THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

STUDIUM PLUS
DUALES STUDIUM

Projektstudium im Sommersemester 2021

Proof of Concept und Implementierung eines filesystems unter Integrity auf embedded Hardware

Vorgelegt von: Jannik Szwajczyk
Ludwig-Erk-Straße 1H
35578 Wetzlar

Matrikelnummer: 5234101

Eingereicht bei
Hochschulbetreuer: Prof. Dr. Carsten Lucke
Fachbetreuer: Guy Sagnes

Unternehmen: Continental Automotive GmbH

Eingereicht am: 30.08.2021

Continental

Sperrvermerk

Der vorliegende Praxisphasenbericht beinhaltet interne vertrauliche Informationen der Firma Continental Automotive GmbH.
Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma Continental Automotive GmbH.

Inhaltsverzeichnis

Abkürzungsverzeichnis

BA Business Area

BU Business Unit

BS Betriebssystem

CE Central Engineering

DS Dateisystem

fs filesystem

IIP Integrated Interior Platform

lfs littlefs

OS open-source

PoC Proof of Concept

TCU Telematic control unit

VNI Vehicle Networking and Information

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Continental

Die Continental AG ist ein weltweit tätiger deutscher Automobilzulieferer mit Hauptsitz in Hannover. Zur Zeit werden circa 230.000 Mitarbeiter weltweit beschäftigt, welche an über 400 Standorten in insgesamt 61 Ländern tätig sind.

Continental ist in fünf Business Areas aufgeteilt mit diversen Business Units pro Business Area. Wetzlar gehört Zentral zur Business Area Vehicle Networking and Information und beherbergt alle drei in Abbildung 1 aufgelisteten Business Units.

Der Standort wurde 1955 von Philips gegründet und 2007 von Continental übernommen. Zur Zeit fokussiert sich das Produkt Portfolio von Wetzlar auf die Entwicklung von digitalen Kombiinstrumenten, sowie Telematic control units.

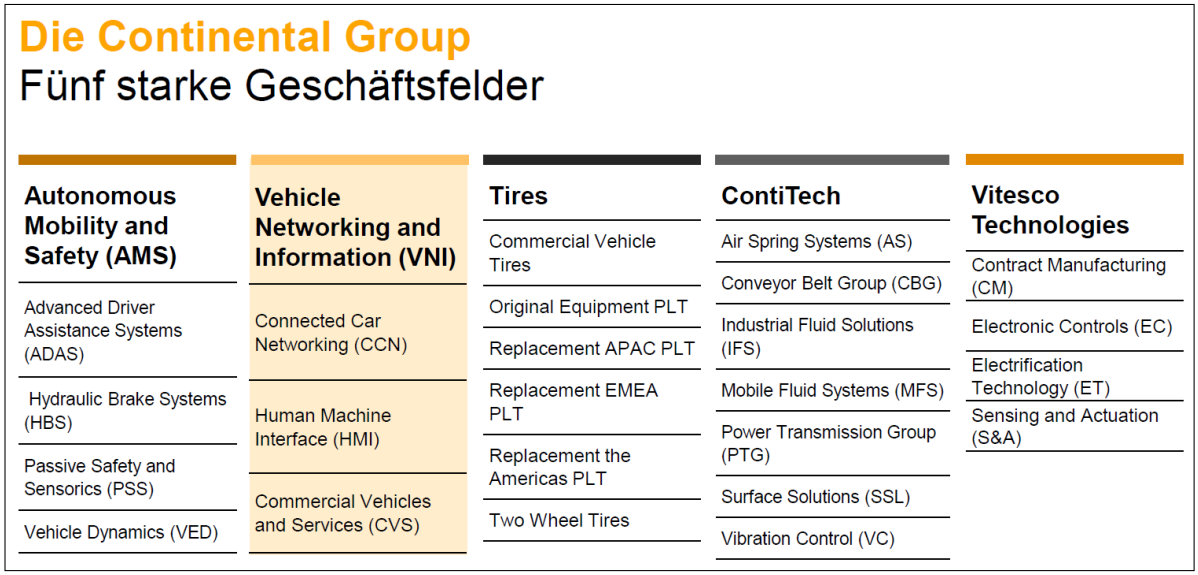


Abbildung 1: 5 Business Areas von Continental

1.2 Abteilung Persistence

Das Persistence Team gehört zur Business Unit Central Engineering. Dabei sind sie Teil des Softwareplatform Projekts Integrated Interior Platform, welches sich mit der Entwicklung einer high performance Softwareplatform für diverse Varianten beschäftigt. Hierbei liegt der

Fokus auf der Wiederverwendbarkeit der Architektur, sowie der Platform für verschiedene Kunden.
Hierbei kümmert sich das Team um den Zugriff auf den Speicher der Hardware, sowie um die Zuteilung der Speicherpartitionen.

1.3 Problemstellung

Für zukünftige Anwendungsfälle sollen einige der Integrated Interior Platform Varianten ein kleines Filesystem bekommen, mit welchem einfache Daten von den Applikationen gespeichert werden können. Allerdings kostet das mit Integrity OS mitgelieferte Filesystem pro produziertem hardware target Geld. Da dieses außerdem noch die benötigten Funktionen übertrifft, soll nach einer kostengünstigen Alternative gesucht werden.

1.4 Proof of Concept

Das Proof of Concept kommt ursprünglich aus dem Projektmanagement und wird genutzt um die Durchführbarkeit eines Vorhabens zu belegen. Dabei kommt es entweder zu einem positiven oder negativen Ergebnis, welche die weitere Projektplanung bestimmt.

Meistens wird bei dem Proof of Concept auch ein Prototyp entwickelt, welcher die Kernfunktionalitäten des Vorhabens aufweist. Gerade bei software orientierten Projekten ist der Prototyp notwendig um Schnittstellen um die Umgebung zu identifizieren.

Mit Hilfe des Proof of Concept können außerdem Risiken und Entscheidungen für den weiteren Projektverlauf minimiert werden. Zum einen werden fürh kritische Anforderungen an die Anwendung validiert, zum Anderen wird das Risiko, sowie das Budget durch den Prototypen minimiert.

1.5 Motivation

Mit einer eigenen Implementierung eines Filesystems kann Geld für die spätere Produktion gespart werden, was je nach Menge der produzierten Steuergeräte eine hohe Summe an Geld sparen kann. Außerdem ist die Software, welche während der Proof of Concept entsteht später in jedem Gerät welches produziert wird vertreten.

1.6 Ziel

Ziel des Projektstudiums ist es ein Proof of Concept für die Implementierung eines Filesystems zu erstellen und die darauffolgende Implementierung des selbigen im Integrated Interior Platform Projekt.

2 Theoretische Grundlagen

2.1 Aufgabenstellung

Es soll ein Proof of Concept erstellt werden um zu überprüfen, inwiefern die Implementierung eines open-source filesystem in das bestehende Projekt möglich ist. Dafür müssen zuerst die grundsätzlichen Anforderungen an das filesystem seitens des Projektes und Systems dokumentiert werden. Danach findet eine testweise Einbindung des filesystem in das bestehende Projekt statt.

Falls dies erfolgreich sein sollte, wird eine vollständige und saubere Implementierung mit samt Dokumentation erfolgen.

2.2 Ausgangslage

Zu Beginn des Projekts gibt es das bereits bestehende System der Integrated Interior Platform, in welchem das filesystem integriert werden soll. Das vorhandene Betriebssystem ist dabei ein Integrity OS von Green Hills Software.

2.3 Embedded filesystem littlefs

Bei littlefs handelt es sich um ein open-source filesystem, welches aus der embedded Welt kommt. Häufig wird es auf kleinen Mikrocontrollern, wie einem ESP32 oder ähnlichen verwendet.

Auf den ersten Blick liefert es jedoch auch die von uns gewünschten Funktionen, welche das filesystem besitzen muss. Dazu gehören Eigenschaften, wie der Erhalt von Daten bei Stromverlust, gleichmäßiger Verschleiß des Arbeitsspeichers, sowie die Nutzung einer begrenzten Menge des Selbigen.

Da die sonstigen Anforderungen an die Funktionalitäten des filesystem relativ gering sind, scheint littlefs eine gute Wahl für die Umsetzung zu sein.

2.4 Roadmap

Um den Start des Projekts zu vereinfachen und strukturieren, soll eine gewisse Roadmap entstehen, welche die wichtigsten Punkte für das Proof of Concept abzudecken.

2.5 Planung der Arbeitspakete

Die Planung der Arbeitspakete erfolgte erst nachdem ein Grundverständnis für das Projekt vorhanden war, sowie ersichtlich ist, dass die Implementierung des filesystem generell möglich ist.
Die Arbeitspakete werden in der Projektmanagement Software JIRA verwaltet.

2.6 Dokumentation des Projekts

Die Dokumentation des Projekts erfolgt parallel zum selbigen über die Wiki-Software Confluence. Dies erfolgt in einem wöchentlichen Meeting, welches zum Austausch über den momentanen Projektstatus, sowie die weitere Planung des Projekts dient.
Außerdem werden die einzelnen Arbeitspakete des Projektes ebenfalls im Confluence dokumentiert, sowie eine abschließende Dokumentation des gesamten Projekts.

3 Proof of Concept

Zum Start des Projekts wird zusammen mit den betreuenden Kollegen eine Roadmap entworfen. Diese bietet eine grobe Übersicht über die Arbeitsschritte und soll gerade zu Beginn bei der Strukturierung helfen.

Hierfür wird eine Confluence Seite eingerichtet, auf welcher die groben Arbeitsschritte, sowie Diagramme und weitere Informationen hinterlegt werden. Außerdem wird eine Unterseite mit erstellt, auf welcher der Austausch im wöchentlichen Projektmeeting dokumentiert wird. Hier können Informationen festgehalten werden, sowie Aufgaben mit zeitlicher Begrenzung zugeteilt werden.

Alle diese Schritte sollen zu einem erfolgreicheren Informationsaustausch, sowie Projektverlauf beitragen.

3.1 Roadmap

Die Arbeitsschritte sind in einem initialen Projektmeeting zusammen mit den Betreuern entstanden. Dabei fand ein Brainstorming statt und alle gesammelten Ideen wurden aufgeschrieben.

Allerdings mussten diese nun noch in eine logische Reihenfolge gebracht und in Kategorien unterteilt werden. Diese geordnete und strukturierte Roadmap wurde im Nachgang auf der Hauptseite im Confluence als "Tasks"hinterlegt.

Dort dient sie als Übersicht zum aktuellen Projektstatus und um nachverfolgen zu können, welche Tätigkeiten noch erledigt werden müssen. So können eventuell benötigte Ressourcen, wie zum Beispiel Testrechner, frühzeitig identifiziert werden.

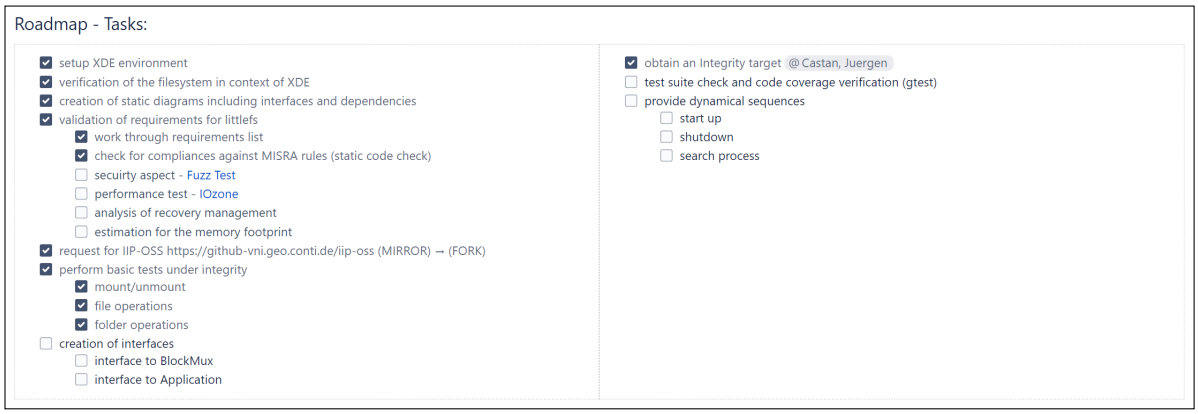


Abbildung 2: Roadmap

Die Roadmap ist in zwei Bereiche aufgeteilt. Auf der linken Seite ist ein chronologisch geplanter Ablauf von grob definierten Arbeitspaketen, auf der rechten Seite hingegen Aufgaben welche nicht zum direkt Entwicklungsprozess zugehörig sind, jedoch trotzdem erledigt werden müssen.

3.2 Einarbeitung

? Zu Beginn des Projekts war eine Einarbeitung in die nachfolgenden Themen unerlässlich.

3.2.1 Dateisysteme

Das Dateisystem ist verantwortlich für die Verwaltung von Daten auf einem Speichermedium und ist ein Teil des Betriebssystems. Dabei kann sich die Art des Dateisystems je nach Speichermedium unterscheiden. Ohne ein Dateisystem wären die Daten im Speicher lediglich ein großer Datensatz und der Anfang, sowie Ende jedes Einzelnen wären nicht erkennbar.
Die Aufgabe des Dateisystem ist es also die Daten zu schreiben, speichern, zu lesen und diese auch verändern zu können, sowie zu löschen. Dabei definiert und spezifiziert das Dateisystem die Eigenschaften der Daten, so zum Beispiel:

- Konvention für den Dateinamen
- Dateiattribute (erstellt, zuletzt bearbeitet, ...)
- Pfadlänge

Die Dateisysteme unterscheiden sich je nach Betriebssystem, da nicht jedes Betriebssystem jedes Dateisystem unterstützt. Nachfolgend eine Auflistung der gängigen Dateisystem nach Betriebssystem.

- Windows: FAT, FAT16, FAT32, NTFS
- macOS: HFS, HFS+
- Linux: Ext2, Ext3, Ext3

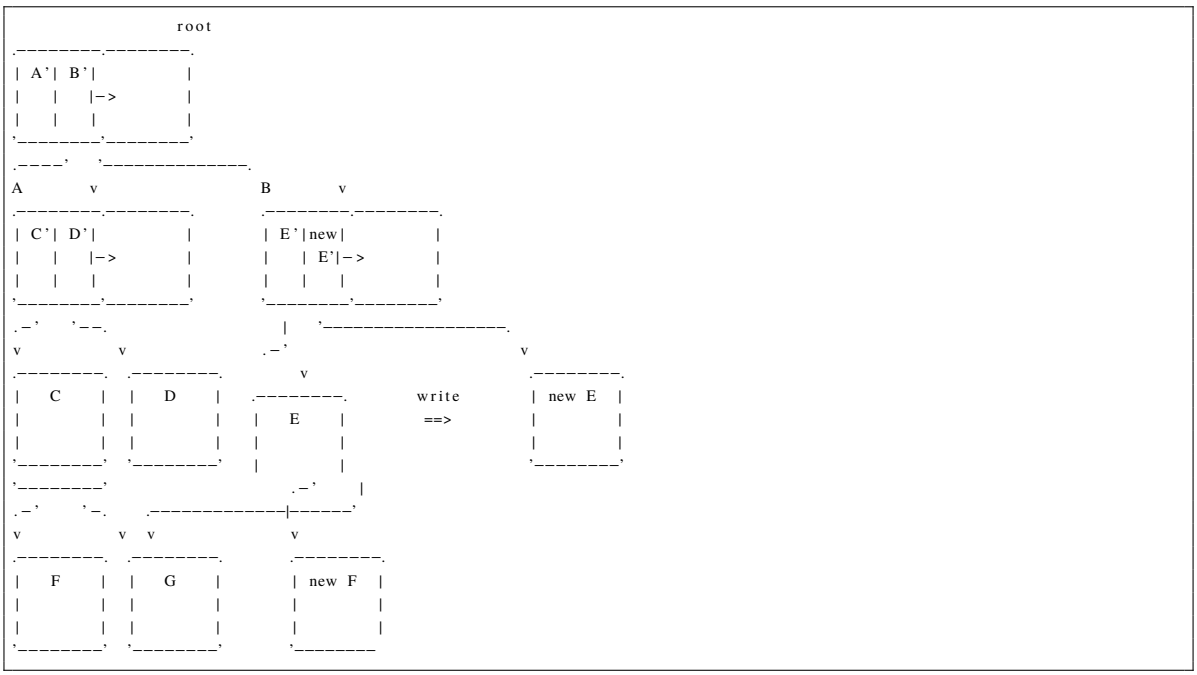
3.2.2 littlefs

Ä little fail-safe filesystemöder auch littlefs ist ein aus der embedded Welt stammendes Dateisystem. Dieses wird normalerweise bei kleineren Mikrocontrollern angewendet, da es keinen großen Funktionsumfang besitzt.

littlefs konzentriert sich auf drei wesentlichen Aspekte:

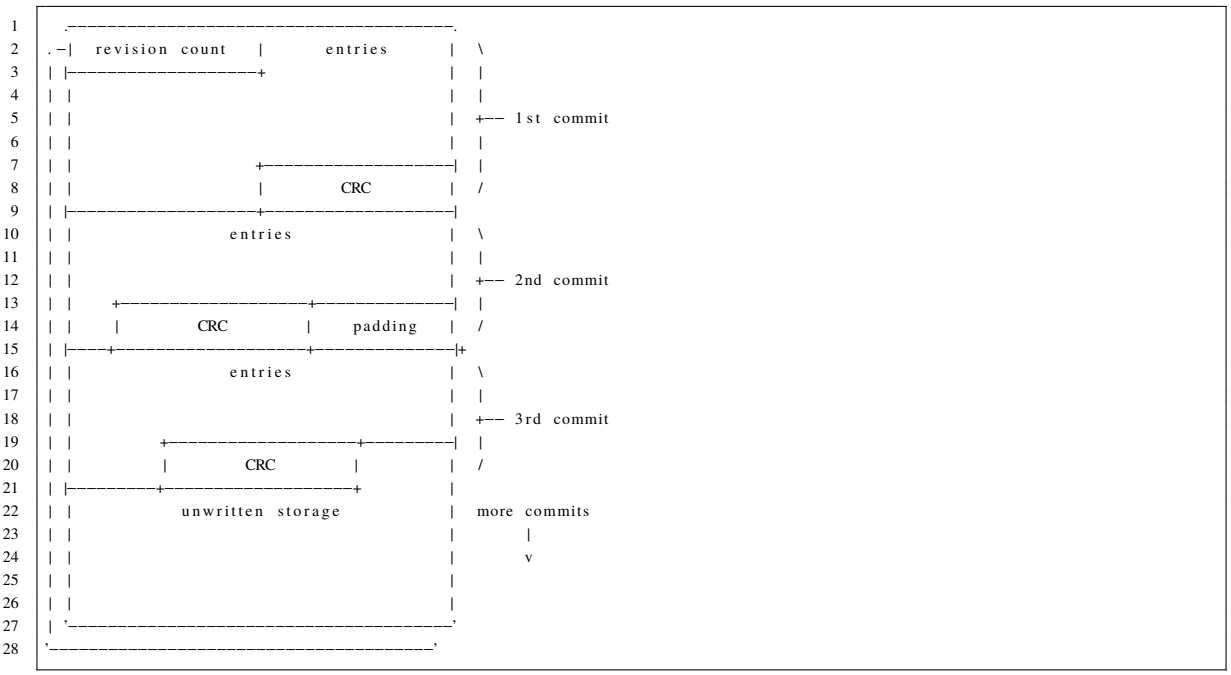
- **Widerstandsfähigkeit gegen Stromausfall**
- **Gleichmäßiger Verschleiß der Hardware**
- **Begrenzter RAM/ROM**

> Kombination aus Logging & Copy on Write
> Aufbau von littlefs als Grafik



> Metadata Pairs > CTZ skip-lists > Block allocator > Wear leveling > Files > Directories > Move problem / Global State

> Metadata pairs, sogar superblock wird als MP gespeichert -> 2 Blocks, 2ter ist Backup während erase cycles oder power lose -> BILD Layout eines metadata blocks



Alle existierenden Dateisystem teilen sich grundsätzlichen Ansätze. Da sich littlefs spezifische Anforderungen in Richtung So nutzt auch littlefs die bereits existierenden Ansätze "Block Based", "Loggingünd "Copy on Write".

Unter "Block Based"versteht man das Aufteilen des Speichers in einzelne Blöcke unter welchen die Daten gespeichert werden. Dieser Ansatz bietet als Vorteile zum einen hohe Operationsgeschwindigkeit sowie eine kleine Speichernutzung des gesamten Dateisystems.

"Logging"nutzt den gesamten Speicher um eine Art Ringbuffer zu erstellen, dabei werden neue Daten einfach an den bisherigen Speicher angehängt. Vorteil hierbei ist das mit Hilfe einer Checksumme leicht Fehler bei einem Verlust des Stroms erkannt werden können, außerdem wird durch die gesamte Nutzung des Speichers dieser gleichmäßig abgenutzt. Allerdings ist die Geschwindigkeit sehr schlecht.

Bei "Copy on Write"wird bei der Änderung der Datenblock kopiert und die Referenz auf den alten geupdated.

3.2.3 IIP

Das zugrunde liegende Betriebssystem der IIP ist ein Integrity RTOS (Real-Time Operating System) der Firma Greenhills. Speziell an diesem Betriebssystem ist die Entkopplung der

Hardware von den einzelnen Applikation, Grund hierfür ist das Ziel an hoher Sicherheit des Betriebssystems. Außerdem ist der Zugriff auf den Source Code innerhalb des Projekts stark limitiert, weshalb keine direkte Einsicht in den Code erfolgen kann.

Dies kann später gegebenenfalls noch zu Komplikationen führe, welche bisher noch nicht berücksichtigt werden können.

3.2.4 Setup der XDE

Das Projekt IIP verwendet eine virtuelle Entwicklungsumgebung in Form einer virtuellen Maschine. Diese basiert auf einem abgewandelten Ubuntu Betriebssystem.

Um für das Projekt arbeiten zu können, soll diese Entwicklungsumgebung genutzt werden. Dazu wird das zugehörige GitHub Repository gecloned und ein Installations-Script gestartet. Nachdem das Skript abgeschlossen war, konnte die virtuelle Maschine gestartet werden.

Hier mussten nun noch einige Tools installiert und konfiguriert werden, so zum Beispiel Visual Studio Code zum arbeiten des Source Codes, wie auch der Greenhills Compiler, welcher vom Projekt zum Bauen der Software genutzt wird.

3.2.5 Implementierung von littlefs unter Linux

Um die Funktionen von littlefs ohne große Implementationen testen zu können, wurde dieses mit Hilfe von fuse unter Ubuntu eingebunden. Fuse steht für "Filesystem in Userspaceünd ist ein Interface um Dateisysteme zum Linux Kernel zu exportieren.

Für littlefs existiert bereits ein Projekt, welches es einem ermöglicht, mit Hilfe eines fuse wrappers, das Dateisystem direkt im user-space zu mounten. Somit können die Funktionen und Limitierungen getestet werden, außerdem wird eine Automatisierung von Tests bezüglich Anforderungen ermöglicht. So können tausende Dateien mit Hilfe eines Python Scripts erstellt werden ohne das dies händisch erfolgen muss.

Um den wrapper nutzen zu können muss FUSE auf dem System installiert sein, sowie das libfuse-dev Paket. Nachdem das Projekt mit make gebaut wurde, kann die Einbindung des Dateisystem vorgenommen werden. Dies erfolgt über die Konsole mit Hilfe von ein paar Befehlen.

```
1 sudo chmod a+rw /dev/loop0
2 dd if=/dev/zero of=image bs=512 count=2048
3 losetup /dev/loop0 image
```

Zeile eins gewährt dem Nutzer Zugriffsrechte auf ein loop-device. Dabei handelt es sich um ein virtuelles Blockgerät, welches als Speicher keine direkte Hardware nutzt, sonder eine Datei. Dies wird genutzt, da der Kernel des open-source nur Dateisysteme einbinden kann, welche ein Blockdevice nutzen. Zeile zwei erstellt ein image, welches in Zeile drei an das loop-device angehängt wird.

```
1 ./lfs --format /dev/loop0
```

Hier wird das Blockdevice formatiert, wobei alle vorhandenen Daten gelöscht werden.

```
1 mkdir mount
2 ./lfs /dev/loop0 mount
```

In Zeile 1 wird ein Ordner mit dem Namen "mountërstellt, welcher in Zeile zwei als Einhängepunkt für littlefs genutzt.

Somit ist littlefs im user-space eingebunden und kann genutzt werden.

3.3 Anforderungen

Da littlefs nun unter Ubuntu eingebunden war, konnte mit der Prüfung der Anforderungen begonnen werden.

Für die IIP existiert ein Excel Dokument, welches die Anforderungen an ein neues Dateisystem beinhaltet. In Zusammenarbeit mit dem Betreuer wurden die für das Projekt wichtigen Punkte identifiziert und die unwichtigen gestrichen.

Nachdem alle Anforderungen überprüft waren, konnte festgestellt werden das littlefs zumindest nach den Anforderungen geeignet erscheint und eine testweise Implementierung in das vorhandene System erfolgen konnte.

3.4 Prototyp - Integration in das System

Nachdem der theoretische Aspekt des Proof of Concept erfolgreich abgeschlossen war, konnte mit einer praktischen Implementierung des Dateisystems in bestehende Projekt begonnen werden. Allerdings sollte dabei lediglich bestätigt werden, ob littlefs unter Integrity läuft.

Dafür sollte das Dateisystem in den Softwarebuild mit aufgenommen werden, sowie eine Überprüfung des Codes mit Hilfe von Klocwork auf die MISRA Anforderungen.

3.4.1 CMAKE & Ninja

Zuerst musste littlefs mit in den Buildprozess eingebunde werden, dafür wurde das komplette Repository in die Subdomäne "filesystem"des "persistence"Pakets mit aufgenommen. Dort liegen auch andere Dateisysteme wie Reliance Edge".

Das Projekt wird mit CMake und Ninja kompiliert, dabei werden die Build-Dateien für Ninja von CMake generiert. Da Ninja performant beim kompilieren von Software ist, wird dies genutzt um die einzelnen Subpakete des Projektes zu kompilieren. Diese werden dann durch CMake zusammen geführt, um so das gesamte Projekt zu bauen.

Da eine bereits vorhandene Build-Struktur im Projekt vorherrscht, mussten die neuen Dateien lediglich in den Build-Prozess mit einbezogen werden. Allerdings existierte keine Dokumentation zu der Struktur von CMake. Dementsprechend mussten die Dateien händisch gesucht und identifiziert werden.

3.4.2 GHS Anpassungen

Bei der Kompilierung des Projekts wird der Compiler von Greenhills verwendet. Dabei fiel auf das es beim Einbinden einiger Header-Dateien zu Komplikationen kam.
Um dieses Problem zu lösen, mussten in den jeweils betroffenen Dateien einer Änderung der Includes vorgenommen werden.

```
1 #ifndef __ghs__
2 #pragma ghs nowarning 193
3 #endif
4 #include <fcntl.h>
5 #include <unistd.h>
```

```
6       #include <errno.h>
7       #ifndef __ghs__
8       #pragma ghs endnowarning
9       #endif
```

Wie drüber stehend zu sehen benötigt der Compiler von Greenhills in diesem Fall defines, welche ihn davon abhalten nach den Header-Dateien zu suchen. Dies hängt damit zusammen, dass der Compiler ansonsten seine eigenen Header-Dateien für die Kompilierung nutzt.

Nach der Anpassung der Includes in allen Dateien, war die Kompilierung des Projekts erfolgreich.

3.4.3 Klocwork

Um den Code auf vorhandene Verstöße gegenüber den MISRA Automotive Richtlinien zu überprüfen, wurde eine Static Code Check Software Namens Klocwork verwendet. Dieses analysiert dabei jede Code Zeile und überprüft diese auf das Einhalten der MISRA Richtlinien, welche gelten um den Code von Automotive embedded Systemen sicherer gegenüber Fehlern zu gestalten.

Da Klocwork normalerweise Teil von CI/CT ist, findet eine Überprüfung der Software normalerweise automatisiert statt, wenn diese auf einem Buildrechner kompiliert wird. Allerdings gibt es auch die Möglichkeit diese lokal auf dem Rechner zu starten.
Dafür wird ein Projekt spezifischer Docker Container, Namens ÏIP-Core"genutzt, welcher die notwendige Konfiguration vorgibt. Dieser muss dafür zuerst gestartet werden, danach kann der nachfolgende Befehl ausgeführt werden:

```
1       iip-core kwshell -pd /srv/workspace/klocwork/IIP_Kilimanjar
```

Damit wird eine grafische Benutzeroberfläche gestartet, in welcher die gefundenen Probleme aufgelistet werden. Die bei littlefs vorhandenen relevanten Verstöße begrenzten sich auf lediglich 8 Fehler, welche im Nachhinein leicht korrigiert werden konnten.

3.5 Prototyp - Testen der Funktionen des Dateisystems

Da alle vorherigen Schritte erfolgreich abgeschlossen worden sind, konnte zuletzt mit dem Testen der Funktionalitäten in der Projektumgebung begonnen werden.

Dafür konnte ein bereits vorhandene Testkomponente mit Namen Apptester verwendet wer-

den.l Diese ermöglicht es Befehle an ein Test-Target zu schicken, welches an einem Testrechner angeschlossen ist.

3.5.1 Anpassen der AppTester Komponente

Zuerst musste die AppTester Komponenten angepasst werden, damit eingehende Befehle die noch zu erstellenden Testfunktionen aufruft.

Dafür muss pro Testfall eine Funktion in der Header-Datei der Komponenten deklariert werden. Außerdem mussten die Aufrufe in der Source-Datein mit eingebunden werden, sowie der Aufruf der eigentlichen Testfunktionen.

```
1   static bool atFsMount(void)
2   {
3           // local variables
4           bool fResult = false;
5
6           // function calls
7           DLT_LOG(atDltContext, DLT_LOG_INFO, DLT_STRING("AT_TC_PERSISTENCY: Mount Filesystem"));
8
9           fResult = littlefs_mount();
10          AtPutResult(fResult == true ? AT_OK : AT_FATAL_ERROR);
11
12          return fResult;
13  }
```

Oben stehend zu sehen ist der Aufruf für das "mounten"von littlefs im User-Space von Integrity. Die DLT_LOG-Funktion erzielt eine Ausgabe auf der Konsole welche Funktion gerade aufgerufen wurde. Danach wird die Testfunktion "littlefs_mount()äufgerufen, welche einen Bool Wert als Rückgabe liefert, um überprüfen zu können ob die Operation erfolgreich war.

3.5.2 Erstellen von Tests

Nachdem die vorhandene Test-Struktur für die neuen Test angepasst wurde, konnten diese erstellt werden. Diese wurden im Scope der File-System Subkomponenten erstellt, da hier bereits existierende Tests für das Reliance Edge Dateisystem liegen.

Bei den Tests sollten lediglich die wichtigsten Funktionen überprüft werden, namentlich: Mount, Unmount, lesen, schreiben, erstellen und löschen von Dateien sowie Ordnern.
Dies sollte dabei zur Verifizierung der wichtigsten Funktionen führen, wodurch eine spätere erfolgreiche Implementierung gewährleistet werden kann.
Für die Tests wird beim Aufruf der Mount-Funktion, welche immer als erstes aufgerufen werden muss, eine Konfiguration des littlefs erstellt. Diese wird auch in den anderen Funktionen genutzt. Zur Vereinfachung der Tests, wurde bei diesen davon ausgegangen, dass Operationen immer nur auf eine einzelne Datei angewendet werden. Dementsprechend wurde ein Handle

für die zuletzt benutzte Datei in der Konfiguration gespeichert, weshalb nicht immer wieder neue Dateien geöffnet werden müssen.

```
bool littlefs_mount(void)
{
        bool result = false;
        int err = 0;

        // block device operations
        cfg.read  = \&lfs_filebd_read ,
        cfg.prog  = \&lfs_filebd_prog ,
        cfg.erase = \&lfs_filebd_erase ,
        cfg.sync  = \&lfs_filebd_sync ,

        // block device configuration
        cfg.read_size = 8,
        cfg.prog_size = 8,
        cfg.block_size = 4096,
        cfg.block_count = 1024,
        cfg.cache_size = 16,
        cfg.lookahead_size = 16,
        cfg.block_cycles = 500,

        lfs.cfg = \&cfg;

        // set the file bd configuration for tests
        cfg.context = (void*) \&filebd_cfg;

        snprintf(lfs_bin_file , LFS_BIN_FILE_SIZE , "%s/%s", RW_FS_MOUNT_POINT, LITTLEFS_BINARY_FILE);
        FSTST_LOG_SIMPLE(FSTST_INFO, 0, lfs_bin_file);

        err = lfs_filebd_create(lfs.cfg, lfs_bin_file);

        if(err != LFS_ERR_OK)
        {
                snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_filebd_create failed , error value: %d", err);
                FSTST_LOG_SIMPLE(FSTST_ERROR, 0, trace_msg_buf);
        }
        else
        {
                // mount the filesystem to a file in an existing read/write filesystem
                err = lfs_mount(\&lfs , \&cfg);

                if (err)
                {
                        snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_mount failed , need to format and mount.");
                        FSTST_LOG_SIMPLE(FSTST_ERROR, 0, trace_msg_buf);

                        err = lfs_format(\&lfs , \&cfg);

                        if(err != LFS_ERR_OK)
                        {
                                snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_format failed , error value: %d", err);
                                FSTST_LOG_SIMPLE(FSTST_ERROR, 0, trace_msg_buf);
                        }
                        else
                        {
                                snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_format successful");
                                FSTST_LOG_SIMPLE(FSTST_INFO, 0, trace_msg_buf);
                        }

                        err = lfs_mount(\&lfs , \&cfg);

                        if(err != LFS_ERR_OK)
                        {
                                snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_mount failed , error value: %d", err);
                                FSTST_LOG_SIMPLE(FSTST_ERROR, 0, trace_msg_buf);
                        }
                        else
                        {
                                snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_mount successful");
                                FSTST_LOG_SIMPLE(FSTST_INFO, 0, trace_msg_buf);
```

```
70                          }
71                  }
72                  else
73                  {
74                          snprintf(trace_msg_buf, TRACE_MSG_BUF_SIZE, "lfs_mount successful");
75                          FSTST_LOG_SIMPLE(FSTST_INFO, 0, trace_msg_buf);
76                  }
77                  result = true;
78          }
79          return result;
80  }
```

! Einzelne Code Abschnitte erklären aus der Mount Funktion als Beispiel

3.5.3 Flashen der Software aufs Target

Nachdem die Tests erstellt worden sind, sowie der Software-Build generiert war, konnte mit dem Testen des Codes auf der eigentlichen Hardware begonnen werden. Dafür wird ein Test-Rechner benötigt, an welchem eine reale Projekt Hardware angeschlossen ist.
Dieser ist über Ethernet, sowie einen CAN-Adapter an die Hardware angeschlossen und ermöglicht es so die Software darauf zu laden und zu starten. Hierfür wird ein Python-Script über die Konsole gestartet, wodurch das Softwareloading auf dem Hardware-Target aktiviert wird. Danach wird die Stromversorgung eingeschaltet und das Target beginnt mit dem Softwareloading, nachdem dieses abgeschlossen wurde, kann dieses neu gestartet werden und die Software ist geladen.

! Grafik vom SWL Prozess

Über das Programm CANoe kann verifiziert werden, dass die Software richtig geladen und das Target gestartet wurde. Dieses fängt die diagnostischen Signale ab, welche von der Hardware auf den Bus versendet werden und zeigt diese in eine grafischen Oberfläche an.

3.5.4 Ausführung der Tests

Um die Tests auszuführen, wird über die Konsole ein Python-Script gestartet, welches noch weitere Parameter beim Aufruf benötigt.

```
1       ./SocketClient.py 030401
```

Hier zu sehen ist der Aufruf der Mount-Funktion.

Dabei haben die Zahlen folgende Bedeutung:

- 03 - Persistence Scope

- 04 - Filesystem/littlefs Subdomain
- 01 - Mount Funktion

! Bild von der Testbench ! Beschreibung des Testprozesses und Rückgabe von der Konsole

3.6 Konklusion Proof of Concept

4 Zusammenfassung und Ausblick

Versicherung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die den benutzten Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen habe ich unter Quellenangabe kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Wetzlar, 30.08.2021