INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# Konzeption und Entwicklung einer generischen Testplattform für Universal-Datenlogger in der Fahrzeugerprobung

Frederik Meerwaldt

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# Konzeption und Entwicklung einer generischen Testplattform für Universal-Datenlogger in der Fahrzeugerprobung

Frederik Meerwaldt

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Dr. Vitalian Danciu
Dipl.-Ing. Simone Ferlin Oliveira

Abgabetermin: 20. Dezember 2011

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 20. Dezember 2011

. . . . . .

*(Unterschrift des Kandidaten)*

In der modernen Fahrzeugerprobung sind Datenlogger, die die Kommunikation von Bussystemen im Fahrzeug überwachen und aufzeichnen nicht mehr wegzudenken. Aufgrund des ständig anwachsenden Funktionsumfangs der Software dieser Datenlogger soll ein automatisiertes Verfahren zum Testen ihrer Funktionsweise entwickelt werden, sodass die erwartete Funktionalität aller Leistungen des Loggers auch in neuen Releases sichergestellt werden kann.

Diese Arbeit befasst sich mit der Analyse der Anforderungen an ein solches Testwerkzeug, sowie der Konzeption der Testumgebung. Hierbei wird eine Hard- und Softwareauswahl getroffen, eine Scriptumgebung für die Testdefinitionen geschaffen, eine flexible Busansteuerung konzipiert, Möglichkeiten zur Störung des Busses simuliert sowie Spannungsverlaufsregelungen und Strommessungen an der Versorgungsseite des Loggers realisiert.

Anschließend wird das Konzept mit entsprechender Hardware in der Programmiersprache C implementiert. Die neue Implementation wird daraufhin anhand eines exemplarischen Tests analysiert und auf ihre Praxistauglichkeit überprüft.

Diese Arbeit wurde bei der BMW Group angefertigt, die einen Arbeitsplatz und alle Teile, sowie professionelle Betreuung und Know-How zur Verfügung stellte. Die Implementierung ist mittlerweile bei der BMW Group, sowie bei dem Loggerhersteller CAETEC im praktischen Einsatz und wird weiterentwickelt und gepflegt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einführung

Die vorliegende Arbeit beschäftigt sich mit dem automatisierten Testen von Datenloggern, Geräte, welche in der Erprobung von Fahrzeugen vor der Serienreife eine große Rolle spielen. Dabei geht es hauptsächlich darum, die Geräte vor ihrem Einsatz in Versuchsfahrzeugen zu überprüfen um das Risiko verloren gegangener Erprobungsresultate, bedingt durch Gerätefehler, zu minimieren.
Um diese Überprüfung durchführen zu können, wird eine Plattform benötigt, welche erweiterbar hinsichtlich der Funktionalität, der Art der verwendeten Datenlogger, sowie der Anzahl der zu testenden Datenlogger ist. Eine solche generische Plattform zur Durchführung von automatisierten Tests an Datenloggern ist, abgesehen von dieser Implementierung, nicht bekannt. In dieser Arbeit wird auf das Konzept und die Implementation einer solchen Plattform eingegangen sowie die praktische Tragfähigkeit überprüft.

## 1.1 Motivation

Bevor ein Fahrzeug in Serie produziert wird und auf dem Markt erhältlich ist, werden Erprobungsfahrten mit Prototypen oder Vorserienmodellen durchgeführt, um Schwächen und Fehler im Fahrzeug zu erkennen und zu beseitigen. Mittlerweile erfolgt in jedem Fahrzeug die Steuerung einzelner Komponenten durch Mikroprozessortechnik, wobei diese Steuergeräte über diverse Bussysteme miteinander vernetzt sind, um Informationen und Daten untereinander auszutauschen. Um diese Daten zu erfassen nutzt man Datenlogger.
Bei Datenloggern handelt es sich um Geräte, die an die verschiedensten Bussysteme im Fahrzeug angeschlossen werden und den Datenverkehr aufzeichnen oder auf bestimmte Ereignisse reagieren. Bei Aufzeichnung des gesamten Datenverkehrs würden Datenmengen entstehen, deren Handhabung stets komplizierter und ineffizienter wird. Daher werden meist nur Daten aufgezeichnet, die für ein bestimmtes, außergewöhnliches Ereignis relevant sind. In der Fachsprache heißen die Prozeduren, die auf ein Ereignis hin ausgeführt werden, *Trigger*. Bei einem solchen Trigger wird der komplette Busverkehr in einem gewissen Zeitraum vor und nach dem Zeitpunkt der Triggerauslösung aufgezeichnet.

Abbildung 1.1: Anzeigeinstrumente im Fahrerblickfeld

Ein Fahrer bekommt zum einen für ihn relevante Messdaten, wie beispielsweise den Status diverser Steuergeräte, Geschwindigkeit oder Temperaturen auf Anzeigedisplays im Blickfeld des Testfahrers übermittelt, vgl. Abbildung 1.1. Dort kann der Testfahrer durch Drücken eines Knopfes einen Trigger auch manuell auslösen, wenn er ein ungewöhnliches Fahrverhalten feststellt. Es besteht ferner die Möglichkeit, dass Datenlogger automatisch auf bestimmte Ereignisse reagieren und Aufzeichnungen starten, beispielsweise wenn ein Steuergeräteausfall vorliegt oder Fahrerassistenzsysteme melden, dass ein Unfall unvermeidbar ist.
Messdaten, wie beispielsweise die Geschwindigkeit, die Motordrehzahl und die aktuelle Position des Fahrzeugs, werden auch kontinuierlich mitgeschrieben. Dies hat den Vorteil, dass man unabhängig von Triggerauslösungen die Testfahrt genauer analysieren kann. Es kann zum Beispiel die Durchschnittsgeschwindigkeit der Fahrt berechnet werden, oder ob der Motor häufig im oberen Drehzahlbereich gelaufen ist, und ob die vorgegebene Fahrtstrecke eingehalten wurde. Die erfassten Messdaten können dann von den entsprechenden Fachabteilungen ausgewertet werden und erlauben somit Rückschlüsse auf das tatsächliche Verhalten der Neuentwicklungen. All diese Möglichkeiten sind in der modernen Fahrzeugerprobung von essentieller Bedeutung und daher sind auch Datenlogger unverzichtbar.
Da die Erprobungszyklen immer kürzer werden und die Anzahl der Erprobungsfahrzeuge steigt, führt dies

zu einem immer häufigeren Einsatz von Datenloggern. Diese Datenlogger stellen eine potentielle Fehlerquelle dar, indem sie die Buskommunikation entweder stören können oder insbesondere nicht die gewünschten Daten aufzeichnen. Weiterhin ist es bedingt durch die erforderliche Funktionsvielfalt der Datenlogger nötig, dass immer mehr Funktionen in den Datenlogger implementiert werden, damit dieser den Ansprüchen der modernen Fahrzeugerprobung gerecht werden kann.
Das Eliminieren des Datenloggers als potentielle Fehlerquelle, sowie das Testen dieser neuen Funktionen macht eine Teststrategie erforderlich. Da es viele Arten von Datenloggern gibt und viele Bussysteme existieren, ist es notwendig, hierfür eine generische Plattform mit einer universellen Teststrategie zu haben. Das heißt, dass man unabhängig von der Art des zu testenden Datenloggers eine Testplattform haben möchte, mit deren Hilfe man verschiedene Datenlogger auf die für die Erprobung wesentlichen Funktionen überprüfen möchte. Wenn neue Funktionen für den Datenlogger implementiert werden, so muss es auch unproblematisch möglich sein, Testroutinen für diese neuen Funktionen in die Testumgebung einzubetten um die korrekte Funktion zu überprüfen.
Es besteht die Möglichkeit, dass durch die Implementierung neuer Funktionen im Datenlogger die Funktionsweise bereits vorhandener und genutzter Funktionen beeinträchtigt oder komplett gestört wird, was zu Problemen bei der Fahrzeugerprobung führt, da Messdaten nicht erfasst wurden und dies nicht zuletzt in wirtschaftlichen Schaden resultiert.
Weiterhin ist es essentiell, dass Datenlogger auch bei nicht vorschriftsmäßiger Handhabung, wie z.B. bei plötzlicher Unterbrechung der Spannungsversorgung, die Daten bis zum Fehlerzeitpunkt verwertbar zugänglich haben und nach Wiederherstellen der Spannungsversorgung wieder ordnungsgemäß ihren Dienst aufnehmen.
Mithilfe der Testplattform ist es möglich neue Softwareversionen der Datenlogger hinsichtlich der Funktionalität der bereits implementierten Funktionen zu überprüfen sowie neue Funktionen vor ihrer Nutzung zu testen, bevor die neue Version automatisiert auf alle Erprobungsfahrzeuge aufgespielt wird.

## 1.2 Aufgabenstellung und Ergebnisse

Die Aufgabenstellung besteht in der Konzeptionierung und praktischen Implementierung einer Testplattform zur automatisierten Überprüfung von Datenloggern hinsichtlich ihrer Funktion und Performance. Überprüfung hinsichtlich ihrer Funktion heißt zu überprüfen, ob der Datenlogger tatsächlich die erwartete und spezifizierte Funktionalität bietet. Unter der Überprüfung hinsichtlich der Performance wird sowohl die Messung der Reaktionszeiten des Datenloggers verstanden, als auch das Verhalten des Datenloggers bei hoher Buslast.
Ein wesentlicher Anspruch an die Testplattform ist die Generizität. Hierbei sollen die im Folgenden genannten Kriterien erfüllt werden.
Zum einen soll die Testplattform unabhängig hinsichtlich der Anzahl und Art der Datenlogger sein, das heißt, es soll möglich sein, mehrere Datenlogger zur gleichen Zeit zu testen, sowie herstellerübergreifend verschiedene Typen von Datenloggern zu testen.
Weiterhin soll die Testplattform unabhängig von Anzahl und Art der Busse sein. Falls ein neuer Bus implementiert werden soll, muss dies machbar sein, ohne die komplette Software umzuschreiben.
Die Testplattform liefert eine Lösung zum automatisierten Testen von Datenloggern, unabhängig von ihrem Typ, Hersteller oder verwendeten Bussystemen.
Als Ergebnis aus dieser Arbeit geht ein Konzept für eine solche Testplattform hervor. Weiterhin wurde dieses Konzept implementiert, wobei die Implementierung bereits erfolgreich bei der BMW Group und dem Loggerhersteller CAETEC im Einsatz ist und weiterentwickelt wird. Diese Implementation hat vier CAN-Busse, simuliert Spannungsverläufe über einen programmierbaren Spannungskonstanter, und besitzt acht Relaisausgänge zum Simulieren von Schaltzuständen und Busunterbrechungen sowie -kurzschlüssen.
Desweiteren ist die Implementation anhand der in Abschnitt 2.1 ausgearbeiteten Kriterien verglichen worden, mit dem Ergebnis, dass alle wesentlichen Anforderungen umgesetzt wurden.

## 1.3 Ziele und Aufbau der Arbeit

Das Ziel der vorliegenden Arbeit ist es, eine Aussage zu treffen, ob und wie es möglich ist, eine Testplattform zur automatisierten Überprüfung von Datenloggern zu konzipieren und zu implementieren. Hierbei sollen die

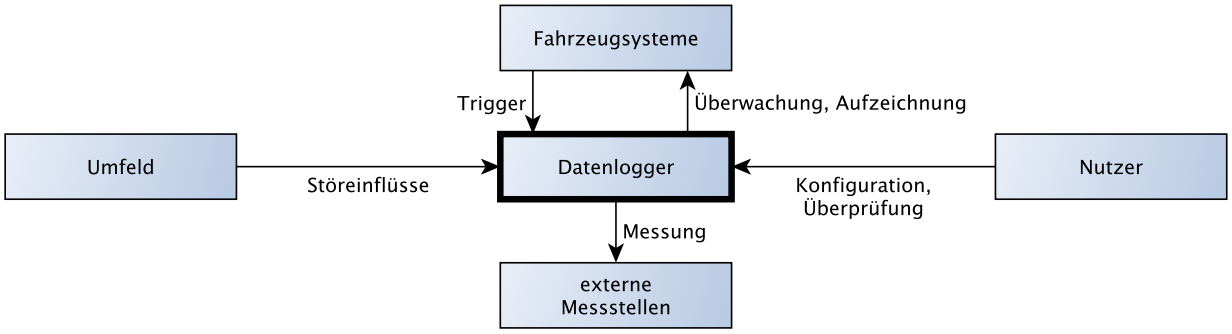


Abbildung 1.2: Überblick über das Einsatzgebiet eines Datenloggers

Kosten und der Aufwand für die Testplattform in einem vertretbaren Rahmen bleiben. Der Datenlogger soll hinsichtlich seiner Funktion und seiner Performance überprüft werden. Weiterhin soll ein Konzept erarbeitet werden, das darstellt, wie die Anforderungen an die Testplattform erreicht werden können. Dieses Konzept soll die Art und Weise darlegen, wie die einzelnen Komponenten der Testplattform zusammenhängen und welche Aufgaben sie in welcher Form erfüllen müssen.
Um über die Tragfähigkeit des Konzeptes zu urteilen wird eine mögliche Implementation erarbeitet und schlussendlich anhand der anfänglich analysierten Kriterien überprüft.

### 1.3.1 Konzept

Ziel der Konzepterstellung ist zu beschreiben, wie die in Abschnitt 2.1 erarbeiteten Anforderungen umgesetzt werden können. Weiterhin wird eine Architektur geschaffen, die die einzelnen Abschnitte der Arbeit in ein Komplettsystem integriert. Die Abbildung 1.2 bietet einen Überblick über das übliche Einsatzgebiet eines Datenloggers. Dabei werden die externen Faktoren, die auf einen Datenlogger einwirken, dargestellt. Diese Faktoren werden im Folgenden näher analysiert, um den Zusammenhang aufzuzeigen und deren Relevanz für die Konzepterstellung zu beleuchten:

- Die *Fahrzeugsysteme* kommunizieren untereinander. Da die Kernaufgabe des Datenloggers das Aufzeichnen und Verarbeiten dieser Busdaten darstellt, ist ein Verfahren für den Zugriff auf die im Fahrzeug verwendeten Kommunikationssysteme notwendig.
- Der *Nutzer* konfiguriert den Datenlogger und überprüft anschließend seine Konfiguration. Bei der Konfiguration bedient er sich Funktionen, welche der Datenlogger bereitstellt. Diese Funktionen müssen automatisiert überprüft werden um die ordnungsgemäße Abarbeitung bereits im Labor sicherzustellen.
- Das *Umfeld* belastet den Datenlogger hinsichtlich Temperaturen, Vibrationen und Leitungsbrüchen bzw. -kurzschlüssen. Es muss eine Möglichkeit geschaffen werden dieses Umfeld zu simulieren.
- Einige Datenlogger stellen auch die Möglichkeit zur Verfügung, Signale *externer Messstellen* auszuwerten, bzw. Signale für externe Geräte zu generieren. Diese Möglichkeit wird als *Analog- and Digital In- and Outputs* (*ADIO*) bezeichnet. Insbesondere existieren:
  - Eingänge für analoge Signale, die Messung von externen Spannung oder externen Widerständen, beispielsweise von Temperaturfühlern.
  - Eingänge für digitale Signale, die Auswertung von externen Schaltzuständen, beispielsweise von zusätzlich angebrachten Schaltern, die zur Validierung der im Fahrzeug fest eingebauten dienen.
  - Ausgänge für digitale Signale, beispielsweise die Ansteuerung von Kontrolllämpchen.
- Da eine Möglichkeit der Beschreibung, welche Tests in welcher Form durchgeführt werden sollen, geschaffen werden muss, ist ein Verfahren für die Steuerung der Plattformkomponenten notwendig.
- Zuletzt wird eine Architektur benötigt, die all diese Punkte vereint.

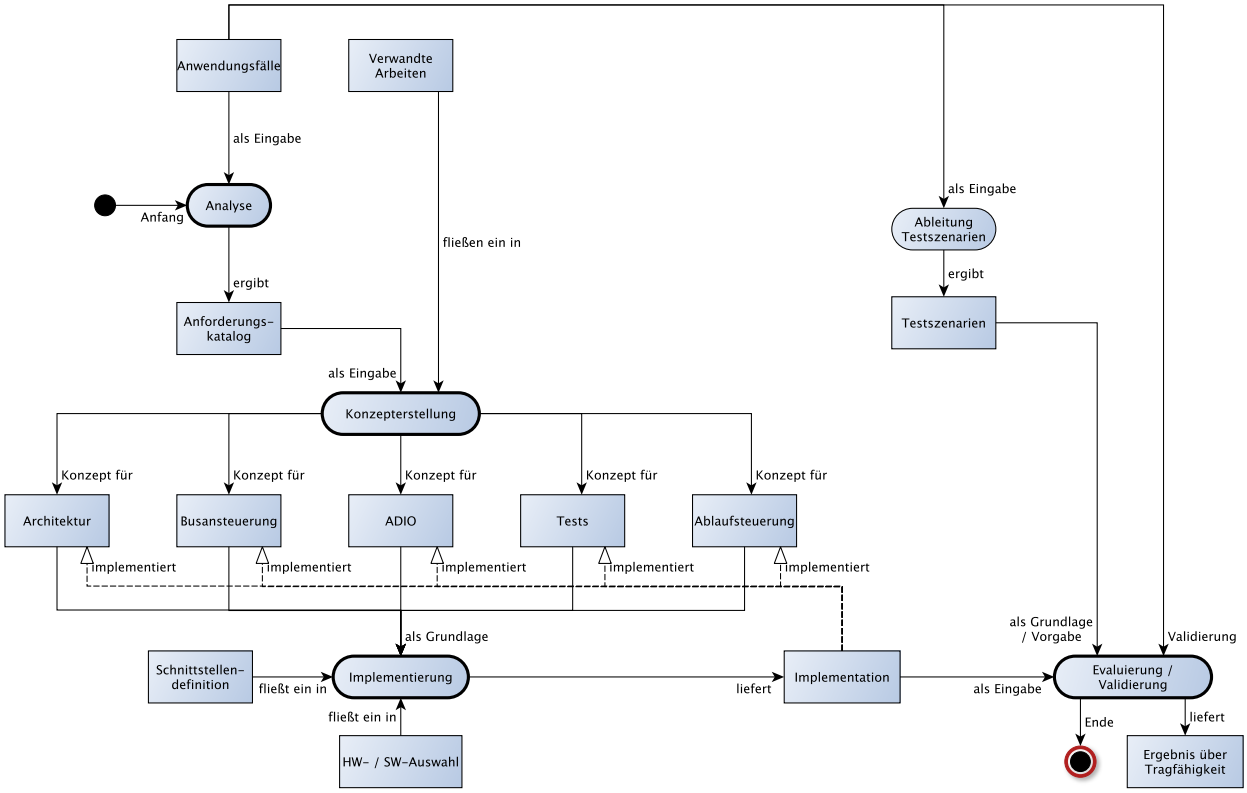


Abbildung 1.3: Vorgehensmodell der Arbeit

### 1.3.2 Tragfähigkeitsnachweis

Ziel des Tragfähigkeitsnachweises ist es, zu erarbeiten, ob eine praktische Umsetzung auf Basis des erarbeiteten Konzepts tatsächlich in dieser Form möglich ist. Ein weiteres Ziel hiervon ist festzustellen, ob sich das Konzept tatsächlich für Performance- und Funktionstests an Datenloggern eignet. Eventuelle Schwierigkeiten und Problemstellungen bei der Implementierung des Konzepts sollen hierbei ebenfalls aufgezeigt werden. Die hier entworfene Architektur wurde im Rahmen dieser Arbeit bei der BMW Group umgesetzt und prototypisch überprüft.

### 1.3.3 Vorgehensmodell und Aufbau

Das Vorgehensmodell in Abbildung 1.3 soll den Aufbau der Arbeit veranschaulichen. Zu Anfang der Arbeit werden in Abschnitt 2.1 Anwendungsfälle analysiert, d.h. es wird analysiert, in welchem Umfeld Datenlogger eingesetzt sind, welche Funktionen erwartet werden und welchen potentiellen Störeinflüssen sie ausgesetzt sind. Hierbei ergeben sich nicht-funktionale sowie funktionale Anforderungen, welche die Basis für die Konzepterstellung bilden. Weiterhin fließen Vorgehensweisen verwandter Arbeiten, die in Abschnitt 3 betrachtet werden, in die Konzepterstellung mit ein.
Die Aufgabe der Busansteuerung, dessen Konzept im Abschnitt 4.5 erläutert wird, ist, dass Tests, welche Buskommunikation erfordern, eine Möglichkeit zur Interaktion mit den Kommunikationssystemen bereitgestellt wird. Dies geschieht unter Berücksichtigung der Anforderung, dass neue Bussysteme, sowie Geräte anderer Hersteller, ohne Änderungen der tatsächlichen Testmodule integrierbar sind.
Die nicht-funktionalen Anforderungen, gepaart mit der Menge der funktionalen Anforderungen ergeben die Aufgabenstellungen an die Architektur, die in den Abschnitten 4.2 und 4.4 erarbeitet werden. Die Ablaufsteuerung definiert im Wesentlichen, wie die Testumgebung zu bedienen sein soll (Usability), wobei hier eine Möglichkeit konzipiert wird, vom Benutzer der Testplattform geschriebene Testabläufe zu interpretieren und

den Inhalt entsprechend auszuführen. Die Ablaufsteuerung stellt auch Komponenten zur Verfügung, welche von allen Teilen verwendet werden können. Damit wird ein Framework für die einzelnen Tests geschaffen.
Die Architektur beschreibt eine Komponentensicht. Sie gliedert die Testplattform funktional in einzelne Komponenten und führt alle Einzelkomponenten zu einer Gesamteinheit zusammen.
Weitere nicht-funktionale Anforderungen bestehen hinsichtlich der Geschwindigkeit des Testsystems sowie einer Kostenanalyse.

Die eigentlichen Testaufgaben sind in Tests, vgl. Abschnitt 4.4, das sind Bausteine die sich in das Gesamtsystem integrieren, ausgelagert. Dies hat den Vorteil, dass die Tests einfach zu pflegen und abstrahiert von der Ablaufsteuerung sind. Ein Modul erweitert die Sprache, in welcher die Testabläufe beschrieben werden, im Folgenden Scriptsprache genannt, durch eigene Parameter sowie möglicherweise eigene Schlüsselwörter.
Die ADIO-Tests sind in dieser Arbeit in Abschnitt 4.7 in Hardware realisiert. Hierbei fließen insbesondere wieder funktionale Anforderungen in die Konzepterstellung ein.
Im Anschluss an die Konzepterstellung wird in Abschnitt 5 eine mögliche und tatsächlich bei der BMW Group realisierten Implementierung vorgestellt. Hierbei wird in Abschnitt 5.1 auf die Anforderungen hinsichtlich Hard- und Software eingegangen und die gesamte Testumgebung mit einem Datenlogger als Testgerät realisiert. Die hier gewonnene Implementierung wird anschließend in Abschnitt 6 evaluiert, wobei ein Testszenario vorgegeben wird, und die Testergebnisse mit den erwarteten Ergebnissen verglichen werden, sowie der Test auch hinsichtlich der anfangs erarbeiteten Anwendungsfälle validiert wird.

## 1.4 Firmenumfeld

Die vorliegende Arbeit wurde bei der BMW Group in der Abteilung EL-432 angefertigt. Wie in Abbildung 1.4[GT 06, S. 12] ersichtlich, geht die Geschichte der Bayerischen Motoren Werke zurück auf das Jahr 1913, in dem die Rapp Motorenwerke GmbH gegründet wurden. Im Juli 1917 wurden diese in die BMW GmbH umfirmiert wobei das Unternehmen ein Jahr später als BMW AG an die Börse ging. [GT 06] Zu Anfang konstruierte die BMW AG Flugzeugmotoren und erst 1923 ihr erstes Motorrad, die R 32, sowie 1929 ihren ersten PKW, den BMW 3/15 PS.[BMW 07]
Die Abteilung EL-432 befasst sich mit der Messtechnik bei der kundennahen Gesamtfahrzeugerprobung, die sich zum einen in die physikalische Messtechnik und zum anderen in die Busanalyse gliedert. Bei der Anfertigung der Arbeit wurde mir jegliche Unterstützung der Abteilung zuteil, sowohl durch kompetente fachliche Unterstützung, als auch durch Unterstützung hinsichtlich benötigter Hardware, Arbeitsplatz, Besprechungsräume, etc.
Hierfür möchte ich mich insbesondere bei den Herren Grill und Taubner bedanken, die mich jederzeit bei meiner Arbeit umfassend unterstützt haben.

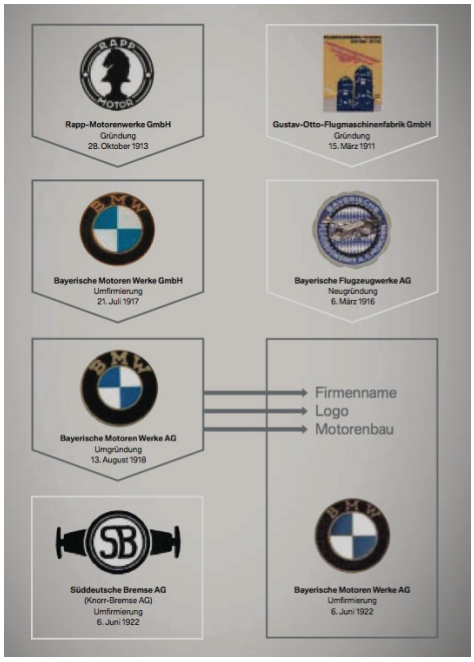


Abbildung 1.4: Entstehung der BMW AG

# 2 Anforderungsanalyse

Um Anforderungen an die Testplattform erarbeiten zu können, wird im Folgenden ein Anwendungsfall analysiert, aus dem daraufhin funktionale, wie auch nicht-funktionale Anforderungen abgeleitet werden.

## 2.1 Analyse eines Anwendungsfalles

Ein Datenlogger zeichnet die Buskommunikation zwischen Steuergeräten auf diversen Bussen im Fahrzeug auf, vergleiche Abschnitt 1.1. Zuerst wird ein Datenlogger im Labor auf korrekte Funktion überprüft, konfiguriert und anschließend im Kofferraum eines Fahrzeuges eingebaut. Daraufhin wird überprüft, ob sich der Datenlogger per WLAN mit dem Rechner, welcher die Daten des Datenloggers abspeichert und Konfigurationen bereitstellt, verbinden kann. Bei jedem Auslesevorgang wird überprüft ob eine aktuellere Konfiguration oder ein Softwareupdate vorliegt und dieses gegebenenfalls eingespielt. Ist die ordnungsgemäße Funktion festgestellt, kann das Erprobungsfahrzeug seinen Dienst aufnehmen. Folgendes Beispiel beschreibt einen typischen Anwendungsfall eines Datenloggers.

Im Sommer steht das Fahrzeug stundenlang in der Sonne, sodass sich der Kofferraum mit dem Datenlogger, vgl. Abbildung 2.1 sehr aufheizt. Ein Datenlogger muss also auch bei +85 °C noch starten und funktionieren können. Das gleiche Auto steht über Nacht in einer Klimakammer, die auf -40 °C herunter gekühlt ist. Dies sind die Spezifikationen des Datenloggers ARCOS des Herstellers CAETEC.

Abbildung 2.1: Datenlogger im Kofferraum verbaut

Der Fahrer steigt morgens ein, startet den Motor und erwartet, dass der Datenlogger funktioniert. Bei diesem Motorstart sinkt die Bordnetzspannung jedoch stark ab, da die kalte Fahrzeugbatterie wenig Leistung liefert und der Anlasser mehr Strom braucht, um den kalten Motor anzudrehen. In manchen Fällen ist die Batterieladung ohnehin nicht mehr so hoch, da einige verbaute Entwicklungssteuergeräte zu viel Strom brauchen oder ständig aktiv bleiben. Wenn es sich um ein Hybrid- oder Elektrofahrzeug handelt, können sich technologiebedingt hochfrequente Störungen in das Bordnetz einkoppeln.
Beim Fahren auch über unwegsames Gelände, schließlich soll evtl. das Fahrwerk erprobt werden, entstehen starke Stöße und Vibrationen, die dem Datenlogger schaden können. Da die Busleitungen im Fußraum des Fahrers angeschlossen werden, passiert es, dass durch eben diese Vibrationen die Leitung an einem Blechteil, welches bei einem Versuchsfahrzeug etwas schärfere Kanten haben kann, scheuert und somit die Busleitungen kurzschließt oder durchtrennt. In beiden Fällen kann der Datenlogger nicht mehr an der Datenkommunikation der Steuergeräte teilhaben. Bei intakter Buskommunikation muss der Datenlogger höhere Buslasten verarbeiten können, da Entwicklungssteuergeräte zur besseren Diagnose mehr Daten auf den Bus senden, als Steuergeräte in Serienreife.
Betrachten wir beispielsweise den Fall, dass bei einer Versuchsfahrt die *Dynamische Stabilitätskontrolle, Fahrdynamikregelung* (*DSC*) ausfällt. Dies registriert der Datenlogger durch:

- Auswertung der vom DSC-Steuergerät gesendeten Statusbotschaft. Ein Steuergerätefehler wird in dieser Statusbotschaft übermittelt.
- Fehler in der Alive-Counter-Botschaft. Dies sind Botschaften, die ein Steuergerät zum Zeichen, dass es aktiv ist, periodisch sendet. Die Botschaften beinhalten einen Zähler, dessen Wert bei jedem Senden der Botschaft um eins erhöht wird. Um die Werte nicht endlos zu erhöhen wird hier nach Erreichen eines Höchstwertes wieder mit dem Niedrigsten angefangen.
  - Ein Fehler liegt vor, wenn die zyklische Alive-Counter-Botschaft nicht mehr vorhanden ist oder das Intervall zwischen zwei Botschaften länger als das vorgegebene Maximalintervall ist.
  - Der Datenlogger registriert einen Fehler, wenn die Alive-Counter-Botschaft einen Fehlerwert enthält (alle Bits konstant 1).
  - Falls eine Lücke im zyklisch versendeten Zähler auftritt, beispielsweise die Folge 0, 1, 3, 4, 5,..., so erkennt der Datenlogger auch einen Fehlerfall.

Durch geeignete Triggerung des Datenloggers soll sichergestellt werden, dass die Fahrzeugbusse bei diesen Fehlerbildern zuverlässig mit Pre- und Posttriggerzeiten aufgezeichnet werden. Ein Trigger ist hierbei ein Konstrukt im Datenlogger, welches den Fehlerfall erkennt und daraufhin die Aufzeichnung auslöst. Bei der Pretriggerzeit handelt es sich um eine Zeitspanne, beispielsweise 20 Sekunden, in denen der Busverkehr vor Auslösung des Triggers mitgeschnitten wird. Dadurch kann analysiert werden, welches Verhalten zu dem Fehlerfall führte. Bei der Posttriggerzeit handelt es sich um eine Zeitspanne, beispielsweise 30 Sekunden, in denen der Busverkehr nach Auslösung des Triggers mitgeschnitten wird. Dies bietet die Möglichkeit, das Verhalten des Fahrzeuges bei bestehender Störung zu analysieren. Bei der Aufzeichnung werden zusätzlich zum Busverkehr auch die Daten von externen, an Analogeingängen angeschlossenen Raddrehzahlsensoren aufgezeichnet, die später mit den vom Steuergerät übersendeten Daten verglichen werden können, um beispielsweise auszuschließen, dass ein Geberdefekt zu dem Fehler geführt hat.

Aus dem Anwendungsfall wird ersichtlich, dass die Herausforderungen an eine automatisierte Überprüfung der Datenlogger sehr hoch sind:

- Flexibilität hinsichtlich des verwendeten Loggertyps und der verwendeten Software-Version.
- Unabhängigkeit hinsichtlich der verwendeten Bussysteme.
- Simulation einer Betriebsumgebung durch Ansteuerung externer Prüfstände wie Klimaschrank, Spannungsquellen, Rüttelprüfstand sowie durch Simulation hoher Buslasten.
- Simulation von Fehlern wie beispielsweise Unterbrechungen und Kurzschlüsse der Bussysteme sowie der Erzeugung von Alive-Counter-Fehlern, beispielsweise durch Aussetzen des Counters.
- Performancemessung des Datenloggers, beispielsweise die Messung der Zeit die der Datenlogger vom Senden einer Botschaft, auf die der Datenlogger reagiert, bis zur Antwort benötigt.

Im Rahmen der Bachelorarbeit soll die Implementierung der Testplattform portabel bleiben. Aufgrund der eingeschränkten Handlichkeit und Verfügbarkeit von Temperaturschränken und Rüttelprüfständen wird daher im Rahmen dieser Arbeit auf die Implementierung dieser beiden Punkte verzichtet. Es ist selbstverständlich denkbar, dies zu einem späteren Zeitpunkt zu realisieren.

Um die zeitnahe Abarbeitung der Daten zu gewährleisten, ist es sinnvoll, dass der Datenlogger auf Ereignisse schnell reagiert und nicht, beispielsweise aufgrund hoher Buslasten, eine zu hohe Verzögerung aufweist. Dazu ist es notwendig, die *Round-Trip-Time* (*RTT*) einer Botschaft zu messen. Da die Nachrichtenverzögerung weitgehend konstant ist, liefert diese Zeit einen guten Anhaltspunkt über die aktuelle Verarbeitungsverzögerung im Datenlogger. Zur Messung soll eine Botschaft vom Testsystem gesendet werden, vom Datenlogger erfasst und mit einer kleinen Änderung wieder zurückgeschickt werden und die Zeitdifferenz zwischen Senden und Empfangen gemessen werden. Wichtig hierbei ist, dass es sich nicht um ein separat zu kalibrierendes System handelt, da diese Kalibrierung einen vermeidbaren Mehraufwand in der Anwendung darstellen würde.

Um eine Testumgebung für Datenlogger bereitzustellen, die einfach konfigurierbar ist und automatisierte Tests durchführt, ist es notwendig, eine Ablaufsteuerung zu implementieren, die den Testablauf anhand der Benut-

zervorgaben durchführt und Testprozesse verwaltet. Für effizienten Betrieb erwünschte Anforderungen für die Ablaufsteuerung sind:

- Erweiterbarkeit hinsichtlich neuer Tests
- Automatisierter Testablauf
- Automatisierte Überprüfung und Auswertung von Testergebnissen

Im Folgenden werden die einzelnen Anforderungen, untergliedert in funktionale und nicht-funktionale Anforderungen, herausgearbeitet und charakterisiert.

## 2.2 Funktionale Anforderungen

Unter funktionalen Anforderungen versteht man solche, aus denen direkte Funktionalitäten des Systems abgeleitet werden können, d.h. die Realisierung einer funktionalen Anforderung resultiert in tatsächlichen Fähigkeiten. Aus dem Anwendungsfall ergeben sich folgende Anforderungen:

**A-1 Senden von Botschaften auf ein Bussystem.** Über die Kommunikationssysteme des Fahrzeugs, die Bussysteme, werden Daten von den Steuergeräten übermittelt. Der Datenlogger empfängt diese Daten und verarbeitet sie indem er auf bestimmte Ereignisse reagiert oder die Daten aufzeichnet. Um dies simulieren zu können ist es notwendig, dass die Testplattform Botschaften auf ein Bussystem senden kann.

**A-2 Empfangen von Botschaften von einem Bussystem.** Die einzige Möglichkeit der Kommunikation zwischen dem Datenlogger und der Testplattform besteht in der Nutzung eines gemeinsamen Bussystems. Das der Datenlogger auf bestimmte simulierte Ereignisse reagieren kann und diese Reaktionen von der Testplattform ausgewertet werden können, ist es notwendig dass die Testplattform Botschaften von einem Bussystem empfangen kann.

**A-3 Nebenläufige Nutzung der Datenbusse.** In einem Testablauf können mehrere Tests parallel laufen, welche alle mit dem Datenlogger über ein gemeinsames Bussystem kommunizieren. Aus diesem Grund ist es wichtig, dass eine nebenläufige Nutzung des Netzzugangs zu den Datenbussen möglich ist, und dieser nicht nur exklusiv von einem Test genutzt werden kann.

**A-4 Simulation von Spannungsverläufen.** Wie in Abschnitt 2.1 beschrieben, bricht die Bordnetzspannung beispielsweise bei einem Motorstart ein. Dies stellt eine Herausforderung an den Datenlogger dar, welche auch mit der Testplattform simuliert werden soll. Daher ist es wichtig, dass eine Möglichkeit geschaffen wird die Spannungsversorgung des Datenloggers zu beeinflussen. Hierbei sind schnelle Reaktionszeiten in der Spannungsversorgung wichtig, wie in Abbildung 2.2 ersichtlich ist, da sich der Spannungsverlauf bei einem Motorstart nach DIN 40839 im ms-Bereich verändert.

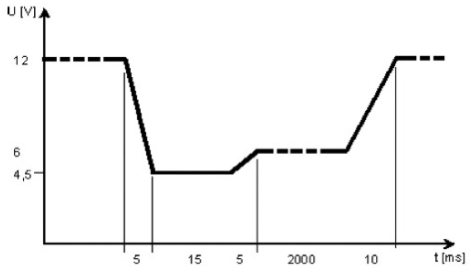


Abbildung 2.2: Spannungsverlauf nach DIN 40839

**A-5 Gemeinsame Nutzung von Spannungsverlaufsdefinitionen.** Es ist denkbar, dass verschiedene Testszenarien immer den gleichen Spannungsverlauf, wie beispielsweise den Motorstart nach DIN 40839 wie in Abbildung 2.2 dargestellt, nutzen. Um Redundanz in den Testbeschreibungen zu vermeiden, ist es wichtig, dass Spannungsverläufe einmal definiert werden und dann in unterschiedlichen Testabläufen genutzt werden können.

**A-6 Simulation des Batteriehauptschalters.** Ein Datenlogger bezieht seine Stromversorgung über das Bordnetz des Fahrzeuges. Bei Versuchsfahrzeugen existiert häufig ein Batteriehauptschalter, welcher das Bordnetz komplett von der Batterie trennt. In der Fachsprache wird dieser dauerhafte Spannungsversorgungsanschluss als *Klemme 30 (KL30)* bezeichnet. Diese KL30 soll durch die Testplattform simulierbar sein.

**A-7 Simulation des Zündungszustands.** Ein Datenlogger bekommt eine zusätzliche Signalleitung, auf welcher nur dann Spannung anliegt, wenn die Zündung des Fahrzeugs eingeschaltet ist. Diese Spannungsversorgung bei Zündung-ein wird in der Fachsprache als *Klemme 15 (KL15)* bezeichnet. Wenn auf KL15 Spannung anliegt, ist dies meist ein Startkriterium für den Datenlogger. Diese KL15 soll durch die Testplattform simulierbar sein.

**A-8 Simulation von Busunterbrechungen.** Wie in dem Anwendungsbeispiel in Abschnitt 2.1 beschrieben, besteht die Möglichkeit, dass Busleitungen, beispielsweise aufgrund scharfer Metallkanten, durchtrennt werden und daher die Buskommunikation zum Datenlogger unterbrochen ist. Diese Busunterbrechung soll durch die Testplattform simulierbar sein.

**A-9 Simulation von Buskurzschlüssen.** Wenn eine scharfe Metallkante die Isolierung der Busleitung durchtrennt und die Leiter miteinander verbindet entsteht ein Kurzschluss auf dem Bus. Dieser Kurzschluss soll durch die Testplattform simulierbar sein.

**A-10 Automatisches Testen der ADIO.** Die analogen und digitalen Ein- und Ausgänge des Datenloggers sollen automatisiert durch die Testplattform auf ihre korrekte Funktion und Reaktionsgeschwindigkeit getestet werden können. Hierzu sollen analogen und digital Ein- und Ausgänge mit Spannungen beschaltet werden, sowie der Schaltzustand der digitalen Ausgänge angezeigt bzw. ausgewertet werden.

**A-11 Simulation von Alive-Countern und deren Fehlerfälle.** Steuergeräte nutzen, wie in Abschnitt 2.1 beschrieben, Alive-Counter um ihre Verfügbarkeit anzuzeigen. Zur Simulation dieser Steuergeräteverfügbarkeit soll die Testplattform einen solchen Alive-Counter simulieren können. Da ein Datenlogger einen Steuergeräteausfall erkennen soll, müssen auch Fehlerfälle des Alive-Counters simuliert werden. Hierzu zählen Botschaftsduplikate, bei denen der Zähler gleich bleibt und verloren gegangene Botschaften, bei denen der Zähler nicht um eins, sondern um zwei im Vergleich zur vorhergehenden Nachricht erhöht ist. Denkbar ist auch, dass ein kurzzeitiger Ausfall eines Steuergerätes simuliert wird, indem das Senden des Alive-Counters für eine bestimmte Zeit eingestellt wird.

**A-12 Erzeugung von Buslast, auch mit definierten Botschaften.** Da die Buslast, wie in Abschnitt 2.1 beschrieben, insbesondere bei Entwicklungssteuergeräten sehr hoch sein kann, ist es möglich, dass die Verarbeitungsgeschwindigkeit oder gar die Funktionalität des Datenloggers mit zunehmender Buslast schlechter wird. Dies soll die Testplattform durch Senden von Botschaften, auf die der Datenlogger nicht reagieren braucht, simulieren. Es soll weiterhin die Möglichkeit bestehen die Botschaften, die zur Erzeugung von Buslast gesendet werden, im Testablauf vorzugeben oder zufällig erzeugen zu lassen.

**A-13 RTT-Messung.** Um die Performance des Datenlogger bestimmen zu können ist es notwendig, die RTT einer Botschaft zu messen. Interessant wird diese RTT Messung insbesondere in der Kombination mit Buslasterzeugung.

## 2.3 Nicht-funktionale Anforderungen

Unter nicht-funktionalen Anforderungen versteht man solche Anforderungen, die für die ordnungsgemäße Funktion der Gesamtplattform notwendig sind, jedoch keine unmittelbare Fähigkeit beschreiben. Hierbei ergeben sich folgende Anforderungen:

**A-14 Eingabemöglichkeit zur Beschreibung des Testablaufs.** Der Nutzer der Testplattform muss den Ablauf des Tests vorgeben können. Aus diesem Grund muss eine Eingabemöglichkeit zur Testplattform bereitgestellt werden, die der Nutzer verwendet, um seinen Testablauf in einer Form darzustellen, die die Testplattform verarbeiten kann.

**A-15 Protokollierung des Testfortschritts und der Testergebnisse.** Um eine Dokumentation über den Testverlauf zu haben, ist es notwendig, dass alle laufenden Tests in einer Protokolldatei protokolliert werden. Hierzu sollen zum einen die Testschritte dargestellt werden und zum anderen die Ergebnisse der jeweiligen Testschritte protokolliert werden.

**A-16 Geschwindigkeit der Testplattform.** Ein Datenlogger soll mithilfe der in dieser Arbeit entworfenen Testplattform auch hinsichtlich seiner Performance untersucht werden. Hierfür ist es notwendig, dass die Geschwindigkeit der Testplattform wesentlich höher ist als die Geschwindigkeit des Datenloggers, sodass bei Performancetests der Datenlogger oder die verwendeten Bussysteme den Flaschenhals darstellen, nicht aber die Testplattform selber.

**A-17 Kosten der Nutzung.** Die Kosten der Nutzung der Testplattform müssen in einer Relation zu den Kosten eines Datenloggers stehen. Ein Datenlogger kostet in der gängigen Ausbaustufe ca. € 10.500, wobei bei diesem Betrag nur der Datenlogger alleine, ohne evtl. notwendiger Peripherie wie externe Messmodule, gerechnet ist. Nachdem der Datenlogger allerdings ein essentieller Baustein in der Fahrzeugerprobung ist, sind die anfallenden Kosten bei einem Ausfall nicht pauschal zu definieren, und variieren je nach verloren gegangenem Erkenntnisgewinn bei einer Erprobungsfahrt. Die realisierte Ausbaustufe der Testplattform hat einen Wert von ca. € 3.500 und liegt damit deutlich unter dem Preis eines einzigen Datenloggers. Bei 100 Erprobungsfahrzeugen, welche mit Datenloggern ausgestattet sind, ist dies ein verschwindend geringer Betrag.

**A-18 Automatisierter Testablauf.** Ein Test soll ohne manuelle Interaktion ablaufen. Somit sind auch Testszenarien denkbar, die eine längere Laufzeit erfordern, da keine kostbare Arbeitszeit auf Überwachung oder Interaktion mit der Testplattform aufgewendet wird.
Weiterhin ist es erforderlich, dass die Testplattform immer über den aktuellen Testfortschritt und -zustand weiß, um Entscheidungen basierend auf dem aktuellen Testverlauf treffen zu können.
Um den automatisierten Testablauf zu gewährleisten ist es notwendig, dass Testergebnisse von der Testplattform selbst automatisch überprüft und ausgewertet werden.

**A-19 Unabhängigkeit hinsichtlich der Anzahl der Busse und dessen Typ.** Datenlogger unterstützen meist mehrere Kanäle verschiedener Bussysteme, was eine Skalierbarkeit der Testplattform erfordert. Diese Erweiterung bezieht sich zum einen auf weitere Bustypen, wobei darauf zu achten ist, dass die Funktionstests unabhängig vom verwendeten Bussystem sind. Zum anderen müssen mehrere Busse desselben Typs möglich sein, um die beim Datenlogger vorhandenen Kanäle auch mit unterschiedlichen Daten beschicken zu können.

**A-20 Unabhängigkeit von der verwendeten Hardwareschnittstelle.** Geräte, welche die Anbindung an das einzelne Bussystem realisieren, sogenannte Businterfaces, werden von unterschiedlichen Herstellern mit unterschiedlichen Leistungsmerkmalen produziert. Sollte sich herausstellen, dass ein anderer Hersteller Leistungsmerkmale bietet, welche für die Testplattform ideal sind, müssen Interfaces dieses Herstellers problemlos angebunden werden können.

**A-21 Erweiterbarkeit hinsichtlich neuer Tests.** Der Funktionsumfang von Datenloggern passt sich ständig den Anforderungen der Fahrzeugerprobung an und nimmt daher stark zu. Dieser Funktionszunahme muss mit einer Erweiterung des Testrepertoires begegnet werden. Aus diesem Grund ist besteht die Möglichkeit, die Testplattform einfach durch neue Tests zu erweitern.

**A-22 Mehrere Datenlogger gleichzeitig testbar.** Aus Gründen der Zeitersparnis, soll es möglich sein, mehrere Datenlogger zur gleichen Zeit zu testen.

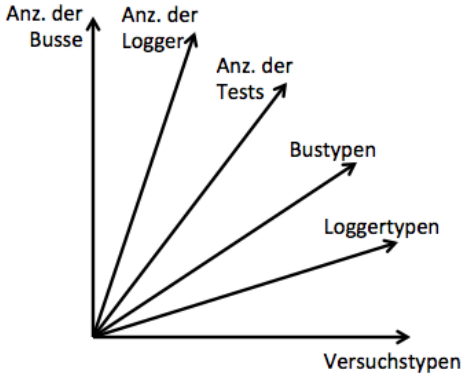


Abbildung 2.3: Generizität der Testplattform

Wie in Abbildung 2.3 ersichtlich, erfordert eine generische Testplattform, dass diese in sechs Dimensionen erweiterbar ist:

1. Anzahl der Busse: Es soll die Möglichkeit bestehen, mehrere Kanäle desselben Bustyps anzusteuern um beispielsweise unterschiedliche Kanäle des Datenloggers mit unterschiedlichen Daten zu beschicken.
2. Anzahl der Logger: Aus Effizienzgründen sollen mehrere Datenlogger gleichzeitig getestet werden können.
3. Anzahl der Tests: Es sollen mehrere Tests parallel ablaufen können, da ein Datenlogger in der eigentlichen Anwendung auch von mehreren Steuergeräten parallel Daten erhalten kann, die er im gleichen engen Zeitfenster verarbeiten muss.
4. Bustypen: Die Testplattform muss die Möglichkeit geben, zusätzliche Bustypen zu unterstützen. Somit ist eine Erweiterung auf weitere, derzeit bereits im Einsatz befindliche Typen möglich. Zusätzlich kann die Testplattform durch Erfüllung dieser Anforderung auch dem technologischen Fortschritt folgen und zukünftige Bussysteme unterstützen.
5. Loggertypen: Unterschiedliche Hersteller produzieren unterschiedliche Datenlogger. Die Testplattform darf nicht auf Datenlogger eines bestimmten Herstellers beschränkt sein, sondern muss generisch alle Datenlogger unterstützen.
6. Versuchstypen: Eine Erweiterbarkeit der Testplattform hinsichtlich der unterstützten Tests muss gegeben sein. Somit können in späteren Phasen mehr Funktionalitäten des Datenloggers automatisiert überprüft werden. Desweiteren kann die Testplattform hinsichtlich neuer Loggerfunktionen erweitert werden, sodass diese auch überprüft werden können.

# 3 Stand der Technik

Im Folgenden soll auf verwandte Methodiken, Testverfahren und Arbeiten eingegangen werden. Hierzu wird zum einen gezeigt, wie der Markt von Datenloggern aussieht. Zum anderen werden zwei große Testverfahren, das *Hardware-in-the-Loop* (*HiL*)-Testverfahren und das *Software-in-the-Loop* (*SiL*)-Testverfahren beleuchtet, auf die im weiteren Verlauf dieses Kapitels genauer eingegangen wird. Anschließend wird ein HiL-Test aus einer anderen Arbeit, der Test eines Motor- und Getriebesteuergeräts, betrachtet und mit der vorliegenden Arbeit verglichen.

## 3.1 Unterscheidungsmerkmale von Datenloggern

Datenlogger, die in der Fahrzeugerprobung eingesetzt werden, sind ein Nischenprodukt und als solches ist die Anbieterzahl dieser Produkte sehr beschränkt. Es gibt unterschiedliche Produkte, eine Vielfalt der vorhandenen Datenloggern ist auf Abbildung 3.1 zu sehen. Alle Datenlogger bewältigen im Wesentlichen das gleiche Aufgabenspektrum. Die Unterscheidung besteht hauptsächlich in der Geschwindigkeit, der Konfigurierbarkeit, der Anzahl und Arten verwendeter Bussysteme sowie der verwendeten Steckverbinder.
Die Unterscheidung hinsichtlich der Geschwindigkeit bezieht sich auf die Verarbeitungsverzögerung, welche mit Hilfe der durch diese Arbeit implementierte Testplattform gemessen werden kann.
Die unterschiedliche Konfigurierbarkeit bezieht sich auf unterschiedliche Scriptsprachen und Funktionsumfänge. Manche Datenlogger bieten Funktionen, wie das Erkennen von Alive-Countern direkt an, bei manchen muss die Funktion in einer eigenen Scriptsprache dazugeschrieben werden.
Weiterhin bieten unterschiedliche Datenlogger unterschiedliche Auslesemethoden, welche von WLAN, über LAN, bis hin zum USB-Stick oder Speicherkarten reichen. Die Anzahl und Arten der Bussysteme variieren je nach Hersteller und Modell. Betrachtet man das *Open Systems Interconnection* (*OSI*) Referenzmodell, hier insbesondere die Schicht 1, die Bitübertragungsschicht, so stellt man fest, dass die verwendeten Steckverbinder auch je nach Hersteller variieren. Die Auswahl der passenden Steckverbinder ist in der Fahrzeugtechnik von großer Bedeutung, da es dort besonders auf beschränkte Platzverhältnisse und Vibrationen ankommt. Um allerdings seinen Datenlogger für jeden Fahrzeughersteller attraktiv zu machen, werden oft spezifische Gerätemodifikationen seitens des Loggerherstellers vorgenommen, indem die Steckverbinder den beim Fahrzeughersteller verwendeten Steckverbindern angepasst werden.
Nach bestem Wissen und Gewissen ist zum Zeitpunkt des Schreibens keine andere Testumgebung bekannt, die sich explizit mit dem Test von Datenloggern beschäftigt hat, sodass hier keine anderen Arbeiten verglichen werden können.

## 3.2 Unterscheidung in HiL- und SiL-Testverfahren

Zum Testen von Hard- und Software haben sich im Wesentlichen zwei Verfahren etabliert. Zum einen handelt es sich um die SiL-Testverfahren, zum anderen handelt es sich um HiL-Testverfahren.
Bei SiL-Testverfahren werden Funktionen in der Software, deren Funktionalität man nicht testen möchte, oder die noch gar nicht implementiert sind aber im Programm eine ausführende Rolle einnehmen, durch Testroutinen ersetzt, welche die vom Programm generierten Daten erfassen, prüfen und entsprechend schlüssige Daten zurückgeben [Bäk 06, S. 206]. Wenn das vorliegende Testszenario in SiL implementiert worden wäre, dann hätte es sich hier angeboten, die Logger-Software dergestalt zu verändern, dass man die Funktionen zum Senden und Empfangen von Botschaften auf dem Bus durch ein Interface an das Testsystem implementiert. Dadurch würde das Testsystem die Botschaften, die auf den Bus gesendet werden, direkt per Softwareinterface erhalten, und den Inhalt entsprechend überprüfen. Das Testsystem würde auch die Empfangsroutinen der

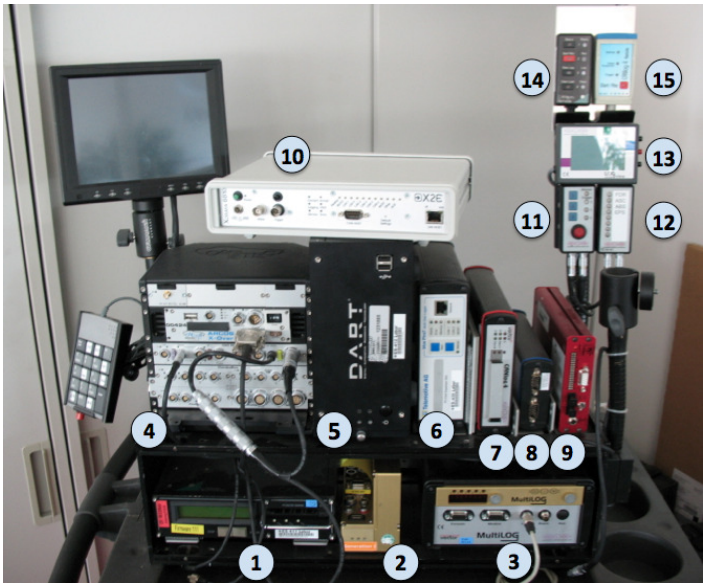


Abbildung 3.1: Datenlogger verschiedener Hersteller
1: CANway CANguru, 2: IPETRONIK M-LOG, 3: GiN MultiLog, 4: CAETEC ARCOS, 5: DART,
6: Telemotive bluePirate, 7: GiN CANlog 4, 8: IXXAT CANcorder,
9: Silicon Systems OPTOLYZER 4 MOST, 10: XORAYA 6810, Fernbedienungen: 11: GiN Remote,
12: GiN LEDs, 13: GiN LogView, 14: CANguru Remote, 15: CANLog 4 Remote

Loggersoftware entsprechend adaptieren, sodass Busdaten generiert und über die Softwareschnittstelle an die Loggersoftware weitergereicht werden können. Dieses SiL-Verfahren hat den Vorteil, dass keine zusätzliche Hardware erforderlich ist, und beispielsweise bei Linux-basierten Datenloggern recht einfach realisiert werden könnte. Allerdings beschränkt man sich hier auf einen Loggertypus, womit die Flexibilität hinsichtlich der Art der Datenlogger, eine der Anforderungen aus Abschnitt 2.3, nicht erfüllt wäre. Außerdem soll der gesamte Datenlogger, sowohl die Hardware, als auch die Software, getestet werden.
Bei Datenloggern, welche eine komplett eigene Firmware mit der Loggersoftware enthalten, ist eine solche Adaption beliebig komplizierter. Der Nachteil von SiL-Testverfahren ist, dass man nur ein Teil des Gesamtsystems testen kann. Man geht davon aus, dass die Interaktion zwischen dem Softwareinterface zur Busansteuerung und den tatsächlichen Busdaten einwandfrei funktioniert, da dies nicht getestet wird. Nachdem in den vorliegenden Testszenarien Hardware auch eine wesentliche Rolle spielt, beispielsweise die Simulation von Spannungsverläufen sowie physikalische Störeinflüsse auf das Bussystem, sind SiL-Testverfahren den HiL-Testverfahren unterlegen. Weiterhin sind die Businterfaces in den Datenloggern auch mit einer eigenen Firmware bestückt, da sie meist über programmierbare Bausteine wie Mikrocontroller oder FPGAs realisiert sind, die bei SiL-Tests nicht getestet werden würden.
Bei HiL-Tests wird die komplette unveränderte Hardware in der Form, wie sie auch im Anwendungszweck eingesetzt wird, in die Testumgebung eingebunden [Bäk 06, S. 204]. So werden im vorliegenden Fall die Busdaten komplett auf einen tatsächlich existierenden Bus gesendet und das Bussystem, wie im Fahrzeug, mit dem Datenlogger verbunden.
Der Nachteil hierbei ist, dass das System beliebig aufwändig wird, da jedes zu testende Element nachgestellt werden muss. Weiterhin können potentielle Fehlerquellen auftreten, hier in Form von Businterfaces, die die Testplattform nutzt und mit einer entsprechenden, vom Hersteller gelieferten API verwendet werden. Da das Testsystem hierauf keinen Einfluss hat, könnte es vorkommen, dass die Testumgebung einen Fehler detektiert, der allerdings durch das Testsystem selbst induziert ist. Der Vorteil ist, dass tatsächliche Hardwarestörungen problemlos simuliert werden können und dass das *Device under Test* (*DUT*) als Ganzes getestet werden kann, und jegliche Art von auftretenden Fehlern simuliert werden kann [SN 04, S. 4].
HiL-Tests unterscheiden sich in Open-Loop- und Closed-Loop-Tests wobei bei Open-Loop-Tests nur Steuerdaten, aber keine Reaktionen simuliert werden und bei Closed-Loop-Tests beides simuliert wird. Der Unterschied wird in Abbildung 3.2 deutlich. Gezeigt ist hier ein HiL-Closed-Loop-Test eines Motor- und Getriebesteuergerätes, welches in Abschnitt 3.3 mit der Testplattform für Datenlogger verglichen wird. Würde man

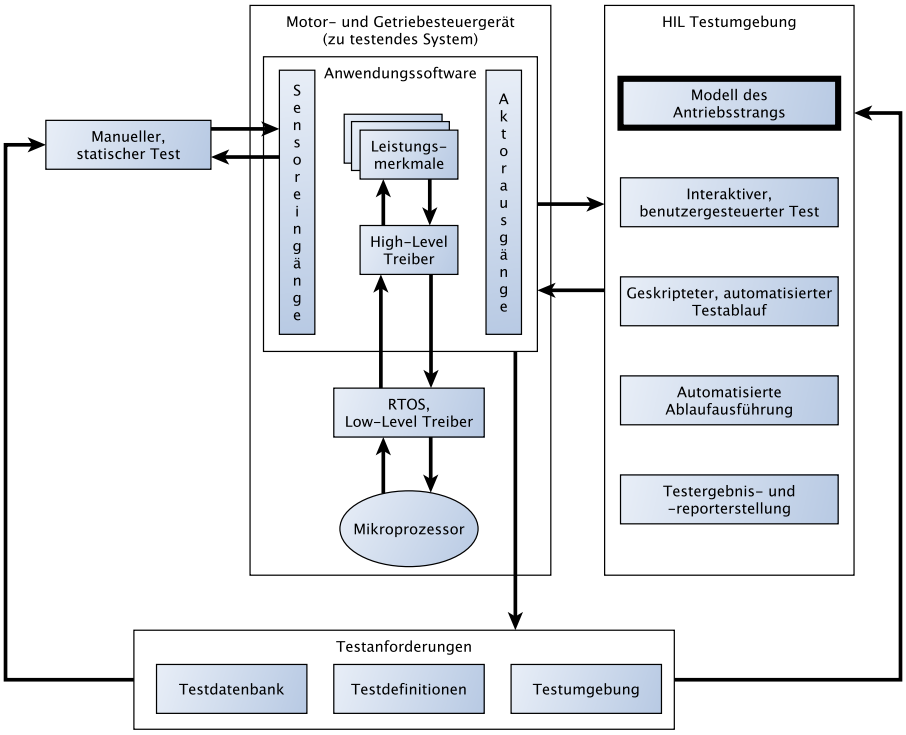


Abbildung 3.2: HIL-Closed-Loop-Test
[SN 04, nach Fig. 10]

das dick umrandete Feld *Modell des Antriebsstrangs* weglassen, so würde es sich um einen Open-Loop-Test handeln, da man das Modell der Reaktionen des Motors bzw. des Getriebes entfernt.
Diese beiden Testverfahren lassen sich am Besten anhand eines Steuergerätes im Fahrzeug beschreiben: Ein Steuergerät für die dynamische Dämpferkontrolle regelt die Fahrwerkscharakteristik je nach eingestelltem Fahrprogramm. Nun ist eine Aufgabe dieses Steuergerätes Daten von Dämpfersensoren auszuwerten um den momentanen Dämpferzustand zu beurteilen und die Dämpfer entsprechend stärker oder weicher abzustimmen. Bei einem HiL-Open-Loop-Test werden nun dem Steuergerät Daten über mögliche Dämpferzustände übermittelt. Gleichzeitig werden die Ausgabewerte vom Steuergerät an die Dämpfer überprüft, wobei überprüft wird, ob die Werte den Erwartungen entsprechen.
Wenn nun ein HiL-Closed-Loop-Test gefahren wird, werden die Ausgabewerte vom Steuergerät an die Dämpfer vom Testsystem interpretiert und die simulierten Sensordaten entsprechend den neuen, vom Steuergerät errechneten Dämpfereinstellungen angepasst. Somit wird die Reaktion, welche die vom Steuergerät errechneten Daten auslösen, mit simuliert. Da sich ein Datenlogger busseitig möglichst transparent verhalten soll, da die Busdaten ausschließlich aufgezeichnet und überprüft werden sollen, ist ein HiL-Closed-Loop-System unmöglich, da es keine Reaktionen auf Busdaten gibt, die zurück auf die Fahrzeugbusse gesendet werden können. Vorliegendes Konzept und Implementierung bezieht sich also auf einen HiL-Open-Loop-Test.

## 3.3 HiL-Testverfahren in anderen Arbeiten

HiL-Testverfahren gehen in die 50er Jahre zurück, wo sie hauptsächlich in der Abwehr sowie Luft- und Raumfahrttechnik eingesetzt wurden. Dieses automatisierte Testverfahren diente seinerzeit hauptsächlich dazu, die Gesundheit der am Test beteiligten Personen zu schützen, indem keine Menschen im Testumfeld anwesend sein müssen. Aufgrund der anfänglich hohen Kosten für die Implementierung von HiL-Systemen haben sich diese erst durch die zunehmende Komplexität von Fahrzeugsystemen in den 90er Jahren im Automobilbereich durchsetzen können. [SN 04, S. 3]

Die Firma dSPACE, ein großer Konzern für die Entwicklung von HiL-Systemen, hat ein HiL-System zum Test eines *Powertrain Control Module (= Motor- und Getriebesteuergerät)* (*PCM*) entwickelt, und stellt folgende Anforderungen an dieses System: [SN 04, S. 5]

1. Kontrolle über den Testablauf und die Möglichkeit von Regressionstests, die Anforderung ist in dieser Arbeit in Abschnitt 2.3 als Punk *A-18* beschrieben.
2. Injizierung von Prüfreizen in Echtzeit, diese Anforderung teilt sich in dem Anforderungskatalog auf. Die Prüfreize sind in Abschnitt 2.2 aufgelistet. Die automatisierte Injizierung in Echtzeit ist in dieser Arbeit in Abschnitt 2.3 als Punkt *A-18* beschrieben.
3. Zugriff auf die Modellparameter und -daten sowie Kontrolle über den Zustand der Simulation, die Anforderung ist in dieser Arbeit in Abschnitt 2.3 in der Anforderung *A-18* beschrieben.
4. Zugriff auf externe Geräte (z.B. Testgeräte, Voltmeter, etc.), die Anforderung ist in dieser Arbeit in Abschnitt 2.2 als Punkt *A-4* bis *A-10* beschrieben.
5. Kontrolle und Zugriff auf das zu testende Gerät (Diagnosewerkzeuge), diese Anforderung entfällt aus unten genannten Gründen.
6. Die Möglichkeit Testdokumentation zu erzeugen und Testdaten abzuspeichern, diese Anforderung ist unter Abschnitt 2.3 als Punkt *A-15* beschrieben.
7. Analyse der Testdaten während des Tests, diese Anforderung ist unter Abschnitt 2.3 als Punkt *A-18* beschrieben

Im Folgenden werden die von dSPACE geforderten Punkte mit dieser Arbeit verglichen und diskutiert.

**Kontrolle über den Testablauf und die Möglichkeit von Regressionstests** Über die in dieser Arbeit beschriebene Ablaufsteuerung wird die erste Anforderung erfüllt, vgl. Abschnitt 4.2. Sie gibt die Möglichkeit, über Ablaufscripte den Testablauf zu steuern, sowie über Hilfsmittel wie Schleifen in Ablaufscripts oder wiederholter Aufrufe von Testmodulen Regressionstests zu durchlaufen. Regressionstests überprüfen wiederholtermaßen Testszenarien, welche in vorhergehenden Produktivversionen bereits erfolgreich durchlaufen wurden. Der Sinn von Regressionstests ist unter anderem die Vermeidung ungewollter Veränderungen an der Software, [MP 07, S. 418] sowie die Vermeidung sporadisch auftretender Fehler durch ständige Wiederholung der Tests. Ein weiteres Problem, welchem mit Hilfe von Regressionstests begegnet werden kann ist, dass ein Test, welcher einmal korrekt ausgeführt und damit bestanden wurde, noch lange nicht bei jedem Aufruf unter jeder Bedingung korrekt ausgeführt wird [MP 07, S. 4]. Wenn ein Test in ständiger Wiederholung ständig erfolgreich war, so wird davon ausgegangen dass er auch zuverlässig funktioniert, da 100% Zuverlässigkeit nicht von Korrektheit unterscheidbar ist [MP 07, S. 44].

**Injizierung von Prüfreizen** Der zweite Punkt, die Injizierung von Prüfreizen, wird in dieser Arbeit hauptsächlich über die Busansteuerung sichergestellt, vgl. Abschnitt 4.5. Hierüber werden Daten an das DUT übertragen, welche eine zu überprüfende Reaktion des Datenloggers auslösen sollen. Über die mögliche individuelle Parametrierung der Module werden Modellparameter und -daten gesetzt.

**Zugriff auf die Modellparameter und -daten, Kontrolle über den Zustand der Simulation** Über den prozessverwaltenden Teil der Ablaufsteuerung, vgl. Abschnitt 4.2 ist das Ablaufscript in der Lage, den Zustand der Simulation zu steuern, womit der dritte Punkt erfüllt wird.

**Zugriff auf externe Geräte** Wesentliche, in dieser Arbeit auch konzeptionierte und implementierte Module, sind die Zugriffsmodule auf externe Geräte, um den vierten Punkt zu erfüllen. In dieser Arbeit wird ein externes Relaismodul, vgl. Abschnitt 4.4.2, sowie die Spannungsversorgung, vgl. Abschnitt 4.4.1 angesteuert. Die Spannungsversorgung kann zum einen mit Werten wie Spannungsverlaufssequenzen, Spannungseinstellungen und Strombegrenzungen parametriert werden und zum anderen auch aktuelle Messwerte, wie den aktuellen Stromverbrauch auslesen.

**Kontrolle und Zugriff auf das DUT** Die Kontrolle und der Zugriff auf das zu testende Gerät sind in diesem Fall nicht erforderlich, womit der fünfte Punkt nicht zu konzipieren ist. Nachdem das Testsystem sich nicht auf das Testen eines bestimmten Datenloggertyps beschränken soll, sondern universell einsetzbar sein soll, wird nur die Funktionalität der Datenlogger überprüft, nicht aber diagnostiziert wenn ein Fehler auftritt, woraus dieser resultiert.

**Möglichkeit Testdokumentation zu erzeugen und Testdaten abzuspeichern** Nachdem alle Testschritte und -ergebnisse mitprotokolliert werden, vgl. Abschnitt 4.2, stellt dies, zusammen mit dem Testablaufscript und den gespeicherten Testdaten die Testdokumentation dar, womit der sechste Punkt erfüllt ist.

**Analyse der Testdaten während des Tests** Nachdem die Testdaten während des Tests von den einzelnen Modulen ausgewertet werden, beispielsweise die Round-Trip-Time bei dem *ping*-Modul, vgl. Abschnitt 4.4.4 ist der letzte Punkt ebenfalls erfüllt.

Es ist erkennbar, dass sich die Anforderungen, wie sie die Firma dSPACE zum Test eines PCM gestellt hat, mit den Anforderungen dieser Arbeit im Wesentlichen decken. Die Ergebnisse aus den gerade betrachteten Arbeiten werden in die nun folgende Konzepterstellung übernommen und dort berücksichtigt.

# 4 Konzeption und Entwurf

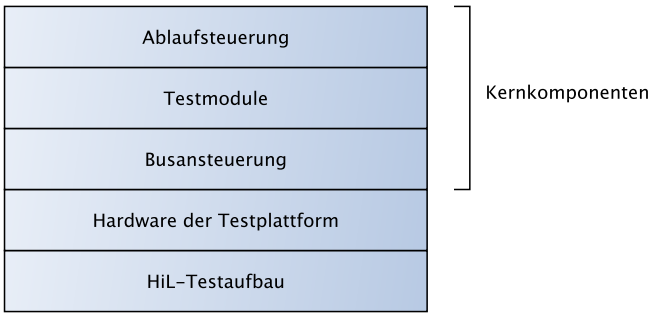


Abbildung 4.1: Teilbereiche der Testplattform

Im Folgenden soll eine Testplattform anhand der im Kapitel 2 erarbeiteten Anforderungen konzipiert werden. Hierbei wird eine Architektur konzipiert, welche sich in fünf Schichten, wie in Abbildung 4.1 ersichtlich, aufteilt. Desweiteren werden die Zusammenhänge und Funktionsweisen der einzelnen Bausteine aus den Schichten in diesem Kapitel erläutert.
Die untersten beiden Schichten sind durch die Testplattform nicht fest vorgegeben. Zum einen kann der HiL-Testaufbau variiert werden, indem beispielsweise einige Testmodule ungenutzt bleiben oder die Verschaltung der Busse geändert wird. Zum anderen kann die Hardware der Testplattform variieren, was beispielsweise die Anforderung A-20 fordert. Somit stellen die obersten drei Schichten die Kernkomponenten der Testplattform dar, auf welche im Folgenden der Fokus gelegt wird.

## 4.1 Architektur

Aufgrund der Komplexität der Testplattform ist es sinnvoll, das Konzept in Teilbereiche, welche funktional abgetrennt werden können zu unterteilen.
Hierfür wird die Testplattform, wie im Modell der Architektur in Abbildung 4.2 dargestellt, in folgende Bereiche unterteilt:

- Die oberste Schicht stellt den Kern der Laufzeitumgebung, die Ablaufsteuerung, dar. Diese realisiert zum einen die Schnittstelle zum Anwender, indem sie die vom Anwender vorgegebenen Ablaufdefinitionen interpretiert, welche in einer eigenen Scriptsprache formuliert werden. Zum anderen ist sie für die Steuerung des Testablaufs verantwortlich, das heißt in dieser Schicht werden Testprozesse aus der darunterliegenden Schicht gestartet, gestoppt und verwaltet. Die Ablaufsteuerung ist in Abschnitt 4.2 genauer beschrieben.
- Testmodule führen die eigentlichen Tests durch. Diese Tests können zweigeteilt gesehen werden. Zum einen der Test selbst, der das technische Verfahren darstellt, und zum anderen die Testspezifikation, sodass die Scriptsprache entsprechend erweitert werden kann. Dieser Teil ist in Abschnitt 4.4 genauer beschrieben
- Busansteuerung, der Teil der Testplattform, der die Kommunikation auf den Bussystemen übernimmt. Dieser Teil ist im Abschnitt 4.5 genauer beschrieben.

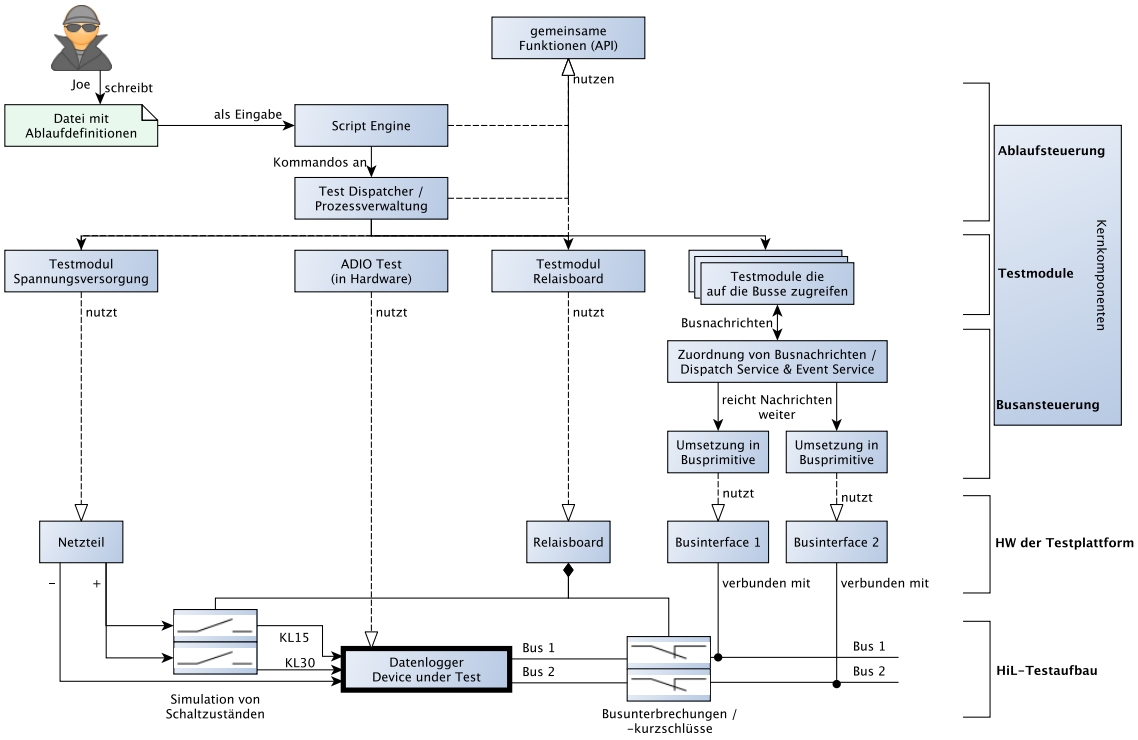


Abbildung 4.2: Architektur der Testplattform

Weiterhin stellt die Testplattform noch *gemeinsam genutzte Funktionen* in Form einer *Application Programming Interface* (*API*) zur Verfügung, die zentral von allen Komponenten der Testplattform genutzt werden können. Die API ist in Abschnitt 4.3 genauer beschrieben.
Auf die weiteren Teile der Architektur, *Hardware der Testplattform* und *HiL-Testaufbau*, hat die Software der Testplattform keinen Einfluss und muss sich möglichst flexibel hinsichtlich dieser beiden Bereiche verhalten.

Eine Gliederung in diese Bereiche ist sinnvoll, da jeder Bereich für sich eigenständig ist. Wie in der Architekturskizze in Abbildung 4.2 zu sehen, ist es beispielsweise irrelevant für die Testmodule, wenn sich das Hardwareinterface, was für die Busanbindung genutzt wird, ändert, da die Busansteuerung einen separaten Teil, und damit eine Abstraktionsschicht, darstellt. Ein weiteres Beispiel ist, dass eine Änderung hinsichtlich der Scriptsprache nicht die Testmodule betrifft, da diese auch abgegrenzt sind.
Jeder Datenlogger verwendet eine Konfiguration, welche vorgibt, wie sich der Datenlogger zu verhalten hat. Die Beschreibungssprache, in der die Konfiguration geschrieben ist, ist allerdings bei den jeweiligen Loggerherstellern verschieden. Der Datenlogger muss beispielsweise bei der Messung der RTT selbst reagieren, indem er bei Empfang einer bestimmten Botschaft eine Antwort versendet. Dies muss in die Konfiguration des Datenloggers eingebaut werden, d.h. es muss vor dem Start der Tests eine Konfiguration in den Datenlogger geladen werden, sodass der Datenlogger für den Testbetrieb konfiguriert ist. Es ist wichtig, dass diese Konfiguration unabhängig vom Testablauf, den der Anwender der Testplattform vorgibt, ist. Das heißt, dass die Datenloggerkonfiguration fest für einen spezifischen Datenloggertyp existiert und gegebenenfalls bei Erweiterung der Testplattform auch erweitert werden muss.

Die Plattform soll generischen Charakter haben, das heißt universell einsetzbar sein. Hier müssen die sechs Dimensionen der Generizität aus Abbildung 2.3 auf Seite 11 beachtet werden:

1. Für die *Anzahl der Busse* bedeutet dies, dass die Busansteuerung beliebig viele Kanäle eines Bussystems verwalten kann und zur Verfügung stellt. Die Schnittstelle zwischen dem Testmodul und der Busansteuerung muss eine eindeutige Zuordnung einer empfangenen oder gesendeten Botschaft zu einem Bus zur

Verfügung stellen.

2. Es muss eine beliebige *Anzahl von Loggern* gleichzeitig testbar sein. Das heißt, dass die Ablaufsteuerung die laufenden Tests den einzelnen *DUT*s zuordnen muss.

3. Die *Anzahl der Tests* darf nicht limitiert sein, das heißt, dass der Test Dispatcher, welcher Teil der Ablaufsteuerung ist, beliebig viele nebenläufige Tests zulässt und verwalten kann.

4. Tests sollen unabhängig vom verwendeten *Bustyp* funktionieren. Dies wird durch eine Abstraktion der Botschaften erreicht, die sowohl die Busansteuerung als auch die Testmodule implementieren. Durch diese Abstraktionsschicht liegt die Aufgabe, gesendete Botschaften entsprechend des Zielbusses aufzubereiten, nicht beim Testmodul, sondern bei der Busansteuerung. Analoges gilt für empfangene Nachrichten.
   Weiterhin soll eine Änderung der verwendeten Hardware nicht auch eine Änderung des Gesamtsystems nach sich ziehen. Das heißt, obwohl beispielsweise ein CAN-Bus bereits implementiert ist, soll ein Austausch des Hardwareinterfaces gegen ein Interface eines anderen Herstellers keine Modifikation der Tests implizieren. Dies wird durch oben genannte Abstraktionsschicht ebenfalls gewährleistet.

5. Die Testplattform soll generisch für alle *Loggertypen* einsetzbar sein. Da es sich um einen HiL-Test handelt, werden die Anschlüsse an den Datenlogger möglichst fahrzeugnah simuliert und es bedarf keiner Änderungen am Datenlogger selbst. Dies macht die Testplattform unabhängig hinsichtlich der Art des zu testenden Datenloggers, da jeder beliebige Datenlogger an die Testplattform angeschlossen werden kann.

6. Eine Erweiterung der *Versuchstypen*, also der Menge der implementierten Tests muss möglich sein. Daher werden die Tests in Module ausgelagert. Ein Testmodul kann die im Abschnitt 4.2 beschriebenen zentralen Funktionen nutzen, soll aber ansonsten so konzipiert sein, dass es als eigenständiger Prozess ohne weitere Abhängigkeiten lauffähig ist. Dies gewährleistet auch die Ausführung als nebenläufigen Prozess. Ein weiterer Vorteil hiervon ist die Übersichtlichkeit des Projekts. Das Testmodul ist ausschließlich in der Script-Engine verankert, sodass eine Änderung oder ein Hinzufügen ohne wesentlichen Aufwand möglich ist.

Für den Bereich der ADIO-Tests bedeutet Generizität, dass eine grundsätzliche Teststrategie erarbeitet werden muss, die nicht nur auf einen spezifischen Datenlogger zugeschnitten ist, sondern allgemein für analoge und digitale Ein- und Ausgänge verwendet werden kann. Diese ADIO-Tests sind in Abschnitt 4.7 genauer beschrieben.

## 4.2 Kern der Laufzeitumgebung, die Ablaufsteuerung

Der Kern der Laufzeitumgebung, auch *Ablaufsteuerung* genannt, stellt im wesentlichen, wie in Abbildung 4.2 ersichtlich, zwei Komponenten zur Verfügung. Die erste Komponente ist die *Script Engine*, welche in Abschnitt 2.3 in Punkt A-14 gefordert wird. Diese stellt die Schnittstelle zwischen dem Benutzer und der Testplattform dar. Der Benutzer schreibt seine Ablaufdefinition in eine Datei, welche er der Script Engine zu Beginn übergibt. Die Syntax dieser Scriptsprache wird im Abschnitt 4.6 erarbeitet. Diese Script Engine, ein sogenannter Interpreter, interpretiert das Ablaufscript und übergibt Kommandos, welche die Ausführung von Tests betreffen an den *Test Dispatcher / Prozessverwaltung*. Dessen Aufgabe ist das Starten bzw. Stoppen des jeweiligen Testmoduls.
Es sind Szenarien vorstellbar, in denen mehrere Tests nebenläufig abgearbeitet werden sollen. Ein Beispiel ist die Generierung von Buslast und parallel dazu die Messung der Antwortzeit des Datenloggers. Aus diesem Grund ist es notwendig, dass der Test Dispatcher die einzelnen Tests entweder nebenläufig, als eigenen Prozess, oder sequenziell, im Prozess der Ablaufsteuerung, ausführt. Diese Art und Weise der Ausführung hat der Benutzer im Ablaufscript anzugeben. Wird der Prozess sequenziell ausgeführt, sind wenig Besonderheiten zu beachten, da der Ablauf gemäß des Ablaufscripts nach Beendigung des Tests einfach fortgesetzt wird. Soll der Prozess allerdings nebenläufig ausgeführt werden, so ist zu beachten, dass die Ablaufsteuerung diese nebenläufige Prozesse verwaltet. Hierunter ist zu verstehen dass die Ablaufsteuerung die Details über den neu

erstellten Prozess abspeichert und auch ein Mechanismus existiert, um den Prozess zu einem späteren Zeitpunkt im Testablauf auch wieder zu beenden. Für diesen Mechanismus ist die *Prozessverwaltung* zuständig, welche für den automatisierten Testablauf, welcher in Anforderung A-18 in Abschnitt 2.3 gefordert wird, notwendig ist.
Solange die Testplattform aktiv ist, ist auch die Ablaufsteuerung als Kern der Testplattform aktiv. Daher muss sich die Ablaufsteuerung bei Beendigung der Software auch um das Aufräumen von Speicherbereichen etc. kümmern, sodass keine unerwünschten Rückstände der Software im System verbleiben und evtl. einen zweiten Aufruf der Software oder gar das gesamte System beeinträchtigen können.
Desweiteren kann eine weitere Instanz der Ablaufsteuerung mit einem anderen Ablaufscript gestartet werden, um die Anforderung A-22 aus Abschnitt 2.3 zu erfüllen und mehrere Datenlogger gleichzeitig zu testen.

## 4.3 Gemeinsame Funktionen

Es gibt einige Aufgaben, die von allen Teilen der Testplattform gleichermaßen genutzt werden. Um unnötige Redundanz zu vermeiden, sollen diese Aufgaben zentral für alle Teile zur Verfügung gestellt werden. Hier ergeben sich insbesondere zwei Bereiche, die zentral zur Verfügung gestellt werden sollen. Zum einen besteht in Abschnitt 2.3, Punkt A-15 die Anforderung hinsichtlich der Protokollierung des Testablaufs. Zu diesem Zweck muss eine zentrale Protokollierungsfunktion existieren, welche, je nach Nutzerwunsch, die Ausgabe entweder auf die Konsole, und/oder in eine Datei schreibt.
Zum anderen ist eine Anforderung aus Punkt A-12 in Abschnitt 2.2, dass für gewisse Module, wie beispielsweise für das Modul der Buslasterzeugung, im Testablauf vorgegeben werden kann, von welcher Form die gesendeten Botschaften sind. Dies impliziert die Definition von Botschaften mit ihrer ID und ihrem Inhalt. Somit kann ein Testmodul eine vordefinierte Botschaft zur Verarbeitung nutzen. Hierzu müssen den Tests zentrale Schnittstellen zum Auslesen des vordefinierten Botschaftsinhalts über die Botschaft-ID zur Verfügung gestellt werden.

## 4.4 Module

Um die Testplattform einfach durch neue Tests erweitern zu können, wie in Abschnitt 2.3, Anforderung A-21 gefordert, sind die Tests in Testmodule ausgegliedert.
Jedes Testmodul soll in sich eine Fehlererkennung implementieren. Beispielsweise soll bei der Kommunikation mit externen Geräten überprüft werden, ob die Kommunikation erfolgreich war. Dies kann protokollabhängig beispielsweise durch Überprüfung von Prüfsummen erfolgen.
Weiterhin gibt jeder Test bei Beendigung des Tests einen Rückgabewert zurück, der im Ablaufscript überprüft werden kann - somit kann die Ablaufsteuerung anhand der Rückgabewerte automatisiert, je nach Testergebnis, den weiteren Testverlauf beeinflussen, welches die Anforderung A-18 aus Abschnitt 2.3 erfüllt.
Wie in Abschnitt 4.1 beschrieben, soll die Konfiguration des Datenloggers unabhängig von den verwendeten Testmodulen einheitlich sein. Dies hat zur Folge dass hinsichtlich der verwendeten Botschaften keine Kollisionen entstehen dürfen. Dabei ist gemeint, dass einzelnen Modulen bereits in der Konzeptionierung bestimmte Bereiche von Botschafts-ID zugewiesen werden, die sie, im Wissen dass diese Botschaften von keinem anderen Testmodul genutzt werden, nach Erfordernis beliebig genutzt werden können. So kann beispielsweise eine zentrale Datenloggerkonfiguration verwendet werden, die auf bestimmten IDs Botschaften hinsichtlich der Messung der RTT erwarten, und auf anderen IDs Alive-Counter-Botschaften senden können ohne dass diese Botschaften miteinander kollidieren.
Es existieren im Wesentlichen zwei Modulklassen. Die eine Modulklasse, vgl. Abbildung 4.3, ist für die Ansteuerung externer Geräte, wie beispielsweise ein Netzteil oder ein Relaisboard zuständig und simuliert somit die Umgebungsbedingungen des Datenloggers. Alle Testmodule dieser Modulklasse haben gemeinsam, dass im Ablaufscript ein eindeutiger Weg zu dem externen Gerät, beispielsweise die Angabe einer Schnittstelle, definiert werden muss.
Die andere Modulklasse, vgl. Abbildung 4.4 generiert Busdaten und simuliert somit den Datenverkehr im Fahrzeug. Alle Testmodule dieser Busklasse haben gemeinsam, dass als Parameter mindestens die Bus ID

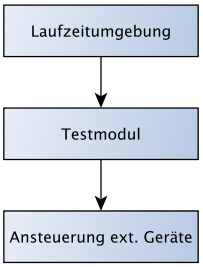


Abbildung 4.3: Modulklasse zur Ansteuerung externer Geräte

mitgegeben werden muss.
In den Folgenden Abschnitten werden Konzepte für einzelne Testmodule erarbeitet. Hierbei werden auch die notwendigen Erweiterungen für die Scriptsprache, die in Abschnitt 4.6 genau angegeben und zusammengefasst wird, beschrieben.

### 4.4.1 Ansteuerung der Spannungsversorgung

Aus Anforderung A-4 in Abschnitt 2.2 geht hervor, dass eine externe Spannungsversorgung in Form eines Netzteils angesteuert werden muss, sodass Bordnetzbedingungen simuliert werden können.
Diese Spannungsversorgung wird über eine gemeinsame Schnittstelle mit der Testplattform verbunden. Es handelt sich also um ein Modul aus der Modulklasse, die ein externes Gerät ansteuern, vgl. Abbildung 4.3
Die Scriptsprache muss also zuerst ein Kommando bereitstellen, mit dem die Schnittstelle zum Netzteil definiert werden kann. Dies geschieht über das Kommando *CONFIG POWER* ⟨*Schnittstelle*⟩, wobei ⟨*Schnittstelle*⟩ abhängig von der Implementierung durch einen eindeutige Beschreibung, wie die Testplattform das Netzteil finden kann, ersetzt werden muss.
Zum anderen wird die Scriptsprache durch das Modul *POWER* erweitert. Es existieren folgende Parameter, welche nach *CALL POWER* bzw. *START n POWER* anzugeben sind:

- Zum Setzen der Spannung genügt ein Parameter in Form einer Dezimalzahl (wobei auch Dezimalbrüche möglich sind), beispielsweise *CALL POWER 13.8* setzt die Ausgangsspannung des Netzteils auf 13.8 Volt
- Zum zusätzlichen Setzen einer Strombegrenzung wird die maximale Stromstärke als Dezimalzahl angehängt, beispielsweise *CALL POWER 13.8 3* setzt die Ausgangsspannung des Netzteils auf 13.8 Volt und begrenzt den Strom auf drei Ampere.
- Um den Ausgang ein- bzw. auszuschalten, wird als Parameter *ON* bzw. *OFF* angegeben. Beispielsweise schaltet der Befehl *CALL POWER ON* den Ausgang an.
- Um einen Spannungsverlauf zu definieren, gibt es zwei Möglichkeiten:
  1. Den Spannungsverlauf kommasepariert in einer separaten Datei beschreiben. Jede Zeile dieser Datei ist in der Form U,$I_{max}$,t wobei U die Spannung in Volt angibt, $I_{max}$ die Strombegrenzung angibt und t die Zeit in ms angibt, die das Netzteil auf diesem Sequenzschritt verweilen soll.
     Um den Spannungsverlauf in dieser Datei im aktuellen Testscript zu verwenden, wird der Parameter *SEQ FILE* ⟨*Dateiname*⟩ verwendet. Beispielsweise lädt der Befehl *CALL POWER SEQ FILE motorstart.pow* den Inhalt der kommaseparierten Datei motorstart.pow als Sequenz in das Netzteil. Dies bedient die Anforderung A-5 aus Abschnitt 2.2, da Spannungsverlaufsdefinitionen in einer separaten Datei von mehreren Ablaufscripts referenziert werden können.
  2. Den Spannungsverlauf direkt im Testscript beschreiben. Hierfür wird für jeden Sequenzschritt eine Zeile mit den Parametern *SEQ U t*, falls die Strombegrenzung nicht geändert werden soll angegeben, bzw. eine Zeile in der Form *SEQ U* $I_{max}$ *t*, falls die Strombegrenzung auf $I_{max}$ geändert werden soll. Die Parameter U und t werden, wie im vorigen Punkt beschrieben, als Dezimalzahl dargestellt.

Beispielsweise definiert *CALL POWER SEQ 13.8 5 1000* einen neuen Sequenzschritt, welcher für eine Sekunde (1000 ms) auf der Spannung 13.8 Volt verweilt und eine Strombegrenzung von fünf Ampere einstellt.

- Um die aktuell definierte Sequenz zu löschen, und somit Platz für eine neue zu schaffen, dient der Parameter *SEQ DEL*, so löscht *CALL POWER SEQ DEL* die aktuelle Spannungsverlaufssequenz.
- Um den aktuell definierten Spannungsverlauf auszuführen dient der Parameter *SEQ ON*, um den Spannungsverlauf zu beenden dient der Parameter *SEQ OFF*.
  Wird kein zusätzlicher Parameter zu *SEQ ON* angegeben, so wird die Sequenz einmal durchlaufen. Optional kann die Anzahl der Durchläufe definiert werden indem die Anzahl als Ganzzahl dem Kommando angehängt wird. Beispielsweise führt der Befehl *CALL POWER SEQ ON 5* die aktuell definierte Spannungsverlaufsdefinition fünf mal aus.
- Um die aktuellen Werte aus dem Netzteil auszulesen dient der Parameter *STAT*. So wird bei einem Aufruf von *CALL POWER STAT* beispielsweise folgende Zeile ausgegeben: *Power constanter current output: 13.8 V, 3.26 A*
  Sollen die Werte periodisch ausgegeben werden, wird der Parameter *SEQ* um die Anzahl der Wiederholungen, sowie den Abstand zwischen den Aufgaben in Sekunden, erweitert. Beispielsweise gibt der Aufruf *CALL POWER SEQ 10 2* alle zwei Sekunden den aktuellen Status aus, und wiederholt dies zehn Mal.

### 4.4.2 Ansteuerung des Relaisboards

Wie in den Anforderungen A-6, A-7, A-8 und A-9 im Abschnitt 2.2 gefordert, muss ein Relaisboard eingesetzt werden, welches Ausgänge durch Kommando von der Testplattform schließen oder öffnen kann. Dies wird benötigt um den Batteriehauptschalter, den Zündungszustand, Busunterbrechungen sowie Buskurzschlüsse zu simulieren.
Dieses Relaisboard muss über eine gemeinsame Schnittstelle mit der Testplattform verbunden werden. Es handelt sich also um ein Modul aus der Modulklasse, die ein externes Gerät ansteuern, vgl. Abbildung 4.3
Die Scriptsprache muss zuerst ein Kommando bereitstellen, mit dem die Schnittstelle zum Relaisboard definiert werden kann. Dies geschieht über das Kommando *CONFIG RELAIS* ⟨*Schnittstelle*⟩, wobei ⟨*Schnittstelle*⟩ abhängig von der Implementierung durch einen eindeutige Beschreibung, wie die Testplattform das Relaisboard finden kann, ersetzt werden muss.
Zum anderen wird die Scriptsprache durch das Modul *RELAIS* erweitert. Dabei werden zwei Parameter akzeptiert. Der erste Parameter ist die Nummer des Relais, welches angesteuert werden soll, der zweite Parameter ist der Schaltzustand, in den das Relais versetzt werden soll (*ON* oder *OFF*). Wenn als Relaisnummer -1 angegeben wird, so werden alle Relais gleichzeitig in den angegebenen Zustand geschaltet.
Beispiele:
Das Kommando *CALL RELAIS 0 ON* schließt den allerersten Relaiskontakt.
Das Kommando *CALL RELAIS -1 OFF* öffnet alle Relaiskontakte.

### 4.4.3 Buslastgenerierung

Anforderung A-12 aus Abschnitt 2.2 fordert, dass die Testplattform auch Buslasten simulieren können muss. Die Berechnung, in welchem Abstand die Botschaften auf den Bus gesendet werden müssen, um die gewünschte Buslast zu erreichen, ist abhängig vom verwendeten Bus. Da dieses Modul mit dem Bus kommuniziert, gehört es zu der Modulklasse, die den Bus nutzen, vgl. Abbildung 4.4.
Als Beispiel dient im Folgenden der *Controller Area Network* (*CAN*)-Bus. Der CAN-Bus kann mit verschiedenen Übertragungsgeschwindigkeiten arbeiten, im Auto ist der CAN-Bus mit einer Übertragungsgeschwindigkeit von 500 kBits/sek im Einsatz. Eine CAN-Botschaft hat eine variable Länge und kann Nutzdaten von bis zu acht Bytes aufnehmen. Zur Simulation von Buslasten empfiehlt es sich, Botschaften mit der maximalen Länge zu versenden, da das Versenden einer jeden Botschaft einen Rechenoverhead darstellt, den es, wenn möglich, zu vermeiden gilt. Eine CAN-Botschaft besteht aus folgenden Teilen:

- 1 Bit: Rahmenanfang

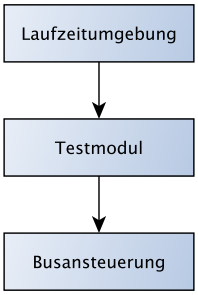


Abbildung 4.4: Modulklasse zum Testen mittels Busbotschaften

- 11 Bits: Identifier
- 1 Bit: RTR (Remote Transmission Request) Bit
- 2 Bits: reserviert für spätere Nutzung
- 4 Bits: Nutzdatenlänge, DLC
- 8*8 = 64 Bits: Nutzdaten
- 15 Bits: CRC-Prüfsumme
- 1 Bit: CRC-Delimiter
- 1 Bit: ACK-Slot
- 1 Bit: ACK-Delimiter
- 7 Bits: End-Of-Frame Bits
- 3 Bits: Zwischenraum zwischen den einzelnen Botschaften

Daraus ergibt sich, dass eine CAN-Nachricht mit acht Bytes Nutzdaten 111 Bits lang ist. Aufgrund der beim CAN-Bus verwendeten *Non Return to Zero* (*NRZ*)-Kodierung kommen allerdings Stopfbits zum Einsatz, die den Gleichspannungsanteil in der Busleitung reduzieren sollen. Dieses Stopfbit werden genau dann eingefügt, wenn fünf Bits in Folge den gleichen Zustand (alle eins oder alle null) haben. Die hiervon betroffenen Teile sind die Teile vom Identifier bis zum ACK-Delimiter, also genau 100 Bits. Es können also im schlimmsten Fall maximal 19 Stopfbits zum Einsatz kommen, wenn alle 5 Bits ein Stopfbit eingefügt wird [Borg 07, S. 68ff.]. Dies ergibt eine maximale Botschaftslänge von 130 Bits.
Das Testmodul muss periodisch Botschaften auf den Bus senden um die Buslast zu erzeugen. Es müssen $\frac{500\frac{kBits}{sek}}{130\frac{Bits}{Botschaft}} \approx 3846\frac{Botschaften}{sek}$ versandt werden, falls eine, in der Realität unmögliche, Buslast von 100% erreicht werden soll. Für eine solche Buslast von 100% muss also alle $\frac{1000000\frac{\mu s}{s}}{3846\frac{Botschaften}{sek}} \approx 260\mu s$ eine Botschaft auf den Bus verschickt werden.
Eine Anforderung ist auch, dass die zu sendenden Botschaften im Vorfeld über die *MSGARRAY*-Funktion in der Scriptsprache, vgl. Abschnitt 4.6, definiert werden können. Das Testmodul muss als Parameter also die Botschaft-IDs der Botschaften, die als Buslast gesendet werden sollen, übergeben bekommen. Das Testmodul muss überprüfen, ob die Botschaft mit der gegebenen ID schon im Vorfeld definiert wurde, ansonsten werden 8 Bytes Zufallswerte generiert, die dann als Nutzdaten für die Botschaft dienen.
Weiterhin muss das Testmodul, wie alle aus der Modulklasse, die mit dem Bus kommunizieren, den Zielbus als Parameter mit übermittelt bekommen. Nachdem die Hardwareinterfaces häufig nicht so schnell aufeinanderfolgende Botschaften senden können, ist es notwendig, zwei Buskanäle, die physikalisch miteinander verbunden sind, zu einem virtuellen Buskanal zusammen zu fassen. Ein solcher virtueller Buskanal bekommt Bus-IDs ab ID 100 zugewiesen. Wird dem Testmodul also ein Kanal mit einer ID $\geq 100$ angegeben, so muss das Testmodul die virtuelle ID zu den beiden tatsächlichen Bus-IDs auflösen und die Botschaften alternierend an diese beiden Businterfaces senden, sodass das einzelne Businterface nur noch die Hälfte der Last bewältigen muss.

Die Scriptsprache wird zur Bündelung von Kanälen um das Kommando *BUNDLE* erweitert. Dieses Kommando nimmt drei ganzzahlige Parameter:

1. Bus-ID des neuen virtuellen Busses
2. Bus-ID des ersten physikalischen Busses
3. Bus-ID des zweiten physikalischen Busses

Ein Beispiel für den Aufruf wäre *BUNDLE 100 0 2*. Mit diesem Aufruf wird ein neuer virtueller Bus mit der ID 100 angelegt, welcher zu den physikalischen Bussen 0 und 2 expandiert. Wird nun dem Buslastmodul die Bus-ID 100 mitgegeben so wird die erste Nachricht auf Bus 0 gesendet, die Zweite auf Bus 2, die Dritte wieder auf Bus 0, usw.
Weiterhin wird die Scriptsprache durch das Modul *LOAD* erweitert. Es existieren folgende Parameter, welche nach *CALL LOAD*, bzw. *START n LOAD* anzugeben sind:

1. Bus-ID entweder eines physikalischen Busses oder eines virtuellen Busses
2. gewünschte Buslast in Prozent (ohne Prozentzeichen)
3. IDs der Botschaften, die zur Generierung der Buslast verwendet werden sollen. Hierbei ist auch die Angabe von Bereichen möglich, z.B. 3-7 für die Botschaften 3, 4, 5, 6 und 7.

Beispielsweise könnte ein Aufruf des Buslastmoduls in der Scriptsprache wie folgt aussehen: *START 1 LOAD 100 50 1-3 5 7-9*, wobei 50% Buslast generiert würde, das (virtuelle) Businterface 100 genutzt würde und die Botschaften 1, 2, 3, 5, 7, 8 und 9 als Buslast gesendet würden. Wenn eine der Botschaften nicht bereits im Vorfeld über ein *MSGARRAY*, vgl. Abschnitt 4.6 definiert ist, werden die Nutzdaten beim Aufruf des Testmoduls zufällig erzeugt.

### 4.4.4 Ping

Die Anforderung A-13 in Abschnitt 2.2 fordert, dass die RTT gemessen werden kann, damit der Datenlogger hinsichtlich seiner Performance beurteilt werden kann. Nachdem diese Messung über ein Bussystem erfolgt, gehört dieses Testmodul zu der Modulklasse, die den Bus nutzen, vgl. Abbildung 4.4.
Die RTT spiegelt die Zeit wieder, die vom Senden eines Pakets bis zum Empfangen des entsprechenden Antwortpakets benötigt wird. Nachdem hier eine Antwort zu empfangen ist, muss der Datenlogger so konfiguriert sein, dass er auf Botschaften mit der festen ID 50 reagiert und eine Antwortbotschaft mit der festen ID 51 versendet. Die Testplattform schreibt den aktuellen Zeitstempel, in $\mu$-Sekunden Genauigkeit in die Botschaft mit der ID 50, die auf den Bus versandt wird. Der Datenlogger versendet bei Erhalt dieser Botschaft eine Antwortbotschaft mit der ID 51, dessen Inhalt identisch ist mit der empfangenen Botschaft. Somit erhält die Testplattform den Zeitstempel zurück, zu dem sie die erste Botschaft auf den Bus geschickt hat und kann die Differenz zwischen der aktuellen Zeit und dem Zeitstempel in der Botschaft berechnen. Dies ist die RTT.
Wie jedes Modul aus der Klasse derer, die mit dem Bus kommunizieren, wird als erster Parameter die Bus-ID übergeben, welche den Bus spezifiziert auf dem die Messung durchgeführt werden soll. Die Wahrscheinlichkeit besteht, dass ein Anwender die Messung periodisch im Hintergrund ausgeführt haben möchte. Aus diesem Grund kann ein Intervall und die Anzahl der Durchgänge spezifiziert werden. Falls eine Messung im Hintergrund ablaufen soll und man nicht jeden Erhalt einer Antwort vom Datenlogger gemeldet bekommen möchte kann man auch angegeben, dass man nur am Ende der Messung eine Zusammenfassung über die durchschnittliche RTT und die gesendeten und empfangenen Botschaften aufgezeichnet haben möchte. Die Scriptsprache wird also durch das Modul *PING* erweitert. Es existieren folgende Parameter, welche nach *CALL PING*, bzw. *START n PING* anzugeben sind:

1. Bus-ID eines physikalischen Busses auf dem die Messung durchgeführt werden soll.
2. Die Zeit in ms, die auf die Antwort des Datenloggers gewartet werden soll, welche gleichzeitig auch die Zeit ist, die zwischen den einzelnen Durchgängen gewartet werden soll.
3. Die Anzahl der Durchgänge, wobei 0 *unendlich* bedeutet.
4. (optional) *SUMMARY*. Wird das Schlüsselwort *SUMMARY* angehängt, so wird nicht bei Erhalt einer Antwort vom Datenlogger jedes Mal eine Meldung aufgezeichnet bzw. ausgegeben, sondern nur bei

Beendigung des Testmoduls eine Zusammenfassung ausgegeben, welche die durchschnittliche RTT, die Anzahl der versandten Botschaften und die Anzahl der empfangenen Botschaften beinhaltet.

### 4.4.5 Alive-Counter

Wie in Anforderung A-11 aus Kapitel 2.2 gefordert, muss ein Alive-Counter simuliert werden. Nachdem ein Alive-Counter über Busbotschaften übertragen wird, gehört dieses Testmodul zur Modulklasse, die den Bus nutzen, vgl. Abbildung 4.4. Aus diesem Grund wird als erster Parameter die Bus-ID auf dem der Alive-Counter testen soll angegeben.
Dieses Testmodul simuliert einen Alive-Counter. Ein Alive-Counter ist ein Zähler in einem beschränkten Wertebereich. So wird Anfang und Ende dieses Wertebereichs als zweiter und dritter Parameter angegeben. Sei $n$ der Endwert des Zählers so läuft der Zähler immer Modulo $n$. Ein Fehlerfall würde dargestellt werden indem der Zählerwert auf $n + 1$ gesetzt wird.
Mit diesem Testmodul soll getestet werden, ob der Datenlogger einen Ausfall des Alive-Counters erkennt und entsprechend der Konfigurationsvorgaben handelt. Aus diesem Grund wird ein Alive-Counter simuliert, welcher auch fehlerhafte Botschaften generieren kann. Mögliche Fehler sollen sein:

- Botschaftsduplikate, dass eine Botschaft zweimal geschickt wird, ohne dass der Zähler erhöht wird
- Botschaftsausfälle, dass eine Botschaft einen um zwei Stellen erhöhten Zähler enthält
- Fehlermeldungen, indem der Zählerwert auf den maximalen Zählerwert + 1 gesetzt wird um einen Fehler anzuzeigen
- Ausfälle, indem für eine spezifizierbare Zeit keine Alive-Counter Botschaften gesendet werden.

Der Alive-Counter wird in einer Botschaft mit der festen ID 52 übertragen. Der Datenlogger muss so konfiguriert sein, dass bei einem Fehler im Alive-Counter, also wenn einer der oben genannten Fälle eintritt, eine Botschaft mit der ID 53 und beliebigem Inhalt geschickt wird, wobei das Testmodul diese Botschaft empfängt und protokolliert ob das Empfangen dieser Fehlermitteilung erwartet war, also ob im Vorfeld ein Fehler erzeugt wurde, oder ob die Fehlermitteilung empfangen wurde, obwohl die Alive-Counter-Werte ordnungsgemäß simuliert wurden.
Die Scriptsprache wird also durch das Modul *ALIVE* erweitert. Es existieren folgende Parameter, welche nach *CALL ALIVE*, bzw. *START n ALIVE* anzugeben sind:

1. Bus-ID des physikalischen Busses auf dem der Test durchgeführt werden soll.
2. Startwert des Alive-Counter-Zählers (beispielsweise 0)
3. Endwert des Alive-Counter-Zählers (beispielsweise 15)
4. Abstand zwischen den einzelnen Alive-Counter-Botschaften in ms
5. (optional) Anzahl der Durchgänge - wird dieser Parameter weggelassen, werden unendlich viele Durchgänge durchgeführt.

Die Fehlererzeugung kann nur stattfinden, wenn der Alive-Counter-Test als nebenläufiger Prozess gestartet wird, da nur dann parallel zur Testausführung das Ablaufscript weiter interpretiert werden kann. Zur Generierung von Fehlern wird die Scriptsprache um das Kommando *ALIVE*, gefolgt von der internen Programm-ID des nebenläufigen Testmoduls, erweitert. Im Anschluss daran wird eines der folgenden Stichwörter erwartet:

- *DUPL* um ein Botschaftsduplikat zu erzeugen
- *SKIP* um eine Nachricht zu überspringen
- *SLEEP* gefolgt von einer Ganzzahl die in ms angibt, wie lange die Ausführung des Alive-Counters unterbrochen werden soll
- *ERR* um eine Fehlernachricht zu erzeugen

## 4.5 Busansteuerung

Wie in Abschnitt 2.3 in den Anforderungen A-19 und A-20 beschrieben, soll die Testplattform unabhängig hinsichtlich der Businterfaces und der Busarten sein.
Alle Funktionen, die das Businterface direkt ansprechen, werden in die Busansteuerung ausgelagert. Die einzelnen Testmodule kommunizieren stets über eine Abstraktionsschicht mit dem Dispatch Service & Event Service der Busansteuerung. Hierdurch wird die Anforderung A-3, die Möglichkeit der nebenläufigen Nutzung des Netzzugangs, aus Abschnitt 2.2 erfüllt. Der Dispatch Service ist dafür zuständig, dass die gesendeten Nachrichten dem richtigen Bus zugeordnet werden. Bei Empfang einer Nachricht muss der Event Service das Testmodul, welches die Nachricht empfangen möchte, darauf hinweisen, dass eine Nachricht für das Modul vorliegt und die Nachricht entsprechend übermitteln. Hierzu muss das Testmodul, bevor es Nachrichten empfangen kann, dem Event Service spezifizieren, welche Nachrichten von welchem Bus an dieses Testmodul weitergereicht werden sollen.
Durch die Abstraktionsschicht der Busansteuerung wird die Unabhängigkeit hinsichtlich der Businterfaces und Busarten erreicht, dies wird in der Architekturskizze in Abbildung 4.2 verdeutlicht.
Die Aufgabe der Busansteuerung ist, dass die Daten tatsächlich auf den Bus gesendet werden, bzw. empfan-

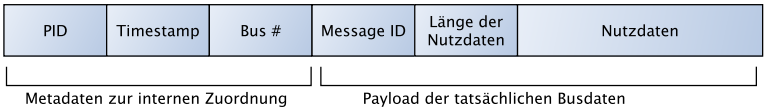


Abbildung 4.5: Format der intern übermittelten Nachrichten

gene Daten tatsächlich zum ausführenden Test gelangen. Dies entspricht den Anforderungen A-1 und A-2 aus Abschnitt 2.2.
Zuerst muss die Busansteuerung erkennen, welche Businterfaces mit welcher Anzahl von Kanälen an die Testplattform angeschlossen sind. Diese Buskanäle bekommen dann eine eindeutige ID zugewiesen, um empfangene Botschaften zuzuordnen, oder zu sendende Botschaften an den richtigen Bus zu übermitteln. Es müssen also zwei Fälle betrachtet werden: Das Senden einer Nachricht auf den Bus und das Empfangen einer Nachricht vom Bus. Beide Fälle implizieren die Abstraktion der Busnachricht.
Wir betrachten exemplarisch die Felder einer CAN-Botschaft, anhand derer analysiert werden soll, welche Parameter unsere Abstraktionsschicht kennen soll:

1. Identifier: Information für den Empfänger und Prioritätsinformation für die Busarbitrierung
2. DLC: enthält die Längeninformation der nachfolgenden Nutzdaten
3. Data: enthält die Nutzdaten des Telegramms

Das erste Feld, die ID der Botschaft muss als Parameter in die Abstraktionsschicht aufgenommen werden.
Das zweite Feld, die DLC kann aufgrund der Datenlänge vor dem Versenden der Nachricht berechnet werden.
Das dritte Feld, die Nutzdaten müssen auch als Parameter in die Abstraktionsschicht aufgenommen werden.
Die Anforderung, dass die Testplattform hinsichtlich der unterstützten Bustypen einfach erweiterbar sein muss, wie in Abschnitt 2.2 beschrieben, muss erfüllt werden. Daher müssen die von der Abstraktionsschicht implementierten Parameter erweiterbar sein, ohne jeden Test, der diese Parameter verwendet, separat anpassen zu müssen. Dies ist der Fall, wenn die Parameter in einer zentralen Datenstruktur gehalten werden, welche für den Fall dass neue Bussysteme spezielle zusätzliche Felder erfordern sollten erweitert werden kann.

Zusätzlich zu den Inhalten der Busbotschaft müssen auch interne Parameter zur Steuerung des Informationsflusses sowie Metadaten zur Botschaft in die Abstraktionsschicht aufgenommen werden:

- Von welchem Bus kam die Nachricht bzw. auf welchen Bus soll sie gesendet werden
- Für empfangene Botschaften: Einen Zeitstempel wann die Botschaft vom System empfangen wurde
- Für empfangene Botschaften: *Process Identification* (*PID*) des Zielprozesses

Der letzte Punkt führt auf eine weitere Anforderung an die Busansteuerung zurück. Diese ist, dass mehrere Tests auch nebenläufig auf den Bus zugreifen können müssen. Aus diesem Grund wird hierfür einen eigenständiger Teil konzipiert, dessen Aufgabe ausschließlich die Busansteuerung ist. Dies hat zur Folge, dass Zugriffskonflikte beim gleichzeitigen Senden an zentraler Stelle nach dem *First in, first out* (*FIFO*) Prinzip gelöst werden. Dabei werden dieser Busansteuerung die zu sendenden Botschaften über die vorhin beschriebene Abstraktionsschicht übermittelt. Um empfangene Botschaften richtig zuordnen zu können muss jeder Test, der Botschaften empfangen soll, sich bei der Busansteuerung anmelden, unter Angabe von:

- Der PID des eigentlichen Tests an den die Nachricht weitergeleitet werden soll
- Des Busses auf dem die zu empfangene Nachricht erwartet wird
- Der ID der Botschaft, die weitergereicht werden soll

Dies führt zu einer Struktur der Nachricht wie sie in Abbildung 4.5 zu sehen ist. Hier sind alle zu übermittelnden Parameter enthalten. Bei Betrachtung der Architektur in Abbildung 4.2 wird die komplette Nachricht, inklusive der Metadaten vom Testmodul zum Dispatch Service weitergeleitet. Hier werden die Metadaten verarbeitet, indem die Nachricht dem entsprechenden Bus zugeordnet wird. Im Anschluss wird der Payload, die tatsächlichen Daten die für die Erstellung der Busbotschaft relevant sind, in die Busprimitive umgesetzt und gesendet. Ein Empfang einer Nachricht erfolgt analog. Zuerst werden die Daten aus der Busbotschaft ausgelesen, anschließend wird die Nachricht an den Event Service weitergeleitet, der die Metadaten erstellt und die Busbotschaft anhängt. Daraufhin wird die komplette Nachricht dem Testmodul übermittelt.
Es ist darauf zu achten dass ein Testmodul bei seiner Terminierung die Busansteuerung entsprechend informiert, sodass keine weiteren Nachrichten an den nicht mehr laufenden Testprozess weitergereicht werden.

## 4.6 Sprachdefinition der Script Engine

Der Nutzer der Testplattform benötigt eine Möglichkeit mit der Testplattform zu interagieren. Die Zielgruppe dieser Testumgebung sind erfahrene Leute, beispielsweise jene, die vor dem Verteilen der neuen Herstellerfirmware diese im Zuge ihrer eigenen Qualitätssicherung nach ihren Anwendungsfällen testen wollen. Eine weitere Zielgruppe sind Firmwareentwickler, die vor der Freigabe einer neuen Version diese automatisiert auf Fehler untersuchen wollen.
Aus diesem Grunde ist eine Bedienung mittels Kommandozeile und eine Testablaufdefinition über eine Scriptsprache in Form von *Ablaufscripts* das Mittel der Wahl, um eine größtmögliche Flexibilität sicherzustellen.
Das Ablaufscript wird von der Ablaufsteuerung eingelesen und abgearbeitet. Die Scriptsprache definiert die Syntax des Ablaufscripts, sodass diese von der Ablaufsteuerung verarbeitet werden kann. Alle Kommandos im Ablaufscript werden grundsätzlich in Großbuchstaben geschrieben. Leerzeichen am Anfang und Ende werden ignoriert und es darf nur eine Anweisung pro Zeile geschrieben werden, womit Steuerzeichen, die das Zeilenende beschreiben, entfallen.

Zum Ausführen von Testmodulen stehen zwei Möglichkeiten zur Verfügung. Die eine Möglichkeit führt das Testmodul sequentiell aus, d.h. die weitere Interpretation des Testscripts erfolgt erst, wenn das Testmodul beendet ist. Um ein Testmodul sequentiell aufzurufen wird der *CALL*-Befehl, gefolgt vom Testnamen und dessen Parameter (s. Abschnitte über die einzelnen Tests), verwendet. Die zweite Möglichkeit ist, das Testmodul nebenläufig aufzurufen. Hierbei wird der Test gestartet und die weitere Interpretation des Ablaufskripts parallel zur Testausführung fortgesetzt. Nachdem diese Tests auch durch das Ablaufscript wieder gestoppt werden können, wird jedem Test beim Starten eine ganzzahlige ID zugewiesen, unter deren Angabe der Test zu einem späteren Zeitpunkt auch wieder gestoppt werden kann. Hierbei ist darauf zu achten, dass die ID pro gestartetem Prozess nur einmal verwendet wird. Der Befehl zum Starten eines nebenläufigen Prozesses lautet *START*, gefolgt von der ID als Ganzzahl, gefolgt vom Testmodulnamen und dessen Parametern. Der Befehl um den Test wieder zu beenden lautet *STOP*, gefolgt von der ID als Ganzzahl, die beim Prozessstart zugewiesen wurde.

Listing 4.1: Beispiel zum Starten und Stoppen von Tests

```
1 CALL POWER ON
  START 1 ALIVE 0 0 14 200
3 SLEEP 5
```

```
STOP 1
```

In der ersten Zeile des Beispielcodes wird das Modul POWER mit dem Parameter ON aufgerufen. Die weitere Ausführung des Ablaufscripts wird so lange hinausgezögert, bis das Modul POWER beendet ist, also bis die Spannungsversorgung eingeschaltet ist. In Zeile zwei wird das Modul ALIVE mit einigen Parametern, die in Abschnitt 4.4.5 dargestellt sind, nebenläufig aufgerufen. Die Zahl nach START gibt die Prozess ID 1 vor. Dabei ist darauf zu achten, dass zwischen dem *START* und dem *STOP*-Befehl kein weiterer Prozess mit der ID 1 gestartet wird. Dies würde sonst zu einem Fehler führen. In der dritten Zeile wird die Ausführung um fünf Sekunden verzögert, sodass der Alive Counter fünf Sekunden läuft bevor er in der letzten Zeile durch den STOP-Befehl, gefolgt von der ID 1 mit der der Alive-Counter auch gestartet wurde, wieder beendet wird.

Es werden, zusätzlich zu den modulspezifischen Ergänzungen der Syntax, Konstrukte bereitgestellt, welche die Ablaufsteuerung beeinflussen. Dies beinhaltet Implikationen, d.h. wenn-dann-Entscheidungen aufgrund der von den Modulen zurückgegebenen Rückgabewerten. Nur so kann der Testablauf aufgrund der Ergebnisse eines bereits durchgeführten Tests beeinflusst werden, was in Anforderung A-18 in Abschnitt 2.3 gefordert ist. Beispielsweise ist ein Konstrukt vorstellbar, dass überprüft ob der Datenlogger antwortet (über das *ping*-Modul), und nur im Falle einer Antwort den Testablauf weiter fortsetzt. Dies wird durch den *IF RETVAL* Operator bereitgestellt. Hiermit kann der Rückgabewert von einem vorher mit *CALL* aufgerufenen Modul überprüft werden. Anschließend erfolgt die Bedingung. Mögliche Bedingungen sind $>\ \langle Ganzzahl \rangle$ für Rückgabewert größer als *Ganzzahl*, $<\ \langle Ganzzahl \rangle$ für Rückgabewert kleiner als *Ganzzahl*, $=\ \langle Ganzzahl \rangle$ für Rückgabewert gleich *Ganzzahl* und $!=\ \langle Ganzzahl \rangle$ für Rückgabewert ungleich *Ganzzahl*. In den folgenden Zeilen folgt der Block, der nur ausgeführt wird, wenn die Bedingung erfüllt ist. Bei Einlesen eines *ELSE* auf einer Zeile wird der darauf folgende Block nur ausgeführt, wenn die vorangegangene *IF*-Bedingung nicht erfüllt ist. Das gesamte *IF*-Konstrukt wird mit einem *ENDIF* auf einer einzelnen Zeile abgeschlossen.

Listing 4.2: Beispiel zum Überprüfen ob ein Logger antwortet

```
CALL PING 0 1000 1
IF RETVAL = 0
        LOG "Logger antwortet"
ELSE
        LOG "Logger antwortet nicht"
ENDIF
```

In der ersten Zeile wird dem Logger eine Botschaft, auf die eine Antwort erwartet wird, gesendet. Es wird eine Sekunde lang (1000 ms) auf die Antwort gewartet. Nachdem das *PING*-Modul fertig ist, wird die zweite Zeile ausgeführt, in der überprüft wird ob der Ping erfolgreich war, also ob das *PING*-Modul den Rückgabewert 0 lieferte. Im Erfolgsfall wird mit der *LOG*-Funktion, die später beschrieben wird, die Nachricht *Logger antwortet* ausgegeben, im Fehlerfall wird die Nachricht *Logger antwortet nicht* ausgegeben.

Weiterhin sind Schleifenkonstrukte für die Testausführung notwendig. Soll beispielsweise der Fall konstruiert werden, dass die Ablaufsteuerung periodisch nur einen *ping* auf den Datenlogger macht und dann immer überprüft ob der *ping* erfolgreich war oder nicht, ist ein Schleifenkonstrukt notwendig. Das Schleifenkonstrukt wird in diesem Fall über den *LOOP*-Operator bereitgestellt. Der *LOOP*-Operator ohne weiteren Angaben führt in eine Endlosschleife, wobei der Block nach dem *LOOP*-Operator bis hin zur *ENDLOOP*-Anweisung endlos hintereinander ausgeführt wird. Soll aus der Schleife herausgesprungen werden, ist dies in einem *LOOP*-Block mit dem *BREAK*-Operator möglich. Wenn eine Schleife nur endlich oft wiederholt werden soll, wird die Anzahl der Wiederholungen als Ganzzahl hinter dem *LOOP*-Operator angegeben.

Listing 4.3: Beispiel zum Warten bis Logger verfügbar

```
LOOP
        CALL PING 0 100 1
        IF RETVAL = 0
                BREAK
        ENDIF
ENDLOOP
```

In der ersten Zeile wird eine Endlosschleife gestartet (da *LOOP* keine Parameter hat). Dann wird ein Ping gesendet, wobei 100ms auf die Antwort gewartet wird. Ist eine Antwort eingetroffen, war der Ping also erfolgreich, wird die *LOOP*-Schleife mit *BREAK* beendet, und die Ausführung wird nach der *ENDLOOP*-Anweisung, also nach der letzten Zeile, fortgesetzt. Ansonsten wird der Logger endlos oft angepingt, bis er antwortet.

Weiterhin stellt die Script Engine noch den Befehl *SLEEP* zur Verfügung, welcher das weitere Einlesen des Ablaufscripts für die angegebene Zeit unterbricht. Wenn nach dem *SLEEP*-Kommando nur eine Zahl angegeben ist, so wird diese Ganzzahl als Sekunden interpretiert. Wird nach der Ganzzahl noch ein *S*, *MS* oder *US* angehängt, so wird die Ganzzahl als Sekunden (standard), Millisekunden bzw. Mikrosekunden interpretiert.
Zuletzt existiert noch der *EXIT*-Befehl mit dem die Testausführung an der Stelle an der dieser Befehl steht beendet wird. Alle nebenläufig ausgeführten Testmodule werden auch ordentlich beendet.

Listing 4.4: Beispiel zum Beenden nach 3,5 Sekunden

```
SLEEP 3500 ms
EXIT
LOG "Das wird nie mehr ausgefuehrt"
```

Im Beispiel wird zuerst 3,5 Sekunden gewartet, anschließend die Ausführung beendet. Die *LOG*-Funktion in der letzten Zeile wird nie ausgeführt, da die Ausführung schon in der zweiten Zeile beendet wird.

Die Scriptsprache muss für einige Tests, beispielsweise für den Alive-Counter Test, Konstrukte für Zahlenbereiche zur Verfügung stellen. So akzeptiert der Alive-Counter Test als Parameter, welche Botschaft-IDs verwendet werden sollen, auch Bereiche in der Form von 1-5, welches dann als 1, 2, 3, 4 und 5 interpretiert werden soll. Dezimalbrüche werden mit einem Punkt (nicht Komma!) getrennt. Freitexte werden zwischen Doppelquotes gesetzt. Eine Zeile, die nicht interpretiert werden soll (Kommentarzeile), beginnt mit einem Strichpunkt.
Die Scriptsprache stellt auch einen *CONFIG*-Befehl zur Konfiguration von einzelnen Modulparametern zur Verfügung. So wird für die Module, die externe Geräte ansteuern, über den *CONFIG*-Befehl die Schnittstelle zu diesem Gerät angegeben, s. Abschnitte 4.4.1 und 4.4.2.

Zusätzlich stellt die Ablaufsteuerung noch zentrale Funktionen zur Verfügung, welche auch in der Scriptsprache abgebildet werden müssen. Hierbei handelt es sich zum einen um die *LOG*-Funktion, gefolgt von einem Freitext zwischen Doppelquotes, wie sie in den vorherigen Beispielen bereits verwendet wird. Diese Funktion schreibt den Freitext in die Logdatei. Zum anderen handelt es sich um die Funktion *MSGARRAY*, die Definition von Botschaftarrays. Dies wird genutzt um Botschaftsinhalte fest zu belegen, wie beispielsweise vom Buslasttest, vgl. Abschnitt 4.4.3, genutzt. Hierbei wird als erster Parameter die ID der Botschaft, die definiert werden soll spezifiziert. Anschließend folgen die Nutzdaten der Botschaft.

Listing 4.5: Beispiel für Definieren zweier Botschaften, die für die Buslasterzeugung genutzt werden

```
MSGARRAY 1 a1 b2 c3 d4 e5 f6 a7 b8
MSGARRAY 2 1a 2b 3c 4d 5e 6f 7a 8b
START 1 LOAD 1 30 1-5
```

Die ersten beiden Zeilen definieren die Botschaften mit der ID 1, bzw. 2. Die letzte Zeile startet die Buslastgenerierung von 30 % Last auf Bus 1, wobei die Nachrichten 1-5, also 1, 2, 3, 4 und 5 genutzt werden. Nachdem die Nachrichten 1 und 2 im Vorfeld über die *MSGARRAY*-Befehle definiert wurden, werden diese Nachrichten genutzt und der Inhalt der Nachrichten 3, 4 und 5 zufällig generiert.

Es besteht die Möglichkeit, dass neue Module die Scriptsprache auch durch neue Befehle erweitern, beispielsweise erweitert das Buslasttestmodul die Scriptsprache durch den Befehl *BUNDLE*, s. Abschnitt 4.4.3.

Als Zusammenfassung aller Befehle und Möglichkeiten der Scriptsprache wird im Folgenden die komplette Syntax, mitsamt der modulspezifischen Erweiterungen in der *Backus-Naur-Form* (*BNF*) angegeben.
Begonnen wird mit den Typdefinitionen, wie in Tabelle 4.1 angegeben:

$$
\begin{aligned}
\langle Digit\rangle &::= 0|1|2|3|4|5|6|7|8|9\\
\langle Number\rangle &::= \langle Digit\rangle\langle Number\rangle|\langle Digit\rangle\\
\langle HexDigit\rangle &::= \langle Digit\rangle|a|b|c|d|e|f\\
\langle HexTupel\rangle &::= \langle HexDigit\rangle\langle HexDigit\rangle\\
\langle Float\rangle &::= \langle Number\rangle.\langle Number\rangle|\langle Number\rangle\\
\langle SeveralHexTupels\rangle &::= \langle HexTupel\rangle\langle SeveralHexTupels\rangle\\
\langle Lowercase\rangle &::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z\\
\langle Uppercase\rangle &::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z\\
\langle SpecChar\rangle &::= .|,|;|:|'|/|\backslash|\%|-|+|*|?|!|\$|\&|(|)|[|]|\{|\}|x|<|>\\
\langle Character\rangle &::= \langle Lowercase\rangle|\langle Uppercase\rangle|\langle Digit\rangle|\langle SpecChar\rangle\\
\langle Sentence\rangle &::= \langle Character\rangle\langle Sentence\rangle|\langle Character\rangle\langle Sentence\rangle\\
\langle QSentence\rangle &::= \langle Character\rangle\langle QSentence\rangle|\langle Character\rangle\langle QSentence\rangle|''\langle QSentence\rangle\\
\langle TextInQuotes\rangle &::= ''\langle Sentence\rangle''\\
\langle NumRange\rangle &::= \langle Number\rangle\ \langle Number\rangle|\langle Number\rangle - \langle Number\rangle|\\
&\quad\ \langle Number\rangle|\langle NumRange\rangle\ \langle NumRange\rangle\\
\langle Operator\rangle &::= <|>|=|!=\\
\langle State\rangle &::= ON|OFF
\end{aligned}
$$

Tabelle 4.1: Typdefinitionen der Scriptsprache in BNF

In der folgenden Tabelle 4.2 wird die Sprachdefinition ohne der Module dargestellt:

$$
\begin{aligned}
\langle Command\rangle &::= \langle Directive\rangle|\langle Comment\rangle|\langle Program\rangle|\langle Config\rangle|\langle MsgArray\rangle|\langle Log\rangle|\langle Sleep\rangle|\langle Extensions\rangle\\
\langle Directive\rangle &::= \langle Loop\rangle|\langle If\rangle|\langle Exit\rangle\\
\langle Loop\rangle &::= LOOP|LOOP\ \langle Number\rangle|BREAK|ENDLOOP\\
\langle If\rangle &::= IF\ RETVAL\ \langle Operator\rangle\ \langle Number\rangle|ELSE|ENDIF\\
\langle Exit\rangle &::= EXIT\\
\langle Comment\rangle &::= ;\langle Comment\rangle\langle QSentence\rangle\\
\langle Program\rangle &::= START\langle Number\rangle\langle ProgNameParams\rangle|CALL\langle ProgNameParams\rangle|STOP\langle Number\rangle\\
\langle Config\rangle &::= CONFIG\ \langle ProgName\rangle\ \langle TextInQuotes\rangle\\
\langle MsgArray\rangle &::= MSGARRAY\ \langle Number\rangle\ \langle SeverelHexTupels\rangle\\
\langle Log\rangle &::= LOG\ \langle TextInQuotes\rangle\\
\langle TimeUnit\rangle &::= S|MS|US\\
\langle Sleep\rangle &::= SLEEP\ \langle Number\rangle|SLEEP\ \langle Number\rangle\ \langle TimeUnit\rangle
\end{aligned}
$$

Tabelle 4.2: Sprachdefinition (ohne Module) der Scriptsprache in BNF

Zuletzt werden in der folgenden Tabelle 4.3 die modulspezifischen Spracherweiterungen, unter dem BNF-Nichtterminalsymbol $\langle Extensions\rangle$ geführt, aufgeführt. Weiterhin werden die Programme mit ihren Parametern, unter dem BNF-Nichtterminalsymbol $\langle ProgNameParams\rangle$ geführt, spezifiziert:

Das folgende Beispiel wird nun nach der vorgegebenen BNF analysiert:

Listing 4.6: Beispiel: Warten auf Logger und Buslast für 10 Sekunden

```
CALL PING 0 100 1
IF RETVAL != 0
        EXIT
ENDIF
MSGARRAY 1 a1 b2 c3 d4 e5 f6 a7 b8
START 1 LOAD 1 20 1-5
SLEEP 10 S
```

$$
\begin{aligned}
\langle Extensions\rangle &::= \langle Bundle\rangle|\langle Alive\rangle \\
\langle ProgNameParams\rangle &::= \langle ProgLoad\rangle|\langle ProgRelais\rangle|\langle ProgPower\rangle|\langle ProgPing\rangle|\langle ProgAlive\rangle \\
\langle Bundle\rangle &::= BUNDLE\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle \\
\langle Alive\rangle &::= ALIVE\ \langle Number\rangle\ \langle AliveErrorGen\rangle \\
\langle AliveErrorGen\rangle &::= SKIP|DUPL|ERR|SLEEP\ \langle Number\rangle \\
\langle ProgLoad\rangle &::= LOAD\ \langle Number\rangle\ \langle Number\rangle\ \langle NumRange\rangle \\
\langle ProgRelais\rangle &::= RELAIS\ \langle Number\rangle\ \langle State\rangle \\
\langle ProgPower\rangle &::= POWER\ \langle ProgPowerSetU\rangle|POWER\ \langle State\rangle| \\
&\quad POWER\ \langle ProgPowerSeq\rangle|POWER\ \langle ProgPowerStat\rangle \\
\langle ProgPowerSetU\rangle &::= \langle Float\rangle|\langle Float\rangle\langle Float\rangle \\
\langle ProgPowerSeq\rangle &::= SEQ\ \langle Float\rangle\ \langle Float\rangle|SEQ\ \langle Float\rangle\ \langle Float\rangle\ \langle Float\rangle\ | \\
&\quad SEQ\ FILE\ \langle TextInQuotes\rangle|SEQ\ DEL| \\
&\quad SEQ\ \langle State\rangle|SEQ\ \langle State\rangle\ \langle Number\rangle \\
\langle ProgPowerStat\rangle &::= STAT|STAT\ \langle Number\rangle\ \langle Float\rangle \\
\langle ProgPing\rangle &::= PING\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle| \\
&\quad PING\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle\ SUMMARY \\
\langle ProgAlive\rangle &::= ALIVE\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle| \\
&\quad ALIVE\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle
\end{aligned}
$$

Tabelle 4.3: Modulspezifische Sprachdefinitionen der Scriptsprache in BNF

```
STOP 1
LOG "Ende des Programms"
```

Auf das Expandieren von Typdefinitionen, wie in Tabelle 4.1 angegeben, wird aus Übersichtsgründen verzichtet.

**Zeile 1**: CALL PING 0 100 1
CALL PING $\langle Number\rangle\ \langle Number\rangle\ \langle Number\rangle$
CALL $\langle ProgPing\rangle \rightarrow$ CALL $\langle ProgNameParams\rangle \rightarrow \langle Program\rangle \rightarrow \langle Command\rangle$

**Zeile 2**: IF RETVAL != 0
IF RETVAL $\langle Operator\rangle\ \langle Number\rangle \rightarrow \langle If\rangle \rightarrow \langle Directive\rangle \rightarrow \langle Command\rangle$

**Zeile 3**: EXIT
$\langle Directive\rangle \rightarrow \langle Command\rangle$

**Zeile 4**: ENDIF
$\langle If\rangle \rightarrow \langle Directive\rangle \rightarrow \langle Command\rangle$

**Zeile 5**: MSGARRAY 1 a1 b2 c3 d4 e5 f6 a7 b8
MSGARRAY $\langle Number\rangle\ \langle SeveralHexTupels\rangle \rightarrow \langle MsgArray\rangle \rightarrow \langle Command\rangle$

**Zeile 6**: START 1 LOAD 1 20 1-5
START $\langle Number\rangle$ LOAD $\langle Number\rangle\ \langle Number\rangle\ \langle NumRange\rangle$
START $\langle Number\rangle\ \langle ProgLoad\rangle \rightarrow$ START $\langle ProgNameParams\rangle \rightarrow \langle Program\rangle \rightarrow \langle Command\rangle$

**Zeile 7**: SLEEP 10 S
SLEEP $\langle Number\rangle\ \langle TimeUnit\rangle \rightarrow \langle Sleep\rangle \rightarrow \langle Command\rangle$

**Zeile 8**: STOP 1
STOP $\langle Number\rangle \rightarrow \langle Program\rangle \rightarrow \langle Command\rangle$

**Zeile 9**: LOG "Ende des Programms"
LOG $\langle TextInQuotes \rangle \rightarrow \langle Log \rangle \rightarrow \langle Command \rangle$

Im Folgenden werden die Anforderungen an die Scriptsprache aus den Abschnitten 2.2 und 2.3 mit den Leistungsmerkmalen der Scriptsprache aus diesem Kapitel verglichen:

- Anforderung A-12, definierte Botschaften: Ist über $\langle MsgArray \rangle$ erfüllt.
- Anforderung A-14, Eingabemöglichkeit zur Beschreibung des Testablaufs: Dieser Abschnitt definiert eine Scriptsprache für Ablaufscripts als Eingabemöglichkeit.
- Anforderung A-18, automatisierter Testablauf: Über die *IF RETVAL* Anweisung können Testergebnisse automatisch überprüft werden und der weitere Testablauf in Abhängigkeit von dem Ergebnis eines vorhergehenden Tests gesteuert werden. Somit ist diese Anforderung erfüllt.

Es ist erkennbar, dass alle Anforderungen an die Scriptsprache durch das vorliegende Konzept abgedeckt werden.

## 4.7 Testen der analogen und digitalen Ein- und Ausgänge

Zum Testen der analogen Eingänge gibt es zwei Ansätze:

1. Es wird ein externes Gerät konstruiert, welches die Option zur Verfügung stellt, die analogen Eingänge mit einer Spannung zu beschalten. Dabei kann die Spannung, die an die analogen Eingänge angelegt wird, aus einer externen Spannungsquelle stammen. Die Spannung kann auch fest von einem Spannungsteiler stammen, damit jeder Eingang mit einer anderen Spannung beschaltet wird.
   Der Nachteil hierbei ist, dass auf dieses Testverfahren nicht automatisiert über die Testplattform eingewirkt werden kann.
2. Die flexibelste, allerdings auch kostenintensivste Lösung ist, dass für jeden analogen Eingang ein Netzteil angesteuert wird, welches die Eingangsspannung bereitstellt. Hierbei kann das Netzteil von der Testplattform angesteuert werden. Dies hat den Vorteil, dass das Testergebnis direkt in Echtzeit und automatisiert ausgewertet werden kann.

Zum Testen der digitalen Eingänge gibt es, analog zu den analogen Eingängen, ebenfalls zwei Ansätze:

1. Es wird ein externes Gerät konstruiert, welches den Schaltzustand eines einzelnen digitalen Eingangs auf Knopfdruck ändert. Eine Rückmeldung könnte durch eine entsprechende Konfiguration des Datenloggers gegeben werden. Beispielsweise könnte eine digitaler Ausgang mit einer LED als Schaltzustandsanzeige genau dann beschaltet werden, wenn der entsprechende Eingang auch beschaltet ist.
   Der Nachteil hierbei ist wieder, dass dieser Test nicht automatisiert stattfinden kann. Als ersten Schritt zur Automatisierung könnte hier in Hardware eine Steuerung implementiert werden, die in einem bestimmten Intervall die Digitalausgänge ein- und wieder ausschaltet (ähnlich zu einem Lauflicht)
2. Die flexiblere Lösung besteht darin, die digitalen Eingänge über ein Relaisboard, welches ohnehin schon in der vorliegenden Arbeit konzeptioniert und implementiert ist, zu schalten. Der Schaltzustand kann dann automatisiert über die Testplattform abgefragt und ausgewertet werden.

Zum Testen der digitalen Ausgänge gibt es wieder zwei Möglichkeiten, die den Schemata der Eingangstests ähneln:

1. Der Schaltzustand der digitalen Ausgänge des Datenloggers könnte in Hardware dargestellt werden, beispielsweise durch Ansteuerung einer LED. Dies wäre eine einfache Möglichkeit, allerdings kann der Schaltzustand nicht automatisiert über die Testplattform ausgewertet werden.
2. Der Schaltzustand der digitalen Ausgänge könnte durch digitale Eingänge direkt an der Testplattform ausgewertet werden. Hierfür müsste eine entsprechende Hardware ausgewählt, und die Unterstützung konzeptioniert und implementiert werden. Weiterhin könnte der Schaltzustand der digitalen Ausgänge

direkt an die im Datenlogger verfügbaren digitalen Eingänge verbunden werden. Dies hätte zum Vorteil, dass keine teure separate Hardware für die Testplattform notwendig ist, und die digitalen Eingänge, zusammen mit den digitalen Ausgängen automatisiert getestet werden können. Der Nachteil hierbei ist, dass bei einem nicht erfolgreichen Test unklar ist, ob der Ein- oder Ausgang defekt ist, da der Test von beidem abhängt.

Durch wählen des jeweils zweiten Lösungsansatzes, wird Punkt A-10 aus Abschnitt 2.2 erfüllt, da somit die ADIO automatisiert getestet werden können.

Bei einem direkten Vergleich des Anforderungskatalogs mit diesem Konzept wird deutlich, dass alle Anforderungen aus dem Katalog in diesen Abschnitten bearbeitet werden. Zuletzt sollen der Vollständigkeit halber noch die beiden fehlenden Anforderungen erwähnt und behandelt werden:

- Die Anforderung A-16 aus Abschnitt 2.3 bezieht sich auf die Geschwindigkeit der Testplattform. Diese muss höher sein, als die Geschwindigkeit des Datenloggers. Dies ist schlussendlich nicht ausschließlich eine Frage der Rechenleistung, sondern auch eine Frage der Effizienz der Testplattformsoftware. Weiterhin spielen hier externe Faktoren, beispielsweise die Leistungsfähigkeit der verwendeten Bushardware eine Rolle. Daher wird diese Anforderung zwar in Abschnitt 5.1 bei der Hard- und Softwareauswahl berücksichtigt, kann aber schlussendlich erst durch Messung der CPU-Auslastung und Buslasten an der fertigen Implementierung bestätigt werden.
- Die Anforderung A-17 aus Abschnitt 2.3 bezieht sich auf die Kosten der Nutzung. Dieser Punkt wird direkt bei der Anforderung abgearbeitet, indem die Kosten der Implementierung mit den Kosten eines einzigen Datenloggers verglichen werden. Nachdem diese Implementierung mit allen Komponenten wesentlich kostengünstiger ist, als ein einziger Datenlogger, wird die Anforderung der Kosten als erfüllt angesehen.

# 5 Implementierung

In diesem Kapitel wird als Tragfähigkeitsnachweis des Konzepts dieses in eine tatsächlich funktionierende Testplattform umgesetzt. Hierfür wird zuerst die Hard- und Software ausgewählt und Schnittstellen definiert. Anschließend wird das Konzept der einzelnen Bausteine auf die verwendete Hard- und Software adaptiert und die dabei auftretenden Problemstellungen und mögliche Lösungen beleuchtet.

## 5.1 Hard- und Softwareauswahl

Zunächst wird anhand objektiver Kriterien ausgewählt, welche Hardware zum Einsatz kommt.


Abbildung 5.1: ARCOS Datenlogger der Firma CAETEC

**Auswahl des verwendeten Datenloggers** Bei der Auswahl des Datenloggers wird darauf geachtet, dass er eine große Vielfalt von Bussen, die verwendet werden können bereitstellt, sowie die Möglichkeit von ADIOs bereitstellt, sodass diese auch getestet werden können. Weiterhin ist es wichtig, diesen Datenlogger nach Anforderung erweitern zu können. Aus diesen Gesichtspunkten fällt hier die Entscheidung auf den ARCOS Datenlogger der Firma CAETEC, vgl. Abbildung 5.1. Er hat, in der bei der BMW Group üblicherweise verwendeten Ausbaustufe, zwölf CAN-Kanäle, das heißt zwölf dedizierte Möglichkeiten mit einem CAN-Bus zu kommunizieren, zwei FlexRay-Kanäle, sowie acht analoge und acht digitale Eingänge und acht digitale Ausgänge.
Der ARCOS Datenlogger kann weiterhin einfach durch Hinzufügen weiterer Module erweitert werden.

**Auswahl des zu implementierenden Bussystems** Desweiteren soll ein Bussystem implementiert werden. Die derzeit im Fahrzeug gängigen Bussysteme sind *Local Interconnect Network* (*LIN*), CAN und FlexRay. LIN und CAN sind, was den Aufbau der Botschaften angeht, sehr ähnlich, FlexRay ist wesentlich

leistungsfähiger, da Übertragungsgeschwindigkeiten von bis zu 10 MBit/s unterstützt werden, aber auch komplizierter, da statische Segmente existieren, bei dem jedes Steuergerät sein eigenes Zeitfenster hat, in dem es Botschaften senden kann. Hierdurch kann FlexRay gewährleisten, dass kritische Daten innerhalb einer bekannten Zeit übertragen werden. Weiterhin ist CAN am weitesten verbreitet, da jedes aktuelle Fahrzeug mindestens einen CAN-Bus hat. Da diese Implementation zeigen soll, dass das Konzept grundsätzlich umsetzbar ist, wird der CAN-Bus implementiert.

**Auswahl der Testplattform** Die nächste Überlegung ist, auf welcher Plattform die Testsoftware laufen soll. Die Firma Vector bietet mit ihrem VN8900 ein CAN- und FlexRay Interface mit acht Kanälen und integriertem Echtzeit-Rechner, welches eine denkbare Basis für die Testplattform wäre. Der Nachteil ist, dass die Programmierung hierfür in *Communication Access Programming Language* (*CAPL*) erfolgen müsste, einer eigens von Vector entwickelten Scriptsprache. Die Flexibilität der Testplattform würde hierunter sehr leiden, schließlich sollen auch im Hinblick auf eine einfache Erweiterung alle Optionen offen gehalten werden. Weiterhin wäre es auch nicht möglich, externe Geräte wie beispielsweise ein Netzteil oder ein Relaisboard anzusteuern. Aus diesem Grund fällt die Entscheidung auf einen handelsüblichen PC, welcher die größtmögliche Flexibilität darstellt.

**Auswahl des Betriebssystems** Die nächste Entscheidung ist die Frage nach dem Betriebssystem. Nachdem aufgrund der Flexibilität ein gängiges PC-Betriebssystem eingesetzt werden soll, stehen Windows und Linux zur Wahl. Beide bieten offensichtlich vergleichbare Funktionen hinsichtlich der Softwareentwicklung. Aus diesem Grund ist diese Entscheidung subjektiver Natur. Nachdem ich bisher Software, welche auch tieferliegende Systemfunktionen nutzt, nur für Unix-basierende Systeme entwickelt habe, wird als Basis für die Testplattform auch Linux eingesetzt.

**Auswahl des Netzteils** Nachdem Spannungsverläufe simuliert werden sollen, wird hierfür ein Netzteil benötigt, welches, wie in Abschnitt 2.2 unter Punkt A-4 beschrieben, schnell auf Spannungswechsel reagieren kann. Weiterhin muss das Netzteil Spannungsverlaufssequenzen unterstützen und das Kommunikationsprotokoll offengelegt sein, dass die Anbindung implementiert werden kann. Die Entscheidung fällt hierbei auf das Netzteil SSP 320-32 der Firma Gossen Metrawatt. Es liefert 320 Watt, bei einer einstellbaren Spannung im Bereich von 0-32 Volt und liefert hierbei 0-18 Ampere. Das Netzteil kann Spannungsverlaufssequenzen abspeichern und aufrufen und lässt sich komplett über RS-232 programmieren. Es bietet eine Einstellzeit von 1 ms, bei einem Lastsprung von 0 auf 100 % und kann Spannungswechsel somit sehr schnell realisieren. Weiterhin bietet es auch die Möglichkeit die aktuelle Stromabnahme über RS-232 auszulesen. Das Protokoll ist komplett offengelegt und im Handbuch beschrieben. Leider können die Sequenzschritte nur im 10ms Abstand definiert werden. Das heißt, obwohl das Netzteil in 1ms auf die Ausgangsspannung hinregelt kann der nächste Schritt erst 9ms später beginnen. Ein Telefonat mit der Herstellerfirma ergab, dass dies auf eine ältere CPU zurückzuführen ist. Ein anderes Modell der Firma hat eine neuere CPU und kann Sequenzen in ms-Auflösung annehmen. Hier ist die Einstellzeit mit 10ms allerdings wesentlich höher, sodass die höhere Auflösung in der Programmierung leider nichts bringt. Die Entscheidung ist trotzdem auf das schnelle Netzteil gefallen. Wenn eine Spezifikation für einen Spannungsverlauf fordert dass in 10ms auf eine bestimmte Spannung hingeregelt werden soll, regelt das verwendete Netzteil in 1ms. Das stellt im Wesentlichen nur härtere Bedingungen für das DUT dar.
Die Technischen Daten zum ARCOS Datenlogger geben diesen mit einer Leistung von 12W an, allerdings zeigt die Praxis, dass die bei der BMW Group verwendete Ausbaustufe mindestens das Doppelte benötigt. Von den Leistungsgrenzen des Netzteils ist dies allerdings auch noch weit entfernt.

**Auswahl der CAN-Busanbindung** Es existieren im Wesentlichen zwei große Hersteller, die Hardwareinterfaces, welche einen CAN-Bus mit einem PC verbinden, herstellen. Dies ist zum einen die Firma Vector, welche ein breites Spektrum Interfaces anbietet, jedoch zum Zeitpunkt des Schreibens dieser Arbeit keine Möglichkeit bereitstellt, diese Interfaces unter Linux mittels einer API anzubinden. Hier wurde auch mit den verantwortlichen Personen bei der Firma Vector gesprochen, die mitteilten, dass eine solche Anbindung im zeitlichen Rahmen dieser Arbeit nicht geplant ist. Zum anderen stellt die Firma IXXAT CAN-Interfaces her.

Diese bieten auch eine API für Linux an. Aufgrund der einfachen Erweiterbarkeit ist die Wahl auf das externe Interface IXXAT USB-to-CAN-II gefallen. Falls mehr Kanäle benötigt werden kann hier einfach ein weiteres Interface extern an den USB angesteckt werden. Obwohl dieses Interface bei der höchsten CAN-Bus-Geschwindigkeit von 1 MBit/s nur eine gemessene Buslast von ca. 40 % hervorrufen kann, fällt die Wahl auf dieses Interface, da für den Fall, dass höhere Lasten benötigt werden, zwei Businterfaces an einen gemeinsamen Bus angeschlossen werden können und somit die doppelte Geschwindigkeit erzielt werden kann.

**Auswahl der PC-Komponenten** Wie in Abschnitt 2.3 unter Punkt A-16 gefordert, muss der PC, auf dem die Testsoftware läuft, schneller sein als der Datenlogger, der getestet werden soll. Der ARCOS Datenlogger setzt eine Intel Core Duo CPU getaktet mit 1,66 GHz ein. Er verfügt über 2 GB DDRAM und verwendet den Intel 945 Chipsatz mit einem mit 633 MHz getakteten Front-Side-Bus. Zum Einsatz kommt eine *Compact-Flash* (*CF*) Karte mit einer Kapazität von 4 GB. Um einen schnelleren PC zu verwenden hat der ausgesuchte PC eine Quad-Code Intel Core i5 2405S CPU mit 2,5 GHz und 6MB Cache. 4 GB DDR3-1333 Arbeitsspeicher ist für den Zweck ausreichend. Wie bereits am Ende des Kapitels 4 diskutiert, ist die Geschwindigkeit nicht ausschließlich eine Frage der Rechenleistung sondern auch eine Frage der Effizienz der Testplattformsoftware. Zu dem Zeitpunkt der Hardwareauswahl wird jedoch davon ausgegangen, dass diese Hardware ausreichend leistungsfähig ist. Dies wird in Kapitel 6 am praktischen Beispiel betrachtet. Als Mainboard kommt das Intel Executive DQ67EPB3 Mini-ITX zum Einsatz, damit der Rechner portabel bleibt. Dieses Mainboard hat auch USB 3.0 Anschlüsse, um schnelle Erweiterungsoptionen anschließen zu können. Als Festplatte kommt eine Solid-State-Disk mit 128 GB zum Einsatz. Dass die Konfiguration des Rechners in der Form ausreichend ist

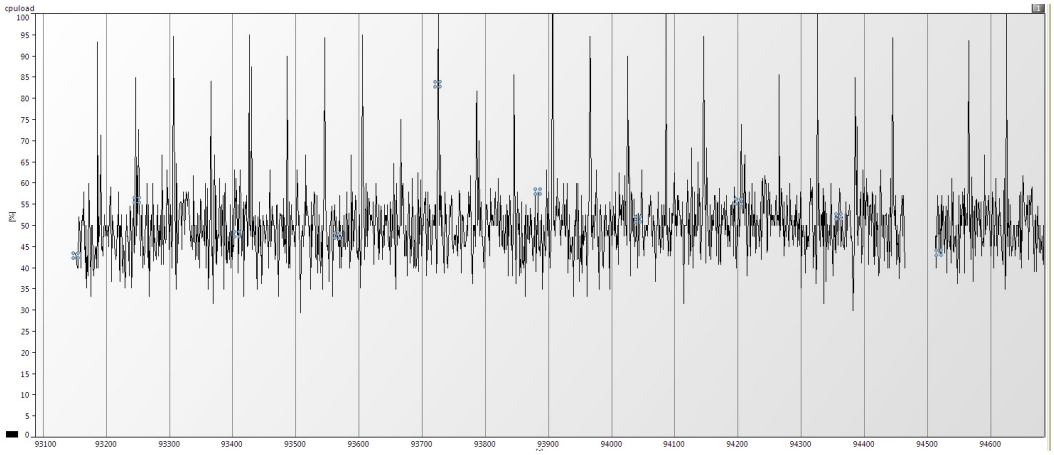


Abbildung 5.2: CPU-Last eines ARCOS-Datenloggers im Produktivbetrieb

zeigt, dass die CPU-Last selbst bei Erzeugung von hohen Buslasten von ca. 85 % auf den Testbussen nicht über 35 % kommt, wohingegen die CPU-Last des Datenloggers bei tatsächlicher Nutzung in einem Fahrzeug, bei wesentlich weniger Buslast als in Testszenarien, bereits im Durchschnitt bei ca. 50 % liegt, siehe Abbildung 5.2.
Nachdem das verwendete Mainboard keine RS232-Schnittstelle mehr zur Verfügung stellt, kommt hier ein USB zu RS232 Umwandler mit einem Prolific PL2303 Chipsatz zur Anwendung, welcher von Linux unterstützt wird.

**Auswahl des Relaisboards** Es werden weiterhin noch vier von der Testsoftware schaltbare Kontakte benötigt (Busunterbrechung, Buskurzschluss, Zündung, Batteriehauptschalter). Hierfür wird ein Relaisboard der Firma Conrad Elektronik eingesetzt. Dieses stellt acht Relaisausgänge zur Verfügung und ist mit einem optionalen USB zu RS232 Konverter onboard ausgestattet. Dieser Konverter nutzt einen Silicon Labs CP2102 Chipsatz, welcher ebenfalls von Linux unterstützt wird.

## 5.2 Schnittstellen zwischen den drei Säulen

Wie aus der Architektur der Testplattform in Abbildung 4.2 auf Seite 18 ersichtlich, unterteilt sich die Plattform in drei Kernkomponenten:

- Die *Ablaufsteuerung*, welche sich sowohl um das Interpretieren der Ablaufscripts, als auch das Starten und Stoppen von Testprozessen kümmert. Die Ablaufsteuerung verwaltet ebenfalls die aktuell laufenden Testprozesse.
- Die *Testmodule* selbst, welche die eigentlichen Testprozeduren beinhalten.
- Die *Busansteuerung*, welche den nebenläufigen Zugriff auf die Bussysteme ermöglicht und eine Abstraktionsschicht zur Verfügung stellt, über die die Testmodule mit dem Bus kommunizieren können.

Die Schnittstelle zwischen der Ablaufsteuerung und den Testmodulen stellt zum einen die Script Engine dar, welche die Ablaufscripts einließt und verarbeitet. Zum Anderen ist eine weitere Schnittstelle die Prozessverwaltung.

Wie in Abschnitt 4.6 des Konzepts dargestellt, wird eine *MSGARRAY*-Funktion zur Verfügung gestellt um den Inhalt bestimmter Botschaften vordefinieren zu können. Diese Funktion ist eine Schnittstelle zwischen der Ablaufsteuerung und den Testmodulen. Die Script Engine der Ablaufsteuerung ließt die Botschaftsdefinitionen aus dem Ablaufscript ein und legt sie in ein *Shared Memory* (*SHM*)-Segment ab, aus welchem das Testmodul die Botschaftsdefinitionen wieder ausließt. Die genaue Funktionalität ist in Abschnitt 5.3.2 dargestellt.

Zum Senden und Empfangen von Busnachrichten stellt der separat laufende Prozess *interfaced*, vgl. Abschnitt 5.4 eine eigene Datenstruktur zur Verfügung, welche die Busnachrichten abstrahiert, s. Abbildung 5.3. Über diese Datenstruktur werden, wie im Abschnitt 4.5 beschrieben, Metadaten der Busbotschaft mitsamt

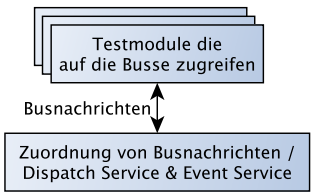


Abbildung 5.3: Abstraktionsebene der Busnachrichten

der Busbotschaft selbst in einer Datenstruktur zur Verfügung gestellt oder als Eingabe angenommen. Bei empfangenen Nachrichten reicht der *interfaced*, welcher die Funktion des *Dispatch Service & Event Service* wahrnimmt, diese Datenstruktur an das Testmodul, welches die Busdaten verarbeitet, weiter. Bei zu sendenden Nachrichten reicht das Testmodul die Busbotschaft in dieser Datenstruktur an den *interfaced* weiter, welcher daraus die tatsächliche Busbotschaft generiert und sendet.
Diese Datenstruktur ist wie in Listing 5.1 definiert.

Listing 5.1: Datenstruktur für die Buskommunikation

```
struct bus_message {            // Message-Q IPC conform struct
        long mtype;             // Message type has to be included
        long timestamp;         // Timestamp of rcv'd message
        int dest;               // Destination bus #
        int id;                 // Message ID
        int msg_len;            // Message length (bytes)
        char message[8];        // Message itself (max. 8 bytes)
};
```

Im ersten Feld *mtype*, welches Aufgrund von Anforderungen aus IPC Messages Queues vom Typ *long* sein muss, wird bei empfangenen Nachrichten die PID des Prozesses gespeichert, welcher die Botschaft weiterverarbeiten soll. Im nächsten Feld *timestamp* wird bei einer empfangenen Botschaft der Zeitpunkt des Empfangs abgespeichert. Im Feld *dest* wird bei empfangenen Botschaften die Bus-ID abgelegt, von der die Botschaft

empfangen wurde und bei gesendeten Botschaften die Bus-ID, auf der die Botschaft gesendet werden soll. Die letzten drei Felder beziehen sich auf die eigentliche Botschaft und stellen die Botschaft-ID, die Länge der Botschaft und die Nutzdaten der Botschaft dar. In Anbetracht der Anforderung der Erweiterbarkeit hinsichtlich neuer Bustypen kann diese zentrale Datenstruktur einmal angepasst werden und greift für alle Nachrichten.
Weiterhin stellt der Interface-Daemon drei *Inter-Process-Communication* (*IPC*) Message-Queues zur Verfügung:

1. *sendq* – die Send-Queue wird von den Testmodulen genutzt indem die Testmodule zu sendende Botschaften in die Datenstruktur *bus_message* adaptieren und auf die Send-Queue legen. Diese wird wiederum vom *interfaced* überwacht und die Nachrichten entsprechend weiterverarbeitet.
2. *rcvq* – die Receive-Queue wird auch von den Testmodulen genutzt indem der *interfaced* empfangene Botschaften für die Testmodule entsprechend auf die Receive-Queue ablegt und dem Prozess des Testmoduls, für den die Nachricht relevant ist, ein *User-Defined Signal 1* (*SIGUSR1*) schickt, sodass dieser die Nachricht aus der Receive-Queue abholt.
3. *configq* – die Config-Queue wird von den Testmodulen genutzt, indem ein Testmodul eine Datenstruktur wie in Listing 5.2 dargestellt auf die Config-Queue legt. Der *interfaced* überprüft periodisch diese Queue auf neue Nachrichten und bearbeitet sie entsprechend. Mithilfe dieser Config-Queue teilt das Testmodul dem *interfaced* mit, welche empfangenen Botschaften für das Testmodul relevant sind und somit vom *interfaced* an das Testmodul geschickt werden sollen.

Listing 5.2: Datenstruktur für die Konfiguration des interfaced

```
struct ifaced_config {
        long mtype;             // 1 or 2, Add or Remove
        pid_t handle_proc;      // PID of Process which handles the packet
        int id;                 // ID of packet to catch
                                // REM: if -1 then all
        int dest;               // Bus # on which to listen for packet (-1 for all busses)
                                // REM: if -1 then all
};
```

Der erste Parameter *mtype* ist entweder 1, falls eine neue Botschaft hinzugefügt werden soll, oder 2 falls eine bereits hinzugefügte Botschaft wieder gelöscht werden soll.

## 5.3 Ablaufsteuerung

Die Ablaufsteuerung unterteilt sich in zwei Teilabschnitte:

- Die *Script Engine*, welche den vom Anwender definierten Testablauf einliest und verarbeitet, sowie
- den *Test Dispatcher und Prozessverwaltung*, welche die Testmodule startet, stoppt und laufende Module verwaltet.

Desweiteren werden in dieser Implementierung auch die *gemeinsam genutzten Strukturen* durch die Ablaufsteuerung zur Verfügung gestellt, da dies einen zentralen Ort hierfür darstellt.

### 5.3.1 Script Engine

Der Anwender der Testplattform hinterlegt den Testablauf in Form eines syntaktisch vorgegebenen Ablaufscripts, vgl. Abschnitt 4.6 in einer Datei. Diese Datei wird von der Testplattform eingelesen und durch die Script Engine interpretiert. Diese nutzt *flex*, um die Eingabedatei in sogenannte Tokens, also kleine Teileinheiten die verarbeitet werden können, zu zerlegen. Die Tokens werden dann an den Parser, welcher durch *bison* realisiert wurde, weitergegeben. *Bison* nimmt die syntaktische Überprüfung vor und erkennt Abfolgen von Token, wobei einer bestimmten Abfolge eine bestimmte Funktion zugewiesen wird. Neben festen Token, die die Schlüsselwörter der Sprache darstellen, *START* wäre z.B. ein solcher fester Token, existieren auch Parameter:

- Ganzzahlen, *NUMBER*-Token, die im Lexer durch den regulären Ausdruck *[0-9]+* erkannt werden und über *yylval.intVal=atoi(yytext);* zu einem Integerwert konvertiert werden.

Es soll auch ein Konstrukt bereitgestellt werden, welches beliebig viele Integerbereiche hintereinander akzeptiert, beispielsweise wird 2-4 6 9 zur Folge 2,3,4,6,9. Dies wird durch den Parser wie in Listing 5.3 dargestellt bewerkstelligt. Auch hier muss beachtet werden, dass die aufgerufene Funktion *numintrange* wieder auf 0 setzt und den alloziierten Speicher des Integerpointers intrange freigibt.

Listing 5.3: Parsen von Zahlenbereichen

```
intrange:
NUMBER {
        // Add number to intrange
        intrange = (int*)realloc(intrange, numintrange * sizeof(int));
        intrange[numintrange] = $1;
        numintrange++;
        $$ = intrange;
}
|
NUMBER TOKMINUS NUMBER {
        int diff;
        int curnum;
        int istart, iend;
        if ($1 < $3) {
                // Right order
                istart = $1;
                iend = $3;
        } else {
                // Swap
                istart = $3;
                iend = $1;
        }
        diff = iend - istart + 1;       // + 1 because we need the start and the end number
        intrange = (int*)realloc(intrange, (numintrange + diff) * sizeof(int));
        for(curnum = istart; curnum <= iend; curnum++) {
                intrange[numintrange + curnum - istart] = curnum;
        }
        numintrange += diff;
        $$ = intrange;
}
|
intrange intrange
;
```

- Dezimalbrüche, *FLOAT*-Token, die im Lexer durch den regulären Ausdruck *[0-9]* .[0-9]+* erkannt werden und mittels *yylval.floatVal=atof(yytext)* zu einem Floatwert konvertiert werden. Da bei einigen Syntaxkonstruktionen Ganzzahlen sowie Dezimalbrüche akzeptiert werden müssen, wird hier mit *floatint*-Token gearbeitet. Dieser ist im Parser wie in Listing 5.4 definiert.

Listing 5.4: Darstellung einer Ganzzahl als Dezimalbruch

```
floatint:
FLOAT           { $$ = $1 }
|
NUMBER          { $$ = (float)$1 }
;
```

- Hexadezimalzahlen, welche aus zwei Hex-Stellen bestehen um ein Byte repräsentieren zu können. Diese Hextupel werden als *HEXTUP*-Token durch den regulären Ausdruck *[a-f0-9][a-f0-9]* erkannt, und über die hextoint-Funktion, welche sscanf nutzt, von einer Hexzahl zu einem Integerwert konvertiert.
  Die Problemstellung besteht darin, wenn ein Hexwert eingegeben wird, der keinen Buchstaben erhält und somit als *NUMBER*-Token erkannt wird. Hierfür wird im Parser ein Konstrukt wie in Listing 5.5 benötigt, um ein *hexnum*-Token zur Verfügung zu stellen.

Listing 5.5: Umwandlung einer Hexzahl $< 10$ in eine Dezimalzahl

```
hexnum:
HEXTUP
|
NUMBER  {
```

```
5         // This number is interpreted as decimal, in fact it is hex - convert it
          // Max. 2 digits so this simple conversion is ok
7         $$ = ($1 / 10) * 16 + ($1 % 10);
  }
9 ;
```

Einige Konstrukte, wie beispielsweise das Definieren von *MSGARRAYs* akzeptieren eine variable Anzahl von Hextupeln als Argumente. Hierfür muss beim Einlesen des ersten Hextupels ein Speicherbereich alloziiert werden, in den das Hextupel abgelegt wird. Bei jedem Auftreten eines weiteren Hextupels muss der alloziierte Speicherbereich um ein Byte vergrößert werden. Die Funktion, die die Hextupel anschließend verarbeitet muss den Speicher wieder freigeben um Memory-Leaks zu vermeiden. Variable Anzahlen von Hextupel werden wie in Listing 5.6 dargestellt durch den Parser akzeptiert.

Listing 5.6: Akzeptieren von beliebig vielen Hextupeln

```
1 varhex:
  varhex hexnum   {
3         $1 = (char*)realloc($1, numargs + 1);
          $1[numargs] = $2;               // Add next item to varhex
5         numargs++;
          $$ = $1;                        // Return Value = arglist
7 }
  |
9 hexnum  {
          tmpchar = (char*)malloc(1);
11        tmpchar[0] = $1;
          numargs = 1;
13        $$ = tmpchar;
  }
15 ;
```

- Stringwerte, welche zwischen hohen Doppelquotes stehen werden als *TEXT*-Token durch den regulären Ausdruck wie in Listing 5.7 gezeigt, erkannt. Dieser Ausdruck stammt aus dem FLEX Manual, A.4.3 Quoted Constructs.

Listing 5.7: Regulärer Ausdruck zur Stringerkennung

```
1 L?\"([^\"\\\n]|
  (\\[’\"?\\abfnrtv])|
3 (\\([0123456]{1,3}))|
  (\\x[[:xdigit:]]+)|
5 (\\u([[:xdigit:]]{4}))|
  (\\U([[:xdigit:]]{8})))*\"
```

Ein Testmodul kann eine flexible Anzahl von Parametern fordern, wie beispielsweise das *LOAD*-Modul beliebig viele Botschafts-IDs und -bereiche als Parameter akzeptiert. Da dieses Testmodul auch nebenläufig ausgeführt werden kann, ist es notwendig vor dem Start des Testmoduls diese Parameter in einen SHM-Bereich zu speichern um die Parameter auch von einem anderen Prozess aus wieder auslesen zu können. Um mehrere Instanzen desselben Testmoduls starten zu können müssen alle Instanzen auf einen anderen SHM-Bereich zugreifen. Hierfür wird der *ftok*-Funktion, die den SHM-Key erstellt, die aktuelle Instanz mitgegeben. Ebenso wird diese Instanznummer dem Testprozess mitgegeben, sodass dieser auch wieder auf den passenden SHM-Bereich zugreifen kann.

Desweiteren stellt die Scriptsprache Schleifen und Konditionen zur Verfügung. Für *LOOP*-Schleifen wird die Funktion *startloop* mit der Anzahl der Schleifendurchläufe (0 für unendlich) als Argument aufgerufen. Die Funktion *startloop* speichert die aktuelle Position des geöffneten Ablaufscripts. Die aktuelle Position ist die erste Zeile der Schleife da die Zeile in dem die Schleife mit *LOOP* eingeleitet wurde bereits eingelesen wurde. Wird im Ablaufscript ein *ENDLOOP*-Token erkannt, so wird überprüft ob die Anzahl der Schleifendurchgänge erfüllt wurde. Wenn dies nicht der Fall ist, wird mittels *fsetpos()* die Dateiposition wieder auf die in *startloop* gespeicherte Zeile gesetzt. Dadurch ist die nächste Zeile, die vom Lexer eingelesen wird, wieder die erste Schleifenzeile.

Bei einem *BREAK*-Token wird solange eine neue Zeile eingelesen ohne den Inhalt an den Lexer zur Interpretation weiterzugeben, bis der *ENDLOOP*-Token erkannt wurde. Erst dann wird mit der normalen Abarbeitung fortgefahren.
Eine Restriktion in dieser Implementierung ist, dass keine zwei Schleifen ineinander schachtelbar sind. Würde man dies wollen müsste man beispielsweise einen Zähler implementieren der die aktuelle Schleifentiefe speichert und für jede Schleife die Anfangsposition speichert.

### 5.3.2 Zur Verfügung gestellte gemeinsam genutzte Strukturen

Da einige Funktionalitäten, allen voran eine gemeinsame Log-Funktion, von allen Testmodulen, sowie von der Ablaufsteuerung gemeinsam genutzt werden, werden diese zentral zur Verfügung gestellt. Die gerade erwähnte Log-Funktion *logit* akzeptiert einen *char* -Pointer als Parameter und schreibt dessen Inhalt mitsamt des aktuellen Zeitstempels in die Logdatei. Ist eine zusätzliche Ausgabe auf der Konsole gefordert wird dort der gleiche Inhalt ausgegeben. Hierbei ist darauf zu achten, dass sowohl Logdatei als auch *stdout* ungepuffert sind, um die richtige Reihenfolge der ausgegebenen Zeilen zu wahren. Der Zeitstempel wird hierbei mit einer Präzision von $\mu$s dargestellt.

Wie in Abschnitt 4.6 des Konzepts dargestellt, wird eine *MSGARRAY*-Funktion zur Verfügung gestellt um den Inhalt bestimmter Botschaften vordefinieren zu können. Hierfür werden diese Botschaften bei Definition in einem SHM-Segment abgelegt. Dieses SHM beinhaltet Datenstrukturen wie in Listing 5.8 dargestellt.

Listing 5.8: Datenstruktur im SHM für MSGARRAYs

```
struct msgarray {
        int msgid;
        int message[8];
        int msglen;
};
```

Im vorliegenden Fall wird die Botschaftslänge fest mit einer Länge von 8 Bytes initialisiert, da aktuell nur der CAN Bus implementiert ist. Wenn zusätzliche Bussysteme implementiert werden ist es denkbar, eine Konstante mit der maximalen Länge einer Busbotschaft einzuführen, die hier für die Größe von *message* verwendet wird, oder die Länge variabel zu gestalten.
*msgid* beinhaltet die ID der Botschaft, *message* die Nutzdaten und *msglen* die Länge der Botschaft. Zum leichteren Zugriff auf diese Strukturen werden zwei Funktionen zur Verfügung gestellt. Zum einen die Funktion addToarray(int msgid, int byte), welche einer abgespeicherten Nachricht mit ID *msgid* ein Byte hinzufügt und als Rückgabewert die aktuelle Länge der Botschaft zurückliefert. Falls die Nachricht noch nicht existiert, wird sie angelegt und das erste Byte hinzugefügt. Diese Funktion wird nur von der Script Engine genutzt, wenn MSGARRAY-Aufrufe im Ablaufscript gefunden werden.
Desweiteren existiert das Gegenstück zu oben genannter *addToArray* -Funktion mit der Funktion *getArray(int msgid)*. Diese Funktion liefert einen Pointer zur Datenstruktur *struct msgarray* zurück, welche die Nutzdaten für die Botschaft mit der ID *msgid* enthält.
Eine Problemstellung hierbei ist, dass die Größe eines SHM-Segments in Linux (im Gegenteil zu IBM AIX oder OpenSolaris) nach der Alloziierung nicht mehr geändert werden kann. Bei Hinzukommen einer neuen Nachricht muss also der gesamte SHM-Bereich in einen lokalen Speicher kopiert werden, der SHM-Bereich freigegeben und anschließend größer wieder alloziiert werden. In Anbetracht der Tatsache, dass es sich hierbei nur um einige Botschaftsdefinitionen handelt, ist dies allerdings kein großer I/O-Aufwand.
Bei Beenden der gesamten Testplattform werden die SHM-Bereiche zentral in einer on_exit()-Routine freigegeben.

### 5.3.3 Test Dispatcher und Prozessverwaltung

Der Test Dispatcher mit der Prozessverwaltung ist neben der Script Engine das Kernstück der Ablaufsteuerung. Bei Starten eines neuen Testmoduls wird die Funktion *prog_wrapper()* aufgerufen, welche als erstes Argument den Testnamen nimmt und als zweites Argument eine 0, falls der Test sequenziell ausgeführt werden soll, also

mittels *CALL*. Die weiteren Argumente entsprechen einer variablen Argumentenliste (...)
Falls der Test nebenläufig ausgeführt werden soll, wird als zweites Argument eine 1 übergeben.
Da Informationen über die laufenden Prozesse evtl. auch von aktuell laufenden nebenläufigen Prozessen benötigt werden können, werden diese Informationen in einem SHM-Bereich abgelegt, welcher eine Aneinanderreihung von Datenstrukturen wie in Listing 5.9 dargestellt ist.

Listing 5.9: Datenstruktur für die Prozessverwaltung

```
1 struct procmgmt {
        pid_t procid;
3       int internalid;
        time_t time_started;
5       char modname[50];
};
```

Im Falle eines nebenläufigen Aufrufs eines Testmoduls wird der SHM-Bereich, wie im vorherigen Abschnitt 5.3.2 für den *MSGARRAY*-SHM-Bereich beschrieben, vergrößert oder verkleinert. Dies ist notwendig, da für jeden neuen nebenläufigen Prozess ein Eintrag in diesem SHM-Bereich stattfindet. Bei einem neuen Prozess wird nach dem fork() die PID des neuen Prozesses registriert, sowie die interne ID, die bei dem *START*-Kommando im Ablaufscript mit angegeben wurde. Diese interne ID wird benötigt um den Prozess über das Ablaufscript auch wieder beenden zu können. Weiterhin wird die Startzeit sowie der Modulname registriert.
Anschließend werden, abhängig vom zu startenden Testmodul, die Parameter aus der variablen Parameterliste ausgelesen und das Testmodul mit diesen Parametern gestartet. Dieses Starten geschieht bei nebenläufiger Ausführung im Kindprozess, da der *fork()*-Aufruf schon bei der Registrierung des Prozesses erfolgt ist.

Wenn ein nebenläufiger Prozess mittels *STOP*-Anweisung im Ablaufscript beendet werden soll, so wird dem Testmodul zunächst ein SIGUSR2 geschickt, sodass dieses ordentlich terminiert. Ist das Testmodul ordentlich implementiert so wird das Auftreten von SIGUSR2 über einen Signalhandler abgefangen und das Modul nach Aufräumen beendet. Wird SIGUSR2 nicht abgefangen, so wird die Standardaktion für dieses Signal durchgeführt, welche das Terminieren des Prozesses darstellt.
Anschließend wird der Eintrag für den Prozess im SHM-Bereich der Prozessverwaltung gesucht und nach Ausgabe von Startzeit, Laufzeit und Programmname für die Logdatei aus dem SHM gelöscht.

### 5.3.4 Implementierung eines neuen Moduls

Beispielhaft wird im Folgenden dargestellt, welche Schritte und Änderungen durchgeführt werden müssen um ein neues Testmodul zu implementieren.
Zuerst wird die eigentliche Funktionalität in einer neuen C-Datei bereitgestellt. Hierbei wird eine Funktion *mod_modulname()* definiert. Anschließend muss das Makefile entsprechend modifiziert werden, damit das Modul in das endgültige Kompilat mit einfließt.
Da die Script Engine die Ablaufscripts einließt und interpretiert, muss sie neue Schlüsselwörter, welche sich auf das neue Testmodul beziehen, erkennen können. Dies geschieht durch die Erweiterung des Lexers. Desweiteren muss die Syntax spezifiziert werden. Dies wird durch die Erweiterung des Parsers erreicht. In Abschnitt 5.3.1 wurde auf die Implementation der Script-Engine und deren Bausteine Lexer und Parser genauer eingegangen.
Im Anschluss an die Script Engine muss die Prozessverwaltung angepasst werden, indem der Modulname und die Anzahl und Typen der Parameter in die Wrapperfunktion prog_wrapper() implementiert werden. Diese Funktion übernimmt das Starten des Moduls entweder als nebenläufigen Prozess oder als sequenziellen Test und registriert den gestarteten Prozess in internen Datenstrukturen. Details zum Test Dispatcher / Prozessverwaltung wurden im Abschnitt 5.3.3 dargestellt.
Es ist zu beachten, dass das Testmodul das Signal SIGUSR2 abfängt und bei Empfang des Signals ordentlich beendet.

## 5.4 CAN-Bus-Ankopplung

Die API, welche die Firma IXXAT zur Ansteuerung der USB-to-CAN-II-Interfaces zur Verfügung stellt, lässt nur exklusive Zugriffe auf das Interface zu. Das heißt, dass nur ein Prozess mit dem Interface kommunizieren kann und Daten senden und empfangen kann.
Die Anforderung A-3 aus Abschnitt 2.2 fordert, dass nebenläufig, d.h. von mehreren Prozessen gleichzeitig, auf die Datenbusse zugegriffen werden kann. Daher ist es sinnvoll, die Busansteuerung komplett in einen eigenständigen Prozess, den *interfaced*, auszulagern, welcher die Ansteuerung der Interfaces übernimmt. Dieser Prozess kommuniziert über IPC-Message-Queues, vgl. Abschnitt 5.2, mit den eigentlichen Testmodulen, welche Buszugriff erfordern.
Die API der Firma IXXAT stellt alle Funktionen zur Verfügung, welche benötigt werden, um den Controller entsprechend zu initialisieren und Datenkommunikation zu betreiben.
Direkt nach dem Starten des *interfaced* wird entweder ein Kindprozess erstellt in dem die Interfaceansteuerung läuft (Daemon-Modus), oder der *interfaced* läuft im Debug-Modus, wobei alle Meldungen zusätzlich zur Logdatei auch auf der Konsole ausgegeben werden.
Die erste Phase nach dem Starten des Daemons ist die Initialisierung aller Controller und das Auslesen deren Fähigkeiten, wie auszugsweise in Listing 5.10 dargestellt. Hierbei ist die aktuelle Anzahl der Controller und Interfaces hart-codiert. Es ist selbstverständlich denkbar, dies über eine externe Konfigurationsdatei dynamisch zu beschreiben.

Listing 5.10: Initialisierungsphase der Hardware des interfaced

```
ctrlConfig.wCtrlClass                   = ECI_CTRL_CAN;
ctrlConfig.u.sCanConfig.dwVer           = ECI_STRUCT_VERSION_V0;
ctrlConfig.u.sCanConfig.u.V0.bBtReg0    = ECI_CAN_BT0_500KB;   // BTR 0: 0x00 is 1000
ctrlConfig.u.sCanConfig.u.V0.bBtReg1    = ECI_CAN_BT1_500KB;   // BTR 1: 0x14 is 1000
ctrlConfig.u.sCanConfig.u.V0.bOpMode    = ECI_CAN_OPMODE_STANDARD | ECI_CAN_OPMODE_EXTENDED;
ctrlIndex = 0;
for (hwIndex = 0; hwIndex < 2; hwIndex++) {
        for (; (((hwIndex == 0) && (ctrlIndex < 2)) || ((hwIndex == 1) && (ctrlIndex < 4)));
           ctrlIndex++) {
                if ((retVal = ECI10A_CtrlOpen(ctrlHandle[ctrlIndex], hwIndex, ctrlIndex % 2,
                    &ctrlConfig)) != ECI_OK) {
                        // [...] Fehlerbehandlung [...]
                }
                sprintf(logmsg, "Successfully opened CAN Controller %d on HW %d, ID %d",
                    ctrlIndex % 2, hwIndex, ctrlIndex);
                logit(logmsg);

                // Getting Controller Capabilities
                if ((retVal = ECI10A_CtrlGetCapabilities(*ctrlHandle[ctrlIndex], &ctrlCaps))
                    != ECI_OK) {
                        // [...] Fehlerbehandlung [...]
                }
                // Ausgabe der Daten der ctrlCaps-Struktur
                // Start the controller
                if ((retVal = ECI10A_CtrlStart(*ctrlHandle[ctrlIndex])) != ECI_OK) {
                        // [...] Fehlerbehandlung [...]
                }
        } // For each controller
} // For each hardware interface
```

Hierbei ist zu erkennen, dass zuerst die Bus-Timing-Register 0 und 1 gesetzt werden, dass die Übertragungsrate 500 kB/s beträgt, ohne Timingverschiebungen. Dies ist die Übertragungsrate die im Fahrzeug standardmäßig verwendet wird. Die anschließende Schleife initialisiert zwei Hardwarecontroller (also zwei Interfaces) mit je zwei Controllern (also zwei Schnittstellen). Im Anschluss an jede Initialisierung wird ausgegeben um welchen Controller es sich handelt. Daraufhin wird der Controller gestartet, d.h. der Transceiver des Controllers aktiviert.

Wenn alle Controller initialisiert sind werden die IPC Message-Queues initialisiert. Hierzu werden IDs als Konstante definiert (ID_SEND_Q, ID_RCV_Q, ID_CONFIG_Q) und mittels *ftok()* ein eindeutiger Schlüssel

für diese Queues erzeugt. Die Erzeugung der Queues erfolgt wie exemplarisch anhand der Send-Queue in Listing 5.11 dargestellt

Listing 5.11: Initialisierungsphase der IPC Message-Queues des interfaced

```
// Send-Queue (ID 1)
sndQ = msgget(ftok("/var/interfaced", ID_SEND_Q), IPC_CREAT);
if (sndQ == -1) {
        sprintf(logmsg, "Can␣not␣create␣send-message-Q:␣%s", strerror(errno));
        logit(logmsg);
        exit(1);
}
sprintf(logmsg, "Send-Message-Queue␣with␣key␣%d␣created",
    ftok("/var/interfaced", ID_SEND_Q));
logit(logmsg);
```

Im Anschluss werden Signal-Handler aufgesetzt, sodass bei Empfangen von SIGHUP oder SIGINT die Controller ordnungsgemäß beendet werden und die IPC Message Queues aufgeräumt werden. Falls die Konstante *DEBUG* gesetzt ist, beispielsweise bei Kompilieren mit *-DDEBUG*, wird SIGUSR2 auch abgefangen und gibt bei Empfangen dieses Signals die aktuelle Weiterleitungsliste für empfangene Nachrichten auf.

Daraufhin setzt eine Endlosschleife ein, in der der *interfaced* zuerst überprüft, ob eine neue Nachricht auf der IPC Config-Queue vorliegt, dann überprüft, ob eine neue Nachricht auf der IPC Send-Queue vorliegt, und anschließend den Empfangspuffer der Controller leert.
Falls eine Nachricht in der Config-Queue vorliegt, wird diese bearbeitet und die interne Weiterleitungsliste entsprechend der Nachricht angepasst. So könnte beispielsweise das *ping*-Modul fordern, dass, wenn auf Bus 3 eine Nachricht mit ID 52 empfangen wird, diese Nachricht an das *ping* Modul, welches gerade die PID 1234 hat, weitergeleitet werden soll.
Falls eine Nachricht in der Send-Queue bereitsteht, so wird diese auf den entsprechenden Bus gesendet. Sollte ein Controller eine Botschaft in seinem Empfangspuffer bereithalten, wird diese ausgelesen und entsprechend der internen Weiterleitungslisten für jeden Prozess, für den die Nachricht relevant ist, auf den Receive-Queue gelegt. Anschließend enthält jeder dieser Prozesse ein SIGUSR1, sodass dieser Testprozess die Receive-Queue ausließt. Hier kommt der Vorteil, dass der Message-Type, das erste Feld in der Datenstruktur auf der Receive-Queue, die PID des empfangenen Prozesses beinhaltet. Die Funktion msgrcv() stellt die Möglichkeit bereit, die Receive-Queue nach dem Message-Type zu filtern. Somit kann das betroffene Modul direkt über die msgrcv()-Funktion nach Botschaften für dieses Modul filtern und selektiv abholen.

## 5.5 Module

Die Testmodule stellen die eigentlichen Testfunktionalitäten bereit. So wird jeder Test in Form eines Moduls zur Verfügung gestellt. Ein solches Modul wird durch den *Test Dispatcher* der Ablaufsteuerung aufgerufen und führt dann eigenständig den Testablauf durch. Die im Rahmen dieser Arbeit implementierten Module sind:

- Ansteuerung der externen Spannungsversorgung zum Testen des Verhaltens des Loggers bei Spannungseinbrüchen
- Ansteuerung eines externen Relaisboards zum Simulieren von Klemmen (KL15 und KL30) sowie Busunterbrechungen und -kurzschlüssen
- Messung der RTT einer Busbotschaft
- Erzeugung von Buslast zum Testen des Verhaltens des Datenloggers unter Last
- Überprüfung der Fehlererkennung bei einem Alive-Counter

### 5.5.1 Ansteuerung der Spannungsversorgung

Das Modul zur Ansteuerung der Spannungsversorgung stellt zwei Funktionen bereit. Die erste Funktion, *mod_power_seqFile()*, ließt eine Spannungsverlaufsdefinition aus einer Komma-Separierten Datei ein. Dabei wird für jeden Sequenzschritt die zweite, und eigentliche Kernfunktion, *mod_power()* aufgerufen. Diese nimmt fünf Parameter:

1. *cmd_id*, welcher beschreibt, was das Modul machen soll:
   - 0 = Netzteilreset durchführen
   - 1 = Ausgang ein- oder ausschalten (je nachdem ob *param4* 0 oder 1 ist
   - 2 = Aktuelle Spannung und Strombegrenzung definieren
   - 3 = Nächstes Element der Spannungsverlaufsdefinition hinzufügen
   - 4 = Spannungsverlauf starten (wobei *param4* die Anzahl der Wiederholungen beinhaltet)
   - 5 = Spannungsverlaufsdefinition löschen
   - 6 = Auslesen der aktuell anliegenden Spannung und der Stromstärke
   - 7 = Spannungsverlaufssequenz stoppen
2. *param1*, welcher, wenn *cmd_id* = 2 oder 3 den Spannungswert beinhaltet
3. *param2*, welcher, wenn *cmd_id* = 2 oder 3 die max. Stromstärke (Strombegrenzer) beinhaltet
4. *param3*, welcher, wenn *cmd_id* = 3 die Dauer des Verharrens auf diesem Sequenzschritt angibt
5. *param4*, welcher, wenn *cmd_id* = 1 oder 4 unterschiedlich genutzt wird

Die Kommunikation zum Netzteil erfolgt mittels eines von Linux unterstützten USB → RS232 Umwandlers. Beim ersten Aufruf wird die RS232-Schnittstelle wie in Listing 5.12 beschrieben, geöffnet.

Listing 5.12: Öffnen der RS232-Schnittstelle

```
if ((ttyFD_power = open(mod_ssp_tty, O_RDWR | O_NOCTTY)) < 0) {
        sprintf(logmsg, "Could not open TTY %s: %s", mod_ssp_tty, strerror(errno));
        logit(logmsg);
        return(1);
}

bzero(&ttyIO, sizeof(ttyIO));
// 9600 baud, 8n1, no handshake, enable receiver
ttyIO.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
// Ignore parity errors
ttyIO.c_iflag = IGNPAR;
ttyIO.c_oflag = 0;      // Raw out
ttyIO.c_lflag = 0;      // Raw in
ttyIO.c_cc[VTIME] = 0;  // Inter-character timer unused
ttyIO.c_cc[VMIN] = 1;   // Block read until 1 char received

tcflush(ttyFD_power, TCIFLUSH);
if (tcsetattr(ttyFD_power, TCSANOW, &ttyIO) != 0) {
        logit("Unable to set TTY attributes");
        return(1);
}
```

Im Anschluss wird der Typ und die Seriennummer des Netzteils abgefragt und in der Logdatei ausgegeben. Die Initialisierung ist dann abgeschlossen. Daraufhin wird das entsprechende Kommando an das Netzteil geschickt. Da der Prozessor des Netzteils ziemlich langsam ist, vgl. Abschnitt 5.1, bricht ein *read()* schon ab, bevor alle Daten empfangen sind. Die Lösung hierzu ist, wie in Listing 5.13 dargestellt, den Lesevorgang so lange zu wiederholen, bis die gesamte, erwartete Länge gelesen wurde.

Listing 5.13: Lesen von Daten aus dem Netzteil

```
bzero(transfer, sizeof(transfer));
```

```
tmpint = 0;
while(tmpint < 49) {
        // Read until we have read 49 bytes
        retVal = read(ttyFD_power, &transfer[tmpint], 1);
        if (retVal == -1) {
                sprintf(logmsg, "Unable to read from power constanter: %s", strerror(errno));
                logit(logmsg);
                return(1);
        }
        if (transfer[tmpint] > 31)      // Ignore stray control characters
                tmpint += retVal;
}
```

Bei der Abfrage nach der aktuell anliegenden Spannung und Stromaufnahme antwortet das Netzteil sehr langsam. Dies ist auch an dem Display ersichtlich, welches sich bei schnellen Spannungsverlaufssequenzen mit Zuständen unter einer Sekunde nicht immer ändert. Aus diesem Grund wird das minimale Intervall für das Auslesen des aktuellen Zustands auf eine Sekunde gesetzt.

### 5.5.2 Ansteuerung des Relaisboards

Wie im vorhergehenden Abschnitt 5.5.1 beschrieben wird auch bei der Ansteuerung des Relaisboards die RS232-Schnittstelle beim ersten Aufruf bzw. Kommando wie in Listing 5.12 dargestellt, initialisiert. Das Relaisboard kann mit weiteren Relaisboards einfach kaskadiert werden, da es bei der Initialisierung von der Testplattform eine ID (immer 0) gesendet bekommt und diesen Initialisierungsbefehl mit der eins erhöhten ID auf einer Ausgangsschnittstelle weiterschickt.
Das Relaisboard implementiert ein Protokoll, bei dem jedes Kommando an das Relaisboard vier Bytes lang ist und das letzte Byte eine XOR-Checksumme über die ersten drei Bytes ist. Weiterhin antwortet das Relaisboard auf jedes Kommando mit einem Status und einer Checksumme hierzu, sodass hier Kommunikationsfehler effizient und zuverlässig über eine einfache Funktion wie in Listing 5.14 dargestellt, erkannt werden können.

Listing 5.14: Überprüfen der Checksumme - Absicherung der Kommunikation

```
int chkchksum(unsigned char *rcvd) {
        if ((rcvd[0] ^ rcvd[1] ^ rcvd[2]) == rcvd[3])
                return TRUE;
        else
                return FALSE;
}
```

Als Argumente für die Funktion mod_relais() dient nur zum einen die ID des zu schaltenden Relais und zum anderen der Zustand in den das Relais versetzt werden soll. Falls die ID = -1 ist, sind alle Relais von der Schaltmaßnahme betroffen. Wie in Abbildung 5.4 ersichtlich, wurde das Relaisboard in ein Gehäuse eingebaut und mit Steckverbindern versehen, wie sie auch beim ARCOS Datenlogger und bei allen Messapplikationen im Fahrzeug verwendet werden. Hierdurch kann ein Testaufbau mit gängigen Kabeln bewerkstelligt werden.

### 5.5.3 Buslasterzeugung

Das Testmodul zur Buslasterzeugung, *mod_load()* bekommt vier Parameter übergeben:

1. Die (virtuelle) Kanalnummer des Zielbusses
2. Die gewünschte Buslast in Prozent
3. Die Instanz zur Zuordnung des SHM-Segments, vergleiche 5.3.1
4. Die Anzahl der Botschaft-IDs die zur Erzeugung der Botschaften genutzt werden sollen um die SHM-Größe richtig zu alloziieren.

Da das verwendete USB-to-CAN-II-Interface von der Firma IXXAT in Tests nur knapp 40 % Buslast generieren konnte, bestand die Lösung darin, zwei Controller zu verwenden, wobei beide an den gleichen Bus


Abbildung 5.4: Eingebautes Relaisboard

angeschlossen sind. Hierfür wurde die Kanalbündelung mittels *CHBUNDLE*, vergleiche Abschnitt 4.6 implementiert. Diese legt die Kanalbündel in einer Datenstruktur, wie in Listing 5.15 dargestellt, in einem SHM-Bereich ab.

Listing 5.15: Datenstruktur zur Beschreibung einer Kanalbündelung

```
struct chb {
        int bundlechannel;
        int ch1;
        int ch2;
};
```

Beim Start des mod_load() Moduls wird zunächst überprüft, ob es sich um einen virtuellen Kanal handelt und in diesem Falle die echten Busnummern aus dem SHM-Segment gesucht. Anschließend wird im lokalen Speicher ein Array von Botschaften in Form von $n$ *struct msgarray*s, vergleiche Listing 5.8, alloziiert. $n$ sei dabei die Anzahl der zu sendenden Botschaften. Dieses Array wird daraufhin entweder mit vordefinierten Botschaften gefüllt, oder, wenn die Botschaft nicht vordefiniert ist, mit zufälligen Werten gefüllt. Anschließend wird SIGALRM abgefangen, da im weiteren Verlauf ein Timer genutzt wird um das periodische Senden einer Botschaft zu veranlassen. Zusätzlich wird das Signal SIGUSR2 abgefangen um das Testmodul zu beenden.
Nun wird die Send-Queue des Interface-Daemons geöffnet um hier die zu sendenden Botschaften ablegen zu können. Der genutzte POSIX per-process-Timer wird jetzt so initialisiert dass dieser, immer wenn eine Nachricht gesendet werden soll, ein SIGALRM sendet. Als Intervall wird die Berechnung wie in Abschnitt 4.4.3 beschrieben, verwendet. Anschließend wird das Konstrukt, wie in Listing 5.16 verwendet, um auf Signale, entweder vom Timer, oder SIGUSR2 zum Beenden, zu warten.

Listing 5.16: Warten auf Signale

```
while(doLoadExit == 0)
        usleep(10 * 1000);
```

Diese Schleife hat den Hintergrund dass, wenn das Modul beendet werden soll, die globale Variable doLoadExit auf 1 gesetzt wird. Ansonsten wird alle 10 Millisekunden diese Variable überprüft, um das Modul auch entsprechend beenden zu können.

### 5.5.4 Ping

Das *Ping*-Modul sendet wie in Abschnitt 4.4.4 beschrieben, eine Botschaft mit ID 50, im Folgenden *Ping_Request* genannt, auf den Bus und erwartet eine Botschaft mit ID 51, im Folgenden *Ping_Reply* genannt, und gleichem

Inhalt als Antwort. Der Inhalt der gesendeten Botschaft ist ein Zeitstempel zur Ermittlung der RTT.
Der Funktion *mod_ping()* werden vier Parameter übergeben:

1. bus_id – die Bus-ID auf die die Botschaften gesendet werden und auf der die Antwortbotschaften erwartet werden.
2. pause_ms – Zeit in ms, die zwischen dem Senden zweier *Ping_Request* Botschaften gewartet wird. Dies ist auch die Zeit die nach dem Senden der letzten *Ping_Request*-Botschaft auf eine Antwort gewartet wird.
3. count – Anzahl der *Ping_Request*-Botschaften, die versendet werden. Falls dieser Parameter 0 ist, bleibt das Modul unendlich aktiv und versendet periodisch *Ping_Request*-Botschaften.
4. beQuiet – ist dieser Parameter 1, so wird nicht für jede Botschaft eine Meldung ausgegeben, sondern nurmehr eine Zusammenfassung vor Terminierung des Moduls ausgegeben.

Zu Anfang werden zuerst die IPC Message-Queues *rcvQ*, *sendQ* und *configQ* gebunden und die PID des Interface-Daemons aus dessen Lockfile ausgelesen. Anschließend werden zwei Signalhandler definiert:

- SIGUSR2, da dieses Signal von der Ablaufsteuerung gesendet wird, wenn das Modul terminieren soll
- SIGUSR1, da dieses Signal vom *interfaced* gesendet wird, wenn eine neue Nachricht für diesen Prozess vorliegt

Als vorbereitende Maßnahme wird der *interfaced* über eine Konfigurationsbotschaft instruiert, Botschaften die mit der ID 51 auf dem definierten Bus empfangen werden, an die laufende Instanz des *mod_ping()* weiterzuleiten. Dies ist notwendig, damit die *Ping_Reply*-Botschaften auch tatsächlich in diesem Modul ankommen.
Nun werden periodisch *Ping_Request*-Botschaften gesendet. Der Zeitstempel wird wie in Listing 5.17 berechnet und repräsentiert die $\mu s$ seit dem 01.01.1970 (UNIX-Zeitstempel).

Listing 5.17: Berechnen des aktuellen Zeitstempels

```
long get_usecs() {
        struct timeval tv;
        long rettime;

        gettimeofday(&tv, NULL);
        rettime = tv.tv_sec * 1000000 + tv.tv_usec;
        return (rettime);
}
```

Nachdem dieser Zeitstempel als *long* repräsentiert ist, wird die ausgehende Botschaft mit einer DLC von *sizeof(long* initialisiert und das Ergebnis von *get_usecs()* als Nutzdaten der Botschaft verwendet. Nach dem Senden der Botschaft wird die Wartezeit von *pause_ms* gewartet. Falls diese Sendeschleife terminiert, also die Anzahl der Durchläufe nicht 0 beträgt, so wird im Anschluss eine Statistik ausgegeben, wieviele Botschaften gesendet und empfangen wurden. Weiterhin wird die durchschnittliche RTT berechnet.
Zur Berechnung der Statistik sowie zur Messung der RTT von jedem einzelnen Ping, werden, wie bereits beschrieben, die *Ping_Reply*-Botschaften von diesem Modul ausgewertet. Hierbei wird der Zeitstempel aus der Botschaft ausgelesen und die Differenz zwischen der momentanen Zeit und der Zeit aus der Botschaft gebildet um die RTT zu ermitteln. Weiterhin wird ein globaler Zähler, der die Anzahl der empfangenen Botschaften zählt, pro empfangener Botschaft um eines erhöht und eine globale Variable, die die Summe aller RTT enthält, angepasst. Über den Quotienten dieser beiden globalen Variablen kann dann bei der Ausgabe der Statistik die durchschnittliche RTT berechnet werden.
Falls das Modul von der Prozessverwaltung über *SIGUSR2* beendet wird, so wird dem interfaced über eine Konfigurationsnachricht mitgeteilt, dass nun keine Busbotschaften mehr an das Modul weitergeleitet werden sollen und anschließend das Modul beendet.
Der ARCOS-Datenlogger wird wie in Listing 5.18 beschrieben konfiguriert, sodass er mit ID 50 empfangene Botschaften kopiert und mit ID 51 wieder zurückschickt.

Listing 5.18: ARCOS Konfiguration zur Unterstützung des ping-Moduls

```
ON EVENT RECEIVE (CAN1, 0 ,50) {
        send(_MESSAGE, CAN1, 0, 51);
}
```

### 5.5.5 Alive-Counter

Der Alive-Counter ist das einzige bis jetzt implementierte Modul, das während seiner Testausführung Anweisungen erhalten kann und ausführen kann. Diese Anweisungen werden im Ablaufscript definiert und beschreiben zu generierende Fehler. Diese Fehler dienen der Überprüfung der im Datenlogger implementierten Alive-Counter Erkennungsmechanismen.
Dem Alive-Counter werden zum Start folgende fünf Parameter übergeben

1. busid – die ID des Busses auf dem der Alive-Counter generiert werden soll.
2. start – der Startwert des Alive-Zählers der die Nutzdaten der Botschaft darstellt.
3. stop – der Endwert des Alive-Zählers. Im darauf folgenden Durchgang fängt der Alive-Zähler wieder bei *start* an.
4. delay_ms – Abstand in ms zweier aufeinanderfolgender Botschaften
5. loops – Anzahl der zu sendenden Botschaften. Falls *loops* 0 ist, so läuft der Alive-Counter unendlich lange.

Zu Anfang werden die drei Message-Queues (*sndQ*, *rcvQ*, *configQ*) des interfaced gebunden um die Buskommunikation zu ermöglichen. Desweiteren wird eine weitere Message-Queue, *acmd*, angelegt, die zur Kommunikation über zu generierende Fehler dient.
Anschließend werden die Signale *SIGUSR1* und *SIGUSR2* abgefangen um zum einen eingehende Nachrichten zu verarbeiten und zum anderen das Modul auf Anfrage der Ablaufsteuerung ordentlich zu beenden.
Nun wird in einem Schleifendurchlauf der Alive-Counter erzeugt und die Botschaft versendet. Die Message-Queue *acmd* soll jede Millisekunde auf neue Nachrichten überprüft werden um gegebenenfalls einen Fehlerwert generieren zu können. Hierbei ist es nicht möglich, eine Schleife zu machen, die die Queue überprüft und anschließend mittels *usleep()* wartet, da ein eingehendes Signal vom interfaced diese Funktion unterbrechen würde und somit fehlerhafte Abstände der gesendeten Botschaften auftreten würden. Zu diesem Zweck wird nach dem Absenden der Alive-Counter-Botschaft der Zeitpunkt für die nächste Botschaft gespeichert. Anschließend wird eine Schleife durchlaufen solange dieser Zeitpunkt noch nicht erreicht ist. Innerhalb dieser Schleife wird überprüft, ob neue Nachrichten in der *acmd* vorliegen. Anschließend wird die verbleibende Zeit bis zum Senden der nächsten Botschaft mithilfe von *usleep* gewartet, höchstens jedoch 1 ms. Die Datenstruktur, wie in Listing 5.19 angegeben, stellt die Nachrichten dar, auf welche der Alive-Counter reagiert und Fehler generiert.

Listing 5.19: Datenstruktur für den Alive-Counter

```
#define ALIVE_ERR_SKIP 1
#define ALIVE_ERR_DUPL 2
#define ALIVE_ERR_PAUS 3
#define ALIVE_ERR_ERRC 4

struct alive_cmd {
        long mtype;     // mtype has to be included is PID of affected process
        short errcmd;   // error command
        int pausetime;  // pause time if errcmd = ALIVE_ERR_PAUS
};
```

*errcmd* stellt einen der in Listing 5.19 definierten Fehlerwert dar, *pausetime* ist nur bei Fehlerwert *ALIVE_ERR_PAUS* relevant und repräsentiert die Zeit in ms, die der Alive-Counter das Senden von Botschaften aussetzen soll.
Wird vom Alive-Counter eine solche Nachricht empfangen, so wird eine globale Variable, die anzeigt, dass jetzt ein Fehler generiert wurde, auf 1 gesetzt und im Anschluss der Fehler generiert.
Auf die Generierung des Fehlers sollte im Normalfall eine Botschaft vom Datenlogger mit ID 53 folgen, da dieser so konfiguriert wird dass bei Fehlern im Alive-Counter eine solche Botschaft versendet wird. Die Folge ist, dass der interfaced dem Alive-Counter-Prozess ein *SIGUSR1* schickt und dieser die *rcvQ* ausließt und somit den Hinweis des Datenloggers über einen Fehler im Alive-Counter erhält. Nun wird entschieden, ob die Nachricht als Folge eines tatsächlich generierten Fehlers empfangen wurde, oder ob die Nachricht empfangen wurde, obwohl kein Fehler generiert wurde und trotzdem die Nachricht empfangen wurde. Als Entscheidungskriterium hierfür wird die oben erwähnte globale Variable, die auf 1 gesetzt wird, falls ein Fehler generiert,

genutzt. Diese wird nach Abarbeitung wieder auf ihren Ursprungszustand 0 gesetzt um Botschaften, die einen vermeintlichen Fehler anzeigen, obwohl keiner vorliegt, erkennen zu können. Als Folge einer empfangenen Fehlerbotschaft vom Datenlogger wird ausgegeben, dass eine solche empfangen wurde und dass diese entweder erwartet oder unerwartet ist.
Das Listing 5.20 stellt die Konfigurationszeilen des ARCOS Datenloggers da, die verwendet werden, um den Alive-Counter zu überwachen. Hier wird ein fester Zählerbereich von 0 bis 14 definiert, wobei 15 einen Fehlerwert darstellt. Als max. Intervall werden 300 ms definiert.

Listing 5.20: ARCOS Konfiguration zur Überprüfung des Alive-Counters

```
CHECKALIVE Alive_1 ('Counter1', 0, 14, 15, 300);

ON EVENT SET (Alive_1){
        Error_Nr=1;
        send(0, CAN1, 0, 53);
}
```

Die Botschaft Counter1 ist mit ID 52 definiert.

## 5.6 Testboard für analoge und digitale Eingänge

Wie in Abschnitt 4.7 beschrieben, verfügen Datenlogger über analoge und digitale Ein- und Ausgänge, die auch getestet werden sollen. Der verwendete CAETEC ARCOS Datenlogger verfügt über acht Digitaleingänge, acht Digitalausgänge und acht Analogeingänge.
Im Konzept sind mehrere Möglichkeiten des Tests angesprochen, realisiert wurde im Rahmen dieser Arbeit die manuelle Testlösung in einfacher Hardware, s. Abbildung 5.5


Abbildung 5.5: ADIO-Testgerät

**Testen der analogen Eingänge** Die analogen Eingänge können durch Drücken des grünen Tasters mit der Versorgungsspannung des Gerätes beschaltet werden, oder eine separate externe Spannung über die roten Buchsen angelegt werden. Hierbei ist darauf zu achten, dass die Eingangsspannung 20V nicht überschreitet, da sonst die in dem Taster integrierte LED Schaden nimmt. Die maximale Eingangsspannung des Analogeingangs des ARCOS Datenloggers beträgt 60V, sodass diese Beschränkung weiter gefasst ist als die des Testgerätes.

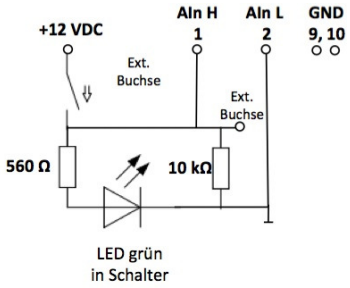


Abbildung 5.6: Beschaltung der Analogeingangstests

Bei allen Analogeingängen ist der Low-Pin auf die gemeinsame Masse gelegt. Um einen Messwert von 0,00 V - 0,01 V bei nicht angeschlossener externer Spannung pro Analogeingang zu erreichen, wird der High-Pin über einen 10k-Pull-Down-Widerstand auf die gemeinsame Masse gezogen.
Die Beschaltung der Analogeingangstests ist in Abbildung 5.6 gezeigt.

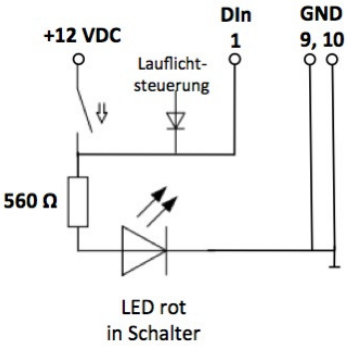


Abbildung 5.7: Beschaltung der Digitaleingangstests

**Testen der digitalen Eingänge** Durch Drücken des roten Schalters wird auf den entsprechenden Eingang die Versorgungsspannung des Testmoduls angelegt. Das Anliegen dieser Spannung wird durch die rote LED im Schalter signalisiert.
Weiterhin besteht die Option durch das eingebaute Lauflichtmodul die Eingänge automatisch zyklisch ein- und wieder auszuschalten. Hierfür wird der grüne Schalter „Auto“ betätigt. Daraufhin werden die Eingänge zuerst aufsteigend und dann absteigend durchlaufen. Die Automatikfunktion wird durch Leuchten der LED im „Auto“-Schalter signalisiert. Der gerade durch die Automatikfunktion geschaltete Ausgang wird durch Leuchten der roten LED des Schalters für den entsprechenden Digitaleingang signalisiert. Die Beschaltung des Digitaleingangstests ist in Abbildung 5.7 gezeigt.

**Testen der digitalen Ausgänge** Die geschalteten Digitalausgänge werden über die blauen LEDs signalisiert. Leuchtet die LED, so ist der jeweilige Digitalausgang geschaltet. Die Beschaltung des Digitalausgangstests ist in Abbildung 5.8 gezeigt.
Mittels der in Listing 5.21 gezeigten Konfiguration werden die Digitaleingänge mit einer Frequenz von 10 Hz abgetastet und die entsprechenden Digitalausgänge in den gleichen Schaltzustand wie die Digitaleingänge versetzt. Hierdurch bekommt der Anwender ein optisches Testresultat, ob die digitalen Ein- und Ausgänge funktionieren. Obwohl die Digitaleingänge eine Abtastrate von 1000 Hz unterstützen würden, unterstützen die Digitalausgänge eine Schaltfrequenz von max. 10 Hz. Daher erfolgt die Abtastung / Schaltung mit 10 Hz.

Listing 5.21: ARCOS Konfiguration zur Ansteuerung der digitalen Ein- und Ausgangstests

```
DIGITAL_OUT DO1 (1, 10, 'D1');
DIGITAL_OUT DO2 (2, 10, 'D2');
DIGITAL_OUT DO3 (3, 10, 'D3');
```

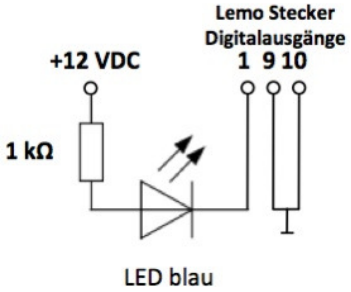


Abbildung 5.8: Beschaltung der Digitalausgangstests

```
4         DIGITAL_OUT DO4 (4, 10, 'D4');
          DIGITAL_OUT DO5 (5, 10, 'D5');
6         DIGITAL_OUT DO6 (6, 10, 'D6');
          DIGITAL_OUT DO7 (7, 10, 'D7');
8         DIGITAL_OUT DO8 (8, 10, 'D8');
```

# 6 Evaluierung

In diesem Kapitel wird die Implementierung auf ihre Tauglichkeit und Konformität mit den am Anfang dieser Arbeit erstellten Anforderungen untersucht. Hierzu wird ein exemplarisches Testszenario konfiguriert und ein Datenlogger mit diesem Testszenario und der Testplattform automatisiert überprüft. Daraufhin werden die Ergebnisse schrittweise analysiert.

## 6.1 Testszenario / Konfiguration

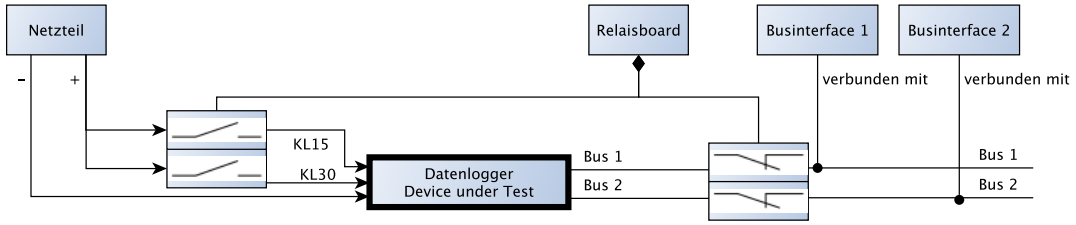


Abbildung 6.1: Skizze des Testaufbaus

Der Testaufbau entspricht der Skizze in Abbildung 6.1. Hierbei sind alle unterstützten Komponenten vertreten. Die verwendete Hardware entspricht der aus dem Abschnitt 5.1. Zusätzlich ist an dem CAN-Bus, der für die Tests verwendet wird, noch ein Vector CANcaseXL angeschlossen, sodass der Datenverkehr auf dem Bus in Echtzeit analysiert werden kann. Ein Bild des Testaufbaus ist in Abbildung 6.2 zu sehen.
Das zum Testen verwendete, kommentierte Ablaufscript ist in Listing 6.1 gezeigt, die dort referenzierte Datei din40839.csv in Listing 6.2.

Listing 6.1: Ablaufscript, welches für Testaufbau verwendet wird

```
; Definiere an welchen Ports Powersupply und Relaisboard haengen
CONFIG RELAIS "/dev/serial/by-id/usb-Silicon_Labs_CP2102_USB_to_UART_Bridge_Controller_0001-if00-port0"
CONFIG POWER "/dev/serial/by-id/usb-Prolific_Technology_Inc._USB-Serial_Controller-if00-port0"
;
LOG "Schalte Spannungsversorgung an, 12.8V, max. 6A"
CALL POWER 12.8 6
CALL POWER ON
SLEEP 1
;
LOG "Schalte KL30 an"
CALL RELAIS 0 ON
SLEEP 3
;
LOG "Schalte KL15 an"
CALL RELAIS 1 ON
SLEEP 1
;
LOG "Pinge bis wir eine Antwort erhalten..."
LOOP
        CALL PING 0 100 1
        IF RETVAL = 0
                BREAK
        ENDIF
ENDLOOP
```

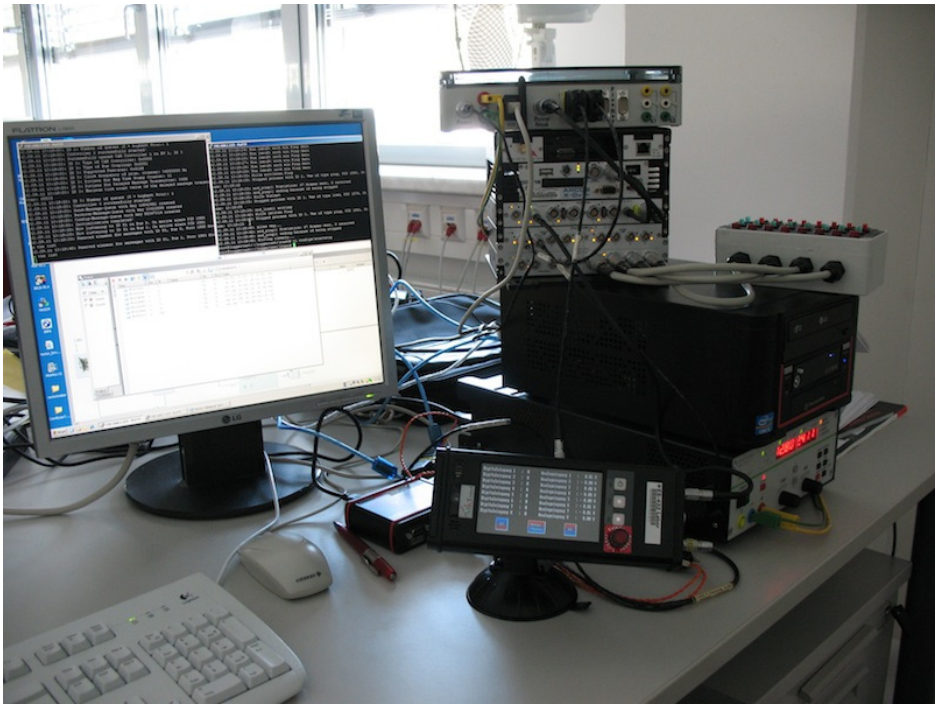

Abbildung 6.2: Die Testplattform im Einsatz

```
LOG "Der Logger antwortet nun..."
SLEEP 1
;
LOG "Aktuelle Stromaufnahme messen"
CALL POWER STAT
SLEEP 1
;
LOG "Teste Alive-Counter"
START 1 ALIVE 0 0 14 200
SLEEP 2
LOG "Ueberspringe einen Wert"
ALIVE 1 SKIP
SLEEP 2
LOG "Wiederhole einen Wert"
ALIVE 1 DUPL
SLEEP 2
LOG "Lasse den Counter fuer 1000ms aussetzen"
ALIVE 1 SLEEP 1000
SLEEP 2
LOG "Generiere Fehlerwert"
ALIVE 1 ERR
SLEEP 2
;
LOG "Jetzt kommt 30% Buslast dazu"
;
; Kanaele 0 und 2 werden zu virtuellem Kanal 100 gebuendelt
BUNDLE 100 0 2
; Botschaften mit ID 1 und 2 definieren wir, der Rest wird zufaellig generiert
MSGARRAY 1 a1 b2 c3 d4 e5 f6 a7 b8
MSGARRAY 2 1a 2b 3c 4d 5e 6f 7a 8b
;
START 2 LOAD 100 30 1-5
;
LOG "Das lassen wir zusammen mit dem Alivecounter so laufen..."
SLEEP 10
;
LOG "Nun weg mit dem Alivecounter"
```

```
62 STOP 1
   SLEEP 5
64 ;
   LOG "Nochmal pingen"
66 CALL PING 0 100 10
   SLEEP 5
68 ;
   LOG "Jetzt Spannungsverlauf DIN 40839"
70 CALL POWER SEQ FILE "configs/din40839.csv"
   CALL POWER SEQ ON 1
72 SLEEP 5
   LOG "Lebt der Logger noch?"
74 CALL PING 0 100 5
   ;
76 LOG "Und weg mit der Buslast"
   STOP 2
78 LOG "Warten und dann raus"
   SLEEP 10
```

Listing 6.2: Spannungsverlaufsdefinition nach DIN40839

```
1 12,6,1
  4.5,6,0.02
3 6,6,2
  9,6,0.01
5 12,6,1
```

Die Spannungsverlaufsdefinition aus Listing 6.2 gibt, wie in Abschnitt 4.4.1 beschrieben, kommasepariert die Werte U,$I_{max}$,t vor. Die erste Zeile bedeutet somit, dass eine Spannung von 12 V anliegen soll, der Strombegrenzer auf 6 A eingestellt wird, und diese Spannung für 1 Sekunde anliegen soll. Wie in Abschnitt 5.1 beschrieben, kann das Netzteil Spannungssequenzdefinitionen nur mit einer maximalen Auflösung von 10ms verarbeiten. Daher wird ein Spannungstal auf der Höhe von 4,5V, welches normalerweise nur 15ms durchfahren werden sollte, in diesem Beispiel 20ms durchfahren. Weiterhin schreibt die DIN 40389 vor, dass die Spannung von 12V auf 4,5V innerhalb von 5ms fällt. Dieser Fall ist wegen den technischen Einschränkungen des verwendeten Netzteils, s. Abschnitt 5.1 nicht beschreibbar, und das Netzteil regelt innerhalb von 1ms auf die Zielspannung hin. Diese beiden Fakten führen zu einem Testablauf unter erschwerten Bedingungen, d.h. wenn der Datenlogger dem Testablauf standhält, dann hält er auch einem Spannungsverlauf nach DIN 40839 statt.

## 6.2 Analyse des Tests

Parallel zur Testausführung wird der Busverkehr auf dem CAN-Bus analysiert und mitgeschnitten. Dies ermöglicht eine detaillierte Testauswertung, wie in Screenshot 6.3 gezeigt.
Das Ablaufscript, wie in Listing 6.1 dargestellt, produziert eine Ausgabe, wie in Listing 6.3 dargestellt.

Listing 6.3: Ergebnis des Testdurchlaufs mit dem Ablaufscript aus 6.1 als Eingabe

```
1 26.10.11 15:19:25.914269: Schalte Spannungsversorgung an, 12.8V, max. 6A
  26.10.11 15:19:25.914344: Calling program power
3 26.10.11 15:19:25.914360: Opening TTY of power constanter
  26.10.11 15:19:26. 21350: Power constanter init successful. Identification: GOSSEN-METRAWATT,SSP32N032RU018P
5 26.10.11 15:19:26. 21398: Power constanter voltage: 12.80 V, max current: 6.00 A
  26.10.11 15:19:26. 71463: Program exited with return value 0
7 26.10.11 15:19:26. 71515: Calling program power
  26.10.11 15:19:26. 71530: Power constanter output turned on
9 26.10.11 15:19:26.121556: Program exited with return value 0
  26.10.11 15:19:27.121648: Schalte KL30 an
11 26.10.11 15:19:27.121672: Calling program relais
  26.10.11 15:19:27.121680: Opening TTY of relais board
13 26.10.11 15:19:27.122445: Initializing relais board
  26.10.11 15:19:27.133372: Relais board with firmware version 11 initialized
15 26.10.11 15:19:27.133408: Turning on relais 0
  26.10.11 15:19:27.141396: Relais board command executed successfully
```

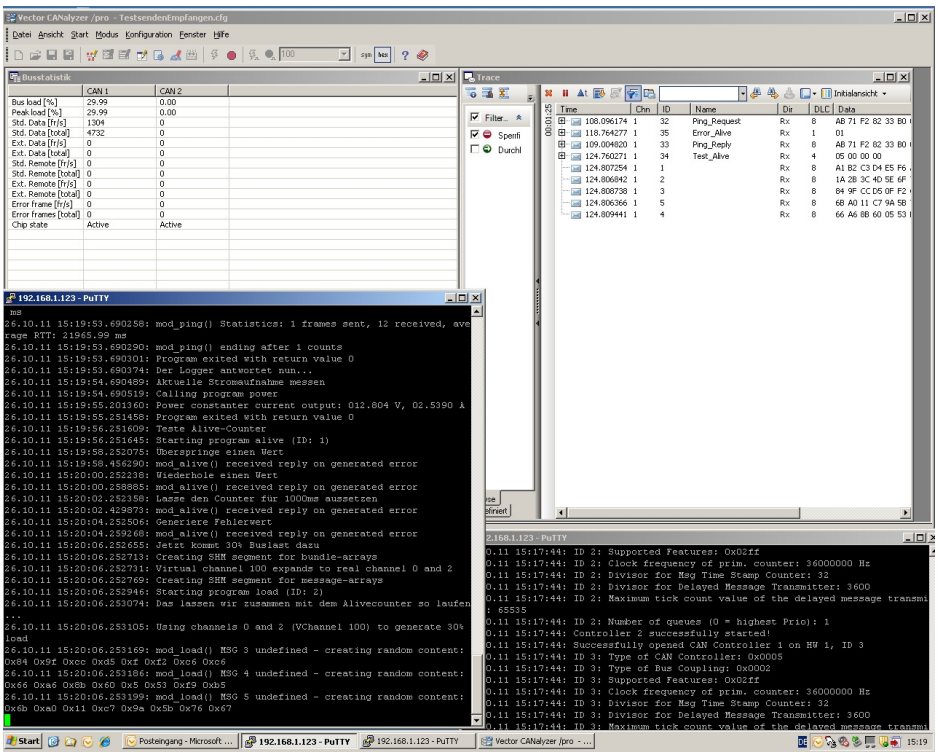


Abbildung 6.3: Screenshot der Testanalyse

```
17 26.10.11 15:19:27.141419: Program exited with return value 0
   26.10.11 15:19:30.141530: Schalte KL15 an
19 26.10.11 15:19:30.141564: Calling program relais
   26.10.11 15:19:30.141573: Turning on relais 1
21 26.10.11 15:19:30.149349: Relais board command executed successfully
   26.10.11 15:19:30.149383: Program exited with return value 0
23 26.10.11 15:19:31.149486: Pinge bis wir eine Antwort erhalten...
   26.10.11 15:19:31.149526: Calling program ping
25 26.10.11 15:19:31.149760: Pinging on Bus# 0 pause 100 ms, 1 times
   26.10.11 15:19:31.249869: mod_ping() Statistics: 1 frames sent, 0 received
27 26.10.11 15:19:31.249906: mod_ping() ending after 1 counts
   26.10.11 15:19:31.249918: Program exited with return value 1
29 [... weitere mod_ping()-Ausgaben ...]
   26.10.11 15:19:53.673907: mod_ping() received answer from logger within 21922.72 ms
31 26.10.11 15:19:53.674230: mod_ping() received answer from logger within 21822.83 ms
   26.10.11 15:19:53.674444: mod_ping() received answer from logger within 21722.81 ms
33 26.10.11 15:19:53.677597: mod_ping() received answer from logger within 21625.74 ms
   26.10.11 15:19:53.681933: mod_ping() received answer from logger within 21529.85 ms
35 26.10.11 15:19:53.685477: mod_ping() received answer from logger within 21433.17 ms
   26.10.11 15:19:53.690258: mod_ping() Statistics: 1 frames sent, 12 received, average RTT: 21965.99 ms
37 26.10.11 15:19:53.690290: mod_ping() ending after 1 counts
   26.10.11 15:19:53.690301: Program exited with return value 0
39 26.10.11 15:19:53.690374: Der Logger antwortet nun...
   26.10.11 15:19:54.690489: Aktuelle Stromaufnahme messen
41 26.10.11 15:19:54.690519: Calling program power
   26.10.11 15:19:55.201360: Power constanter current output: 012.804 V, 02.5390 A
43 26.10.11 15:19:55.251458: Program exited with return value 0
   26.10.11 15:19:56.251609: Teste Alive-Counter
45 26.10.11 15:19:56.251645: Starting program alive (ID: 1)
   26.10.11 15:19:58.252075: Ueberspringe einen Wert
47 26.10.11 15:19:58.456290: mod_alive() received reply on generated error
   26.10.11 15:20:00.252238: Wiederhole einen Wert
49 26.10.11 15:20:00.258885: mod_alive() received reply on generated error
   26.10.11 15:20:02.252358: Lasse den Counter fuer 1000ms aussetzen
51 26.10.11 15:20:02.429873: mod_alive() received reply on generated error
   26.10.11 15:20:04.252506: Generiere Fehlerwert
```

```
26.10.11 15:20:04.259268: mod_alive() received reply on generated error
26.10.11 15:20:06.252655: Jetzt kommt 30% Buslast dazu
26.10.11 15:20:06.252713: Creating SHM segment for bundle-arrays
26.10.11 15:20:06.252731: Virtual channel 100 expands to real channel 0 and 2
26.10.11 15:20:06.252769: Creating SHM segment for message-arrays
26.10.11 15:20:06.252946: Starting program load (ID: 2)
26.10.11 15:20:06.253074: Das lassen wir zusammen mit dem Alivecounter so laufen...
26.10.11 15:20:06.253105: Using channels 0 and 2 (VChannel 100) to generate 30% load
26.10.11 15:20:06.253169: mod_load() MSG 3 undefined - creating random content: 0x84 0x9f 0xcc 0xd5 0xf 0xf2
26.10.11 15:20:06.253186: mod_load() MSG 4 undefined - creating random content: 0x66 0xa6 0x8b 0x60 0x5 0x53
26.10.11 15:20:06.253199: mod_load() MSG 5 undefined - creating random content: 0x6b 0xa0 0x11 0xc7 0x9a 0x5
26.10.11 15:20:16.253211: Nun weg mit dem Alivecounter
26.10.11 15:20:16.253291: Stopped process with ID 1. Was of type alive, PID 3347, Starttime: 15:19:56
26.10.11 15:20:21.253423: Nochmal pingen
26.10.11 15:20:21.253476: Calling program ping
26.10.11 15:20:21.253492: Pinging on Bus# 0 pause 100 ms, 10 times
26.10.11 15:20:21.258490: mod_ping() received answer from logger within 4.93 ms
26.10.11 15:20:21.358743: mod_ping() received answer from logger within 5.14 ms
26.10.11 15:20:21.458509: mod_ping() received answer from logger within 4.85 ms
26.10.11 15:20:21.558387: mod_ping() received answer from logger within 4.67 ms
26.10.11 15:20:21.657643: mod_ping() received answer from logger within 3.87 ms
26.10.11 15:20:21.758491: mod_ping() received answer from logger within 4.66 ms
26.10.11 15:20:21.859293: mod_ping() received answer from logger within 5.40 ms
26.10.11 15:20:21.959538: mod_ping() received answer from logger within 5.63 ms
26.10.11 15:20:22. 58624: mod_ping() received answer from logger within 4.66 ms
26.10.11 15:20:22.159619: mod_ping() received answer from logger within 5.59 ms
26.10.11 15:20:22.254093: mod_ping() Statistics: 10 frames sent, 10 received, average RTT: 4.94 ms
26.10.11 15:20:22.254130: mod_ping() ending after 10 counts
26.10.11 15:20:22.254141: Program exited with return value 0
26.10.11 15:20:27.254294: Jetzt Spannungsverlauf DIN 40839
26.10.11 15:20:27.254344: Opening file configs/din40839.csv with power sequence list
26.10.11 15:20:27.254392: Power constanter Sequence 1: U: 12.00 V, IMAX: 6.00 A for 1.00 seconds
26.10.11 15:20:27.304495: Power constanter Sequence 2: U: 4.50 V, IMAX: 6.00 A for 0.02 seconds
26.10.11 15:20:27.354598: Power constanter Sequence 3: U: 6.00 V, IMAX: 6.00 A for 2.00 seconds
26.10.11 15:20:27.404709: Power constanter Sequence 4: U: 9.00 V, IMAX: 6.00 A for 0.01 seconds
26.10.11 15:20:27.454831: Power constanter Sequence 5: U: 12.00 V, IMAX: 6.00 A for 1.00 seconds
26.10.11 15:20:27.504957: Calling program power
26.10.11 15:20:27.504982: Power constanter sequence started, 1 repetitions
26.10.11 15:20:27.555052: Program exited with return value 0
26.10.11 15:20:32.555198: Lebt der Logger noch?
26.10.11 15:20:32.555254: Calling program ping
26.10.11 15:20:32.555490: Pinging on Bus# 0 pause 100 ms, 5 times
26.10.11 15:20:33. 55860: mod_ping() Statistics: 5 frames sent, 0 received
26.10.11 15:20:33. 55900: mod_ping() ending after 5 counts
26.10.11 15:20:33. 55911: Program exited with return value 1
26.10.11 15:20:33. 55945: Und weg mit der Buslast
26.10.11 15:20:33. 55978: Stopped process with ID 2. Was of type load, PID 3348, Starttime: 15:20:06
26.10.11 15:20:33. 55995: Warten und dann raus
26.10.11 15:20:33. 56018: mod_load() exiting
26.10.11 15:20:43. 56111: Exiting cleanly...
```

Der Testablauf in 6.1 wurde an einer Stelle gekürzt, weil hier im 100ms Abstand ein *ping* durchgeführt wurde. Dies erzeugte in der Logdatei für jeden Ping fünf Zeilen. Da der Datenlogger in diesem Beispiel in ca. 21 Sekunden hochgefahren ist und anschließend antwortet, würden $\frac{\frac{1000ms}{1s}}{\frac{1Durchlauf}{100ms}} \cdot 21s \cdot 5\frac{Zeilen}{Durchlauf} = 1050$ Zeilen in die Logdatei geschrieben werden.
Weiterhin zeigt die Testausführung, dass das erwartete Testergebnis eintritt. Der ARCOS Datenlogger spezifiziert nicht, dass Spannungseinbrüche nach DIN 40839 soweit abgefangen werden können, dass der Datenlogger weiter läuft. Diese Spannungsverlaufsdefinition könnte aber trotzdem von Bedeutung sein, beispielsweise um das Aufstartverhalten des Datenloggers nach Spannungseinbruch zu analysieren.

## 6.3 Vergleich mit Anforderungsanalyse

Im Folgenden werden die Anforderungspunkte aus den Kapiteln 2.2 und 2.3 tabellarisch in Tabelle 6.1 zusammengestellt. In dieser Tabelle wird analysiert und begründet in wiefern die jeweilige Anforderung in der

tatsächlichen Implementierung umgesetzt wurde.

| **Beschreibung** | **Erfüllt / Begründung** |
|---|---|
| Senden von Botschaften auf ein Bussystem | ✓ Durch Busansteuerung möglich, s. Abschnitt 5.4 |
| Empfangen von Botschaften von einem Bussystem | ✓ Durch Busansteuerung möglich, s. Abschnitt 5.4 |
| Unabhängigkeit vom verwendeten Businterface | ✓ Hardwareansteuerung durch separaten Prozess für Busansteuerung, dadurch keine Testmodifikation notwendig wenn anderes Interface verwendet wird, s. Abschnitt 5.4 |
| Nebenläufige Nutzung der Datenbusse | ✓ Der Prozess für die Busansteuerung nutzt die Datenbusse exklusiv, die Botschaften für die Tests nutzen jedoch alle IPC Message-Queues, welche nebenläufig beschrieben und gelesen werden können. Siehe Abschnitt 5.4 |
| Simulation von Spannungsverläufen | ✓ Modul *mod_power* implementiert die Netzteilansteuerung und die Programmierung von Spannungsverläufen, s. Abschnitt 5.5.1 |
| Gemeinsame Nutzung von Spannungsverlaufsdefinitionen | ✓ Modul *mod_power* kann die Spannungsverlaufsdefinitionen auch von externen Dateien, welche aus mehreren Testabläufen referenziert werden können, lesen. Siehe Abschnitt 5.5.1 |
| Simulation des Batteriehauptschalters | ✓ Der Batteriehauptschalter wird durch ein Relais auf dem Relaisboard simuliert. Dieses wird über *mod_relais* angesteuert, s. Abschnitt 5.5.2 |
| Simulation des Zündungszustands | ✓ Der Zündungszustand wird durch jeweils ein Relais für KL15 und ein Relais für KL30 auf dem Relaisboard simuliert. Dieses wird über *mod_relais* angesteuert, s. Abschnitt 5.5.2 |
| Simulation von Busunterbrechungen | ✓ Busunterbrechungen werden durch ein Relais auf dem Relaisboard simuliert. Dieses wird über *mod_relais* angesteuert, s. Abschnitt 5.5.2 |
| Simulation von Buskurzschlüssen | ✓ Buskurzschlüsse werden durch ein Relais auf dem Relaisboard simuliert. Dieses wird über *mod_relais* angesteuert, s. Abschnitt 5.5.2 |
| Automatisches Testen der ADIO | ○ Für die analogen Eingänge trifft dies nicht zu. Jedoch wird der Datenlogger so konfiguriert, dass jeder geschaltene Digitaleingang den entsprechenden Digitalausgang schaltet. Wird die *Auto*-Funktion des ADIO-Testboards genutzt, so schaltet das Testboard automatisiert alternierend die Digitaleingänge, welche durch die Loggerkonfig die Digitalausgänge schalten. Siehe Abschnitt 5.6 Somit können die digitalen Ein- und Ausgänge automatisiert getestet, nicht aber automatisiert ausgewertet werden. Eine alternative Lösung ist in Abschnitt 4.7 beschrieben. |
| Simulation von Alive-Countern und deren Fehlerfälle | ✓ Das implementierte Testmodul *mod_alive*, wie in Abschnitt 5.5.5 beschrieben, simuliert einen Alive-Counter und entsprechende Fehlerfälle. Das Testmodul funktioniert und der erzeugte Alive-Counter wird vom Datenlogger richtig ausgewertet. |
| Erzeugung von Buslast, auch mit definierten Botschaften | ✓ Das Modul *mod_load*, wie in Abschnitt 5.5.3 beschrieben, erzeugt Buslast entsprechend der Nutzervorgaben. Mittels der *MSGARRAY*-Funktion können Botschaften auch definiert werden. |
| RTT-Messung | ✓ Das Modul *mod_ping* überprüft die RTT und wertet diese entsprechend aus. Siehe Abschnitt 5.5.4. |

| | |
|---|---|
| Eingabemöglichkeit zur Beschreibung des Testablaufs | ✓ Der Benutzer der Testplattform erstellt Ablaufscripte in einer für diese Testplattform erarbeiteten Scriptsprache. Die *Script Engine*, siehe Abschnitt 5.3.1, interpretiert diese Ablaufscripte und führt den Testablauf gemäß dieser Ablaufdefinition aus. |
| Protokollierung des Testfortschritts und der Testergebnisse | ✓ Ein Teil der gemeinsam zur Verfügung gestellten Strukturen, s. Abschnitt 5.3.2, ist die *logit*-Funktion, welche, je nach Parameter beim Softwarestart, die Ausgaben entweder nur in eine Logdatei, oder auch auf die Konsole ausgibt. Somit wird der Testfortschritt und die Testergebnisse mitprotokolliert. |
| Geschwindigkeit der Testplattform | ✓ Bei der Auswahl der PC-Komponenten in Abschnitt 5.1 wurde darauf geachtet, dass diese schneller sind, als der zum Einsatz kommende Datenlogger. Desweiteren beträgt die max. gemessene CPU Auslastung der Testplattform im Einsatz bei hohen Buslasten 30 %. |
| Kosten der Nutzung | ✓ Die Kosten der Testplattform sind in der Anforderungsanalyse, vgl. Abschnitt 2.3, verglichen worden und sind im Verhältnis zu den Kosten des Fahrversuches und der Datenlogger sehr gering. |
| Automatisierter Testablauf | ✓ Der Testablauf erfolgt vollautomatisch indem die Testdefinition abgearbeitet wird. Es ist keine Nutzerinteraktion möglich oder erforderlich. |
| Unabhängigkeit hinsichtlich der Anzahl der Busse und dessen Typ | ✓ Neue Busse erhalten neue ID die nur im Ablaufscript angegeben werden müssen, Abstraktion der Busbotschaften sorgt für Typunabhängigkeit, s. Abschnitt 5.2 |
| Unabhängigkeit von der verwendeten Hardwareschnittstelle | ✓ Durch Abstraktion der Busbotschaften muss nur die Busansteuerung, s. Abschnitt 5.4 an die neue Hardwareschnittstelle angepasst werden. Die Ablaufsteuerung und die Testmodule müssen nicht verändert werden. |
| Erweiterbarkeit hinsichtlich neuer Tests | ✓ Der Abschnitt 5.3.4 beschäftigt sich mit der Implementierung eines neuen Testmoduls, was problemlos möglich ist. |
| Mehrere Datenlogger gleichzeitig testbar | ✗ Die aktuelle Implementierung unterstützt über die Scriptsprache nur den Test eines Datenloggers. Es wäre denkbar, mehrere Testprozesse aufzurufen, die jeweils andere Busse nutzen, sodass mehrere Datenlogger gleichzeitig getestet werden können. Hierzu müssten kleinere Modifikationen hinsichtlich der SHM-Strukturen erfolgen, da diese einen festen Schlüssel nutzen, welcher unabhängig von der Prozess-ID des Testprozesses ist und somit mehrere Instanzen den gleichen SHM-Bereich nutzen würden, was zwangsläufig zu Problemen führt. Würde die Schlüsselgenerierung die PID des Hauptprozesses mit einbeziehen und an die Testmodule weitergeben, so könnten mehrere Instanzen parallel betrieben werden. |

Tabelle 6.1: Zusammenfassung der Anforderungen an die Testplattform

# 7 Zusammenfassung und Ausblick

Schlussendlich wird der Erkenntnisgewinn dieser Arbeit festgehalten und ein Ausblick über die zukünftigen Möglichkeiten hinsichtlich Einsatz und Erweiterung dieser Arbeit gegeben.

## 7.1 Zusammenfassung

Die Absicht dieser Arbeit ist zu zeigen, dass es möglich ist, eine generische Testplattform zum Test von Datenloggern zu konzipieren. Hierbei wurden folgende Schwerpunkte und Ziele gesetzt:

- *Erweiterbarkeit auf neue Busse*. Neue Busse müssen mit wenig Aufwand implementiertbar sein, ohne dass die Testmodule hierzu bearbeitet werden müssen. In der vorliegenden Arbeit wurde hierzu eine Abstraktionsschicht zwischen dem Bussystem und den Testmodulen gebildet, sodass die Daten, die das Testmodul auf den Bus senden will in einem internen Format an einen separaten Prozess übermittelt werden, welcher sich um die physikalische Busanbindung kümmert.
- *Erweiterbarkeit hinsichtlich anderer Datenlogger*. Die Testplattform soll in der Lage sein, alle Typen von Datenloggern zu testen. In dieser Arbeit werden daher keine spezifischen Schnittstellen, wie beispielsweise die Ethernet-Schnittstelle des Datenloggers, genutzt. Vielmehr werden die Informationen in der Form generiert, wie sie ein Datenlogger ähnlich von einem Testfahrzeug erhalten würde. Dies garantiert, dass die Testplattform Datenlogger verschiedenen Typs überprüfen kann.
- *Erweiterbarkeit hinsichtlich neuer Tests*. Es wurden exemplarisch im Zuge dieser Arbeit einige Testmodule konzipiert und implementiert. Diese Testmodule können allerdings bei weitem nicht die komplette Bandbreite der Fähigkeiten eines Datenloggers abdecken. Desweiteren nimmt der Funktionsumfang der Datenlogger mit dem technologischen Fortschritt und steigenden Anforderungen stetig zu. Aus diesem Grund ist es wichtig, dass die Testplattform hinsichtlich neuer Tests erweitert werden kann. Testmodule werden in der vorliegenden Arbeit als Module implementiert, wobei neue Module mit geringem Aufwand in das Gesamtkonstrukt integriert werden können.
- Zur *Darstellung der tatsächlichen Nutzbarkeit* ist es notwendig, dass die tatsächlichen Testszenarien mithilfe der Testplattform abgebildet werden können. Hierfür wurden essentielle Testmodule, wie ein RTT-Test, ein Alive-Counter-Test, die Ansteuerung eines externen Relaisboards und eines externen Netzteils sowie ein Buslasttest implementiert. Diese Testmodule zeigen im Gesamtkontext der Arbeit auf, dass einige Anforderungen an die Ablaufsteuerung wie die Erweiterbarkeit, die Unabhängigkeit von den Modulen zu den Bussystemen, etc. erfüllt wurden.
- Hinsichtlich der *Bedienbarkeit* wurde auf eine Scriptsprache wertgelegt, sodass die Testszenarien durch Ablaufscripts definiert werden können, ohne dass der Ablauf im Quellcode vorgeschrieben werden muss und anschließend die Software erneut kompiliert werden muss. Diese Arbeit nutzt eine mächtige Lexer / Parser-Implementation zur Verarbeitung dieser Ablaufscripts. Diese Sprache folgt einer generischen Spezifikation und ist leicht erweiterbar. Hierdurch ist eine flexible Bedienbarkeit durch Fachpersonal geschaffen.

## 7.2 Erkenntnisgewinn dieser Arbeit

Diese Arbeit zeigt, dass es möglich ist, Datenlogger automatisiert zu überprüfen. Hierbei ist es erforderlich, klare architektonische Strukturen zu wahren, um die Übersichtlichkeit zu behalten. Desweiteren würde die Erweiterbarkeit sehr darunter leiden, wenn diese Strukturen durchbrochen würden. Wenn neue Datenlogger

zum Einsatz kommen, oder eine neue Firmwareversion für bestehende Datenlogger aufgespielt werden muss, können diese im Vorfeld mittels der Testplattform überprüft werden. Durch diese automatisierte Überprüfung kann vor dem Praxiseinsatz zeitsparend überprüft werden, ob alle Funktionen, die für den Einsatz relevant sind, mit dem neuen Datenlogger, bzw. der neuen Firmwareversion auch funktionieren. Hierdurch wird vermieden, dass eine neue Firmwareversion, welche einen Fehler in bestehenden Funktionen aufweist, aufgespielt wird und der Fehler erst nach einiger Einsatzzeit entdeckt wird. Hierbei können bereits immense Kosten entstanden sein, da die Erkenntnisse einiger Erprobungsfahrten eventuell nicht korrekt, unvollständig, oder gar nicht aufgezeichnet wurden.
Die vorliegende Arbeit ist eine gute Basis für eine flexible und erweiterbare Testplattform für Datenlogger. Weitere Funktionalitäten können auf diese Plattform gut aufgesetzt werden und die Leistungsfähigkeit dadurch gesteigert werden. Es ist denkbar, dass alle Datenlogger einen Test mit dieser Testplattform durchlaufen müssen, bevor sie praktisch eingesetzt werden um spätere Fehler auszuschließen und somit Kosten und Aufwand einzusparen.

## 7.3 Zukünftige Möglichkeiten

Es wäre denkbar zu allen Funktionalitäten des Loggers ein Testmodul zu schreiben. Dadurch könnte festgestellt werden, dass auch in einer neuen Softwareversion die bestehenden Funktionen nach Spezifikation funktionieren.
Desweiteren können weitere externe Geräte angebunden werden, welche Betriebsumgebungen im Fahrzeug genauer simulieren. Denkbar wäre hier eine Anbindung an einen Klimaschrank um einsatznahe thermische Bedingungen zu simulieren oder die Anbindung einer Rüttelplattform um einsatznahe mechanische Belastungen zu simulieren.
Zusätzlich wäre es möglich zu überprüfen ob die Daten, die auf dem Bus ankommen auch tatsächlich den Daten entsprechen, die das Testmodul gesendet hat. Dies bewirkt, dass Probleme, welche durch fehlerhafte Übertragungen hervorgerufen wurden, zum Testzeitpunkt aufgedeckt werden und nicht fälschlicherweise als Fehler des Datenloggers ausgegeben werden. Dies könnte realisiert werden, indem weitere Hardwareinterfaces an die Bussysteme angeschlossen werden, welche den Busverkehr als Validierungsdaten an die Testplattform übermitteln. Die Testplattform vergleicht daraufhin ob dies auch die Daten sind, die tatsächlich übertragen werden sollen. Unterscheiden sich die gesendeten Daten und die Validierungsdaten so sind entweder fehlerhafte Daten auf den Bus übertragen worden, oder Daten falsch vom Bus ausgelesen und an die Testplattform übermittelt worden.
Eine wichtige zukünftige Erweiterungsmöglichkeit ist die Anbindung weiterer Bussysteme wie beispielsweise FlexRay, LIN oder *Media Oriented Systems Transport* (*MOST*). Die Fa. CAETEC arbeitet bereits an einer entsprechenden Erweiterung der Testplattform auf zusätzliche Bussysteme.
Um die Bedienbarkeit zu vereinfachen ist eine grafische Oberfläche denkbar. So könnte über ein Webinterface der Testablauf grafisch zusammengestellt und visualisiert werden, und aus den Nutzervorgaben ein Ablaufscript generiert werden. Weiterhin könnte der aktuelle Testfortschritt grafisch dargestellt werden. Dies hätte den Vorteil, dass der Benutzer nicht die Syntax der Scriptsprache kennen muss, sondern nur über fachliches Wissen wie sein Testszenario aussehen soll, verfügen muss.
Es wäre ebenfalls denkbar, echte Busdaten aus Fahrzeugen einzuspielen und das Verhalten des Datenloggers zu analysieren. Solche Busdaten können aus automatisierten Aufzeichnungen von Datenloggern im Fahrzeug gewonnen werden, bei denen der Datenlogger ein problematisches Verhalten gezeigt hat. Eine weitere Möglichkeit ist den aufgezeichneten Busverkehr im Vorfeld etwas zu modifizieren um Fehlerszenarien, die der Datenlogger erkennen muss, zu provozieren.
Weiterhin könnten die Testabläufe zum Test der ADIO-Module vollständig automatisiert werden, sodass keine manuelle Interaktion mehr notwendig ist. Dieser Ansatz ist detailliert im Konzept in Abschnitt 4.7 beschrieben.

# Abkürzungsverzeichnis

**ADIO** Analog- and Digital In- and Outputs

**API** Application Programming Interface

**BNF** Backus-Naur-Form

**CAN** Controller Area Network

**CAPL** Communication Access Programming Language

**CF** CompactFlash

**DSC** Dynamische Stabilitätskontrolle, Fahrdynamikregelung

**DUT** Device under Test

**FIFO** First in, first out

**HiL** Hardware-in-the-Loop

**IPC** Inter-Process-Communication

**LIN** Local Interconnect Network

**MOST** Media Oriented Systems Transport

**NRZ** Non Return to Zero

**OSI** Open Systems Interconnection

**PCM** Powertrain Control Module (= Motor- und Getriebesteuergerät)

**PID** Process Identification

**RTT** Round-Trip-Time

**SHM** Shared Memory

**SIGUSR1** User-Defined Signal 1

**SiL** Software-in-the-Loop

# Literaturverzeichnis


[BMW 07] BMW GROUP: *Stationen einer Entwicklung*. BMW Group Presse- und Öffentlichkeitsarbeit, 2007. 154 p.

[Borg 07] BORGEEST, KAI: *Elektronik in der Fahrzeugtechnik*. Vieweg+Teubner Verlag, 2007. 360 p.

[Bäk 06] BÄKER, BERNARD A.: *Moderne Elektronik im Kraftfahrzeug: Innovationen, Neuentwicklungen, Anwendungen, Praxisberichte*. Expert-Verlag, 2006. 260 p.

[GT 06] MANFRED GRUNERT, FLORIAN TRIEBEL: *Das Unternehmen BMW seit 1916*. BMW Group Mobile Tradition, 2006. 596 p.

[MP 07] MAURO PEZZÈ, MICHAL YOUNG: *Software Testing and Analysis*. John Wiley & Sons, Inc., 2007. 488 p.

[SN 04] SYED NABI, JACE ALLEN, ET AL (Herausgeber): *An Overview of Hardware-In-the-Loop Testing Systems at Visteon*. SAE International, 3 2004, http://www.dspace.de/ftp/papers/2004-01-1240_Download.pdf . 2004 SAE World Congress, Detroit, Michigan.