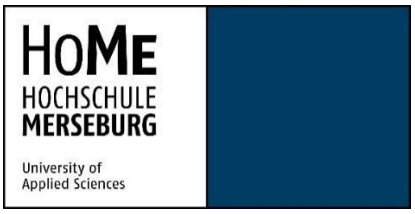
emotas
embedded
communication

Hochschule Merseburg (FH)

University of Applied Sciences



Fachbereich Ingenieur- und Naturwissenschaften

Fachgebiet: Angewandte Informatik

Bachelorarbeit

Zur Erlangung des Grades Bachelor of Science (B. Sc.)

Entwicklung und Bewertung einer Modultest-Umgebung für eingebettete Systeme unter Verwendung von automatisierten Testlösungen

Vorgelegt bei

Erstprüfer: Prof. Dr. -Ing. Rüdiger Klein

Zweitprüfer: Dipl. Ing. Andreas Böbel

eingereicht von:

Name: Alawak, Ziad

Vorwort

An dieser Stelle möchte ich gerne die Gelegenheit nutzen, um die Motivation und die wichtige Ziele dieser Forschung zu erläutern.

Die Technologie der eingebetteten Systeme ist ein wesentlicher Bestandteil, der heutzutage in vielen elektronischen Geräten, Computern, Flugzeugen und Verkehrsmitteln zu finden ist. Ohne sie wären diese Geräte nutzloses Alteisen.

Aufgrund der Komplexität der eingebetteten Systeme bzw. Mikrocontroller geht ab und zu das verwendete Systeme schief, das natürlich immer richtig funktionieren muss.

Die Zuverlässigkeit dieser Systeme ist von entscheidender Bedeutung, und Modultests sind ein wesentlicher Bestandteil, um diese Zuverlässigkeit zu gewährleisten.

Diese Arbeit konzentriert sich auf die Analyse verfügbarer automatisierter Modultest Lösungen, Design und Implementierung einer Modultest Umgebung für eingebettete Systeme, die auf die Bedürfnisse eingebetteter Systeme zugeschnitten ist.

Danksagung

Ich danke Gott, der mir Erfolg gegeben hat, mein Studium abzuschließen und diese wissenschaftliche Arbeit zu beenden.

Ich möchte mich bei den Herren Prof. Dr.-Ing. Rüdiger Klein und meinem Zweitprüfer Dipl. Ing. Andreas Böbel für ihre wertvolle Unterstützung und Anleitung bedanken. Ebenso gilt mein Dank der Firma emotas embedded communication GmbH, bei der ich meine Bachelorarbeit durchgeführt habe. Ich schätze ihre Hilfe von den zuverlässigen Entwicklern, die mir immer zur Seite standen.

Darüber hinaus bedanke ich mich recht herzlich bei allen Professoren und Lehrern, die sich sehr bemüht haben, ihr Bestes zu tun, um das nötige Wissen und die Fähigkeiten zu vermitteln, um in der Informatik herausragende Leistungen zu erzielen.

Besonders danken möchte ich meinen Eltern. Ihr habt wegen mir viele Nächte durchgemacht und nie gezögert, mir eure ganze Begeisterung und Unterstützung zu schenken, damit ich meine Ziele nicht aus den Augen verliere. Ohne eure Ermutigung stünde ich heute nicht hier.

Außerdem bedanke ich mich bei meiner Familie und Freunden für ihre Ermutigung während meines Lebens und auch meiner Studienzeit.

Ich möchte auch in liebevoller Erinnerung an meine Großväter danken, die zwar nicht mehr unter uns weilen, deren Liebe, Weisheit und Ermutigung aber immer ein Teil von mir sind. Ruhe in Frieden.

Abkürzungen

CPU : [Englisch: Central Processing Unit], [Deutsch: Zentrale Verarbeitungseinheit].

OS : [Englisch: Operating System], [Deutsch: Betriebssystem].

ROM : [Englisch: Read Only Memory], [Deutsch: Festwertspeicher].

RAM : [Englisch: Random Access Memory], [Deutsch: Arbeitsspeicher].

RTOS : [Englisch: Real Time Operating System], [Deutsch: Echtzeitbetriebssystem].

LSB : [Englisch: Least Significant Byte], [Deutsch: Niederwertigstes Byte].

MSB : [Englisch: Most Significant Byte], [Deutsch: Höchstwertiges Byte].

TI : Texas Instruments (Firma Name).

ARM : [Englisch: Advanced RISC Machines], [Deutsch: Fortgeschrittene RISC Maschinen].

SRAM : [Englisch: Static Random Access Memory], [Deutsch: Statischer Arbeitsspeicher].

API : [Englisch: Application Programming Interface], [Deutsch: Anwendungsprogrammierschnittstelle].

CI/CD : [Englisch: Continuous Integration and Continuous Delivery], [Deutsch: Kontinuierliche Integration und Kontinuierliche Auslieferung].

KI : Künstliche Intelligenz.

MCU : [Englisch: Master Control Unit], [Deutsch: Hauptsteuereinheit].

CCS : Code Composer Studio (Environment Name).

HALCoGen : [Englisch: Hardware Abstraction Layer Code Generator], [Deutsch: Codegenerator für die Hardware-Abstraktionsschicht].

IDE : [Englisch: Integrated Development Environment], [Deutsch: Integrierte Entwicklungsumgebung].

1 Einleitung

1.1 Eingebettete Systeme

Ein eingebettetes System bzw. *(embedded System*) ist eine Sammlung von informationsverarbeitende Systeme und Hardware, die sich in den elektronischen Produkten wie z.B. Computer, Handy und verschiedene Systeme befindet, um die Kontrolle und die Steuerung der Datenverarbeitungsaufgeben zu übernehmen, wobei sie auch programmierbar sein könnte. [1]

Es besteht ein Unterschied zwischen den herkömmlichen Computer, die für eine Vielzahl von Anwendungen konzipiert sind, und die eingebettete Systeme, die oft für bestimmte Aufgabe optimiert ist, und in einem größeren System integriert ist.

Die Entwicklung von dem Software des eingebetteten System unterscheidet sich von der herkömmlichen PC-Systemen, weil sie stärker auf die Ein und Ausgabefunktionen konzentriert, und für jedes System neu entwickelt werden muss. [2] [3]

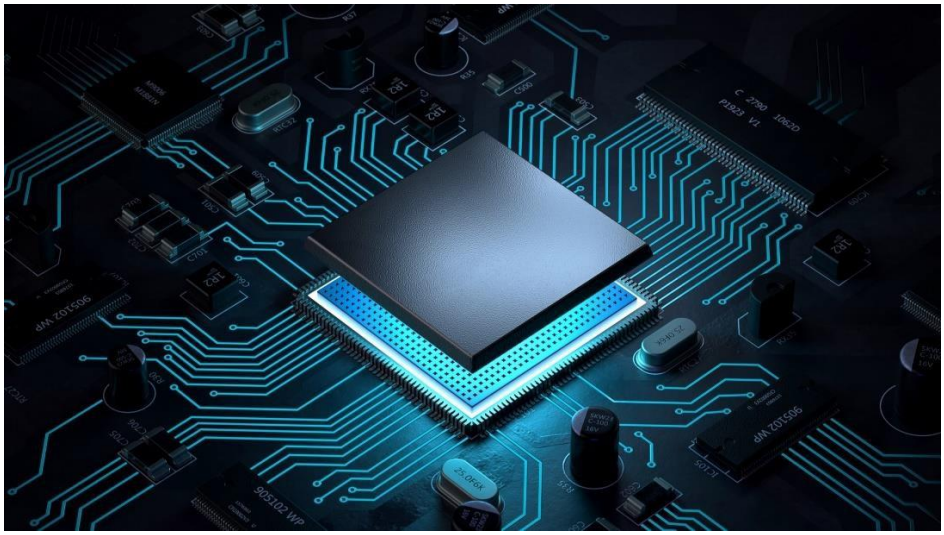


Abbildung 1:(Image credit: Shutterstock)

1.1.1 Eine Übersicht

Die Hardware von eingebetteten Systemen besteht normalerweise aus Mikroprozessoren oder Mikrocontrollern, wobei dieser Prozessor ein Kernstück von einer integrierten Schaltung ist, und aus Echtzeitanforderungen basiert.

[1] (1)
[2] (1)
[3] (2)

Dieser Mikroprozessor hat nur ein **CPU** und zusätzlich einen Speicher und auch Peripheriegeräte.

Sie befinden sich häufig in Geräten wie z.B. die Fahrzeugsteuerungen, Robotern und medizinischen Geräten.

Einfache eingebettete Systeme sind in der Lage, ohne Betriebssystem zu funktionieren und werden in der Maschinensprache der **CPU** programmiert.

Jedoch können die komplexeren Systeme auf spezialisierte **OS** zurückgreifen, besonders wenn die Echtzeitumgebungen bedient werden müssen. Eingebettete Firmware, die in den Speicher eines Geräts geschrieben wird, spielt eine wichtige Rolle bei der Steuerung der Geräte.[4] [5]

1.1.2 Geschichtliche Entwicklung der Eingebettete Systeme

Die Geschichte der eingebetteten Systeme stammt aus 1960er Jahren und hat sich Schritt für Schritt mit in der Mikroelektronik verknüpft. Einer der wichtigste Meilenstein dieser Entwicklung war, ist die Arbeit von ***Charles Stark Draper*** im Jahr 1961. Er entwickelte einen innovativen integrierten Schaltkreis, um die Größe und das Gewicht des Apollo-Führungscomputer zu verringern. Dieser Computer war den ersten Art der integrierte Schaltkreis, und war richtig entscheidend für die Echtzeitdaten-Erfassung von den Flügen während der ***Apollo-Missionen***.

Ein anderer wichtiger Schritt in der Entwicklung von eingebetteten Systemen erfolgte Mitte der 1960er Jahre. ***Autonetics***, das heute ein Teil von der amerikanische Firma ***Boeing*** ist, hat den ***D-17B-Computer*** für das ***Minuteman***-I-Raketenleitsystem entworfen.

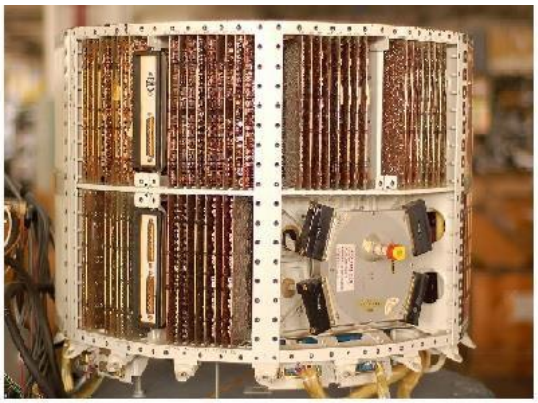
Dieser wird oft als das erste in Serie produzierte eingebettete System anerkannt.

Mit der Übergang zur ***Minuteman*** II im Jahr 1966 kam das ***NS-17***- Raketenleitsystem, das für seine intensive Nutzung von integrierten Schaltkreisen bekannt ist.

Im Jahr 1968 war ein weiteres Einsatz eines eingebetteten System in einem Fahrzeug: Der deutsche Firma ***Volkswagen 1600*** nutzte ein Mikroprozessor zur Steuerung ihre elektronischen Kraftstoffeinspritzsystems.

[4] (3)
[5] (4)

Minuteman I



Autonetics D-17 guidance computer from a Minuteman I missile

Abbildung 2:Fig 2: Minuteman-I missile and Autonetics D-17

Ein besonderer Wendepunkt war am Ende der 1960er und Anfang der 1970er Jahre, da die Preise von den Schaltkreisen sanken und die Verwendung richtig stark zunahm.

Am Anfang der 1970er herstellte die amerikanische Firma ***Texas Instruments*** ihren ersten Mikrocontroller, der zu ***TMS 1000-Serie*** gehört.

Dieser Mikrocontroller wurde im Jahr 1974 rausgekommen, der ***4-Bit-Prozessor***, **ROM** und **RAM** enthielt.

Während des Jahres 1971 brachte die Firma ***Intel*** mit dem ***4004*** den ersten Mikroprozessor heraus, und später von 8-Bit ***8008*** und dem 8080 im Jahr 1974.

Ende des 1980er begann eine neue Ära von eingebetteten Operating System und zwar: im Jahr 1987 veröffentlichte ***Wind River*** das **RTOS *VxWorks***, und danach folgte ***Microsoft*** 1996 mit ***Windows Embedded CE.Gegen*** Ende der 1990er begann die Embedded ***Linux*** Ära, und wegen seine Flexibilität wird heutzutage in einer Vielzahl von eingebetteten Geräten verwendet.[6]

1.1.3 Arten von Embedded System

Dies sind entscheidende Systeme in Bereichen, die zeitkritische Prozesse erfordern. Beispiele für Bereiche, in denen solche Systeme unentbehrlich sind, sind die Luft- und Raumfahrt, die Verteidigung und die Automobiltechnik. Dazu gehören Steuerungssysteme für Flugzeuge und Fahrzeuge, die Daten von Sensoren analysieren und überwachen, sowie Raketenabwehrsysteme. Diese Systeme sind in zwei Unterkategorien unterteilt: „Soft Real-time Embedded Systems“ und „Hard Real-time Embedded Systems“.[7]

1.1.3.1 Soft Real-time Embedded Systems

[6] (2)
[7] (5)

Diese Systeme sind in Bezug auf die Antwortzeit ziemlich flexibel. Verzögerungen können die Systemleistung beeinträchtigen, jedoch kein unmittelbares Risiko darstellen. Typische Beispiel ist ein Computersystem, das Umweltdaten wie Temperatur und Luftfeuchtigkeit analysiert sowie Online-Meetings, die manchmal aufgrund der Internetgeschwindigkeit Verzögerungen erfahren.[8]

1.1.3.2 Hard Real-time Embedded Systems

Diese Systeme müssen strikte Zeitvorgaben erfüllen und könnten durch Unterbrechungen schwerwiegender Folgen ausgesetzt sein. Beispielsweise, das Raketenabwehrsystem könnte durch Verzögerungen lebensgefährlich sein.[9]

Kriterium	Soft Real-Time Systeme	Hard Real-Time Systeme
Reaktionszeit	Flexibel, Verzögerungen sind akzeptabel und führen nicht zu kritischen Fehlern	Strikt, Verzögerungen sind inakzeptabel und können zu schwerwiegenden Fehlern führen
Konsequenzen einer Verzögerung	Leistungsminderung möglich, aber ohne gravierende Auswirkungen	Kann kritische Auswirkungen haben, möglicherweise katastrophale Folgen
Wichtigkeit der Pünktlichkeit	Pünktlichkeit ist wünschenswert, aber nicht entscheidend	Pünktlichkeit ist absolut entscheidend für die Systemfunktionalität
Einsatzgebiete	Anwendungen, bei denen Verzögerungen nicht lebensbedrohlich sind	Mission-kritische Anwendungen
Flexibilität	Höhere Flexibilität in Bezug auf die Verarbeitung	Weniger Flexibilität, da die Einhaltung strenger Zeitvorgaben erforderlich ist

1.1.4 Endianness:

Das Endianness ist ein Begriff, der die Reihenfolge von Bytes in den Computer oder in Mikrocontroller bezeichnet, um das maschinelle Wort zu lesen und zu speichern.

Ähnlich wie die arabische Sprache von rechts nach links und die englische von links nach rechts gelesen wird, behandelt Endianness die Reihenfolge, in der Bytes innerhalb eines Computerspeichers gelesen werden. Diese Richtung des Lesens ist entscheidend, da Inkompatibilitäten zwischen verschiedenen Systemen auftreten können, insbesondere wenn Daten zwischen Computern, die Bytes in unterschiedlicher Reihenfolge lesen, ausgetauscht werden.[10] [11]

[8] (5)
[9] (5)
[10] (6)
[11] (7)

Endianness unterteilt sich in zwei Haupttypen:

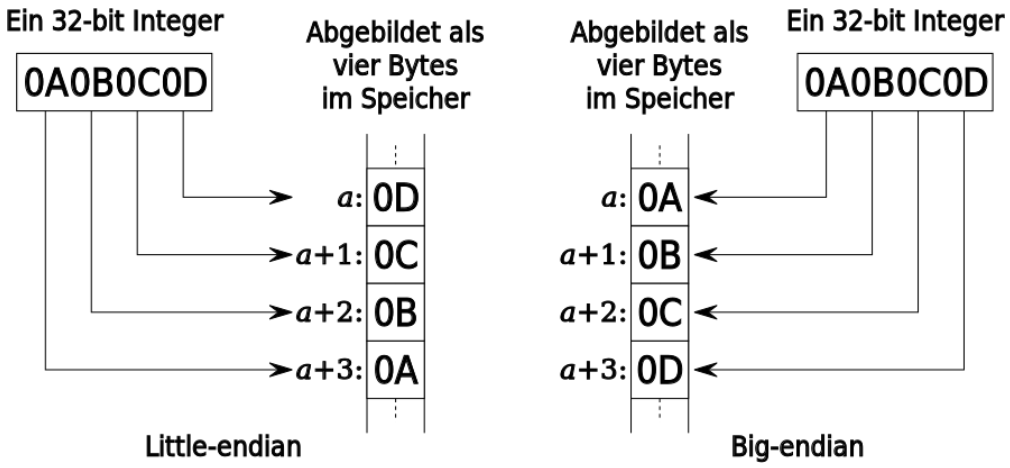


Abbildung 3: Ganzzahl 168.496.141 Integer

1.1.4.1 Little Endian

Ein Vorteil von Little-Endian besteht darin, dass bei zunehmenden Werten neue Ziffern oder Bytes am Anfang der Zahl hinzugefügt werden müssen. Zum Beispiel benötigt eine wachsende Ganzzahl mehr Platz am Beginn der Speicherung. Bei einer in Little-Endian gespeicherten Zahl bleiben die ursprünglichen Ziffern unverändert am Ende, während neue Ziffern an der "vorderen" Speicheradresse hinzugefügt werden. Dies kann bei bestimmten Operationen, wie beispielsweise beim Erweitern von Zahlenwerten, zu einer einfacheren und effizienteren Verarbeitung führen.

Code in c :

```c
#include <stdio.h>

union EndianTest {
    int value;
    char bytes[sizeof(int)];
};

int main() {
    union EndianTest test;
    test.value = 0x12345678; // An example integer value

    printf("Integer value: 0x%X\n", test.value);
    printf("Bytes in memory: ");
    for (int i = 0; i < sizeof(int); i++) {
        printf("%02X ", (unsigned char)test.bytes[i]);
    }
    printf("\n");

    return 0;
}
```

Abbildung 4:Endian Code in C

Ausgabe im Fall Little Endian:

```
Integer value: 0x12345678
Bytes in memory: 78 56 34 12
```

Abbildung 5:Ergebnis des Little Endian Code in C

test.value ist auf 0x12345678 gesetzt. In einem Little-Endian-System wird das niedrigwertigste Byte (Least Significant Byte, **LSB**) zuerst gespeichert.

Die Reihenfolge der Bytes im Speicher wird umgekehrt zur Reihenfolge, in der sie in einer typischen hexadezimalen Darstellung erscheinen. Deshalb wird 78 (das **LSB** von 0x12345678) zuerst gespeichert, gefolgt von 56, 34 und 12.[12]

1.1.4.2 Big Endian

[15] (9)

In einem Big-Endian-System wird beispielsweise eine Hexadezimalzahl wie A1B2 exakt in dieser Reihenfolge im Speicher abgelegt. Wenn A1 in der Speicheradresse 3000 gespeichert wird, befindet sich B2 an der Adresse 3001. Im Vergleich dazu speichert ein Little-Endian-System diese als B2A1, wobei B2 an der Adresse 3000 und A1 an der Adresse 3001 steht.

Ausgabe im Fall Big Endian:

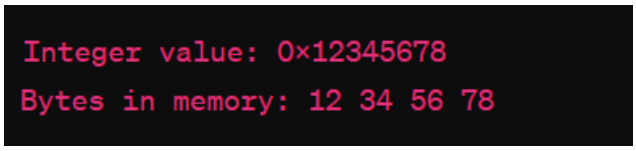


Abbildung 6:Ergebnis des Big Endian Code in C

test.value ist immer noch auf 0x12345678 gesetzt.

Da es sich um ein Big-Endian-System handelt, wird das höchstwertige Byte (**MSB**) zuerst gespeichert. Daher erscheint 12 zuerst, gefolgt von 34, 56 und 78.[13]

1.1.5 STM32 Boards

Sind moderne Mikrocontroller, die zur einen 32-Bit Familie gehört, und auf dem **Arm *Cortex-M-*** Prozessor basiert, wobei sie mehrere Vorteile und Fähigkeiten hat wie z.B. geringen Stromverbrach, Echtzeitfähigkeiten und Signalverarbeitung.

Diese Boards sind eigentlich ideal für eine breite Palette von Anwendungen, von kleinen Projekten bis hin zu umfassenden Plattformen, dank ihrer Vielfalt an Tools und Software, die die Projektentwicklung unterstützen.

STM32 bietet auch ein offenes Entwicklungsumfeld (***STM32*** IDE), das eine flexible, einfache und erschwingliche Methode zur Entwicklung innovativer Geräte und Anwendungen darstellt. Es basiert auf ***STM32 Nucleo-Boards*** und der ***STM32Cube***-Software. [14][15]

[13] (7)
[14] (8)
[15] (9)

Abbildung 7:STM32-Cortex

1.1.6 Texas Instruments (TMS570LS3137) [16]

Es handelt sich um einen fortgeschrittenen Mikrocontroller, der von der amerikanischen Firma ***TI*** hergestellt wird und zur ***TMS570*** Serie gehört. Diese Serie ist bekannt für ihre ausgeprägten Sicherheits- und Zuverlässigkeitsmerkmale.

1.1.6.1 Architektur [17]

Der ***TMS570LS3137*** basiert auf der **ARM *Cortex-R4F*** Kernarchitektur. Diese Architektur ist speziell für Echtzeitanwendungen konzipiert.

1.1.6.2 Speicher und Leistung [18]

Der ***TMS570LS3137*** hat bis zu 1,25 MB integrierten Flash-Speicher und bis zu 192 KB RAM. Er bietet eine hohe Leistung, die für anspruchsvolle Anwendungen notwendig ist.

1.1.6.3 Anwendungsbereiche [19]

[16] (10)
[17] (10)
[18] (10)
[19] (10)

Der ***TMS570LS3137*** wird häufig in sicherheitskritischen Anwendungen eingesetzt, wie z.B. in der Automobilindustrie, in industriellen Steuerungssystemen und in Flugzeugsystemen.

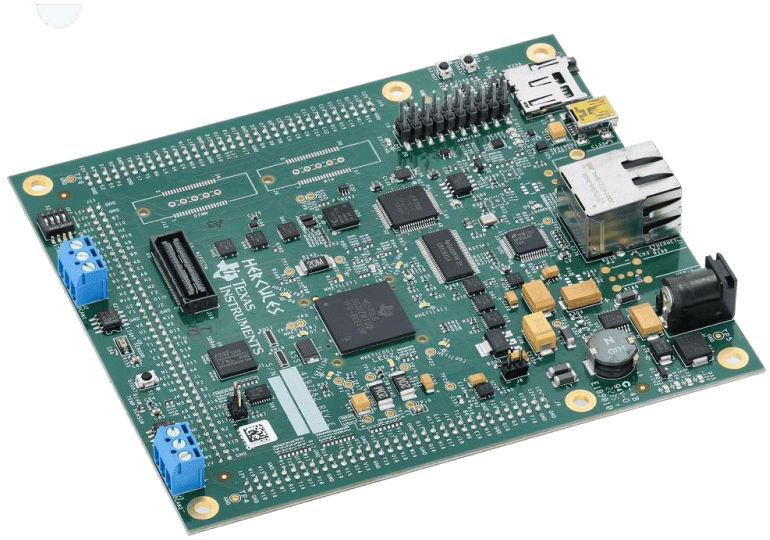


Abbildung 8:TMDS570LS31HDK

1.2 Hintergrund und Bedeutung von Modultests in eingebetteten Systemen

1.2.1 Vertiefung des Hintergrundes

Entscheidende Schritte im Entwicklungsprozess umfassen die Prüfung und Bewertung von eingebetteten Systemen. Dadurch wird sichergestellt, dass das System gemäß den spezifizierten Anforderungen funktioniert. Auch wenn kein Testverfahren eine absolut fehlerfreie Garantie bieten kann, konzentriert es sich dennoch darauf, potenzielle Fehler und Störungen auf ein Minimum zu beschränken. Die Testverfahren für eingebettete Systeme sind zahlreich und haben jeweils ihre individuelle Bedeutung bei der Bewertung des Produkts. Die Überprüfung der korrekten Funktionsweise, der Ein- und Ausgabe sowie der Reaktionszeiten des Systems kann mithilfe von Tests durchgeführt werden. Insbesondere bei Echtzeitsystemen mit zeitkritischen Anforderungen sind solche Tests sehr hilfreich. Bei diesen Tests ist es ebenfalls wichtig, die Systemverfügbarkeit zu berücksichtigen - also wie lange es dauert, bis ein Fehler oder eine Störung auftritt.

1.2.2 Softwarequalität in eingebetteten Systemen

In eingebetteten Systemen spielt die Softwarequalität eine zentrale Rolle.

Embedded Systeme benötigen eine hochwertige Software für ihre Sicherheit, Zuverlässigkeit und Leistung. Es besteht die Gefahr von funktionalen Problemen und ernsthaften Sicherheitsrisiken durch Fehler in der Software, insbesondere in kritischen Branchen wie

Medizintechnik oder Automobilindustrie. Um sicherzustellen, dass die Software zu jeder Zeit integer und zuverlässig bleibt, sollten robuste Testprozeduren implementiert werden.

1.2.3 Entwicklung von eingebetteter Software

Es gibt spezifische Schwierigkeiten bei der Entwicklung von eingebetteter Software. Zu diesen gehören Ressourcenbeschränkungen wie Speicher und Verarbeitungskapazität sowie Echtzeit-Anforderungen. Darüber hinaus ist es erforderlich, die Software Um mit den erhöhten Anforderungen an das Testing aufgrund dieser Faktoren umzugehen, müssen spezielle Herangehensweisen und Tools verwendet werden.

2 Modultests

2.1 Eine Übersicht

Aufgrund der hohen Präzision, die von eingebetteten Systemen gefordert wird, insbesondere da sie heutzutage in allen elektronischen Geräten, besonders in professionellen, vorhanden sind, wurde es notwendig, dass Ingenieure und Entwickler eine Methode zum Prüfen der Qualität und Funktionsweise der Systeme finden, was sie zur Entwicklung von Modultests anregte. Trotz der vielen Unternehmen, die zur Entwicklung dieser Werkzeuge beigetragen haben, besteht weiterhin ein großer Bedarf an Entwicklung. Es ist erwähnenswert, dass eine der größten Herausforderungen darin besteht, einen Framework zu finden, der alle Funktion und die darin enthaltenen Programmieranweisungen überprüft, aber es ist notwendig, Methoden zu finden, um diese Codes auch mit den verfügbaren, wenn auch unkonventionellen Werkzeugen zu testen.

2.2 Geschichtliche Entwicklung der Modultests

Es hat eine deutliche Weiterentwicklung bei Modultests im Verlauf der Zeit gegeben. Einmal oft manuell durchgeführt und zeitaufwendig gewesen sind sie jetzt ein automatisierter, effizienter und integraler Teil des Prozesses der Softwareentwicklung. Dank der Evolution von Testwerkzeugen und -Methoden sind Modultests heute ein wesentlicher Bestandteil der Qualitätssicherung.

2.3 Methoden und Techniken der Modultests

Die Qualitätssicherung in der Softwareentwicklung beinhaltet essenziell die Durchführung von Modultests, um die Funktionalität und Korrektheit einzelner Softwaremodule sicherzustellen.[20]

2.3.1 Nichtinkrementelles Testen

[23] (11)

ermöglicht das parallele Testen von Modulen durch die Simulation abhängiger Module mittels STUBs. Diese Methode stößt jedoch auf Herausforderungen, insbesondere bei der Erstellung der STUBs, die komplexe Abhängigkeiten simulieren sollen. Ein wesentlicher Nachteil dieser Methode liegt in der späten Integration der getesteten Module, was die Identifikation von Integrationsfehlern erschwert.[21]

2.3.2 inkrementelle Testen

Im Gegensatz dazu steht das inkrementelle Testen, bei dem Module schrittweise hinzugefügt und in Kombination getestet werden. Diese Methode teilt sich in zwei Ansätze: das Top-Down- und das Bottom-Up-Testen. Beim Top-Down-Testen beginnt der Testprozess bei den obersten Modulen, wobei STUBs für die Simulation der noch nicht getesteten untergeordneten Module verwendet werden. Dieser Ansatz ist besonders nützlich für das Testen von Schnittstellen und Interaktionen zwischen den Modulen. Ein Nachteil des Top-Down-Testens ist die Notwendigkeit, komplexe STUBs zu entwickeln, um die untergeordneten Module zu simulieren.[22]

2.4 Anwendungsbereiche von Modultests

Modultests haben in allen Phasen der Softwareentwicklung eine große Bedeutung. Die Isolierung von einzelnen Komponenten in der Software ermöglicht es Entwicklern, Fehler rechtzeitig aufzudecken und zu korrigieren.[23] In eingebetteten Systemen sind Modultests von großer Bedeutung, da Fehler in diesen einen weiten Einfluss haben können und sogar zu Katastrophen führen.

3 Ziele und Aufbau der Arbeit

3.1 Einführung

Als ich mit der Arbeit begann hatte ich eine große wesentliche Herausforderung daran, drei geeignete Framework herauszufinden, die schon für eingebettete Systeme entwickelt wurde. Das Problem eigentlich stand daran, wie man alle benötigte Funktionen von den Framework richtig nachvollzieht und effektiv einsetzt.

Nach intensiver Recherche und sorgfältiger Analyse konnte ich verschiedene Framework finden, die speziell für C/C++ entwickelt wurden, und die es die Entwicklern ermöglichen, die Funktionen von eingebettete System effektiv zu testen.

Diese Erfahrung war für mich enorm wertvoll. Sie hat nicht nur mein Verständnis für die Komplexität von Testverfahren in eingebetteten Systemen enorm vertieft, sondern auch meine Fähigkeit zur kritischen Analyse und Auswahl der richtigen Werkzeuge stark verbessert.

---

[21] (16)

[22] (16)

3.2 Vertiefte Auseinandersetzung mit Framework

Meine nächsten Schritte bestanden darin, mich gründlich mit jedem der ausgewählten Frameworks auseinanderzusetzen, nachdem ich potenzielle Optionen gefunden hatte. In diesem Zusammenhang musste man sich mit den grundlegenden Funktionen vertraut machen, die Syntax für das Verfassen von Testfälligen lernen sowie eine Bewertung der Vor- Nachteile jeder Plattform vornehmen. Meine spätere Entscheidung darüber, welches Framework am besten den spezifischen Anforderungen von Embedded System entspricht, basierte auf dem Verständnis dieser Einzelheiten.

3.3 Fokussierung auf ein Framework

Am Ende meiner eingehenden Recherche zu den drei Frameworks fokussierte ich mich schließlich auf eins von ihnen. Ich hatte die Aufgabe, eine Wahl zu treffen hinsichtlich des Frameworks das am ehesten den Anforderungen gerecht wird sowie nahtlos in die entwickelten eingebetteten Systems integriert werden kann. Es wurde intensiv überlegt, um den passenden Kandidaten auszusuchen. Sowohl positive als auch negative Aspekte wurden detailliert betrachtet und eine umfangreiche Auswertung der praktischen Anwendung durchgeführt.

Mein Verständnis in den Bereichen Softwareentwicklung und Qualitätssicherung konnte durch die Erreichbarkeit dieser Ziele nicht nur erweitert werden, sondern es half auch dabei, eine fundierte Wahl bei der Suche nach dem passenden Unit Testing Framework für Embedded Systems zu treffen

4 Analyse verfügbarer Modultests-Umgebungen

Die Analyse ist einer der wichtigste Aufgabe von dem Entwickler ist, um herauszufinden, welches Framework am besten für die Software bzw. Hardware Anforderungen passend ist.

In dieser Forschung wurde besonders auf die Frameworks konzentriert, die für die eingebettete Systeme geeignet ist, weil manche Frameworks sind nur für sehr speziellen entwickelt.

Und jetzt kommen wir zu die besten meine Meinung nach Frameworks, die ich während diese Forschung verwendet habe und zwar:

4.1 **Unity**:

Aufgrund seine Einfachheit und Fokussierung auf die eingebettete Systeme ist ***Unity*** Framework sehr berühmt zwischen den Entwickler geworden. In Szenarien mit begrenzten Ressourcen, wie sie oft in eingebetteten Systemen vorkommen, ist es möglich Test-Driven Development (TDD) durchzuführen.[24]

Die Entwickler von diesem Framework haben versucht, Funktionen zu entwickeln, die die abgedeckt hat, alle Fälle und Bedürfnisse von eingebetteten Systemen zu testen.

[24] (12)

Als bestes Framework kann Unity meine Meinung nach wegen seiner speziellen Ausrichtung auf eingebettete Systeme angesehen werden. Leichtgewichtig und ressourcenschonend eignet es sich besonders gut für den Einsatz in eingebetteten Systemen mit begrenztem Speicherplatz.

Beispiel 1:

```c
#include "unity.h"

// Global variables or resources can be declared here.

void setUp(void) {
    // This code is executed before each test.
}

void tearDown(void) {
    // This code is executed after each test.
}

int add(int a, int b) {
    return a + b;
}

void test_addition(void) {
    TEST_ASSERT_EQUAL(4, add(2, 2));
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_addition);
    UNITY_END();
    return 0;
}
```

Abbildung 9: Code in C für Modultest 1 Bsp. in Unity Framework

In diesem Beispiel wurde die add Funktion getestet. Die Tests Funktion hat nur ein void Parameter, und in der wird die passenden Funktion von Unity Framework aufgerufen.

setUp(void) ist dafür zuständig, die Testumgebung in einen bekannten Anfangszustand zu versetzen, wobei sie um eine gemeinsame Ressourcen zu initialisieren verwandet werden könnte.

tearDown(void) wird nach der Ausführung jedes Tests aufgerufen, um alle Aktionen rückgängig zu machen. Sie wird genutzt, um Ressourcen freizugeben, temporäre Dateien zu löschen oder den Zustand zu bereinigen, dass den nächsten Test nicht durch der vorherige Test beeinflusst wird.

Weil in diesem Fall um ein ganz normale mathematische Operation geht, braucht man nicht mehr als TEST_ASSERT_EQUAL(), wobei in den ersten Parameter die erwartete Wert geschrieben muss, und in den zweiten die getestete Funktion mit ihre Parameters.

Die main() muss UNITY_BEGIN() Funktion beinhalten, die für das Starten des Testes verantwortlich ist, dann innerhalb der RUN_TEST() muss nur den Name des getestete Funktion als Parameter sein, wobei man für jeder getesteten Funktion ein RUN_TEST() braucht.

Am Ende des Testes wird mit UNITY_END() das Test beendet.

Die erwartete Ausgabe ist :

```
test_addition.c:9:test_addition:PASS

-----------------------
1 Tests 0 Failures 0 Ignored
OK
```

Abbildung 10:Ergebnis vom Modultest 1 Bsp. in Unity Framework

In der Ausgabe wird die Ergebnisse des Testes angezeigt. Wenn alle Tests schon passt dann wird (PASS) von dem Unity ausgeliefert.

1 Tests = steht für den Anzahl die durchgeführte Tests

0 Failures = steht für den Anzahl die nicht bestandenen Tests

0 Ignored = steht für die von dem Frameworks ignorierten Tests

Und in der letzten Zeile OK, um sicherzustellen, dass alles in Ordnung ist.

Beispiel 2:

```c
#include "unity.h"

void setUp(void) {
}

void tearDown(void) {
}

int add(int a, int b) {
    return a + b;
}

void test_addition1(void) {
    TEST_ASSERT_EQUAL(4, add(2, 2));
}

void test_addition2(void) {
    TEST_ASSERT_EQUAL(12, add(9, 5));
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_addition1);
    RUN_TEST(test_addition2);
    UNITY_END();
    return 0;
}
```

Abbildung 11:Code in C für Modultest 2 Bsp. in Unity Framework

In diesem Beispiel wurde sie zwei Funktionen : test_addition1() und test_addition2() getestet, wobei die erwartete Ergebnisse von test_addition2() nicht richtig ist.

Die Ausgabe ist :

```
test_addition.c:9:test_addition1:PASS
test_addition.c:13:test_addition2:FAIL: Expected 12 Was 14

-----------------------
2 Tests 1 Failures 0 Ignored
FAIL
```

Abbildung 12:Ergebnis vom Modultest 2 Bsp. in Unity Framework

Hier merkt man, dass bei der zweiten Funktion einen Fehler aufgetreten ist, und es wurde gezeigt, welche von den User die eingegebene Wert ist, und wie ein sein soll.

Außerdem hat sich die Ergebnisse in den 2 letzten Zeilen geändert, und FAIL, weil ein von den Tests falsch war.

4.2 Minunit

MinUnit ist ein minimales Unit-Test-Framework für C/C++, das vollständig in einer einzigen Headerdatei enthalten ist. Es ermöglicht die Definition und Konfiguration von Test-Suiten sowie einige praktische Arten von Aussagen. MinUnit gibt eine Zusammenfassung mit der Anzahl der durchgeführten Tests, Anzahl der Prüfungen und der verstrichenen Zeit aus. [25]

Beispiel:

```c
#include <stdio.h>
#include "minunit.h"

// Function to check if a number is even
int is_even(int num) {
    return (num % 2) == 0;
}

// Test function
MU_TEST(test_even_number) {
    mu_assert(is_even(2), "Failed: 2 is an even number");
    mu_assert(is_even(4), "Failed: 4 is an even number");
}

MU_TEST(test_odd_number) {
    mu_assert(!is_even(1), "Failed: 1 is an odd number");
    mu_assert(!is_even(3), "Failed: 3 is an odd number");
}

MU_TEST_SUITE(test_suite) {
    MU_RUN_TEST(test_even_number);
    MU_RUN_TEST(test_odd_number);
}

int main(int argc, char *argv[]) {
    MU_RUN_SUITE(test_suite);
    MU_REPORT();
    return 0;
}
```

Abbildung 13:Code in C für Modultest 1 Bsp. in MinUnit Framework

[25] (13)

In diesem Beispiel wird getestet, ob die Zahlen 2, 4, 1 und 3 gerade sind oder nicht.

Zu dem braucht man die MU_TEST() Funktion, die in der MinUnit-Testbibliothek bereitgestellt ist, um eine Testfunktion zu definieren. Mit dieser Funktion wird signalisiert, dass ein Testfall dargestellt ist.

Anhand die mu_assert() haben wir die Möglichkeit, eine bestimmte Bedingung zu überprüfen.

Wenn die Bedingung erfüllt ist, fährt der Test fort und falls er jedoch falsch ist, schlägt der Test fehl, und wird eine Fehlermeldung in der Konsole ausgegeben.

Dieser Funktion hat zwei Parameter, wobei der erster Parameter für die getestete Funktion ist, und der zweite für die Fehlermeldung falls den Test nicht bestanden hat.

Für die Ausführung der Tests, ist die MU_RUN_SUITE() Funktion geeignet, wobei sie nur ein Parameter für den Name des getestete Funktion hat.

Um einen Bericht über die Testergebnisse zu generieren und auszugeben, wird die MU_REPORT() verwendet. Dieser Bericht enthalt alle notwendige Informationen von dem Test wie z.B. wie viele davon erfolgreich waren und wie viele fehlgeschlagen sind.

Die Ausgabe ist :

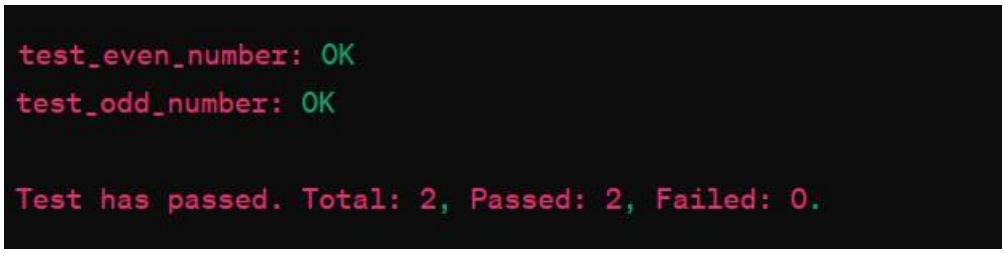


Abbildung 14:Ergebnis vom Modultest 1 Bsp. in MinUnit Framework

Beispiel 2:

```c
#include <stdio.h>
#include "minunit.h"

// Function to check if a number is even
int is_even(int num) {
    return (num % 2) == 0;
}

// Test function
MU_TEST(test_even_number) {
    mu_assert(is_even(2), "Failed: 2 is an even number");
    mu_assert(is_even(4), "Failed: 4 is an even number");
}

MU_TEST(test_odd_number) {
    mu_assert(!is_even(1), "Failed: 1 is an odd number");
    mu_assert(!is_even(3), "Failed: 3 is an odd number");
}

MU_TEST(test_odd_number2) {
    mu_assert(!is_even(2), "Failed: 2 is an odd number");
}

MU_TEST_SUITE(test_suite) {
    MU_RUN_TEST(test_even_number);
    MU_RUN_TEST(test_odd_number);
    MU_RUN_TEST(test_odd_number2);
}

int main(int argc, char *argv[]) {
    MU_RUN_SUITE(test_suite);
    MU_REPORT();
    return 0;
}
```

Abbildung 15:Code in C für Modultest 2 Bsp. in MinUnit Framework

Hier wird eine Funktion mit deiner falschen erwarteten Ergebnisse hinzufügt.

Die Ausgabe ist :

```
test_even_number... ok
test_odd_number... ok
test_odd_number2... FAIL (Failed: 2 is an odd number)

Tests run: 3, Failures: 1
```

Abbildung 16:Ergebnis vom Modultest Bsp. in MinUnit Framework

Wegen des Fehlers bei der test_odd_number2() Funktion hat das Framework die Fehlermeldung ausgegeben.

4.3 CPPUTest

ist ein Unit-Test-Framework, das sich auf C und C++ fokussiert. Das benutzt der Prinzipien von Test Driven Development.

Der Grund, der es zu seiner Ausbreitung führte, ist die Benutzerfreundlichkeiten und sein einfacher Ansatz, der einen attraktiven Option für Entwickler macht.[26]

Beispiel :

```cpp
#include "CppUTest/CommandLineTestRunner.h"
#include "CppUTest/TestHarness.h"

// Function to add two numbers
int add(int a, int b) {
    return a + b;
}

TEST_GROUP(AddFunctionTests) {
    int result;

    void setup() {
        // Initialization before each test case
    }

    void teardown() {
        // Cleanup after each test case
    }
};

TEST(AddFunctionTests, AddTwoPositiveNumbers) {
    result = add(2, 3);
    CHECK_EQUAL(5, result);
}

// Additional test cases can be added here

int main(int argc, char **argv) {
    // Run all the tests
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

Abbildung 17:Code in C für Modultest Bsp. in CPPUTest Framework

In diesem Beispiel testen wir ebenfalls die add() Funktion, und um den Test Name zu definieren, braucht man die TEST_GROUP() Funktion, wobei der nur einen Parameter hat.

In dieser Funktion sollte die setUp() Methode enthalten, die vor jedem Testfall aufgerufen wird und die tearDown(), die nach jedem Testfall auch aufgerufen wird.

Testfall AddTwoPositiveNumbers: Innerhalb der Testgruppe wird ein Testfall definiert, der die Funktion add testet, indem er überprüft, ob die Summe von 2 und 3 gleich 5 ist.

CHECK_EQUAL() ist eine Makroanweisung, die überprüft, ob result gleich 5 ist.

[26] (14)

In der main() ruft CommandLineTestRunner::RunAllTests auf, um alle definierten Testfälle auszuführen.

Die Ausgabe ist :

```
..
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, [time] ms)
```

Abbildung 18:Ergebnis vom Modultest Bsp. in CPPUTest Framework

4.4 Bewertungskriterien

Feature/ Kriterium	Unity	Minunit	CPPUTest
Typ	Test Framework	Test Framework	Einheitstest Framework
Vorteile	1- Einfachheit und Leichtigkeit im Einrichten. 2- Gute Dokumentation. 3- Fokus auf eingebettete Systeme. 4- Unterstützung von Mock-Tests	1-Einfachheit und Leichtigkeit. 2-Geringe Abhängigkeiten. 3-Kompakte Größe Klare Fehlerberichte.	1-Einfache Anwendung. 2- Unterstützung für Mocking. 3- Integration mit anderen Tools
Nachteile	1- Begrenzte Funktionalität für komplexe Anwendungen - Kein eingebautes Build-System	1-Begrenzte Funktionalität: Minunit ist eher einfach und bietet nicht die umfangreichen Funktionen und Erweiterbarkeitsoptionen. 2-Begrenzte Integration. 3-Keine Unterstützung für Mocking. 4-Begrenzte Teststeuerung	1- Begrenzte Funktionalität für komplexe Projekte. 2- komplexe Installation. 3- Fehlende erweiterte Funktionen.
Besonderheiten	1- Entwickelt für eingebettete Systeme. 2- Stark in Szenarien	1- Einfaches Test-Framework für die C-Programmiersprache. 2-Es ist darauf spezialisiert, Tests in C	- Speziell für Einheitstests und TDD

	mit begrenzten Ressourcen.	zu erstellen und zu automatisieren.	

5 Modultests in der Praxis

5.1 Modultests mit IDE

Das erste Ziel bestand darin, wie man mit der vom Hersteller gefertigten **IDE** ein Unit Testing Framework zum Laufen zu bringen, um sicherzustellen, dass das System getestet werden könnte.

Als erstes Board habe ich ***STM32-L496ZG*** benutzt, und deswegen war der ***STM32Cube Programmer*** dafür geeignet.

Mein zweites Board war ***TMS570LS3137*** von ***Texas Instruments*** und dafür ist natürlich der

Code Composer Studio geeignet.

5.1.1 STM32Cube Programmer

Abbildung 19:STM32 CubeIDE Logo

STM32Cube Programmer ist eine Desktop-App-Umgebung, die speziell für die Programmierung von ***STM32***-Mikrocontrollern entwickelt wurde. Die Benutzeroberfläche des ***STM32Cube*** Programmers bietet zahlreiche Optionen, die die Entwicklung auf ***STM32***-Boards vereinfachen. Beim Erstellen eines neuen Projekts kann der Benutzer das Modell des Boards auswählen, um eine passende Vorlage für das gewählte Board zu finden. Das **IDE** übernimmt anschließend die Konfiguration der Hardware-Eigenschaften und Software-Bibliotheken, die für das ausgewählte Board relevant sind.

Damit wird der Einstieg in die Entwicklung erleichtert und gewährleistet, weil die Projekte von Anfang an mit den richtigen Einstellungen und Ressourcen ausgestattet sind.

Man kann jeden Pin am Mikrocontroller einstellen, wobei basierend auf dieser Konfiguration der Basiscode generiert wird.

Darüber hinaus bietet der ***STM32Cube Programmer*** Funktionen zur Fehlerdiagnose und zum Debugging, was besonders nützlich ist, um Probleme während der Entwicklung effizient zu identifizieren und zu beheben.

Der Unterstützt verschiedene Programmiersprachen wie z.B. C/C++ und Assembly für hardwarenahe Programmierung.

Die Software kann mit anderen Entwicklungsumgebung und Werkzeugen, wie z.B. ***STM32CubeMX*** integriert werden, was die Entwicklungsprozesse erleichtert.

Außerdem bietet er einen Zugriff auf eine umfangreiche Beispielprojekte, die das Verständnis der Funktionsweise der Mikrocontroller erleichtern und als Anfangspunkt für eigene Projekte dienen können.

5.1.1.1 Integration des Frameworks in IDE von STM32 Boards

Eine der größten Herausforderungen dieser Forschung war die Integration von ***Unity*** in die ***STM32Cube IDE***, da es als Fremdcode gilt, der auf dem Board geladen werden soll. Nach der Erstellung des Projekts wird der Template-Code in verschiedene Ordner aufgeteilt:

-**Binaries**: Hier wird am Ende die .elf-Datei generiert, falls der Aufbau des Projekts erfolgreich war. Diese .elf-Datei wird mit Hilfe von Debug-Tools auf das Board geflasht, wobei ***ST-Link*** im Hintergrund verwendet wird.

-**Core**: Dieser Ordner enthält fast den gesamten wichtigen Code des Boards, darunter:

-**Inc**: Hier befinden sich alle vom **IDE** generierten Header-Dateien.

-**Src**: In diesem Ordner sind alle Quellcodes, wie z.B. main.c, der die main()-Funktion enthält, oder Syscalls.c, der Call-Funktionen wie _write(), _lseek() und _open() beinhaltet.

-**Startup**: Hier befindet sich der .s-Code, der für jedes Board spezialisiert und in Assembly geschrieben ist. Der Code beginnt mit der Einstellung des Stack-Pointers auf den Anfang des Stack-Speichers. Er ruft SystemInit auf, eine Funktion, die typischerweise die Systemuhr und andere grundlegende Hardware-Einstellungen konfiguriert. Der Code kopiert Initialisierungswerte aus dem Flash-Speicher in den **SRAM** für den Datenbereich. In diesem Ordner habe ich die unit_test-Quell- und Header-Dateien abgelegt, die für das Testen der Funktionen wichtig sind. Die Datei test_runner.c enthält die Funktion mainTest(), die nicht main() genannt werden darf, da die Ausführung des Codes von main() in der Datei main.c ausgeht. Die mainTest()-Funktion startet und beendet alle getesteten Methoden und ist dafür geeignet, die Ausgabe des ***Unity*** Frameworks zu liefern.

Für die Testfälle benötigen wir einen Ordner, in dem die Testfunktionen geschrieben werden.

-**Drivers**: Dieser Ordner enthält die Hardwareabstraktionsschicht (HAL) und die Low-Level (LL) Treiber sowie die CMSIS (Cortex Microcontroller Software Interface Standard) Kerntreiber.

-**HAL**-Treiber: Diese bieten eine abstrahierte und portable Codebasis zur Steuerung der Hardwareperipherie des ***STM32***. Sie erleichtern die Entwicklung, indem sie komplexe Low-Level-Details verbergen und eine einfachere **API** für die Peripheriegeräte bieten.

-**LL**-Treiber: Diese sind näher an der Hardware und bieten eine leichtgewichtige, optimierte Bibliothek.

Für das Unit Testing müssen zwei neue Ordner erstellt werden, die in diesem Bereich liegen sollten. Der erste enthält unser Unity Framework: die Dateien unity.c, unity_internals.h und unity.h. Im zweiten schreiben wir unsere echten Funktionen in .c-Dateien und dazugehörige Header.

5.1.1.2 Probleme und deren Lösungen

Wie bereits erwähnt, stellt die Integration von Fremdcode in ***STM32Cube*** **IDE-**Projekte eine Herausforderung dar, insbesondere weil neu generierte Ordner von der **IDE** nicht automatisch erkannt werden. Diese Nichterkennung kann zu erheblichen Komplikationen im Entwicklungsprozess führen. Nachfolgend werden einige der Probleme und deren Lösungen detailliert beschrieben:

Das Hauptproblem liegt darin, dass die von der **IDE** generierten Pfade nicht manuell hinzufügen werden könnten. Dies führt dazu, dass der Compiler dieses Dateien während des Build-Prozesses nicht findet.

Um dieses Problem zu lösen, müssen die Pfade in den Projekteigenschaften (Properties) manuell aktualisiert werden.

Und damit beinhaltet das Hinzufügen der Pfade zu den Include-Verzeichnissen der neuen Ordner, sodass der Compiler auf die Header-Dateien zugreifen kann.

Zusätzlich müssen die Pfade zu den Source-Dateien in den Build-Einstellungen hinterlegt werden, damit diese bei der Kompilierung berücksichtigt werden.

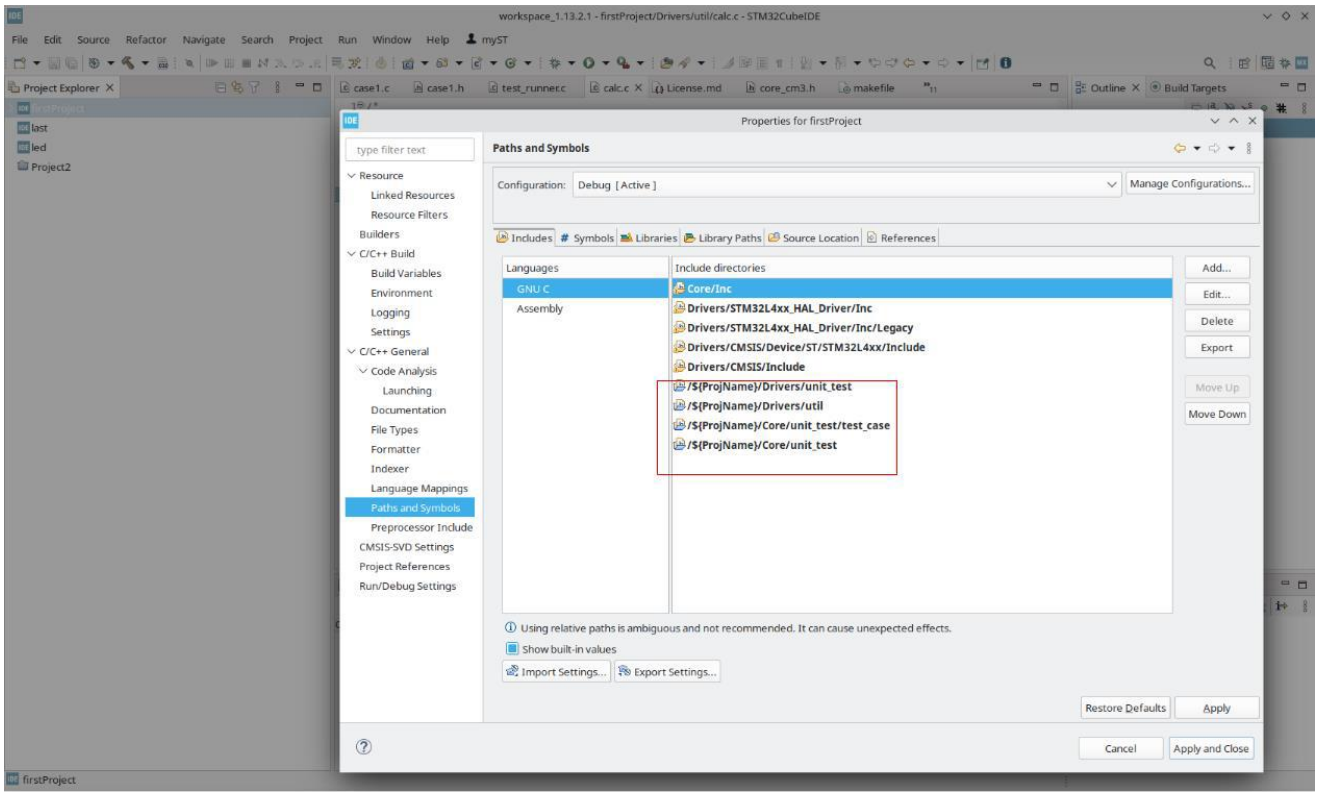


Abbildung 20:Properties für das Hinzufügen von Unity Frameworks Dateien

Bei der Integration externer Bibliotheken in das ***STM32Cube***-Projekt ist es entscheidend, dass diese mit der Hardware des ***STM32***-Boards kompatibel sind. Dies erfordert häufig eine Anpassung des Codes der Bibliotheken, insbesondere wenn sie für andere Mikrocontroller-Architekturen entwickelt wurden.

Beim Debugging von Fremdcode können unerwartete Probleme auftreten, insbesondere wenn der Code nicht für die ***STM32***-Hardware optimiert ist. Eine gründliche Analyse der Ausführung und der Speichernutzung ist erforderlich, um solche Probleme zu identifizieren und zu beheben. Hilfreich kann hierbei die Nutzung von Debugging-Tools sein, die detaillierte Informationen über den Zustand des Mikrocontrollers während der Ausführung liefern.

5.1.1.3 Auswertung der Ergebnisse im Terminal von IDE

Für die Auswertung der Ergebnisse des Unit-Tests, die auf dem Board laufen, ist es notwendig, die Ausgabe auf die seriellen Schnittstelle zu lenken. Dafür braucht man die Funktion _write(). Diese Funktion ist in der Systembibliothek bereits existiert, und muss in diesem Fall für die spezifischen Anforderungen des Projekts angepasst werden.

Die Anpassung ermöglicht es, die Ausgabe der Unit-Tests im Terminal der ***STM32Cube*** **IDE** anzuzeigen.

Die angepasste _write() – Funktion sieht wie folgt aus :

```c
int _write(int file, char *ptr, int len) {
    int DataIdx;
    for (DataIdx = 0; DataIdx < len; DataIdx++) {
        ITM_SendChar(*ptr++);
    }
    return len;
}
```

Abbildung 21:Write Funktion für STM32 Board

int file: Dieser Parameter wird in dieser Funktion nicht verwendet, ist aber Teil der Standard-Signatur der _write()-Funktion.

char *ptr: Dies ist ein Zeiger auf das Array von Zeichen, das gesendet werden soll.

int len: Dies gibt die Länge des zu sendenden Zeichenarrays an.

ITM_SendChar: Diese Funktion ist Teil der CoreSight Debugging-Schnittstelle und sendet ein einzelnes Zeichen zum Host-Computer.

Durch die Implementierung dieser Funktion in der main.c-Datei kann die Standardausgabe (z.B. printf) so umgeleitet werden, dass die Ausgabe über die serielle Schnittstelle erfolgt.

Diese Funktion bietet den Vorteil, dass die Ergebnisse der Unit-Tests in Echtzeit überwacht werden können.

Außerdem ermöglicht diese Funktion eine klare Trennung zwischen den Testergebnisse und anderen Ausgaben des Boards, was die Analyse und Auswertung der Testergebnisse vereinfacht.

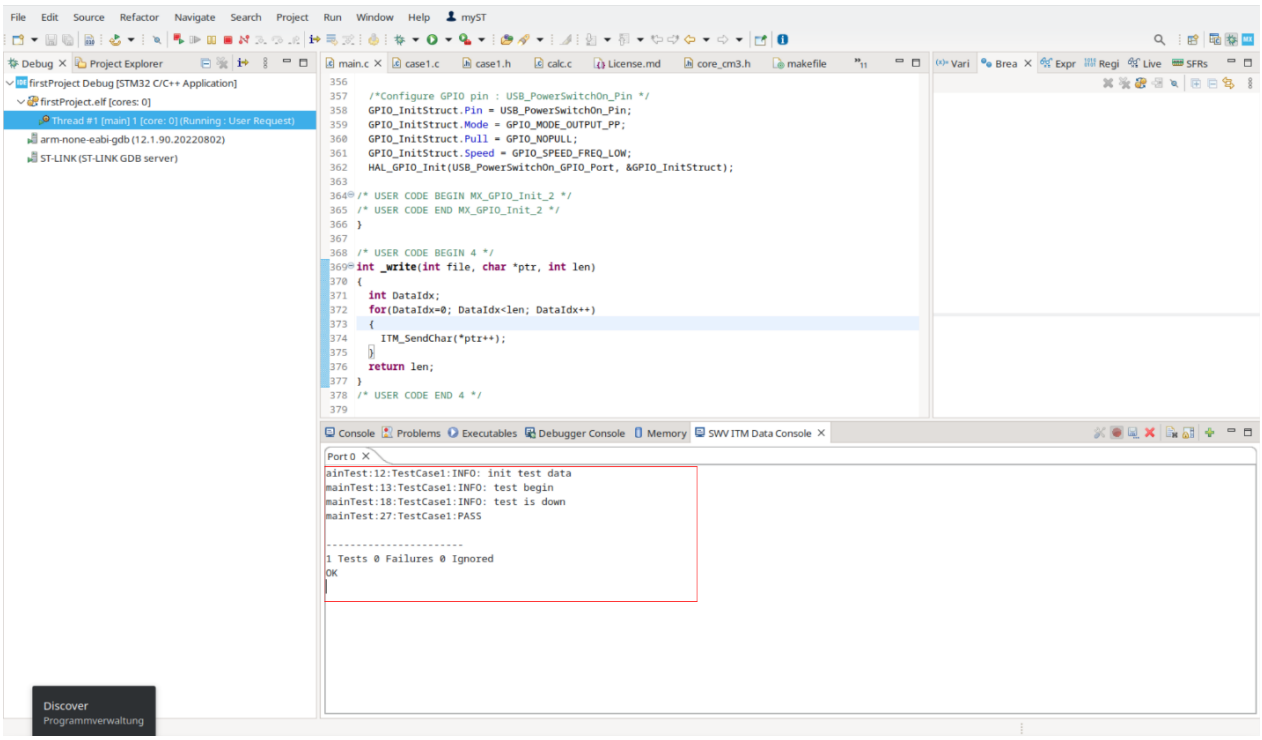


Abbildung 22:Ergebnisse von Unity in STM32 CubeIDE

5.1.2 Code Composer Studio

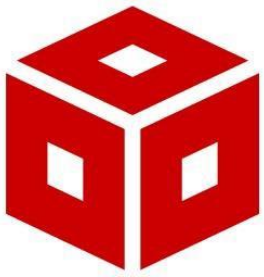


Abbildung 23:Code Composer Studio

ist eine fortgeschrittene integrierte IDE, die von Texas Instruments speziell für Mikrocontroller und Signalprozessoren entwickelt wurde. Die bietet eine umfassende Lösung, die das Programmieren, Debuggen und die Analyse von Embedded-Systemen vereinfacht.

CCS unterstützt viele TI-Hardwareplattformen, was die Entwicklung von Systemen vereinfacht.

Dazu gehören ***MSP430***-Ultra-Low-Power-Mikrocontroller, ***Tiva***-C-Serie **ARM-*Cortex-M4*-MCU**s, Hercules Safety-Mikrocontroller und eine Vielzahl von TI-DSPs.

Innerhalb des ***CCS*** findet den Entwickler mehrere Werkzeugen und Funktionen und dazu zählen:

TI-Compiler und Debugger: Diese Tools sind speziell für TI-Hardware optimiert und ermöglichen eine effiziente Code Entwicklung und Fehlerbehebung.

Automatisierte Code-Generierung: Tools wie **HALCoGen** generieren Code für spezifische Hardwarekonfigurationen, was die Entwicklung beschleunigt und vereinfacht.

Um ein effiziente Nutzung und schnelles Lernen zu ermöglichen, bietet ***CCS*** einen direkten Zugang zu einer Bibliothek von Softwarebeispiele und technischer Dokumentation.

5.1.2.1 Integration des Frameworks in IDE von Texas Instrument Boards

Als zweite Herausforderung meiner Forschung war die Einbindung des Unity Frameworks in die (***CCS***) Entwicklungsumgebung, insbesondere wegen der spezifischen Werkzeugkette, die von Texas Instruments (TI) Boards unterstützt wird. Nach der Erstellung eines neuen Projekts in ***CCS*** mussten bestimmte Anpassungen vorgenommen werden, um das externe Framework effektiv zu integrieren. Die Projektstruktur in ***CCS*** weist Ähnlichkeiten mit der von ***STM32Cube*** **IDE** auf, allerdings gibt es einige Unterschiede, die berücksichtigt werden müssen:

Binaries: Am Ende des Build-Prozesses wird die .out-Datei in diesem Ordner generiert. Diese Datei ist das finale Binärfile, das auf das Board geflasht wird. Im Gegensatz zu der .elf-Datei, die in STM32-Projekten verwendet wird, bietet die .out-Datei eine für TI's Debugging-Tools optimierte Formatierung.

Includes: Dieser Ordner enthält alle vom System generierten Header-Dateien. Diese sind notwendig für den Zugriff auf die Hardware-Funktionen des ***TI*** Boards und müssen korrekt in das Projekt eingebunden werden, um eine nahtlose Integration und Interaktion mit dem Unity Framework zu gewährleisten.

Debug: hat eine besondere Bedeutung für die Entwicklung, weil er das Makefile und die Tools, die für das Flashen und Debuggen des Projekts erforderlich sind enthält. Dieser Ordner spielt eine besondere Rolle im Entwicklungsprozess, da er die notwendigen Werkzeuge bereitstellt, um das Programm auf Fehler zu prüfen und auf das Board zu laden. Die Integration des Unity Frameworks erfordert eine genaue Konfiguration dieser Tools, um sicherzustellen, dass das Framework korrekt mit der Hardware kommuniziert und getestet werden kann.

source : enthält nicht nur den sys_main.c Code, in dem die main() Funktion definiert ist, sondern auch andere wichtige Dateien, die für die Ausführung notwendig ist.

Dazu enthält er auch die Linker-Konfigurationsdatei, bekannt als sys_link.cmd. Dieser Datei konfiguriert, wie der Linker die verschiedenen Speicherbereiche des Mikrokontroller oder DSPs zuweist und verwendet.

Es ist erwähnenswert, dass die korrekte Konfiguration des Source-Ordners und der sys_link.cmd Datei für den Erfolg des Projekts sehr entscheidend ist, um der Code ordnungsgemäß zu kompilieren, linken und auf dem ***TI*** Board ausführen zu können.

Für das Framework brauchte ich zusätzliche Ordner zu erstellen, die genau wie bei ***STM32*** Cube **IDE** strukturiert ist.

5.1.2.2 Probleme und deren Lösungen

Es ist leider nicht möglich, unter ***Linux*** direkt ein Projekt mit **HALCoGen** zu erstellen.

Aus diesem Grund wurde entschieden, die initialen Schritte unter ***Windows*** mit **HALCoGen** durchzuführen und dann das Projekt für die weitern Entwicklungsphasen nach ***Linux*** zu übertragen.

Unter Verwendung von **HALCoGen** unter ***Windows*** wurde der Grundstein für das Projekt gelegt.

Ein wesentliches Problem bei der Integration des ***Unity*** Frameworks in das ***CCS*** war, dass die Entwicklungsumgebung die Pfade zu den neuen Framework-Verzeichnissen nicht automatisch erkannte. Dieses Problem ähnelte einer Herausforderung, die bereits bei der Verwendung der ***STM32 Cube* IDE** aufgetreten war. Die Lösung bestand ebenfalls darin, die notwendigen Pfade zum ***Unity*** Framework manuell in die Projekt-Einstellungen unter die Inhaltsverzeichnisse (Properties) hinzuzufügen.

Innerhalb dieser Einstellungen gibt es eine Option, um zusätzliche zu spezifizieren. Durch das Hinzufügen der Pfade zum ***Unity*** Framework in dieses Einstellung konnte ***CCS*** die erforderlichen Dateien während des Kompilierungs- und Linkprozesses finden.

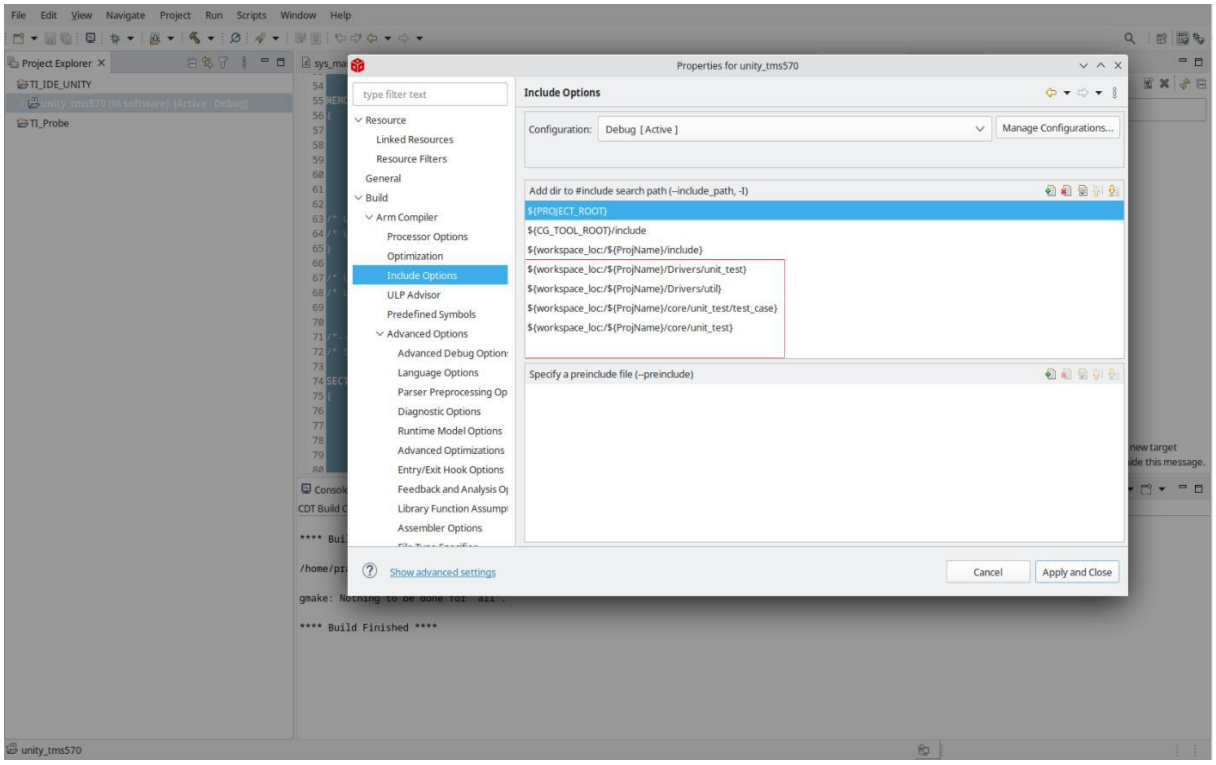


Abbildung 24:Die Einstellung von Properties in STM32 CubeIDE.

Ein weiters aufgetretenes Problem bestand darin, dass der PC unter Linux die USB-Verbindung zur Hardware nicht sofort erkennen konnte. Dieses Problem ist besonders relevant, das eine stabile USB-Verbindung für das Flashen und Debuggen der Hardware unerlässlich ist.

Um dieses Problem zu beheben, war es notwendig, den USB-Treiber für die Hardware manuell zu installieren und zu konfigurieren.

5.1.2.3 Auswertung der Ergebnisse im Terminal

Um die Ausgabe von einem Programm, der auf dem ***TI*** Board zu im Terminal zu zeigen, wird die sciDisplayText() verwendet, wobei dieses Funktion auf den UART Zeichnen sendet, und mit sciBASE_T *sci die Ausgabe zum Lesen ermöglicht.

```
void sciDisplayText(sciBASE_t *sci, uint8 *text, uint32 length) {
    while(length--) {
        while ((UART->FLR & 0x4) == 4); /* Warten, bis UART nicht beschäftigt
        sciSendByte(UART, *text++);     /* Senden des Zeichens */
    };
}
```

Abbildung 25:Standard Write Funktion für TI Board

Diese Methode funktioniert gut für die direkte Ausgabe von Zeichenketten über ein seriellen Interface. Jedoch trat bei der Integration des Unity Frameworks zur Durchführung von Unit-Tests eine Komplikation auf: Unity verwendet standardmäßig die Funktion (void)putchar(), die in stdio.h definiert ist, für die Ausgabe von Testergebnissen. Diese Standardimplementierung von putchar() war nicht kompatibel mit der erforderlichen Methode zur Ansteuerung des seriellen Interfaces des Boards

Um dieses Problem zu lösen und die Ausgabe der Testergebnisse des Unity Frameworks über das serielle Interface des Boards zu ermöglichen, wurde entschieden, die Standardimplementierung von putchar() zu überschreiben. Die neue Implementierung wurde so angepasst, dass sie direkt mit dem für das Board spezifischen UART-Hardwaremodul kommuniziert:

```
int putchar(int c) {
    while ((UART->FLR & 0x4) == 4)
        ; // Warten, bis UART nicht beschäftigt ist
    sciSendByte(UART, (uint8)c); // Senden des Zeichens
    return (c);
}
```

Abbildung 26:Spezifische Write Funktion für Unit Test Ergebnisse. TI Board

Diese Anpassung ermöglicht es, dass die von Unity für die Testausgaben verwendete putchar()-Funktion nun die Textausgaben korrekt über das serielle Interface des Boards ausgibt. Durch das Überschreiben der putchar()-Funktion konnte die Ausgabe der Unity Testergebnisse im Terminal angezeigt werden, was die Überprüfung der Testläufe vereinfacht und beschleunigt.

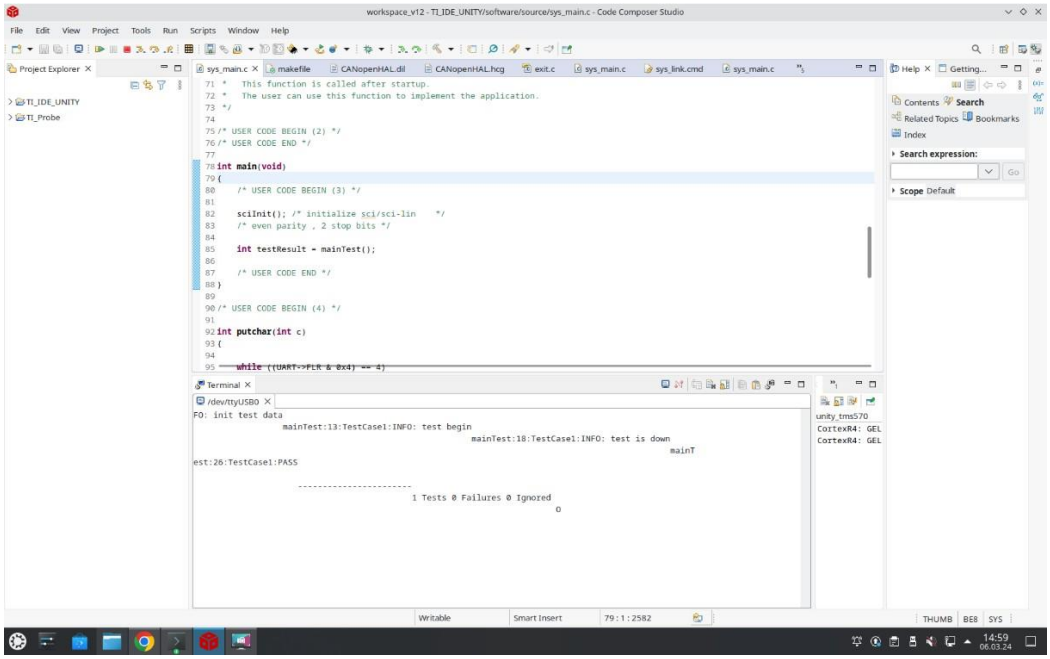


Abbildung 27:Ergebnisse von Unity von TMS in CCS

5.2 Modultests ohne IDE

5.2.1 Übersicht

In der modernen Entwicklung von eingebetteter Systemen ist die Fähigkeit, Modultests effizient und automatisiert durchzuführen, einer entscheidender Bedeutung.

Der Hauptziel dieser Forschung liegt in der Entwicklung und Implementierung einer Umgebung für Modultest, die unabhängig von einer integrierten **IDE** ist, um die Durchführung von Tests zu vereinfachen und zu automatisieren, indem die Abhängigkeit von spezifischen Entwicklungswerkzeugen minimiert wird.

Ein wesentlichen Grund für die Entwicklung einer solchen Testumgebung ist die Notwendigkeit, eingebettete Systeme regelmäßig und zuverlässig testen zu können, auch außerhalb der regulären Arbeitszeiten wie beispielweise über Nacht.

Die Integration ein (CI) System wie Jenkins ermöglicht es, dieses Tests automatisch und periodisch durchzuführen, was zu Verbesserung der Softwarequalität und einer Reduzierung von Entwicklungszeiten führt.

5.2.1.1 Warum Modultests ohne IDE?

Die Entscheidung, Modultests ohne die Verwendung einer **IDE** durchzuführen, wurde aus mehreren Gründen getroffen:

1- Unabhängigkeiten von Entwicklungsplattformen: Durch die Loslösung von spezifischen IDEs wird einer größere Flexibilität in der Entwicklungsumgebung erreicht. Dies ermöglicht es Entwicklerteams, ohne an die Einschränkung einer bestimmten **IDE** gebunden zu sein.

2- Automatisierung: Die Fähigkeit, Tests ohne manuelle Eingriffe durchzuführen, ist für die Implementierung effektiver CI/CD-Pipelines unerlässlich. Eine IDE-unabhängige Testumgebung lässt sich leichter in Automatisierungstools wie Jenkins integrieren, was automatisierte Builds und Tests rund um die Uhr ermöglicht.

3- Effizienzsteigerung: Durch die Automatisierung der Testprozesse können Entwicklungszyklen beschleunigt und die Zeit bis zur Markteinführung (Time-to-Market) verkürzt werden. Automatisierte Tests stellen sicher, dass Fehler frühzeitig erkannt und behoben werden, was die Qualität des Endprodukts verbessert.

4- Nachttests: Die Möglichkeit, Tests automatisch während der Nachtstunden auszuführen, ist ein großartiger Vorteil. Diese Tests können umfangreicher und zeitaufwendiger sein, da sie die reguläre Arbeitszeit nicht beeinträchtigen. Das Ergebnis sind umfassend getestete Systeme am nächsten Morgen, was die Effizienz des Entwicklungsprozesses steigert.

5.2.2 Design der Testumgebung

Die Entwicklung einer Testumgebung für eingebettete Systeme ohne die direkte Verwendung **IDE** stellte eine Herausforderung dar, die innovative Lösung erforderte. Nach einer umfassenden Analyse verschiedener Automatisierungsmöglichkeiten wurde entschieden, dass sie Verwendung von Makefile ist die beste Optimale Lösung.

Der Einsatz von Makefiles ermöglicht eine hohe Flexibilität und Automatisierung des Build und Testprozesses und erleichtert zudem die Integration in Jenkins für kontinuierliche Testzyklen.

Die Implementierung der Testumgebung basiert auf einem sorgfältig konzipierten Makefile, das den gesamten Prozess von der Kompilierung über das Linking bis hin zur Ausführung der Tests steuert.

Folgende Aspekte sind für erfolgreiche Umsetzung entscheidend:

1- Modulare Struktur: Die klare Trennung von Quellcode, Testdateien und Bibliotheken in der Verzeichnisstruktur des Projekts ermöglicht eine übersichtliche Organisation und einfache Erweiterbarkeit der Testumgebung.
2- Automatisierte Build-Prozesse: Durch die Definition spezifischer Build-Targets im Makefile können unterschiedliche Konfigurationen für Debugging und Release Builds erstellt werden, was eine flexible Anpassung an verschiedene Test- und Entwicklungsszenarien erlaubt
3- Integrative Testdurchführung: Die Einbindung von Modultests, insbesondere unter Verwendung des Unity Frameworks, wird durch spezielle Makefile-Targets realisiert. Dies erlaubt die Integration der Testausführung in den Build-Prozess und die automatisierte Rückmeldung der Testergebnisse.

4- Direkte Hardware-Interaktion: Um die Tests direkt auf den Embedded Boards ausführen zu können, beinhaltet das Makefile Mechanismen zur Steuerung der Hardware-Programmierung und -Kommunikation. Dies schließt das Flashen des Programms auf das Board und das Auslesen der Testergebnisse über serielle Schnittstellen oder andere Kommunikationskanäle ein.

5.2.2.1 Komponenten des Build-Prozesses

Der Build-Prozess umfasst sowohl den Source Code des Systems und des zu testenden Systems als auch die Quellcodes des Unity Framework. Zusätzlich sind die Header-Datei erforderlich, die die Definition bzw. Initialisierung der Funktionen und Variablen enthalten.

Für jedes Board sind spezifische Compiler Flags und Tools erforderlich, um den Code korrekt zu kompilieren.

Es ist notwendig, die **CPU**- und **MCU**-Spezifikationen genau zu bestimmen. Die **MCU**-Definition umfasst Angabe zur Architektur (z.B. **ARM *Cortex-M4***), was direkte Anweisung auf die Auswahl der Compiler-Flags hat. Diese Spezifikationen sind richtig entscheidend, um den Code so zu kompilieren, dass er mit der Hardware des Boards kompatibel ist.

Da die Entwicklungsumgebung (***Linux*** 64-Bit) und die benutze Plattformen (***STM32 L496ZG*** und ***TI TMS570*** 32-Bit) unterschiedliche Archetektur haben, sind Cross-Compiler Flags notwendig.

Ein Linker Script ist erforderlich, um zu definieren, wie die verschiedenen Teile des Programmes im Speicher der Hardware angeordnet werden sollen.

Dieses Script ist für jeden Typ von Board spezifisch und bestimmt, wie der Speicherplatz (z.B. für Code, Daten und Stack zugewiesen wird.

Für das Flashen des generierten Executables auf das Board ist ein spezifisches Flash-Tool erforderlich, das mit der Zielhardware kompatibel ist. Dieses Tool wird verwendet, um das Programm über eine Schnittstelle (z.B. USB oder seriell) direkt auf das Board zu übertragen.

5.2.3 Implementierung der Testumgebung

In diesem Abschnitt werde ich die Schritte der Implementierung vom Makefile erklären und zwar:

Es beginnt in der Regel mit der Definition des Ziel-Targets, das den Namen des Endprogrammes angibt. Optional werden Variablen wie DEBUG für den Debug-Modus festgelegt. Dieses Variablen steuern wichtige Aspekte des Kompilierungsprozesses.

Für jede Quelldatei definiert das Makefile eine Regel, wie diese zu einer Objektdatei kompiliert wird.

Diese Regeln nutzen die festgelegten Compiler-Optionen und generieren Objektdateien im spezifizierten Build-Verzeichnis.

Nachdem alle Quelldateien kompiliert wurden, wird das finale ausführbare Programm durch das Linken der Objektdateien erzeugt. Die Linker-Optionen und das Linker-Script steuern diesen Prozesses, indem sie festlegen, wie die Objektdateien im Speicher angeordnet werden.

Im Fall des STM32 wird am Ende das Makefile ein .elf File und im Fall des TMS570 ein .out File erzeugt, die für das Flashen des Programmes auf die Hardware benötigt werden.

Das Makefile enthält in der Regel auch einen speziellen Befehl (flash), der das Flash-Tool aufruft, um das Programm auf die Zielhardware zu übertragen.

5.2.4 Auswertung der Testergebnisse

Die Auswertung der Testergebnisse ist ein entscheidender Schritt im Testprozess für eingebettete Systeme, um sicherzustellen, dass die Software den Anforderungen entspricht und fehlerfrei funktioniert.

Zunächst muss das Makefile so konfiguriert sein, dass es nicht nur das Programm kompiliert und auf das Gerät flasht, sondern auch automatisch die definierten Tests durchgeführt.

Die Testergebnisse werden direkt von der seriellen Schnittstelle des Embedded Boards gelesen. Dies kann mit Tools in Linux wie *cat* oder *screen* erfolgen, die in der Lage sind, die Ausgabe von der seriellen Schnittstelle zu erfassen und in eine Textdatei umzuleiten.

Um die Testergebnisse zu analysieren kann das Makefile so erweitert werden, dass es Skripte oder Programme aufruft.

Für die Analyse der in Textdatei gespeicherten Testergebnisse werden zwei Python-Skripte verwendet. Der erste Skript ist zur Generierung eines Exit-Codes und ein weiters zur Erstellung einer detaillierteren XML-Datei.

5.2.4.1 Exit-Code-Generator

Das erste Skript prüft die Testergebnisse aus das Vorhandensein von *FAIL* -Einträgen. Findet es solche Einträge, gibt es einen Exit-Code **1** (Error) zurück; andernfalls **0** (OK). Dieser Exit-Code wird in einem Textdatei gespeichert und kann verwendet werden, um den Erfolg oder Misserfolg der Tests aus einer höheren Ebene Schnell zu identifizieren.

```python
def check_test_result(file_path):
    try:
        with open(file_path, 'r') as file:
            content = file.read()

        if 'FAIL' in content:
            return 1
        else:
            return 0
    except FileNotFoundError:
        print("Error: File not found.")
        return 1

def write_exit_code(file_path, exit_code):
    with open(file_path, 'w') as file:
        file.write(str(exit_code))
```

Abbildung 28:Python Code. 1 Analyser

5.2.4.2 JUnit XML-Report

Das zweite Skript liest ebenfalls den gleichen Testdatei, und analysiert die Testergebnisse detaillierter, um einen JUnit XML-Report als .xml zu generieren.

Dieser Report enthält Informationen zu jedem Testfall, einschließlich der Anzahl der durchgeführten, bestandenen und fehlgeschlagenen Tests. Die XML-Struktur ist für die Integration in **CI/CD** Pipelines wie Jenkins geeignet, da sie eine detaillierte Ansicht der Testergebnisse ermöglicht.

6 Fallstudien und Anwendungsszenarien

6.1 Fallstudie 1: STM32

Die erste Fallstudie konzentriert sich auf die STM32 Mikrocontroller-Familie von STMicroelectronics, die für ihre Vielseitigkeit und Leistungsfähigkeit in einer breiten Palette von Embedded-Anwendungen bekannt ist.

6.1.1 Vorbereitung und Konfiguration

Ein wesentlicher Vorteil bei der Arbeit mit STM32 ist die Möglichkeit, über die STM32CubeMX Software direkt ein Makefile-Projekt zu erstellen. Dieser Prozess beginnt mit der Auswahl des spezifischen Board-Typs innerhalb der STM32CubeMX-Oberfläche, woraufhin ein standardisiertes Makefile für dieses Board generiert wird. Dieses Makefile enthält bereits

vordefinierte Compiler-Flags, Tools und Regeln, die für den Aufbau des Projekts benötigt werden, einschließlich der Spezifikation der **CPU** (-mcpu=cortex-m4) und der Mikrocontroller-Einheit (**MCU**).

6.1.2 Build-Prozess

Die im Makefile definierten Regeln steuern den Build-Prozess, beginnend mit der Kompilierung der Quell- und Header-Dateien, über das Linken der benötigten Objekte, bis hin zum finalen Programm, das auf dem Mikrocontroller ausgeführt werden kann.

6.1.3 Flash-Prozess

Für das Flashen des Programms auf das STM32-Board wurde das Tool ***st-flash*** verwendet, das speziell für STM32-Mikrocontroller geeignet ist und die Adresse ***0x8000000*** für den Flash-Vorgang nutzt. Da st-flash nicht standardmäßig auf Linux-Systemen installiert ist, muss es zusätzlich eingerichtet werden, wobei auch der PREFIX ***arm-none-eabi-*** zu definieren ist.

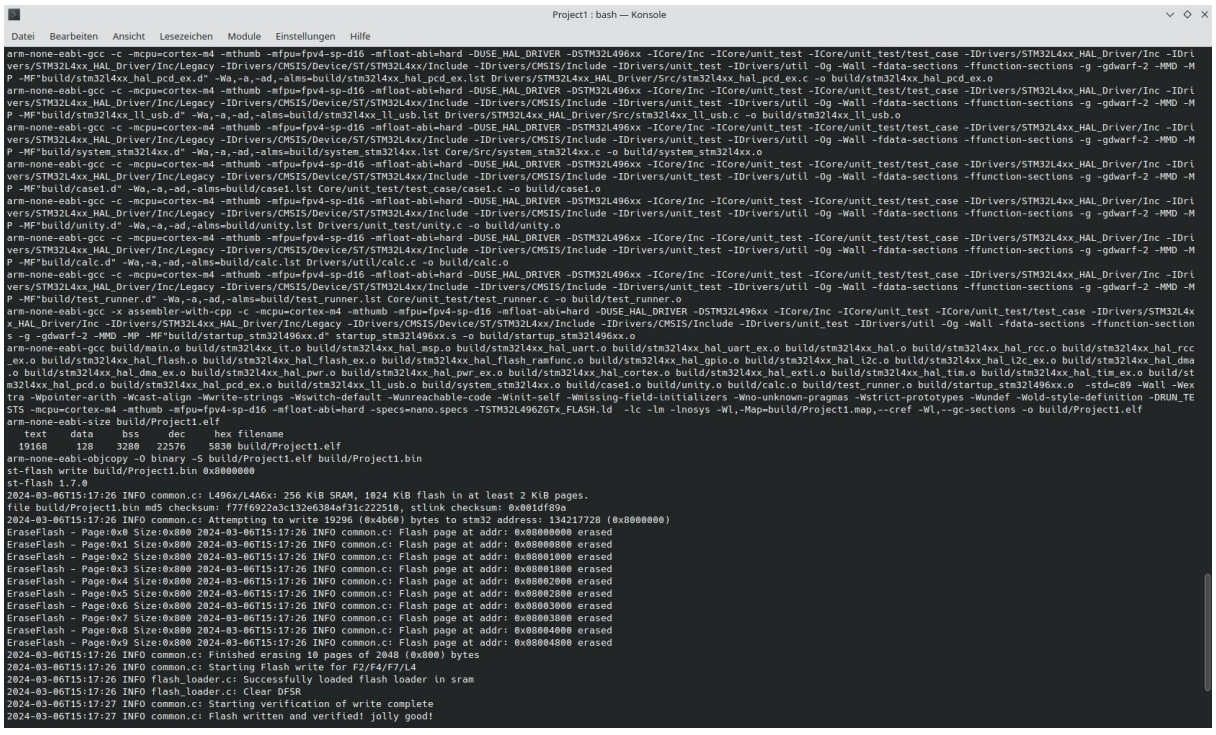


Abbildung 29:Flashprozess von STM32 Makefile

6.1.4 Herausforderungen bei der USB-Konfiguration

Eine besondere Herausforderung stellt die Erkennung des USB-Ports dar. ***Linux***-Systeme erkennen nicht immer sofort die Verbindung zum ***STM32***-Board, was zusätzliche Konfigurationen erfordert, insbesondere wenn nicht als root (***sudo***) gearbeitet wird. Darüber

hinaus kann sich der Name des USB-Ports (***ttyACM0***, ***ttyACM1*** etc.) ändern, was eine manuelle Anpassung im Makefile für die automatisierte Testumgebung erforderlich macht.

die Ausgabe der seriellen Schnittstelle eines ***STM32*** Mikrocontrollers auf einem Linux-System in eine Textdatei zu speichern, veranschaulicht ein typisches Problem in der Automatisierung von Testumgebungen für eingebettete Systeme. Die Entscheidung, ***screen*** anstelle von ***cat*** für diesen Zweck zu verwenden, war eine pragmatische Lösung, die auf spezifischen technischen Einschränkungen und Anforderungen basiert.

6.1.4.1 Warum *Screen* statt *Cat*?

cat ist ein einfaches Unix-Utility, das zum Lesen von Dateien und zur Ausgabe ihres Inhalts auf die Standardausgabe verwendet wird.[27]

In der Praxis hat sich gezeigt, dass das ***cat*** besonders effektiv für das Lesen statischer Dateien oder für Geräte mit einem kontinuierlichen Datenfluss eingesetzt werden kann, stößt es bei der Interaktion mit Geräten, die sporadische Daten senden oder eine initiale Konfiguration erfordern, an seine Grenzen.

STM32 Mikrocontroller könnten aufgrund ihres begrenzten Speichers Schwierigkeiten haben, eine kontinuierliche Datenübertragung über die serielle Schnittstelle zu verarbeiten, was zu Datenverlust führen kann, wenn die Ausgabe schneller erfolgt, als ***cat*** sie verarbeiten kann. ***screen*** hingegen bietet eine Lösung, da es eine Terminal-Sitzung simuliert, die besser mit den Charakteristiken der seriellen Kommunikation zurechtkommt.

Der Einsatz von ***screen*** im Non-Interactive-Modus mit den Optionen -L (Logging aktivieren), -dmS (Starten einer benannten Session im Detached-Modus) und anschließendem Beenden der Session über screen -S [session_name] -X quit ermöglicht es, die Ausgabe der seriellen Schnittstelle effizient in eine Textdatei umzuleiten, ohne dass eine manuelle Interaktion erforderlich ist.

Dieser Ansatz löst das Problem der manuellen Steuerung von ***screen***, da die Erstellung und das Beenden der Session vollständig über das ***Makefile*** gesteuert werden können. Die Konfiguration sieht wie folgt aus.:

```
usb_config_and_flash:
    @echo "Configuring USB..."
    @screen -L -Logfile usb.txt -dmS usb_session /dev/ttyACM0 115200
    @sleep 5  # Die Dauer kann angepasst werden
    @screen -S usb_session -X quit
    @echo "USB configuration completed."
```

Abbildung 30:Konfigurations und Flashs Befehl von STM32 Makefile

[27] (15)

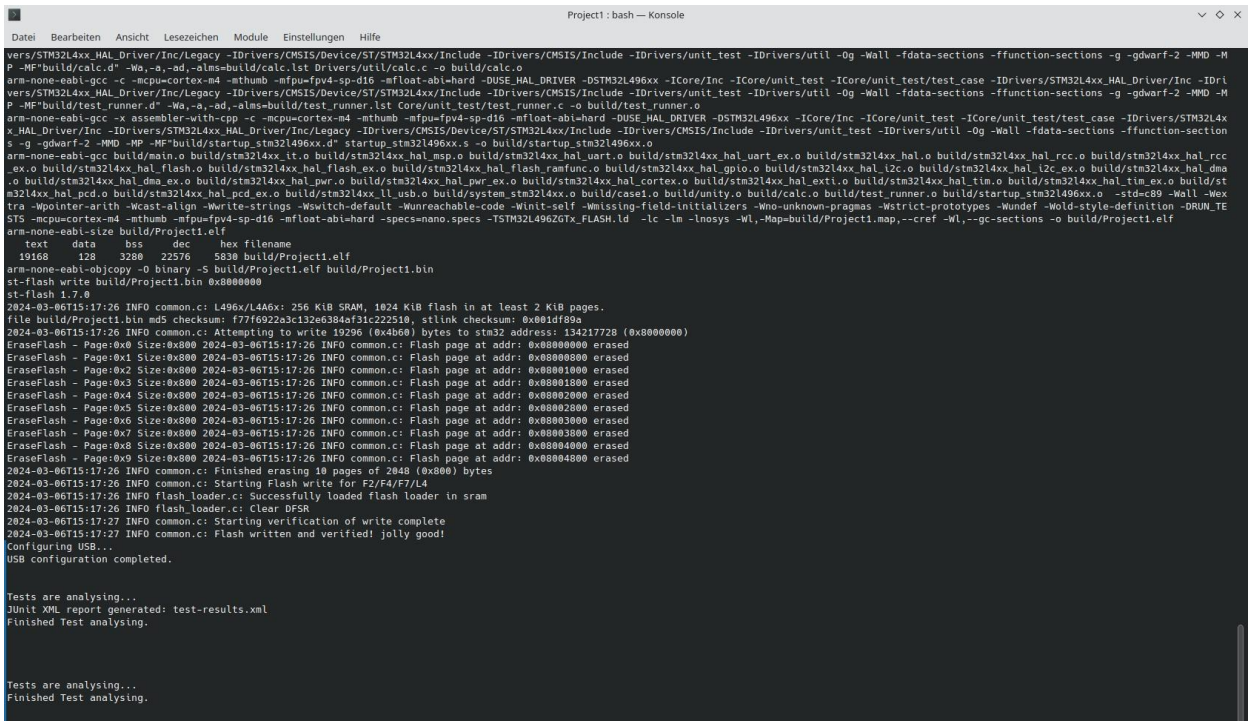


Abbildung 31::Endergebnisse nach dem Ausführen von STM32 Makefile

6.2 Fallstudie 2: TMS570

Die Entwicklung einer Testumgebung für den ***TMS570*** Mikrocontroller von ***TI*** stellte eine besondere Herausforderung dar, insbesondere im Vergleich zum ***STM32***. Die Komplexität des ***TMS570*** und das Fehlen eines äquivalenten Tools wie ***STM32CubeMX***, das automatisch ein ***Makefile*** generieren könnte, erforderten eine tiefgehende Auseinandersetzung mit der Softwarestruktur und dem Aufbau von Projekten für diese Plattform.

6.2.1 Anpassung des Makefiles

Ohne die Unterstützung eines automatischen ***Makefile***-Generators war es notwendig, den Aufbau des ***Makefile***s von Grund auf zu konzipieren. Dazu gehörte das Verständnis darüber, wie das ***CCS*** von ***Texas Instrument***s Projekte aufbaut, einschließlich der verwendeten Flags, Compiler und der spezifischen CPU-Konfiguration des ***TMS570***. Die detaillierte Analyse des ***CCS*** ermöglichte es, ein funktionsfähiges Makefile zu erstellen, das den Anforderungen des ***TMS570*** gerecht wird und eine .out-Datei generiert, die auf dem Mikrocontroller geflasht werden kann.

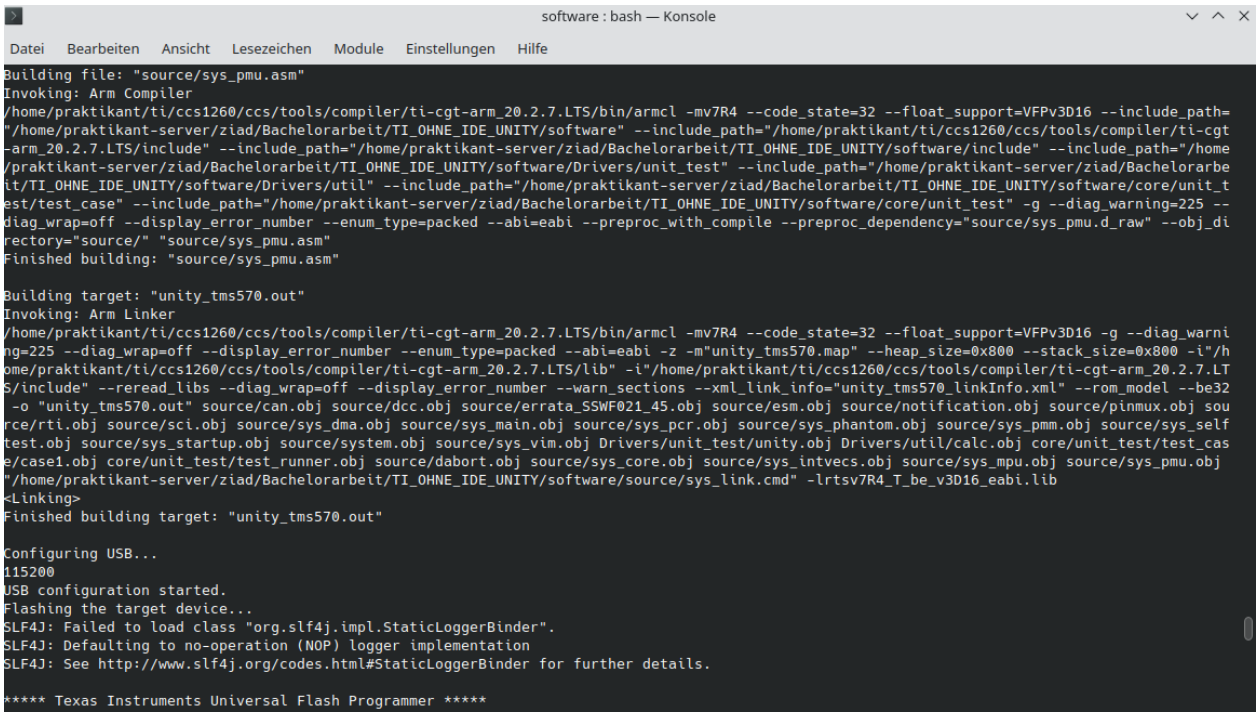


Abbildung 32:Bauen von TMS Projekt mit Makefile

6.2.2 Flash-Prozess

Die Suche nach einer geeigneten Methode zum Flashen des Programms führte zur Entdeckung des ***Uniflash***-Tools von ***TI***. ***Uniflash*** bietet die Möglichkeit, Befehle über die Kommandozeile auszuführen, was eine nahtlose Integration in das ***Makefile*** ermöglichte. Die Verwendung von ***Uniflash*** löste das Problem des Flashens und stellte sicher, dass das Programm erfolgreich auf dem ***TMS570*** Mikrocontroller ausgeführt werden konnte.

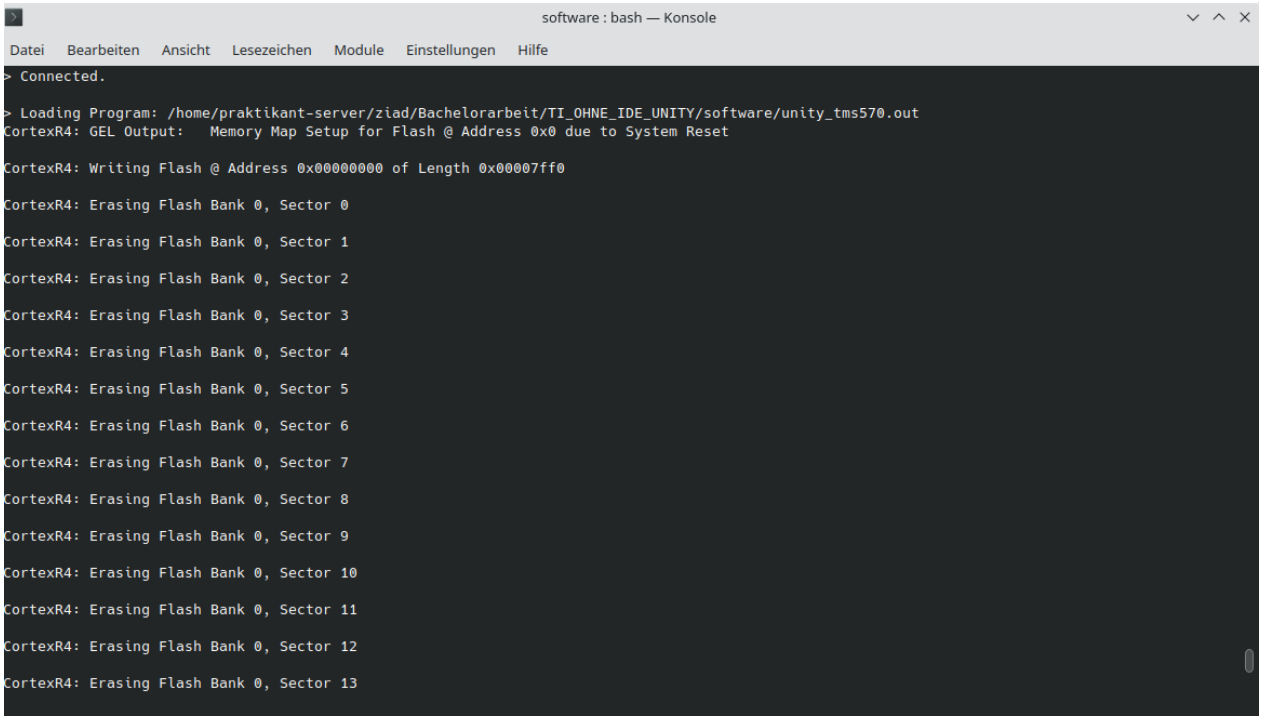


Abbildung 33:Flashprozess von TMS Makefile

6.2.3 Ausgabe der seriellen Schnittstelle

Ein weiteres zentrales Problem war die Darstellung und Speicherung der Ausgabe der seriellen Schnittstelle in einer Textdatei. Aufgrund der Erfahrungen mit dem ***STM32*** wurde beschlossen, ***cat*** zu verwenden, um die Ausgabe von dem Board in usb.txt umzuleiten. Die Verwendung von ***cat*** in Kombination mit stty zur Konfiguration der seriellen Schnittstelle ermöglichte es, die Testergebnisse effektiv zu erfassen.

Dieser Teil beginnt die Definition eines Makefile-Targets namens **usb_config_and_flash**, das für die Konfiguration der USB-Verbindung und das Flashen des Programms auf den ***TMS570*** Mikrocontroller verwendet wird.

```
usb_config_and_flash:
    @echo "Configuring USB..."
```

Abbildung 34:Konfigurations und Flashs Befehl von TMS570 Makefile (1)

Um einer Nachricht auszugeben, die den Start der USB-Konfiguration signalisiert, wurde die Befehl verwendet:

```
@stty -F /dev/ttyUSB0 raw speed 115200
```

Abbildung 35:Konfigurations und Flashs Befehl von TMS570 Makefile (2)

Hier wird der ***cat***-Befehl verwendet, um die Ausgabe von /dev/ttyUSB0 in die Datei usb.txt umzuleiten. Der Prozess wird im Hintergrund ausgeführt (&), und die Prozess-ID (PID) des ***cat***-Befehls wird in cat_pid.txt gespeichert. Dies ermöglicht es, den ***cat***-Prozess später gezielt zu beenden.

```
@cat /dev/ttyUSB0 > usb.txt & echo $$! > cat_pid.txt
```

Abbildung 36:Konfigurations und Flashs Befehl von TMS570 Makefile (3)

Das ***Makefile*** pausiert für 5 Sekunden, um sicherzustellen, dass genügend Zeit für die Initialisierung der USB-Verbindung zur Verfügung steht. Diese Zeit kann je nach Bedarf angepasst werden.

```
@sleep 5  # Adjust this duration as necessary
```

Abbildung 37:Konfigurations und Flashs Befehl von TMS570 Makefile (4)

Dieser Teil startet den Flash-Prozess des Mikrocontrollers mit dem ***Uniflash***-Tool (PFLASH). Die Variablen VFLASH, FILE_PATH und TARGET werden verwendet, um den Pfad zur .out-Datei und die notwendigen Optionen für das Flashen zu spezifizieren.

```
@echo "Flashing the target device..."
@$(PFLASH) $(VFLASH) $(FILE_PATH)$(TARGET).out -targetOp run
```

Abbildung 38:Konfigurations und Flashs Befehl von TMS570 Makefile (5)

Erneut wird eine Pause eingelegt, um dem Flash-Prozess Zeit zu geben, abgeschlossen zu werden.

```
@sleep 5  # Adjust this duration as necessary
```

Abbildung 39:Konfigurations und Flashs Befehl von TMS570 Makefile (6)

Diese Bedingung überprüft, ob cat_pid.txt existiert und nicht leer ist. Falls ja, wird der ***cat***-Prozess, der die serielle Ausgabe liest, mittels seiner PID beendet (kill-Befehl), und die cat_pid.txt-Datei wird gelöscht.

```
    @if [ -s cat_pid.txt ]; then \
        kill `cat cat_pid.txt` && rm cat_pid.txt; \
    fi
```

Abbildung 40:Konfigurations und Flashs Befehl von TMS570 Makefile (7)

Zum Schluss wird eine Nachricht ausgegeben, die das erfolgreiche Abschließen der USB-Konfiguration und des Flash-Prozesses bestätigt.

```
    @echo "USB configuration and flash completed."
```

Abbildung 41:Konfigurations und Flashs Befehl von TMS570 Makefile (8)

Und die ganze Implementierung sieht dann so aus:

```
usb_config_and_flash:
    @echo "Configuring USB..."
    @stty -F /dev/ttyUSB0 raw speed 115200
    @cat /dev/ttyUSB0 > usb.txt & echo $$! > cat_pid.txt
    @echo "USB configuration started."
    @sleep 5  # Adjust this duration as necessary
    @echo "Flashing the target device..."
    @$(PFLASH) $(VFLASH) $(FILE_PATH)$(TARGET).out -targetOp run
    @sleep 5  # Adjust this duration as necessary
    @if [ -s cat_pid.txt ]; then \
        kill `cat cat_pid.txt` && rm cat_pid.txt; \
    fi
    @echo "USB configuration and flash completed."
```

Abbildung 42:Konfigurations und Flashs Befehl von TMS570 Makefile

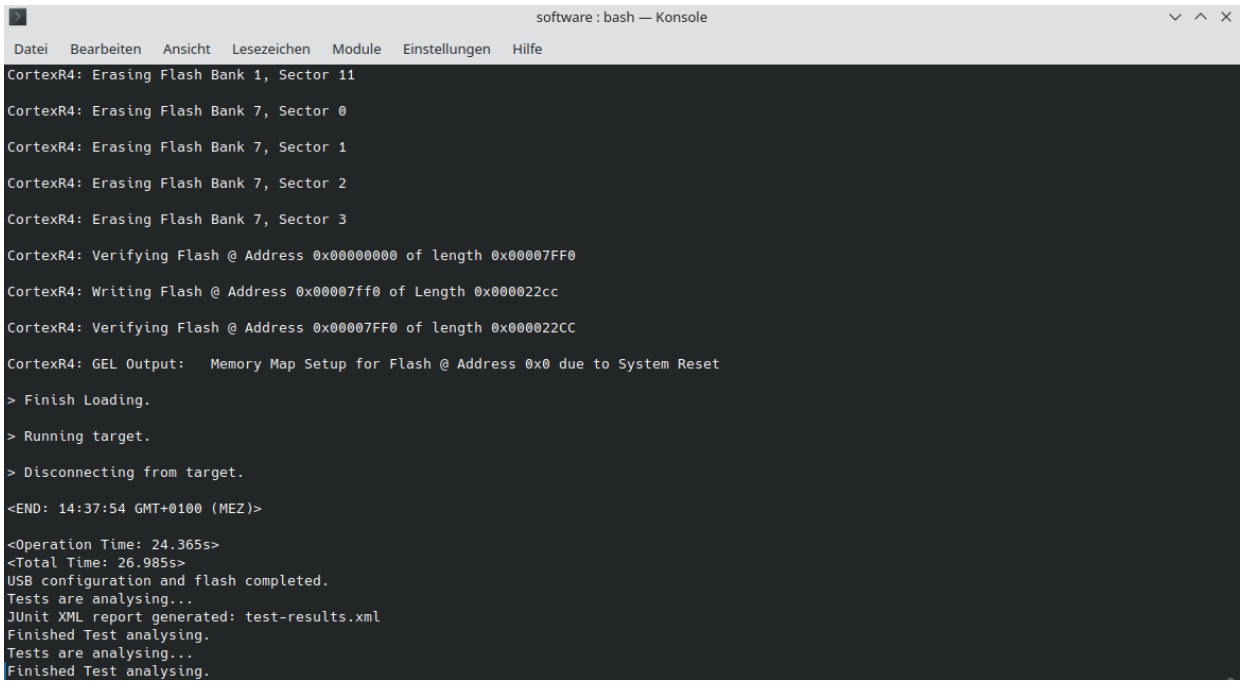


Abbildung 43:Endergebnisse nach dem Ausführen von TMS Makefile

6.2.4 Stabilität des USB-Gerätenamens

Im Gegensatz zum ***STM32***, bei dem sich der Name des USB-Geräts (**ttyACM0**, **ttyACM1** usw.) je nach Situation ändern kann, wurde bei der Arbeit mit dem ***TMS570*** festgestellt, dass der Name des USB-Geräts (**ttyUSB0**) stabil bleibt. Dies vereinfacht die Automatisierung im ***Makefile*** erheblich, da der USB-Gerätename nicht jedes Mal angepasst werden muss, wenn das Gerät neu verbunden oder der ***PC*** neu gestartet wird.

7 Diskussion und Bewertung

Die Entwicklung und Implementierung einer integrierten Entwicklungsumgebung (**IDE**)-freien Testumgebung für Mikrocontroller-Systeme wie den ***STM32*** und ***TMS570*** stellt eine innovative Methode dar, um den Entwicklungsprozess von On-Board-Systemen zu optimieren. Durch diesen Ansatz ließen sich der Build- und Testprozess erfolgreich automatisieren und eine hohe Flexibilität im Hinblick auf unterschiedliche Hardware-Konfigurationen erzielen. Allerdings war dieser Prozess auch mit spezifischen Herausforderungen und Limitationen verbunden, die im Fokus der nachfolgenden Diskussion und kritischen Bewertung stehen.

7.1 Effektivität der Testumgebung

Mit der erstellten Testumgebung durch wurde den Erfolg des Automatisierung des Build und Testprozesses für die beide Mikrocontroller-Typen bewiesen. Besonders hervorzuheben ist die Fähigkeit, den gesamten Testzyklus – von der Kompilierung über das Flashen bis hin zur Ausführung der Tests und der Erfassung der Ergebnisse – ohne manuelle Eingriffe

durchzuführen und damit ist trägt die Automatisierung zur Reduktion von Fehler bei, die durch manuelle Prozesses passieren könnten, und verbessert die Effizienz des Entwicklungszyklus.

7.2 Kritische Bewertung und Limitationen

Obwohl die Testumgebung eine Reihe von Vorteilen bot, zeigten sich bei ihrer Entwicklung und Anwendung eine Reihe von Limitationen :

7.2.1 Anpassungsbedarf beim STM32

Ein zentrales Problem bildete die Instabilität des USB-Gerätenamens beim ***STM32***. Dieser ändert sich je nach Situation und Systemkonfiguration (z. B. von ***ttyACM0*** zu ***ttyACM1***) und muss daher von Hand im ***Makefile*** entsprechend angepasst werden, um eine korrekte Kommunikation zu ermöglichen. Diese Notwendigkeit zur regelmäßigen Anpassung kann den Grad der Automatisierung einschränken und den Entwicklungsprozess verlangsamen.

7.2.2 Komplexität beim Erstellen des Makefiles für TMS570

Im Gegensatz zum ***STM32*** unterstützen Tools wie ***STM32CubeMX*** das Erstellen von ***Makefile***s. Für den ***TMS570*** war jedoch ein manueller Ansatz notwendig, der einen erheblichen Mehraufwand und die Auseinandersetzung mit der Toolchain und Projektstrukturen erforderte.

7.2.3 Wartungs- und Skalierbarkeitsprobleme

Die Pflege und Weiterentwicklung der Testumgebung, v. a. beim Anpassen an neue Mikrocontroller oder Änderungen in der Toolchain, stellt eine durchgehende Herausforderung dar. Zudem kann das Skalieren der Umgebung auf größere oder teamspezifische Projekte zusätzliche Komplexität einführen.

7.3 Ausblick und zukünftige Entwicklungen

Die Erfahrungen und Erkenntnisse aus der Entwicklung und Implementierung der Testumgebung ohne **IDE** für ***STM32*** und ***TMS570*** Mikrocontroller bieten eine umfangreiche Grundlagen für zukünftige Verbesserungen und Entwicklungen in der Automatisierung von Testprozessen Embedded Systems.

7.3.1 Dynamische Erkennung von USB-Geräten

Eine automatisierten Lösung zur Erkennung den wechselnde USB-Gerätenamen könnte den Prozess vereinfachen und erhöht auch den Automatisierungsgrad weiter.

7.3.2 Erweiterte Tool-Unterstützung

Die Entwicklung oder Integration zusätzlicher Tools, die den Prozess der Makefile-Erstellung ähnlich wie ***STM32CubeMX*** für ***STM32***, würde den Setup-Aufwand für weitere Mikrocontroller-Plattformen reduzieren.

7.3.3 KI-gestützte Testumgebung

Eine attraktive Entwicklung wäre die Schaffung einer **KI**-gestützten Testumgebung, die in der Lage ist, die Funktionalität von Software in eingebetteten Systemen umfassend zu testen. Diese Umgebung würde nicht nur auf vordefinierten Tests basieren, sondern könnte selbstständig neue Testfälle generieren, die mögliche Fehlerquellen und Schwachstellen in der Software herausfinden. Insbesondere bei der Änderung oder Erweiterung von Funktionen könnte die **KI** die Relevanz bestehender Test bewerten und bei Bedarf neue Test generieren, um die modifizierten Funktionen effektiv abzudecken.

8 Schlussfolgerung

„I never lose. I either win or learn.“ (Nelson Mandela)

in dieser Forschung wurde intensiv erforscht, wie eine automatisierte Testumgebung für eingebettete Systeme, besonders für den Mikrocontroller ***STM32*** und ***TMS570***, entwickelt und implementiert werden kann. Das Ziel war es, eine sinnvolle Lösung für die Herausforderungen und Abhängigkeiten zu finden, die mit der integrierte Entwicklungsumgebung **IDE**s verbunden sind.

Es ist mein Bestreben gewesen, eine solche Lösung zu finden, die nicht nur die Abhängigkeit von IDEs reduziert, sondern auch neue Perspektiven für den Unit Test in eingebetteten Systemen, insbesondere deren Automatisierung, eröffnet.

Durch die erfolgreiche Umsetzung der Testumgebung konnte gezeigt werden, dass eine effiziente und effektive Testautomatisierung möglich ist, die eine Steigerung von der Produktivität und der Softwarequalität mit sich bringt.

Ich hoffe, dass die in dieser Forschungsarbeit erzielten Ergebnisse als Meilenstein am Anfang der Entwicklung in diesem Bereich angesehen werden können. Mein Anliegen war nicht nur praktische Lösung für bestehende Probleme zu bieten, sondern auch den Weg für zukünftige Forschungen und Entwicklung zu ebnen.

Abschließend möchte ich zum Ausdruck bringen, dass ich hoffe, mit dieser Forschung einen wertvollen Beitrag zur Gemeinschaft der Embedded-Systementwickler geleistet zu haben. Die Auseinandersetzung mit den Limitationen bestehender Ansätze und die Entwicklung neuer Methoden zur Testautomatisierung sollen nicht nur die tägliche Arbeit von Entwicklern

erleichtern, sondern auch die Grundlagen für eine fortlaufende Verbesserung und Innovation in diesem spannenden und sich ständig weiterentwickelnden Feld legen.

9 Literaturverzeichnis

1. **Informationstechnik, Bundesamt für Sicherheit in der.** Eingebettete Systeme. [Online] Feburar 2020. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium_Einzel_PDFs/07_SYS_IT_Systeme/SYS_4_3_Eingebettete_Systeme_Edition_2020.pdf?_blob=publicationFile&v=1#:~:text=Eingebettete%20Systeme%20sind%20informationsverarbeitende%20Systeme,n.

2. **Earls, Alan R.** computerweekly. [Online] Mai 2020 . https://www.computerweekly.com/de/definition/Embedded-System-Eingebettetes-System.

3. **wikipedia.** wikipedia. [Online] https://de.wikipedia.org/wiki/Eingebettetes_System.

4. **HACK, CARSTEN.** e-hack. [Online] 29. Juli 2020. https://www.e-hack.de/was-ist-ein-eingebettetes-system-definition/.

5. **Brett Daniel.** trentonsystems. *trentonsystems.* [Online] 22. Juli 2022. https://www.trentonsystems.com/en-us/resource-hub/blog/what-are-embedded-systems.

6. **wikipedia.** wikipedia. [Online] https://de.wikipedia.org/wiki/Byte-Reihenfolge.

7. **SLR.** embetronicx. [Online] October 28, 2023. https://embetronicx.com/tutorials/p_language/c/little-endian-and-big-endian/.

8. **wikipedia.** *en.wikipedia.* [Online] https://en.wikipedia.org/wiki/ARM_Cortex-M.

9. **st.** *st.* [Online] https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers.html.

10. **ti.** ti. [Online] https://www.ti.com/lit/ds/symlink/tms570ls3137.pdf?ts=1708581811745&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTMS570LS3137.

11. **Khan, Salman.** LAMBDATEST. [Online] https://www.lambdatest.com/learning-hub/unit-testing.

12. **VanderVoord, Mark and Karlesky, Mike.** Udemy. [Online] Dezember 2020. https://www.udemy.com/course/unit-testing-and-other-embedded-software-catalysts/.

13. **Siñuela Pastor, David .** github. [Online] https://github.com/siu/minunit.

14. **Cpputest.** cpputest.github. [Online] https://cpputest.github.io/.

15. **Wikipedia.** wikipedia.org. [Online] wikipedia. https://en.wikipedia.org/wiki/Cat_(Unix).

16. **Wüst, Prof. Dr. K. , Kneisel, Prof. Dr. P. und Aßmann, Tim .** homepages.thm. [Online] 01. 03 2005. https://homepages.thm.de/~hg6458/BlockseminarvortraegeWS05/Testfallentwurf_und_Modultests.pdf.

Abbildungen :