Szoftver tesztelés

Bevezető

A tesztelés témájának az államvizsga projektemet választottam, ez egy mikrovezérlő alapú rendszer amely C++-ban van megírva. Amelyhez tartozik egy PCB terv és a szoftver a mikrovezérlőn ami a programot futtatja és egyben teszteli saját magát is amennyiben ez be van állítva. A tesztelés célja, hogy az eredményeket helyesen kiszámolja és hogy az elvárásoknak megfelelően működjön. Ilyen például alkatrész nem felismerése, amikor fel kellene ismerje. Az ilyen problémákat jó hamar felismerni, mivel az ilyen rendszeren amennyiben a végső felhasználó találja meg a hibát akkor az problémát okoz és komplikált az összes rendszer frissítése a piacon. Sok modern alkalmazás frissítést interneten keresztül el tudja végezni, viszont itt nincs ilyen lehetőség, mivel a rendszernek nincs internet kapcsolata. Mivel a rendszer mindenki számára elérhető kell legyen, még azoknak is akiknek minimális számítógépes ismerete van, nem biztos, hogy mindenki el fogja tudni majd végezni a frissítést ezen kívül, mivel ez önálló rendszerként is működhet, így lehet nincs számítógép a közelben.

Alapos teszteléssel kisebb az esély (nem 0), hogy valamilyen katasztrofális hiba keletkezzen. Lehetséges, hogy egyes esetekben nem fog helyesen működni, viszont az általános esetekben a felhasználó nem fog hibákat észlelni.

Mivel a rendszer mikrovezérlő alapú, így az általános teszt könyvtárak nem alkalmazhatóak, különböző limitációk miatt. Ezért létrehoztam egy osztály sablont, mely csak akkor jön létre, amennyiben tesztelni akarjuk a rendszert, ez lefutás során kiírja, hogy melyik osztály melyik tesztje mit hibázott, ez szabadon változtatható, így a teszt eset megírásakor pontosan megadható, hogy mi kellett volna történjen, de mi történt helyette. Lehetséges exception-ok tesztelése is, mivel sok osztály tud exception-t dobni, ami a rendszer leállását eredményezné és még az exception üzenetét se írná ki a Serial Porton keresztül amin keresztül látható a rendszer kimenetei, így bár láthatóvá válik, hogy milyen okból állt le a rendszer. Így lényegében egy egyszerűsített teszter keretrendszer lesz létrehozva.

Sok esetben nem lehet csak ilyen osztály tesztelni mindent, sok esetben az adat nem biztos, hogy elér a célhoz (szakadt vezeték), vagy az adatmennyiség túl nagy, hogy egy mikrovezérlős rendszeren limitált erőforrásokkal tesztelni lehessen. Az ilyen esetekben külső tesztekkel lehet ezt elvégezni, ilyen például egy oszcilloszkóp ami segítségével tesztelhetőek gyors jelek anélkül, hogy a rendszer működését megzavarnánk, viszont az oszcilloszkópok drágák így nem mindig alkalmazhatóak.

Működés

A kódbázis OOP alapú, ahol a különböző modulok különböző osztályokba van szedve. Minden modulnak van egy interface osztálya amely megadja a használható függvények listáját és annak paramétereit. Ebben van minden függvény előtt egy rövid komment, hogy mit csinál, mik a paraméterei és mit fog visszatéríteni és esetlegesen milyen exceptionokat tud dobni. Hasonlóan az alábbi példához :

```
/*
*Get Measurement data by name
@param measurement,(const std::string &measurement), which output mode was used during the measurement, for example
 "405"
 @return (std::vector<double>) with 3 value, avaraged voltage of that setting if exist
 *THROW NOSUCHMEASUREMENT if there is no such saved measurement
*/
virtual std::vector<double> getMeasurement(const std::string &measurement) const = 0;
```

A kód egyszerűen átalakítható tesztelő módban, amikor csak a tesztek futnak és nem futtatja a kód többi részét, így más nem fogja zavarni a kód működését, csak ami szükséges. Így egy elszigetelt teszt eset jön létre amely tökéletes a komponens tesztekre, mivel nem lesz esetlegesen még egy másolat, vagy egy másik szál, függvény belezavarjon a tesztbe. Erre a #define TESTS definiálásával valósul meg. Ha definiálva van akkor a csak a teszt esetek futnak és nem a fő kód, ami #ifndef TESTS között van, míg a teszt esetek a #ifdef TESTS közt vannak. A másik probléma a tesztelés során, hogy az ADC alapértelmezetten próbál olvasni a portról, viszont ez tesztelés során nem alkalmazható, mivel akkor külsőleg csatlakoztatni kell pont egy olyan eszközt, mint a teszt eset, ami lassú, mivel manuálisan kell elvégezni és limitált teszt lehetőségek vannak, mivel nincs lehetőség minden eset tesztelésére, mivel az nagy mennyiségű különböző eszközre lenne szükség. Erre a lehetőség az alapértelmezett portról olvasást áttéríteni adat kérésre exceptionokon és üzenetein keresztül. Erre a lehetőség a #define ADCDISABLE -vel lehetséges, ami ha definiálva van akkor az alapértelezett port olvasásról áttér az adat kérésre, így egyedi adatokat lehetséges megadni. Ezek a global.h -ban találhatóak meg, ebben az esetben mindkettő definliálva van, így a test eset van érvényben és az ADC -ről nem lehet olvasni.

```
//if commented out then the normal program runs without test cases
//if not then only the test cases run
#define TESTS

//disables ADC read
#define ADCDISABLE
```

A fő részek különböző mappákban vannak elhelyezve, amely segíti az osztályok megtalálását a projekt fejlesztése során. Viszont a kódbázis túl hosszú, így csak azok a részek lesznek leírva, amelyek fontosak ehhez a projekthez. Az osztályok abban az esetben amikor egy másik osztályt használnak akkor annak az osztálynak az interface osztályát megkapja paraméterként a konstruktorban és elmenti egy saját tárolóban, így nincsenek limitálva csak egy osztályra és könnyen kicserélhető egy hasonló osztályra ami ugyan azt az interface-t implementálja.

Áramköri rajz:

A projekt tartalmaz egy áramköri rajzot is, amin a mikrovezérlő és a többi szükséges IC található. Ennek az a lényege, hogy a komponenseket összekösse egy átlátható és egyszerű módon. Lehetséges minden pontot egy-egy külön vezetékkel összekötni, viszont ez sokkal költségesebb és a vége egy átláthatatlan vezeték csomó lesz, miközben nagyobb az esély a vezeték megszakadására és a jelek zajosabbak lesznek, ami ebben az esetben a legtöbb helyzetben nem okoz gondot, mivel a jelek digitálisak, viszont az analóg feszültségek érzékenyek az ilyenre, így ezt csökkenteni kell.

Metadológia

A rendszer kimenete a tesztelt komponens adatai, minél nagyobb pontossággal.

A rendszer fel kell ismerje az ellenállásokat, kondenzátorokat, diódákat és tranzisztorokat, miközben meghatározza a lábkiosztását.

A rendszer minél gyorsabb legyen, 5 másodpercnél tovább nem tarthat egy futás, ez is csupán abban az esetben amikor nagy méretű kondenzátorok (nagyobb mint 25 mF) tesztelésre kerül sor, ebben annyi időbe telik míg feltelik a legkisebb ellenálláson keresztül, vagy 30s-ig amikor ad egy hibát, hogy „akkumulátor”.

A rendszer ellenállások esetén 5%-os pontosságot kell elérjen 5 Ohm és 5 Mega Ohm értékek közt, miközben a maximális mérési tartomány 0 és 10 Mega Ohm közt található.

A rendszer kondenzátor esetében a pontosság 5%-alatt kell legyen, és a mérési tartomány 100nF és 85mF közt található.

A diódák esetében meg kell határozza a dióda nyitó feszültségét és az irányát. A nyitó feszültség pontossága 5% alatt kell legyen és 2 inverz dióda esetén mindkét dióda értékét kimutatni.

A rendszer meg kell tudja különböztetni az NPN és a PNP típusú tranzisztorokat és meghatározni, hogy MOSFET típusúak-e. Az tranzisztor erősítési értékét 10%-os pontossággal, míg a nyitó feszültséget és a feszültség esést 5%-os pontossággal meghatározni. Amennyiben mérhető parazita kapacitása abban az esetben ezt is kiszámolni 10%-os pontossággal.

Tesztek

Automatizált szoftver tesztek:

Mivel a mikrovelő nem tud biztosan láthatólag üzeneteket kiírni, így a Serial Porton keresztül küldi az üzeneteit. Ehhez nem szükséges külső eszköz, csupán az USB kábel ami egyben a tápfeszültséget is adja (viszont sok USB kábelnek nincs adat vezetéke, így erre oda kell figyelni). Amennyiben az USB a számítógéphez van csatlakoztatva akkor kell egy program ami képes a Serial Porton küldött üzenetek kiírására. Ezt a Putty programmal

A tesztek egy része a mikrovezérlő program kódjára összpontosított, itt a számítások helyességére és a funkcionalitások működése van tesztelve. Ezek a tesztek automatikusan le tudnak futni és a helyességüket ellenőrzik, a működésük hasonlóan működik, mint a Gtest, vagyis kiírják a teszt eredményét, hogy a teszt sikeres volt vagy sem. Ezek sikertelen teszt esetén kiírják a hibát és tovább futnak, hiba esetén meg azt is kiírja, hogy mennyi kellene a helyes eredmény legyen és hogy melyik osztály melyik tesztjében történt a hiba. Sikeres lefutás esetében az osztály és teszt eset kerül kiírásra.

```
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 50.0000 instead it is 53.9394!
```

A tesztelő osztály a következőképpen néz ki:

A konstruktorban az osztály megkapja a tesztelendő osztályt és a printer osztályt amivel majd ki fogja írni a teszt sikerességét, vagy a hibáit. Miután a konstruktor meghívódott és a privát adattagok értékadása után még a konstruktorban meghívódnak a teszt függvények, melyek tesztelik az osztályt, így külsőleg nincs szükség a függvényeket meghívni, de lehetőség van rá.

```
#pragma once

#include "../../../Calculate/include/BaseCleanInput.h"
#include "./TestPrinter.h"

class BaseCleanInputTest
{
private:
    BASECLEANINPUT *inputTest;
    TESTPRINTER* testprinter;
public:
    BaseCleanInputTest(BASECLEANINPUT *inputTest, TESTPRINTER* testprinter);
    ~BaseCleanInputTest();

    void AVGVoltage();
    void IsAnythingConnected();
};
```

A TESTPRINTER osztályban lehetőség van az üzenetek kiírására, egy stringben meg kell adni, hogy melyik teszt volt futtatva és az hiba esetén lehetséges a számított és helyes érték kiírására is. A successPrinter függvényt a sikeres teszt esetében kell használni, ami a paraméterekből egy könnyebben átláthatóbb üzenetet generál és kiírja a Serial Porton keresztül. Az errorPrinter is hasonlóan működik. A description paraméter opcionális, de lehet itt is megadni üzeneteket.

```
#pragma once
#include <iostream>

class TESTPRINTER
{
private:
    /* data */
public:
    TESTPRINTER(/* args */) {}
    ~TESTPRINTER() {}
    void errorPrinter(const std::string &functionName, const double calculatedValue, const double correctValue) const;
    void errorPrinter(const std::string &functionName, const std::string& description) const;
    void succesPrinter(const std::string &functionName, const std::string& description) const;
};
```

Nem automatizált szoftver tesztek:

Nem minden tesztet lehet automatizálni, legfőképpen a teszt mérete miatt. Mivel a memória eléggé limitált és a processzort sem lehet sokáig lefoglalni így más módszerrel kell próbálkozni. Ez legfőképpen a külső kommunikációnál fontos amikor nagy adat mennyiségű adat kerül elküldésre és nem lehet biztosra menni, hogy helyesen érkezik meg és a másik eszköz helyesen értelmezi. Ezt manuálisan lehet a legolcsóbban elvégezni, mivel egy ilyen teszt eszköz beszerzése költséges.

A kijelző tesztelése ezzel a módszerrel történt, ahol több hibát is felfedeztem. Mivel a mikrovezérlő egyik lábát nem helyesen forrasztottam, így nem mindig érintkezett és a jelek nem mindig érkeztek meg a kijelzőhöz. A mikrovelő SPI-n keresztül kommunikál, viszont a kijelző parancsok esetében 8 bit-es adatokat vár, míg adat esetében 16 bit-et, így átállítás nélkül az alapértelmezett 8 bitenként küldte még akkor is ha 16 bites értéket kapott meg kiküldésre, amivel a 16 bites szín utolsó 8 bitjét elhagyta és a színek nem megfelelő módon jelentek meg. Ezt egy új függvénnyel javítottam ki, amivel egyszerűen lehet változtatni a kiküldendő adat méretét.

```
void SPI::changeFormat(bool dub)
{
    if (dub)
        spi_set_format(spi_Hw_inst, 16, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
    else
        spi_set_format(spi_Hw_inst, 8, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
}
```

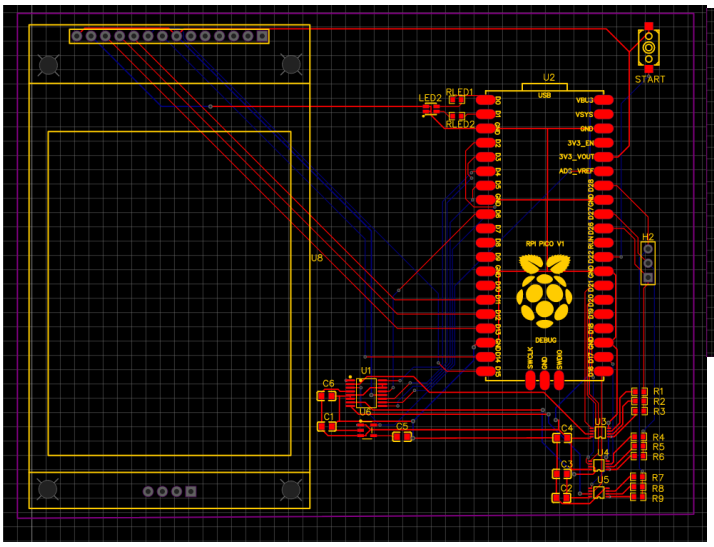
Ezen kívül végeztem egy stabilitás tesztet is, viszont még a kijelző maximális SPI órajel többszörös meghaladása esetén sem történt látható elváltozás, minden helyesen jelent meg.

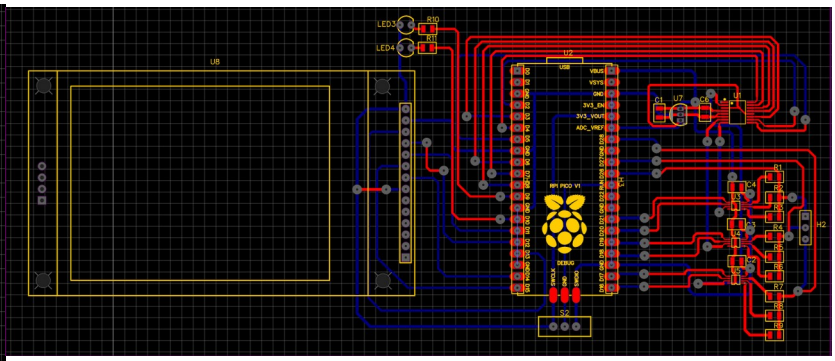
Áramkör terv tesztelése:

Ez a tesztelés 2 lépésben történt meg. Első lépésben a fő paraméterek meghatározása, minta vonal vastagság és távolságok majd a második lépésben a megvalósíthatóság tesztelése.

Az első verzió esetén a tervrajz elkészítése után a terv átment a teszt első fázisán, viszont több probléma is felmerült amikor az megvalósíthatóság és a standardok betartására nézve. Az egyik főbb probléma, hogy a nem a minimális vezeték vastagsággal kell mindent vezetni, hanem csak ha vastagabbal nem lehetséges, az ajánlott sokkal vastagabb, mint a minimális, hogy a gyártás során egy esetleges hiba sokkal kisebb eséllyel okozzon problémát. Ezen kívül a standardok be nem tartásából következő hibák, amik szintén gondot okozhatnak majd a későbbiekben. Ilyen a mikrovelő közvetlen beforrasztása a nyomtatott áramkörbe és nem egy socketbe, amiből könnyedén ki lehet cserélni. A vezetékek vezetésénél is figyelni kell, hogy magas frekvenciás jelek ne legyenek analóg jelek közelében és a kanyarodási szögek betartása is, hogy nem ajánlott sem a hosszú átlók, sem a derékszögek használata. Ezért, amitől a teszt első fázisán átment a terv attól a megvalósítási problémák miatt újra kellett tervezni, míg megfelelt az elvárásoknak. Ezen kívül a THC (through hole components) komponensek esetében nehéz vagy lehetetlen ugyanazon az oldalon oldalon forrasztani, mint a komponens, így míg az első verzióban ez többször megtörtént, addig a második verzióban ezek a hibák kijavításra kerültek és ebben az esetben a vezetékek a lap másik oldalán találhatóak, így nem zavarják a forrasztást.

A lenti ábrán a bal oldali az első verzió látható, míg a jobb oldalon az átjavított verzió.





Észrevételek:

A kijelző a rendszeren nem ugyan olyan sorrendben tárolja az adatokat, mint ahogyan a mikrovezérlő, így küldés előtt meg kell forgatni, hogy a kijelzőn az elvárt eredmény jelenjen meg, ezt egy rövid függvény segítségével valósítható meg. Enélkül a függvény nélkül a színek nem úgy jelennének meg, mint ahogyan elvárt lenne és nem lehetne a 16 bites színkódokat használni külső forrásokból közvetlenül.

```
uint16_t COMMON::swap_bytes(uint16_t color)
{
   color = (color & 0xFF00) >> 8 | (color & 0x00FF) << 8;
   color = (color & 0xF0F0) >> 4 | (color & 0x0F0F) << 4;
   color = (color & 0xCCCC) >> 2 | (color & 0x3333) << 2;
   color = (color & 0xAAAA) >> 1 | (color & 0x5555) << 1;
   return color;
}
```

Mivel a mikrovezérlő egy ARM M0+ processzoron alapszik így nincs beépített floating point szupportja, viszont lehet kódban használni, mivel szoftveresen meg van valósítva, viszont ezzel azt a problémát észleltem, hogy a pontossága nem elégséges, így a számítások akár 5-10%-os hibákat eredményezhet ami túl nagy, hogy pontos számításokra lehessen alkalmazni.

```
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 50.0000 instead it is 53.9394!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 220.000 instead it is 237.333!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 550.000 instead it is 593.333!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 1000.000 instead it is 1078.79!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 3300.00 instead it is 3560.00!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 5000.00 instead it is 5393.94!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 10000.0 instead it is 10787.9!
ERROR! BASECLEANINPUT BaseCalculateTest calcResistance error, value supposed to be 50000.0 instead it is 53939.4!
Sleeping
```

Egy főbb probléma amit tesztelés alatt vettem észre, hogy az ADC (Analog Digital Converter) nem 0-t ad amikor 0V feszültség van kapcsolva és nem a maximális 2^{12} amikor a tápfeszültség. Ezt egy lineáris korrekcióval korrigáltam, mivel ez is nagyban növeli a pontosságot és pontosabban lehet ilyen feszültséget rákapcsolni. Lehetőség lehetne több mérési pont beiktatására, viszont arra külső eszközök kellenek, míg ez a megoldás más projektek esetében is működik és gyorsan kiszámítható.

```cpp
uint16_t *ADCCORRECTER::offsetCorrection(uint16_t *samples,const uint16_t samplesSize)
{
    if (samplesSize == 0 || samples == nullptr)
    {
        throw NOTSUPPOSEDTOREACHTHIS("ADC correcter got samplesize 0 or null array");
    }
    for (int i = 0; i < samplesSize; i++)
    {
        samples[i] = (samples[i] * VCCOffset) - gndOffset;
    }
    return samples;
}
```