでテストしてみた。

```
#include "CppUTest/TestHarness.h"
#include <time.h>
#include <string.h>
#include "2038.h"

TEST_GROUP(FutureProblemTestGroup)
{

};

TEST(FutureProblemTestGroup, pre2038problem)
{
        char date_str[256];

        memset(date_str, 0x00, sizeof(date_str));
        time_t tmr = 2147483647; /* UTC:2038/01/19 03:14:07 日本(UTC+9): 2038/01/19 12:14:07 */

        get_date_string(tmr, date_str, sizeof(date_str));
        STRCMP_EQUAL("2038/01/19 03:14:07", date_str);
}

TEST(FutureProblemTestGroup, 2038problem)
{
        char date_str[256];

        memset(date_str, 0x00, sizeof(date_str));
        time_t tmr = 2147483647; /* UTC:2038/01/19 03:14:07 日本(UTC+9): 2038/01/19 12:14:07 */
        tmr += 1; /* UTC:1901/12/13 20:45:52 日本(UTC+9): 1901/12/14 05:45:52 */

        get_date_string(tmr, date_str, sizeof(date_str));
//      STRCMP_EQUAL("1901/12/13 20:45:52", date_str);  // このテストは失敗することを確認した
        STRCMP_EQUAL("2038/01/19 03:14:08", date_str);  // テスト成功: 2038年問題対応済みであることがわかった
}
```

- テストの結果、2038年問題に対応していることがわかった。
  今思えばtime_tの型を調べれば早く2038年問題に対応しているかわかりますね😅

テストに注目し、より良い組込みソフトウェア開発を考える

1. テストのための時間を捻出する
2. テストをしやすくする便利なツールを使う
3. クラウド環境（Arm Virtual Hardware）でテストする
4. テストをデグレード発見レーダーとして使う
5. テストを資産とする
6. ChatGPTを使ってテストする
7. テストからより良い設計を導く

1. テストのための時間を捻出する

時間がないからテストできない。
→考えればつぎの例のように時間を捻出できることもある。

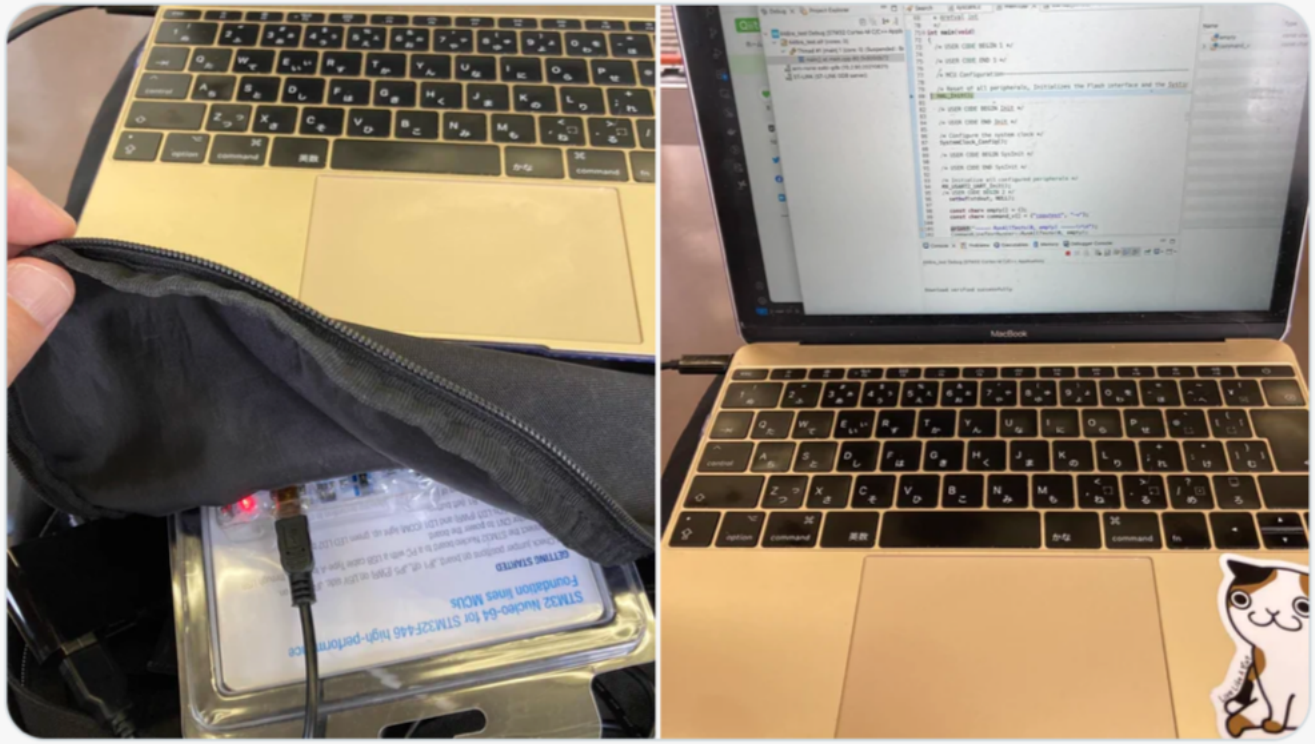
k-abe@組み込まれた猫使い🧙‍♂️ @juraruming · 7月5日

通勤電車内でマイコンボードに給電してデバッグします（仕事ではなく個人のプロダクト）。

ポイントはSTM32マイコンボードのプラスチック外装を切り欠き、USB給電していることです。

これによりマイコンボードをプラスチック外装で保護しつつデバッグできます✌️

通勤電車の時間は貴重ですからね。



2　10　345

k-abe@組み込まれた猫使い🧙‍♂️
@juraruming

>ポイントはSTM32マイコンボードの
>プラスチック外装を切り欠き、
>USB給電していることです。

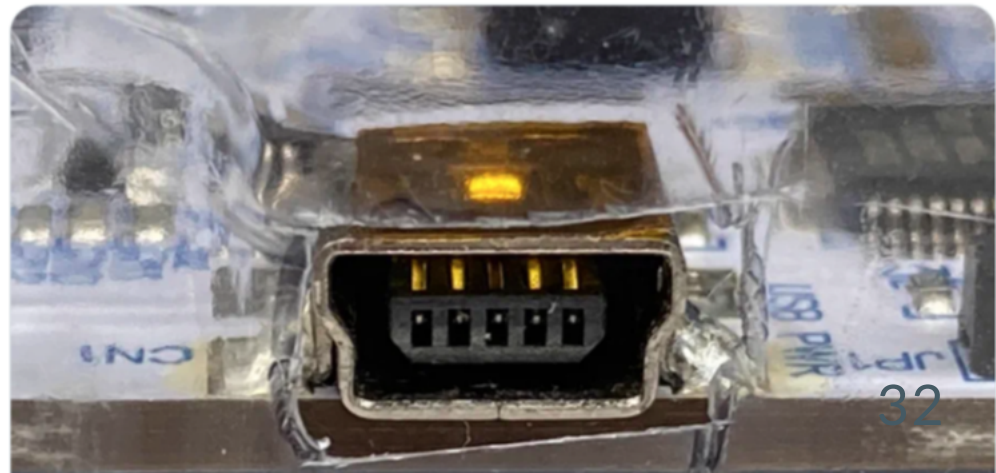
拡大するとこんな感じです。

メリット:
・カバー外さなくてもデバッグ可能。
・ユーザーボタン、リセットボタンはプラスチック外装の上からでも押せた。

デメリット:
・切り欠き部分からの異物混入のリスク有り。
・プラスチック外装のままだとピンヘッダにワイヤを接続できない。

マイコンボード単体でデバッグする、
ピンヘッダに配線しない、
移動中もデバッグしたい、

という方にはお勧めできる改造かと思います。



前ページの電車内デバッグの例はネタだが、

開発現場でも考えれば時間を捻出できるポイントはあるかもしれない。

2. テストをしやすくする便利なツールを使う

1. NTShell
2. GoogleTestとFake Function FrameworkによるTOPPERSアプリケーションの単体テスト自動化
3. FFF（Fake Function Framework）
4. サニタイザ
5. FTDIデバイス MPSSE

1. NTShell
   NTShellでテスト環境を整えるでも紹介した軽量シェル

Natural Tiny Shell (NT-Shell)

ターゲットでテストをしたいとき、テストのコマンドを受けつける窓口にしている。
容易に組み込むことができるので重宝している。
個人的に大好きなOSS。

2. GoogleTestとFake Function FrameworkによるTOPPERSアプリケーションの単体テスト自動化

2022 第12回 TOPPERS活用アイデア・アプリケーション開発コンテスト 活用アイデア部門 銀賞のアイデア

TOPPERS活用アイデア・アプリケーション開発コンテスト

- コンテスト応募資料
- 教材コンテンツ

GoogleTestとFFF（Fake Function Framework）で実機不要な自動単体テスト環境を構築できる。

3. FFF（Fake Function Framework）

FFFのことは【GoogleTestとFake Function FrameworkによるTOPPERSアプリケーションの単体テスト自動化（以降、単体テスト自動化）】で知った。
単体テスト自動化はGoogleTestとFFFを組み合わせて、TOPPERSアプリケーションをテストしていたがFFFはベアメタルなシステムにも使える。
スタブをつくったりテストしやすくなるためのいろいろな機能があるのでFFFのGitHubのREADMEを読んでみることを薦める。

FFF

4. サニタイザ

組込み装置の実機確認を行うと時間がかかる。
テスト工数は**実機確認よりも前の早い段階**で不具合が見つけられればも増加しない。
たとえば、コンパイル時に不具合がみつけられれば実機確認よりもテストのコストが安くなる。
そういった場合は

- アドレスサニタイザ
- リークサニタイザ

を使い、メモリに関するエラーがないかチェックするのも良い。

5. FTDIデバイス MPSSE

I2C・SPIのI/Fをもつセンサーからデータ取得し動作するシステムがあるとする。センサーを制御するマイコン部分のハードウェアがないから動作確認が進められない、というケースがあるかもしれない。

FTDI社のMPSSEケーブルがあればPCとセンサーを接続しセンサーの制御プログラム動作確認を進めることが可能。
ソフトウェアのツールだけではなく、ハードウェアのツールも効果的に使えばテスト工数を減らす場合がある。

FTDI社のMPSSEケーブル：
https://ftdichip.com/product-category/products/cables/usb-mpsse-spi-i2c-jtag-master-cable-series/

3. クラウド環境（Arm Virtual Hardware）でテストする

Arm Virtual Hardwareというクラウドの開発環境がある。
クラウドで開発することで

- 実機レス
- CIのサイクル回す

など従来の実機を使ったテストと比べてメリットがありそう。

4. テストをデグレード発見レーダーとして使う

冒頭話したTDD学習のきっかけはデグレードに困っていたため。
もしTDDの環境が整備されており、新機能実装時に前に成功していたテストが失敗すればデグレードに早く気づき、テスト工数を減らすことができていた。

テストを蓄積し、活用していくことでデグレードを早期に発見するレーダーのような役割を果たすことができる。

5. テストを資産とする

テストを蓄積することでデグレード発見レーダーになる、と書いた。
そのほかにもテストが資産となること例について考えた。

例えば出荷テストプログラムをTDDで使ったテストをベースにして書く、などに使えるかもしれない。
TDDでテストした観点で出荷用テストプログラムを書けば、1から出荷用テストプログラムを書くよりも早く、確実なプログラムになりそう。

6. ChatGPTを使ってテストする

ChatGPTを使ったソフトウェア開発・TDDについて言及している書籍を購入したので共有する。

ソフトウェア開発にChatGPTは使えるのか？
――設計からコーディングまでAIの限界を探る
小野哲　著

【5-2 TDDによるテストからの実装】でChatGPT（本ではGPT-4を使用）でTDDを試行していた（コードはPython）。

- テストコード、関数の入出力仕様をプロンプトに入力し、テストケースを満たすコード生成を依頼する。
- 生成されたコードを見て、他のコード生成を依頼する
  何度でも爆速でコードを生成してくれる。
- テストをパスするコードを生成してくれたらリファクタリングを依頼する

まさにTDDのプロセスをChatGPTで回していた。
気になった方は確認してみては如何でしょうか?
私も読書だけでプロンプトはまだ試せていないので確認します🎶

7. テストからより良い設計を導く

TDDをはじめてみると楽しい。
TDDのレッド→グリーン→リファクタリングをリズムよく回せた時は
特に楽しく、快感。

でもふと思う。
テストすることが目的だったのか?、と。

TDDはその名のとおりテストから駆動する開発手法だった。
何を駆動するかというと【良い設計】。

テストしやすい構造は良い設計といえる。

組込みでTDDをおこなう際の定番本【テスト駆動開発による組み込みプログラミング】でも後半は設計原則SOLIDに触れている。

良いテストを書き、
良い設計を駆動・導き出し、
高凝集・疎結合な構造のソフトウェアをつくる。

そのような構造のソフトウェアは変更しやすく、流用もしやすい。
チーム、組織、会社によい資産となりビジネスの競争力が高まる。

アイスクリーム型のテスト体制からピラミッド型のテスト体制に以降していけばテスト工数の削減、エンジニアの労働時間削減で成長のための学習時間創出にもつながる。

時間の創出はエンジニアの幸福度UPにもつながる。

これからもテストに注目し、より良い組み込みソフトウェア開発を考えていきます。

ご清聴ありがとうございました🙇

宣伝です。

テストしやすいソフトウェアは良い設計、ということで設計（SOLID原則）について学ぶ勉強会を開催しています。

もしよければ一緒に学びませんか?
次回は9/28(木) 20:45-21:45 Xのスペースで下記で開催予定です。

- **ソフトウェア設計原則【SOLID】を学ぶ #3 依存性逆転の原則**

お待ちしています🎶

こちらは先日開催した勉強会です。
ソフトウェア設計原則【SOLID】を学ぶ #2 インターフェース分離の原則

参考資料

1. youtube: 「サバンナ便り～自動テストに関する連載で得られた知見のまとめ～」t wada（和田 卓人）
2. speakerdeck: サバンナ便り～自動テストに関する連載で得られた知見のまとめ（2023年5月版）
3. STM32CubeIDEにCppUTestを環境構築し、STM32マイコンでTDDする
4. STM32CubeIDEにCppUTestを環境構築し、STM32マイコンでTDDする(2) ～シリアル通信でCppUTestを実行する～

5. ソフトウェア開発にChatGPTは使えるのか？
   ――設計からコーディングまでAIの限界を探る
   小野哲　著
6. テスト駆動開発による組み込みプログラミング
   ―C言語とオブジェクト指向で学ぶアジャイルな設計
   James W. Grenning　著、蛸島 昭之　監訳、笹井 崇司　訳