Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Diplomarbeit

Christoph Kutzera

Qualitätsverbesserungen im Bereich Treibersoftware und hardwarenaher Programmierung am Beispiel automatischer Komponenten-, Integrations- und Regressionstests bei der Firma Garz & Fricke

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christoph Kutzera

Qualitätsverbesserungen im Bereich Treibersoftware und hardwarenaher Programmierung am Beispiel automatischer Komponenten-, Integrations- und Regressionstests bei der Firma Garz & Fricke

Diplomarbeit eingereicht
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin : Prof. Dr. rer. nat. Bethina Buth
Zweitgutachter : Prof. Dr. rer. nat. Stephan Pareigis

Abgegeben am 12. Juni 2008

Christoph Kutzera

Thema der Diplomarbeit

Qualitätsverbesserungen im Bereich Treibersoftware und hardwarenaher Programmierung am Beispiel automatischer Komponenten-, Integrations- und Regressionstests bei der Firma Garz & Fricke

Stichworte

Test, Softwaretest, Automatischer Test, Regressionstest, Integrationstest, Komponententest, Hardwaretest, Windows CE, Embedded, hardwarenahe Programmierung, Treiber, CAN

Kurzzusammenfassung

Der Funktionsumfang von kleinen und eingebetteten Systemen nimmt stetig zu. Das Testen dieser Gerätekategorie muss dem Rechnung tragen. Der Testprozess eines Herstellers von Embedded-Geräten mit dem Betriebssystem Windows CE soll mittels ausgewählter Testmethoden verbessert werden. Am Beispiel eines CAN-Treibers werden Tests mit hohem Automatisierungsgrad dargestellt und angewandt. Auf der Komponentenebene als auch auf der Integrationsebene werden Testfälle erstellt. Die Ergebnisse bzgl. Aufwand und Qualität werden reflektiert und auf den Entwicklungsprozess weiterer Komponenten in diesem Bereich übertragen.

Christoph Kutzera

Title of the paper

Quality improvements with driver software development and programming of hardware systems executed by examples on automatic component, integration and regression tests at the company Garz & Fricke

Keywords

Test, Testing, Software test, Automatic test, Regression test, Integration test, Component test, Hardware test, Windows CE, Embedded System, driver, CAN

Abstract

The functionality of small embedded devices gets more complex. The testing of these must take care of this development. The testprocess of a designer and manufacturer of embedded hardware devices with Windows CE shall be improved in quality by choosen testmethods. Tests with high automatism will be illustrated and executed on an example part of the system (CAN driver). On the component and the integration layer testcases will be designed. The results will be analyzed with the focus on effort and quality and will be assigned to the development process of further components in this area.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Der Studiengang Technische Informatik an der Hochschule für Angewandte Wissenschaften Hamburg (HAW) beschäftigt sich mit informationsverarbeitenden Systemen für technische Anwendungen. Computer-Engineering, Prozesslenkung sowie Verteilte Systeme gehören zu den Kernbereichen des Studiengangs.

Der Autor dieser Diplomarbeit ist im Rahmen seiner Tätigkeit bei der Garz & Fricke GmbH mit der Entwicklung von Software für Embedded-Systeme betraut. Embedded (engl., eingebettet) ist die verallgemeinernde Bezeichnung für alle Computersysteme, die in eine Anwendung eingebettet sind, wie z. B. eine Anzeige- oder Steuereinheit auf einem Schiff.

Diese Diplomarbeit beleuchtet das Thema Softwaretests in diesem Bereich. Die große Bandbreite der Embedded-Produkte bringt viele Herausforderungen mit sich. Konkret wird anhand der aktuellen Situation versucht, strukturierte Testmethoden und Prozesse anzuwenden. Der Fokus wird dabei von der Gesamtsicht auf einzelne Teilbereiche, Produkte und Komponenten geführt. Testmethoden werden ausgesucht und an Beispielen veranschaulicht. Anforderungen für die Produkte und Komponenten werden überprüft und im Sinne der Testbarkeit erläutert. Ein zentrales Element ist das Vorbereiten und Erstellen einer Umgebung, die die automatische Ausführung aller Tests erlaubt.

1.1 Motivation

Entwicklungsprodukte in der Softwarebranche unterliegen einer Besonderheit: Man sieht ihnen ihre Fehler und Qualität nicht an. Bei vielen anderen Produkten kann schon ein prüfender Blick einer erfahrenen Fachkraft reichen, um Fehler aufzudecken. So kann z. B. jede elektronische Baugruppe per Sichtkontrolle getestet werden, ob etwa Bauteile fehlen, Kurzschlüsse erkennbar sind oder die Qualität der Lötstellen mangelhaft ist.

Software hingegen ist nur bedingt sichtbar. Der Quellcode offenbart seine Zustände und Logik jeweils nur einer kleinen Gruppe, den meist in ihrem jeweiligen Bereich sehr erfahrenen Softwareprogrammierern. Und das meist erst nach langer Einarbeitungszeit. Schon für kleinere Projekte kommen schnell 1000 und mehr Zeilen Quellcode zusammen. Größere Vorhaben stemmen nicht selten Quelltextlängen von über einer Million Quellcodezeilen unter

Mitarbeit von vielen Programmierern. Selbst nach einem Studium des Quelltextes bleibt die Frage nach der Fehleranfälligkeit ungeklärt. Ob die Logik in jedem Zustand zufriedenstellend arbeitet, lässt sich nur für kleine, isolierte Teile einer Software theoretisch überdenken. Sobald die Zusammenhänge größer werden, und das werden sie mit dem Zusammenkoppeln verschiedener Module sehr schnell, steigt die Anzahl der Möglichkeiten rapide an.

Um weiterhin den Überblick zu behalten und sicherzustellen, dass das Software-Produkt seinen Anforderungen genügt, ist gut geplantes und kontinuierliches Testen sehr wichtig. Knappe Budgets und Zeitpläne veranlassen die Verantwortlichen jedoch oft dazu, dem Testen nicht die nötige Aufmerksamkeit zu schenken. Doch wie viel Aufmerksamkeit ist tatsächlich nötig? Wie sieht ein gut geplanter Test aus? Was ist überhaupt zu testen? Viele Fragen stellen sich einem Entwicklerteam während eines Projekts. Um diese zu beleuchten, ist eine exakte Kenntnis des Projekts, seiner Ergebnisse und Anforderungen, vor allem aber seines Umfeldes nötig.

Verschiedene Umgebungen stellen sehr unterschiedliche Herausforderungen an die Projektierungs- und Testphasen. Große Unternehmen können aufgrund ihrer Mitarbeiterzahl und zur Verfügung stehender Mittel ganze Arbeitsgruppen, bis hin zu Abteilungen, mit Testaufgaben betrauen. Natürlich müssen diese Prozesse auch angemessen geplant und gesteuert werden.

Der Autor versucht Testprozesse in einem mittelständischem Unternehmen umzusetzen. Hier gelten ganz andere Maßstäbe. Mittel und Mitarbeiterzahl sind oft stark begrenzt. Und auch die Produktart zählt nicht zur einfachen Sorte: Der Embedded-Bereich zeichnet sich durch eine hohe Integration aus Hard- und Software aus. Dementsprechend können sich auch beide gegenseitig stark beeinflussen: Fehlverhalten oder einfach nur geringe Abweichung in der Hardware löst unerwartetes Verhalten in der Software aus und umgekehrt.

Genau diese vielfältige Aufgabenstellung stellt die besondere Herausforderung dar. Das Ziel dabei nicht aus dem Auge lassend, Produkte zu entwickeln, die Kunden auch bereit sind zu bezahlen, widmet sich der Autor dem Thema Qualitätsverbesserungen in ausgewählten Bereichen.

1.2 Über diese Arbeit

Das erste Kapitel bietet dem Leser eine Einleitung in das Thema. Die aktuelle Situation und einige Randbedingungen werden kurz umrissen und münden in Motivation und Vision.

Im nachfolgenden Kapitel 2 wird die aktuelle Situation bei Garz & Fricke detailliert beschrieben. Produkte und Entwicklungsprozesse werden beleuchtet, mit dem Ziel, die größten Po-

tentiale für Qualitätsverbesserungen auszumachen. Defizite werden aufgezeigt, bewertet und eine exakte Aufgabenstellung wird erstellt.

Das Kapitel 3 (Grundlagen) beschreibt das Basiswissen für Softwaretests.

Am Beispiel einer Komponente aus einem Gesamtsystem (Kapitel 4) werden Testmethoden der Komponenten- und Integrationstests angewendet. Nachdem Aufbau und Funktion der Beispielkomponente erklärt wurden, werden die Anforderungen im Einzelnen betrachtet. Das Erstellen verschiedener Testfälle und Bewerten der Ergebnisse wird detailliert beschrieben und stellt einen Schwerpunkt dieser Arbeit dar.

Ein weiterer Schwerpunkt liegt auf der Automatisierbarkeit der Testdurchführung und wird im Kapitel 5 behandelt. Die Schaffung der Umgebung mit allen Randbedingungen wird beschrieben und realisiert. Die erstellten Testfälle werden dabei in den Gesamtablauf einbezogen.

Das abschließende Kapitel 6 fasst die bearbeitenden Inhalte und Ergebnisse kurz zusammen. Offene Punkte, aber auch Reflektionen dieser Arbeit auf den weiteren Entwicklungsprozess bei Garz & Fricke, werden aufgegriffen.

Die Planung aller Aufgaben dieser Arbeit wurde in einem Projektplan erfasst und regelmäßig mit dem Ist-Zustand abgeglichen. Der Projektplan ist in einem Gantt-Diagramm im Anhang A dargestellt.

Die weiteren Anhänge enthalten sowohl Dokumentationen zu den im Test benutzten Softwarekomponenten als auch Ausschnitte der Implementierungen der Testfälle selbst.

Auf alle benötigten Spezifikationen und Datenblätter wird im Literaturverzeichnis verwiesen.

Das Glossar enthält Definitionen der verwendeten Fachbegriffe und Abkürzungen.

1.3 Aufgabenstellung / Vision

Die Softwareentwicklung im Bereich der Treiber und hardwarenahen Programmierung soll erhöhten Qualitätsstandards genügen. Gängige Methoden zum Testen von Software sollen ausgesucht und angewandt werden. Die Fehlerrate einer Entwicklung soll damit gesenkt werden können. Der Aufwand muss dabei im Verhältnis zum Produkt stehen. Exemplarisch werden ausgesuchte Testmethoden für eine Komponente des Gesamtsystems angewandt. Automatisierbarkeit, Nachvollziehbarkeit und Wiederholbarkeit stehen dabei im Vordergrund.

1.4 Hinweis zu Markennamen

Alle in dieser Arbeit verwendeten Firmen- und Produktnamen sind Warenzeichen und / oder eingetragene Warenzeichen ihrer jeweiligen Hersteller in ihren Märkten und / oder Ländern.

2 Ist-Situation

Die Beschreibung der aktuellen Situation der Produktenwicklung bei Garz & Fricke steht im Fokus dieses Kapitels. Es wird zuerst das allgemeine Umfeld erläutert, um anschließend auf die Produkte einzugehen. Hard- und Software-Eigenschaften werden in getrennten Kapiteln beschrieben. Die Anforderungen an das Gesamtgerät schließen die Betrachtung der Produkte ab.

2.1 Das Umfeld

Ein Produktbereich der Firma Garz & Fricke ist die Entwicklung von 32-Bit-Embedded-Plattformen für industrielle Einsatzzwecke. Die Produktpalette umfasst verschiedene Computermodule, die den Kern eines Systems darstellen und einzeln verkäuflich sind sowie darauf aufbauende Gesamtlösungen, die weitere Funktionalitäten (z. B. Display) in verschiedenen Ausbaustufen beinhalten. Dazu kommen kundenspezifische Entwicklungen, die oft auf diesen Produktkomponenten beruhen, sich im Detail aber unterscheiden. Garz & Fricke tritt meist als Original Equipment Manufacturer (OEM) auf, d. h., liefert diese Produkte an Kunden, die diese mit einem Mehrwert versehen und als Endprodukt weitergeben. Das bedeutet das die Applikation, also ein Programm, das die tatsächliche Anwendung eines Gerätes realisiert, beim Kunden umgesetzt wird. Garz & Fricke liefert die Lösungen mit einem Standardbetriebssystem. Damit herrscht eine klare Trennung der Verantwortlichkeiten. Die Softwareentwickler von Garz & Fricke passen das Betriebssystem der Hardwareplattform an. Die Applikationsentwicklung erfolgt ausschließlich beim Kunden.

Die weiteren Geschäfts- und Produktbereiche sollen hier noch erwähnt werden, sind aber nicht Teil dieser Arbeit. Produkte für den Verkaufsautomatenbereich basieren auf 16-Bit Mikrocontrollern und einer Software, die eigens für diese Plattform geschrieben wird. Es wird kein Betriebssystem eingesetzt. Ein kleines, separates Softwareteam ist für die Systemprogrammierung und Applikationslogik zuständig.

Die hausinterne Lohnfertigung produziert Eigenentwicklungen sowie fremde Elektronikbaugruppen. Automatisiertes und manuelles Bestücken von Leiterplatten mit Elektronikbauteilen, Lackieren, Montieren und Testen von Baugruppen gehören zu den Kernaufgaben. Das

Testen der produzierten Baugruppen wird am Rande behandelt und in soweit beschrieben, wie es im Zusammenhang mit dem Testen der Software in der Entwicklungsabteilung steht.

2.2 Die Testobjekte

Im Embedded-Bereich betrachtet man die Funktionalität eines Gerätes als Einheit zwischen Hard- und Software. Die Hardware wird oftmals für eine spezielle Umgebung und Aufgabe entwickelt und die Software ist auf diese spezielle Hardware angepasst.

Um die Software zu verstehen, ist eine detaillierte Kenntnis der Hardware nötig. Im nächsten Kapitel werden die Hardware und ihre Einsatz-Varianten vorgestellt. Erst dann folgt eine Beschreibung der Software.

Auch die Aufgabe des späteren Gesamtsystems könnte helfen, ein besseres Verständnis zu entwickeln. Aufgrund der Trennung zwischen Applikationsprogrammierung beim Kunden und Betriebssystemprogrammierung bei Garz & Fricke kann die Gesamtaufgabe jedoch nicht bei Garz & Fricke getestet werden. Oftmals ist diese auch gar nicht bekannt.

2.2.1 Hardware

Die zu testende Software wird auf verschiedenen 32-Bit-Systemen ausgeführt. Häufig findet ein Modulkonzept Verwendung. Der Kern eines Systems mit CPU, RAM und Festspeicher wird auf einer getrennten Platine untergebracht, bei Garz & Fricke Minimodule genannt. Weitere Namen sind System on Module (SoM) oder Computermodule bzw. CPU-Module. Auch der Name Single Board Computer (SBC) findet Anwendung, doch handelt es sich hier um zwei zusammengesteckte Platinen, die allein nicht lauffähig wären, weshalb der Begriff nicht ganz passend ist. Die CPUs im Embedded-Bereich kommen in einem sog. System on a Chip (SoC) vor, d. h. neben der Hauptrecheneinheit werden viele weitere Funktionseinheiten auf demselben Stück Silizium integriert. Controller für SDRAM[1], UART[2], USB[3], SPI[4], I2C[5] sind Standard. Funktionen für Audio, Ethernet und Grafikcontroller, ja sogar 3D-Beschleunigungseinheiten werden in einige Modelle integriert.

Bei Garz & Fricke sind insgesamt sechs verschiedene Minimodul-Baureihen vorhanden, für die Software entwickelt wird bzw. wurde. Die Architekturen sind SH3, SH4, ARM9 und

[1] Synchronous Dynamic RAM (SDRAM)
[2] Universal Asynchronous Receiver Transmitter (UART)
[3] Universal Serial Bus (USB)
[4] Serial Peripheral Interface (SPI)
[5] Inter-Integrated Circuit (IIC)

Produkt/Baseboard	SH4	SH3	ARM9	ARM11
TAZ		x		
GPS Compass	x			
TU500		x	x	
JUPITER			x	x
TOCEOS				x

Tabelle 2.1: Verwendung der Computermodule in verschiedenen Produkten

ARM11, wobei nur noch die ARM Varianten heute weiter entwickelt und vermarktet werden. Einige davon sind zu einem großen Teil pinkompatibel, so dass ein älteres Produkt durch Austausch des Minimoduls neue Funktionen oder bessere Performance erhalten kann. Eine Übersicht über alle Computermodule der Baureihe „Downunder“ findet sich im Anhang D. In Tabelle 2.1 ist die Zuordnung der verschiedenen Minimodule zu den Produkten dargestellt.

Die Minimodule werden auf ein sog. Baseboard aufgesteckt, welches zumindest die Spannungsversorgung, die benötigten Steckverbinder und einige wenige Bauteile zur Ankopplung externer Signale enthält. Die meisten Computermodule führen den gesamten Datenbus nach außen, womit auf dem Baseboard auch weitere Bausteine untergebracht werden können. Dazu zählen Controller Area Network (CAN), zusätzliche UARTs, PCMCIA[6] und häufig produktspezifische Funktionen in programmierbarer Hardware (CPLD[7] oder FPGA[8]).

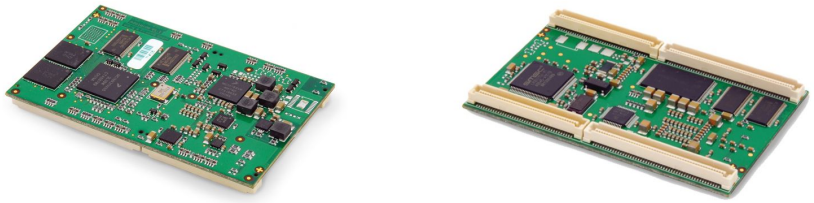


Abbildung 2.1: Ober- und Unterseite des ARM11-Computermoduls Adelaide

Der Fokus der aktuellen Entwicklung richtet sich auf das zuletzt entwickelte Computermodul mit dem Namen Adelaide (Bild 2.1)[11]. Der verwendete SoC ist der i.MX31 von Freescale, basierend auf einem ARM11 Kern. ARM11 ist eine Prozessorarchitektur der Firma ARM Ltd. für mobile Geräte. Der i.MX31 zeichnet sich durch eine hohe Integration vieler Peripherie-Controller bei gleichzeitiger Verwendung von Stromspartechniken aus. Die hohe Leistungsfähigkeit beruht auf der Taktfrequenz von 512 MHz, den 128 kB großen Caches,

[6] Personal Computer Memory Card International Association (PCMCIA)
[7] Complex Programmable Logic Device (CPLD)
[8] Field Programmable Gate Array (FPGA)

dem DDRRAM[9] Interface, der Vector Floating Point Unit (VFP) der MPEG4-Encoding- und der 3D-Einheit.

Die Produkte TAZ, GPS Compass und TU500 sind abgeschlossene Projekte. Hier findet keine Weiterentwicklung statt.

Abbildung 2.2: Vor- und Rückseite des Jupiters

Das Adelaide, im Folgenden ARM11-Modul oder einfach Computermodul genannt, hat mit dem Jupiter-System ein sehr universelles Referenzsystem, auf dem es eingesetzt wird. Das Jupiter-System (Bild 2.2)[11] ist ein Baseboard und bietet ein Display mit Touch sowie sehr viele Schnittstellen: USB, Ethernet, RS-232, CAN, digitale Ein-/Ausgänge, analoge Ein-/Ausgänge, SPI, IIC, RTC[10], programmierbares CPLD und Audio-Ein-/Ausgänge. Neue Projekte werden fast nur noch auf Basis des ARM11-Moduls entwickelt.

2.2.2 Software

Die 32-Bit-Computermodule werden ausschließlich mit Standardbetriebssystemen eingesetzt. Windows CE und Linux stehen den Kunden hierbei zur Auswahl. Die Softwareabteilung von Garz & Fricke konzentriert sich aufgrund ihrer Erfahrung in diesem Bereich auf die Windows CE Entwicklung. Entsprechende Linux Anpassungen werden von Partnerfirmen betreut.

[9] Double Data Rate RAM (DDRRAM)
[10] Real Time Clock (RTC)
[11] Quelle: Garz & Fricke Marketingmaterial

Analog zum pinkompatiblen Minimodul-Konzept muss auch die Software in hohem Maße wiederverwendbar sein. Dazu kann der Quellcode verschieden organisiert sein.

- Im Idealfall kann dieselbe Software auf unterschiedlichen Produkten zum Einsatz kommen, ohne dass sie erneut übersetzt wird. Unterschiedliche Hardware wird dabei zur Laufzeit erkannt und entsprechende Treiber werden geladen.
- Derselbe Quellcode wird mit unterschiedlichen Einstellungen wie zum Beispiel Compilerschalter oder Defines übersetzt. Die übersetzte Software ist dann jeweils nur auf einer Hardwareplattform ausführbar.
- Für eine Hardwareplattform kann es eine separate Quellcodebasis geben.
- Für eine Hardwareplattform kann es mehrere Quellcodes geben, um zum Beispiel besondere Kundenwünsche zu realisieren.

Da die Hardwareplattformen zu einem großen Teil identisch sind, gibt es häufig auch Mischformen. Sowohl für Linux als auch für Windows CE ist der zweite Fall der am meisten benutzte.

2.2.3 Windows CE

Die Firma Microsoft liefert mit Windows for Consumer Electronics ein modularisierbares Betriebssystem mit sehr vielen Softwarekomponenten im Standardlieferumfang. Die unterstützten Architekturen sind ARM, MIPS, SuperH und x86. In punkto Funktionalität und Funktionsvielfalt steht CE dem Desktop Betriebssystem Windows 2000 und XP in kaum etwas nach. Die CE-Application Programming Interface (API) ist eine Untermenge der Win32-API und daher größtenteils kompatibel. Zusätzlich ist es echtzeitfähig und erlaubt die Ausführung auf Systemen mit sehr eingeschränkten Ressourcen. Mindestvoraussetzung ist das Vorhandensein einer Memory Management Unit (MMU). Darüber hinaus kann das System vollständig auf die Beschaffenheit der Hardware angepasst werden. Softwarekomponenten können beliebig hinzu gelinkt oder auch weggelassen werden. Damit lässt sich ein Minimalsystem ohne Display Support auf deutlich unter 2 MB Laufzeitimage-Größe konfigurieren. Systeme mit Display haben hingegen eine typische Größe von 10 bis 30 MB.

Zum Zeitpunkt des Erstellens dieser Arbeit sind für das ARM11-Modul noch nicht alle Treiber und Funktionalitäten fertiggestellt. Daher gibt es hier den größten Testbedarf. Entwickelt werden alle Komponenten für Windows CE 5.0. Auch Windows CE 6.0 wird in Kürze eine Rolle spielen.

Für die Anpassung an die Hardware ist Garz & Fricke zuständig. Das sind im Wesentlichen alle Treiber, die direkten Hardwarezugriff erfordern. Die Tabelle 2.2 gibt eine Auflistung aller Treiber wieder, die bei Garz & Fricke entwickelt wurden bzw. derzeit in Planung sind. Die

Komponente	Art	Interface
Display	GWES Treiber	Windows API
Touch	GWES Treiber	Windows API
OAL/HAL	Kernel Library	Windows API
Flash-Speicherdisk	Block Treiber	Windows API
RS-232	Streaminterface Treiber	Windows API
Ethernet	NDIS Treiber	Windows API, Socket API
USB	USB Treiber	Windows API
CAN	Streaminterface Treiber	Eigene API
SPI	Streaminterface Treiber	Eigene API
IIC	Streaminterface Treiber	Eigene API
RTC	Streaminterface Treiber	Windows API
Digital IO	Streaminterface Treiber	Eigene API
Analog IO	Streaminterface Treiber	Eigene API
System Update	Applikation	GUI & Kommandozeile
Fernkonfiguration	PC Applikation & Service	GUI
CPLD Update	Applikation	GUI & Kommandozeile
CPLD Drehknopf	Streaminterface Treiber	Eigene API

Tabelle 2.2: OEM-Softwarekomponenten

Treiberart gibt an, ob es ein generisches Streaminterface ist (das Streaminterface wird in Kap. 4.3 näher vorgestellt) oder ein spezielles auf die Funktionalität einer Komponente zugeschnittenes Modell ist. Der Interfacetyp gibt an, ob eine vordefinierte API von Microsoft benutzt, eine eigene API erstellt oder andere Bedienungsmethoden wie GUI oder Kommandozeile verwendet werden. Es sollen an dieser Stelle nicht alle diese Komponenten im Detail vorgestellt werden, da dies außerhalb des Rahmens dieser Arbeit liegt. Die Tabelle soll einen Überblick verschaffen, welche Teile des Systems zugreifbar und veränderbar sind. Die meisten Komponenten werden dem versierten Leser bekannt vorkommen und die wenigen Spezialitäten werden hier noch kurz eingeordnet.

Das GWES ist das Grafiksubsystem in Windows CE und enthält Display und Touch Treiber. OEM Adaption Layer (OAL) und Hardware Abstraction Layer (HAL) sind in einer Kernel Library angesiedelt, die ein OEM für sein Gerät bereitstellen muss. Hier sind grundlegende Dinge wie Startup-Code des Betriebssystems, Interruptverarbeitung, CPU-Besonderheiten und Speichermapping der Hardware untergebracht. Das Complex Programmable Logic Device (CPLD)-Update ermöglicht das Beschreiben des CPLDs im laufenden Betrieb von der CPU aus.

Bevor das Betriebssystem zur Ausführung kommt, muss es in den Speicher geladen werden. Dieses erledigt der Bootloader, auch Binary Input Output System (BIOS) genannt. Nach dem

Einschalten muss in erster Linie die CPU und das RAM in einem sog. Startup-Code initialisiert werden. Danach wird das Speichermedium, das das Betriebssystem enthält, initialisiert und das Image in den Arbeitsspeicher kopiert.

Garz & Fricke verwendet für die ARM11-Computermodule RedBoot als Bootloader für Windows CE als auch für Linux. RedBoot steht unter der GNU Public License.

2.2.4 Windows CE Test Kit

Die Entwicklungsumgebung für Windows CE bringt das Windows CE Test Kit (CETK) mit. Dieses besteht aus einem Framework und einigen Testfällen für gängige Standardkomponenten sowie dazugehöriger Dokumentation. Ein Großteil des Quellcodes liegt offen vor, vor allem die Implementierung der Testfälle. Die Funktionalitäten des Frameworks sind nicht im Quellcode einsehbar.

Die Testmodule können per Kommandozeile direkt auf dem Prüfling ausgeführt werden. Sollen mehrere Tests nacheinander ausgeführt werden, lässt sich dies in einer grafischen Oberfläche auf einem PC einstellen. Zur Kommunikation mit dem Prüfling nutzt das CETK Ethernet. Testmodule gibt es für folgende Treiber: RS-232, USB-Speichermedien, SD-Speicherkarten, Touch, Display, RTC, OAL/HAL, Flashdisk. Es gibt noch einige weitere, die aber für die Jupiter-Plattform nicht relevant sind. Alle haben gemeinsam, dass es gängige Komponenten sind und die Schnittstelle in Windows CE berücksichtigt ist.

2.2.5 Anforderungen

Wie bereits in einem vorherigen Kapitel erwähnt, besteht das Endprodukt aus Hard- und Software. Der Kunde erwartet ein funktionierendes Gesamtsystem, das weder einen Hardwaredefekt, noch eine fehlerhafte Software enthält.

Um Hardwaredefekte auszuschließen, muss jedes Gerät vor der Auslieferung getestet werden. Diese Thematik wird am Rande in einem weiteren Kapitel behandelt (Kap. 2.3.2). Ein wichtiger Aspekt wird jedoch sein, dass die Hardwaretests von den hier geleisteten Arbeiten profitieren können, also eine Wiederverwendbarkeit entsteht.

Hauptthema dieser Arbeit ist das Testen der Software. Dazu stellt sich zuallererst die Frage, was getestet werden soll. Eine Auflistung der Softwarekomponenten von Garz & Fricke gibt es in der Tabelle 2.2. Der Großteil dieser Software ist lediglich die Implementierung eines Schnittstellen-Standards, wie z. B. RS-232. Hier geht es darum, alle mit der Hardware möglichen Baudraten zu unterstützen und fehlerfreie Kommunikation mit allen Einstellungen zu betreiben. Zusätzliche Anforderungen für solche Bustreiber können sein:

- Die Kommunikationsgeschwindigkeit sollte stets das höchstmögliche erreichen.
- Die Belastung des Gesamtsystems (Rechenzeit) sollte minimal gehalten werden, um die Applikation bei Berechnungen nicht auszubremsen. Insbesondere zeitbasierte Berechnungen verlieren an Genauigkeit, wenn sie sich die Rechenzeit teilen müssen.

Es sei hier schon angemerkt, dass diese Art der Anforderungen aus Sicht des Testens unbrauchbar ist. Wirklich testbar ist ein Produkt mit diesen Vorgaben nicht, da sich ohne weiteres nicht entscheiden lässt, wann das „höchstmögliche“ erreicht wurde. Dieses Thema und nötige Maßnahmen werden im Kap. 4.2 besprochen.

Diese Anforderungen treffen auch auf die anderen Bustreiber zu: CAN, SPI, IIC, Ethernet und USB. Auch der Flash-Speicherdisk-Treiber hat äquivalente Anforderungen bzgl. Performance und Systemlast.

Bei der Betrachtung des Gesamtsystems ergeben sich einige weitere Punkte.

- Können alle Schnittstellen parallel und ohne Einschränkungen benutzt werden?
- Wie ist das Verhalten unter dieser Last? Ist die Performance noch ausreichend?
- Verläuft ein Dauerbetrieb problemlos?
- Wie robust reagiert das System bei Fehlern?
- Ist die Benutzungsfreundlichkeit und Dokumentation zufriedenstellend?

2.3 Die Ausgangslage

Qualitätssichernde Maßnahmen wie das Testen von Software sollten in der Regel in einer Beschreibung des Entwicklungsprozesses zu finden sein. Garz & Fricke verfügt jedoch nicht über einen definierten und gefestigten Entwicklungsprozess. Prozessbeschreibungen gibt es vereinzelt auf Unternehmensebene für einige wenige Schlüsselprojekte, doch gehen diese nicht auf Softwaretests ein. Daneben gibt es Richtliniendokumente in der Entwicklungsabteilung. Diese beschreiben den Umfang der Softwareentwicklungsaufgaben allerdings so detailliert, dass sie auf die aktuellen Projekte und Werkzeuge kaum anwendbar sind.

In der Praxis sind die Projektverantwortlichen in ihrer Tätigkeit eingespielt. Es werden Releasezyklen mit Testphasen auch mit Einbeziehung der Kunden geplant und durchgeführt. Die Zusammenarbeit mit dem Management ist gut und profitiert davon, dass beide Geschäftsführer mit ihrer technischen Ausbildung Verständnis für Problemsituationen in der Softwareentwicklung entgegenbringen können.

2.3.1 Softwaretests

Während der Entwicklung von Softwarekomponenten ist der jeweilige Entwickler für die Durchführung von Tests verantwortlich. Er implementiert Hilfsprogramme, die die entwickelten Softwarekomponenten ansprechen. Die Entwicklung der Hilfsprogramme erfolgt nicht strukturiert oder nach einem Vorbild. Es liegen sehr viele Testprogramme vor, deren Wiederverwendungsrate sehr niedrig ist. Die Gründe dafür sind verschieden, oftmals jedoch fehlt die Dokumentation für der entsprechenden Testumgebung.

Bei der Freigabe einer Softwareversion ist es ein beruhigendes Gefühl, zu wissen, dass bekannte Fehler sich nicht wieder eingeschlichen haben. Je nach Softwareprojekt existiert ein sog. Abnahmetest, ein Anweisungsdokument, das den Ablauf aller durchzuführender Tests beschreibt.

Für die ARM11 Computermodule gibt es diesen Abnahmetest noch nicht. Für die Vorgängergeneration, die ARM9 Computermodule, sind Abnahmetests vorhanden. Diese sind jedoch nicht ohne weiteres auf die neue Generation übertragbar. Die Softwarebasis der ARM11 Serie wurde komplett von den Vorgängermodellen abgekoppelt: APIs, Bedieninterfaces, Fernzugangsschnittstellen uvm. hat sich grundlegend geändert. Die Tests der einzelnen Module sind teilweise übertragbar, sofern eine Windows API zugrunde liegt. Die Testabdeckung war jedoch nicht sonderlich hoch und müsste erneut überprüft werden. Auch die Einbeziehung der bereits mitgelieferten Tests von Microsoft, dem CETK, ist kaum vorhanden.

Die Abnahmetests sollten aus Zeitgründen nicht von einem Softwareentwickler durchgeführt. Wünschenswert wäre eine hohe Automatisierung, jedoch ist dazu recht viel Equipment notwendig. Neben den Tests der reinen Datenübertragung gibt es noch etliche Tests, die Bedienereingriffe erfordern. Wenn es sich dabei um die Bedienung der Oberfläche handelt, so lässt sich eine Automatisierung mit entsprechenden Bibliotheken verwirklichen. Es kommt aber ebenfalls vor, dass physische Schnittstellen bedient werden müssen, also Kabel ein- und ausgesteckt werden müssen.

2.3.2 Hardwaretests

Die Firma Garz & Fricke produziert ihre Elektronikbaugruppen selbst. Um sicherzustellen, dass jedes produzierte Gerät auch seinen Anforderungen genügt, durchläuft es einige Tests bevor es zum Kunden geliefert werden kann. Für einfachere Elektronikplatinen genügen optische Tests wie Automatische Optische Inspektion (AOI) und die Prüfung mittels In Circuit Test (ICT). Dabei werden die einzelnen Verbindungen auf der Leiterkarte mittels passendem Nadeladapter kontaktiert und mit einem automatischen Messgerätesystem auf ihre Eigenschaften hin geprüft. Dieses Verfahren stößt an seine Grenzen, wenn Mikrocontroller mit

komplizierten Signalfolgen oder Bussysteme, wie Ethernet und USB, getestet werden sollen.

Hier kommen die sog. Funktionstests zum Zuge. Ein Modul wird dabei auf seine Funktion geprüft. Es sei das Beispiel RS-232 Schnittstelle aufgezeigt: Testprogramme auf dem Prüfling und auf dem angeschlossenen PC bedienen die Schnittstelle gegenseitig so, dass alle Leitungen einmal in ihrer maximalen Geschwindigkeit stimuliert wurden.

Diese Hardwaretests sind in Form von Softwaretests vielleicht schon vorhanden. Auch dort musste die Funktion der Schnittstelle schon geprüft werden. Die Softwaretests prüfen jedoch weitaus mehr, als zum Testen der Hardware, speziell der Leiterkarte, benötigt wird. Der wesentliche Unterschied ist, dass es für die Hardwaretests auf eine besonders schnelle Ausführung ankommt.

2.4 Die Werkzeuge

Obwohl der Entwicklungsprozess bei Garz & Fricke nicht besonders dokumentiert ist, versucht die Entwicklungsabteilung sich mithilfe von Werkzeugen und eingespielten Verfahren zu strukturieren.

Der Autor hat schon vor dieser Arbeit einige unterstützende Maßnahmen im Bereich Qualitätsmanagement des Entwicklungsprozesses federführend eingeführt. Dazu gehört das Austauschen des Versionverwaltungssystems und der Fehlerdatenbank. Diese Systeme werden in den folgenden Kapiteln erklärt, wobei die Entwicklungsgeschichte und bisherige Erfahrungen mit einbezogen werden.

2.4.1 Versionsverwaltung

Die Softwareentwicklungsabteilung von Garz & Fricke hat schon früh die Bedeutung einer Versionskontrolle erkannt. Um paralleles Arbeiten an einem Softwareprojekt zu ermöglichen sowie die Historie des Quellcodes aufzuzeichnen, setzte man zuerst das Werkzeug Visual Source Safe von Microsoft ein.

Visual Source Safe glänzt vor allem durch gute Integration in die gängigen Entwicklungsumgebungen von Microsoft und anderer Hersteller. Doch bei zunehmender Projektgröße und -komplexität zeigt es sehr bald Schwächen (siehe auch Übersichts- und Vergleichstabelle [Fish]).

Die Entwickler von Garz & Fricke hatten sich jedoch noch ein anderes Problem geschaffen. Alle Projekte waren in einer Datenbank enthalten. Neben der schlechten Performance kam

es immer wieder zu administrativen Problemen mit der Datenbank. Durch die schlechten Erfahrungen mit diesem Programm, und nicht zuletzt auch durch mangelnde Schulung, waren die Entwickler nicht besonders vertraut mit Versionsverwaltungssystemen.

Vor einigen Jahren wurde deshalb ein neues Werkzeug gesucht. Sehr schnell kristallisierte sich [Subversion] als das Tool der Wahl heraus. Neben der einfachen Handhabung und der Open Source Lizenz überzeugte jedoch die Art der Versionierung. Eine Revisionsnummer wird nur noch auf den gesamten Zustand einer Datenbank vergeben und nicht pro Datei. Viele Entwickler hatten mit dem bisherigen System vor allem dann schlechte Erfahrungen gemacht, wenn unbedingt eine sehr alte Version wiederhergestellt werden musste. Durch die typischen Umstrukturierungen, die die Projektdateien und Verzeichnisse üblicherweise in mehreren Jahren erfahren, war es sehr mühselig bis unmöglich, einen alten Zustand wiederherzustellen. Um nicht die selben Fehler wie vorher zu machen, wurden die Projekte nun in separate Datenbanken („Repositories“) gelegt. Zusätzlich wurde über alle neuen Möglichkeiten von Subversion geschult. Zum Zeitpunkt des Verfassens dieser Arbeit liegt die Einführung etwa drei Jahre zurück. Nach dieser Einsatzzeit kann die Einführung von Subversion als ein Erfolg betrachtet werden.

Subversion protokolliert alle Änderungen am Quellcode mit dem Kommentar des Entwicklers. Mit ein wenig Disziplin und einigen Regeln werden Arbeitsabläufe transparenter für die Teamkollegen. So ist jeder Entwickler angehalten, folgende Punkte zu beachten:

- Zu einer Quellcodeänderung gehört immer ein Kommentar.
- Ein Kommentar sollte mit dem Komponentennamen beginnen und dann eine Auflistung der Änderungen enthalten. Die Beschreibung der Änderung sollte ein Mittelmaß zwischen sehr detailliert und sehr kurz sein.
- Ein Commit-Vorgang (Übergeben einer Quellcodeänderung ins Repository) sollte nur ein Thema oder eine (Teil-) Aufgabe enthalten. Änderungen des Quellcodes für verschieden Zwecke, zum Beispiel Hinzufügen von Komponente X und Beheben des Fehlers Y, sollen in separaten Commit-Vorgängen realisiert und kommentiert werden.
- Quellcode in der Datenbank muss in jedem Fall kompilierbar sein.

Vor allem der dritte Punkt erlaubt eine gute Nachvollziehbarkeit der Arbeitsschritte. Beim Durchsuchen der Versionshistorie können zu jedem einzelnen Commit-Vorgang auch die jeweiligen Änderungen im Quellcode (auch für verschiedene Dateien) angezeigt werden. Das erlaubt sogar das Wiederverwenden von Änderungen. Gut abgegrenzte Änderungen, wie zum Beispiel das Beheben eines Fehlers, lassen sich dann sogar automatisch auf einen Softwarezweig anwenden, der zwar die gleiche Softwarebasis hat, aber den fehlerhaften Code noch enthält. Diesen Vorgang nennt man Merge (engl., zusammenführen).

Für den Fall der Fehlerbehebung kann man die Informationsverknüpfung noch etwas weiter denken: Wenn eine Änderung aus Gründen einer Fehlerbehebung gemacht wurde, so ist auch interessant, welcher Fehlerfall dazugehört. Diese Information steckt in einer Fehlerdatenbank, welche im nächsten Kapitel besprochen wird.

2.4.2 Fehlerdatenbank

Wenn Fehler in der Software gefunden werden, sollten diese strukturiert dokumentiert werden. Wichtig dabei ist, welche Version(en) der Software betroffen ist (sind), wer den Fehler gefunden hat, wie er sich reproduzieren lässt usw. Alle diese Informationen sollte derjenige liefern, der den Fehler gefunden hat. Der Eintrag in eine Fehlerdatenbank erlaubt aber durchaus weitere Möglichkeiten, als nur den Fehler zu dokumentieren. Ein neuer Fehlereintrag kann eine Benachrichtigung (für gewöhnlich eine Email) auslösen. Der Projektleiter oder das Entwicklerteam entscheiden dann über das weitere Vorgehen. Dem Eintrag kann ein Verantwortlicher Entwickler zugewiesen werden, womit sich die Entwickler untereinander synchronisieren können um nicht einen Fehler versehentlich mehrmals zu bearbeiten. Der Verantwortliche dokumentiert seine Arbeitsschritte, Einschätzungen, Empfehlungen und Lösungen ebenfalls unter diesem Eintrag. So entsteht eine Historie für jeden Fehlereintrag. Verschiedene Zustände eines Eintrags (New, Open, Closed, ...) helfen dabei die Übersicht bei sehr vielen Einträgen zu behalten. So wird die Projektleitung sich z. B. für alle neuen und offenen Fehler interessieren. Mit der Zeit wächst eine Fehlerdatenbank zu einer Wissensdatenbank heran, wo auch die bereits abgearbeiteten Fehler für z. B. neue Projekte interessant sind.

Bei Garz & Fricke wurde bis vor wenigen Monaten vor Beginn dieser Arbeit das Produkt Testtrack der Firma Seapine eingesetzt. Testtrack wird für exakt die oben genannten Aufgaben beworben und seine Funktionalität setzt diese auch um. Vorausgesetzt, alle Beteiligten dokumentieren ihr Vorgehen auch sorgsam. Hier zeigten sich jedoch die Schwächen von Testtrack. Die Mitarbeiter waren mit der Bedienung nicht zufrieden und benutzten es daher nicht in dem Maße, wie es nötig gewesen wäre. Das größte Manko allerdings war die fehlende Möglichkeit, einen Fehlerfall mit Änderungen im Sourcecode zu verknüpfen. Grundsätzlich ist dies bei Testtrack zwar vorgesehen, aber das Versionskontrollsystem Subversion wird nicht unterstützt. Auch war der Hersteller nicht in der Lage mitzuteilen, ob und wann eine Unterstützung erwartet werden kann.

Bei der Suche eines Nachfolgers wurde auf die Punkte der Bedienbarkeit und der Verknüpfung zum Sourcecode besonders geachtet. Das noch sehr junge Projekt [Trac] stach dabei mit einer sehr guten Integration von Subversion hervor. Trac erlaubt es, Fehlereinträge und Sourcecodeänderungen direkt zu verknüpfen, und zwar in beide Richtungen. In der Praxis

funktioniert das so: Wenn der Entwickler einen Fehlereintrag bearbeitet hat und Änderungen des Sourcecodes in das Subversion-Repository übergibt (Commit), so gibt er im Kommentar zusätzlich die Fehlernummer und einen Status mit an. Im Hintergrund aktualisiert das Trac-System den referenzierten Fehlereintrag mit den neuen Informationen: Kommentar, Revisionsnummer und Status. Der Fehlereintrag listet somit alle Arbeitsschritte des Entwicklers vollständig auf. Der Kommentar eines Commit erlaubt das schnelle Nachvollziehen der Änderungen und die Revisionsnummer erlaubt das Darstellen der exakten Änderungen im Sourcecode (häufig als Diff-View bezeichnet) Zeile für Zeile. Andersherum liefert die Historie in Subversion zu jedem Commit eine Fehlernummer. Aus beiden Systemen ist ein Querreferenzieren mit einfachen Mitteln möglich.

2.4.3 Projektmanagement

Das Adelaide-Projekt war eines der ersten, in dem Trac zum Einsatz kam. Da das Projekt zum Zeitpunkt der Einführung noch sehr jung war, konnte auf eine Übernahme von Fehlereinträgen aus dem alten System verzichtet werden. Das Trac-System ist jedoch mehr als nur eine Fehlerdatenbank. Es integriert die Versionsverwaltung, ein Wiki, eine Meilensteinverwaltung und ein Ticketsystem. Das Ticketsystem wird benutzt, um Fehlereinträge, wie im vorhergehenden Kapitel beschrieben, abzuspeichern. Ein Ticket muss allerdings nicht zwingend ein Fehlereintrag sein. Jede beliebige Aufgabe lässt sich in einem Ticket ablegen. Dazu bringt das System in seiner Standardkonfiguration drei Tickettypen mit: defect, enhancement und task. Damit lassen sich auch ganze Arbeitspakete in das System einpflegen, womit auch die normale Implementierungsarbeit, die den größten Teil darstellen sollte, genauso nachvollziehbar wird, wie die Fehlerbehebung. Jeder Änderung im Sourcecode kann zusätzlich zum Kommentar auch die Beschreibung, Metainformationen und Historie eines Tickets zugeordnet werden. Im Nachhinein lässt sich damit die oft gestellte Frage, warum eine Änderung am Sourcecode durchgeführt wurde, beantworten. Vor oder während der Implementierung stellt sich dagegen die Frage: Welche Arbeiten wurden bereits erledigt? Auch das kann damit sehr detailliert beantwortet werden.

Jedes Ticket hat als eine wesentliche Strukturierungsmöglichkeit einen Zustand. Dies erlaubt das schnelle Auffinden von Tickets, die z. B. noch abgearbeitet werden müssen. In Abbildung 2.3 ist das Zustandsdiagramm eines Tickets dargestellt und im Folgenden kurz beschrieben. Nach Erstellung eines Tickets hat es den Zustand `new`. Wenn ein Entwickler die Verantwortung übernimmt oder zugewiesen bekommt, befindet sich das Ticket im Zustand `assigned`. Die informativen Zustände `infoneeded` und `inwork` deuten an, dass zusätzliche Informationen benötigt werden oder das Ticket gerade aktiv bearbeitet wird. Der Zustand `closed` bedeutet, dass das Ticket nicht mehr relevant ist, weil z. B. der ein Fehler behoben wurde oder entschieden wurde, dass er nicht behoben wird. Der Zustand `intest` bedeutet, dass prüfende Arbeiten noch zu erledigen sind.

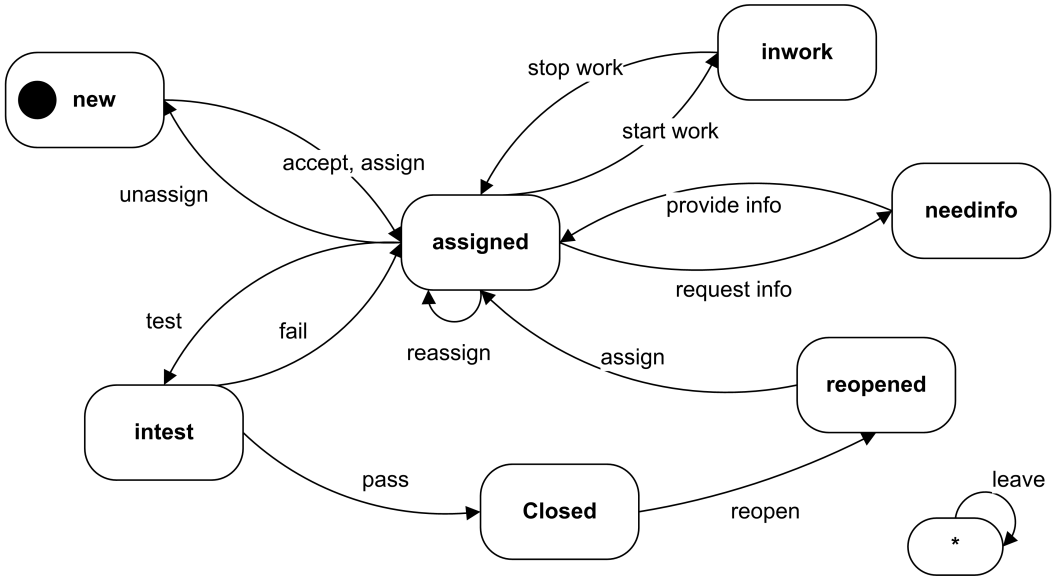


Abbildung 2.3: Workflow eines Tickets im Trac-System

Die Meilensteinverwaltung in Trac ist wiederum mit dem Ticketsystem gekoppelt. Pro definierten Meilenstein können diesem beliebig viele Tickets zugeordnet werden. Aus Sicht der Projektplanung lässt sich damit der Überblick bewahren, welche Arbeiten noch zu erledigen sind, bevor ein Meilenstein erreicht wird. Auf Ticketebene gibt es Zustände, die Aufschluss darüber geben, ob ein Ticket noch relevant ist. Auf Meilensteinebene werden einfach alle zugeordneten Tickets zusammengezählt, wobei sie nach ihrem Zustand unterschieden werden. Wenn alle Tickets eines Meilensteins closed sind, ist auch das Ziel des Meilensteine erreicht.

Das integrierte Wiki unterstützt alle Bereiche von Trac, indem Beschreibungen von Meilensteinen und Tickets in der einfach zu erlernenden Wiki-Syntax erfolgen können. Dadurch lassen sich insbesondere die einfachen und effektiven Verlinkungsmechanismen eines Wikis nutzen. Neben dem bekannten CamelCase[11] lassen sich auch Links zu Tickets und Sourcecodeänderungen einfach umsetzen. Ein eigener Wiki-Bereich dient der allgemeinen Dokumentation eines Softwareprojektes.

Durch die Integration dieser Teile fallen auch sehr nützliche Funktionen ab, wie eine Suche oder eine Zeitlinie. Ebenso, wie die Suche alle Teile des Systems indiziert (Wiki, Tickets, Subversion-Kommentare), so zeigt die Zeitlinie die Veränderungen in allen diesen Teilen

[11] Ein CamelCase ist ein zusammengesetztes Wort, bei dem beide Wortanfänge groß geschrieben werden, woraus ein Wiki automatisch einen Link erstellt, ohne dass der Autor sich spezieller Sprachkonstrukte bedienen muss.

chronologisch an. Dies hilft insbesondere dann, wenn ein Entwickler sich über die Aktivitäten der letzten Tage informieren will.

2.5 Aufgabenstellung

Die Beschreibung der Ist-Situation in diesem Kapitel fördert viele Arbeitspakete zutage. Als größter Punkt sei ein wiederholbarer und weitestgehend automatisierter vollständiger Test zu nennen. Dieser sollte alle von Garz & Fricke entwickelten Komponenten testen können.

Für den Rahmen dieser Diplomarbeit wird eine Komponente herausgesucht, an der typische und geeignete Testmethoden angewandt werden können. Der Treiber für die CAN-Schnittstelle eignet sich dafür gut, da hierfür noch kein Test existiert. Zudem ist sein Aufbau sehr typisch, es gibt daher Parallelen zu vielen anderen Treibern.

Die Grundlagen der Softwaretests werden erarbeitet und dem Leser zum Verständnis präsentiert. Komponententests und Integrationstests werden am Beispiel eines Teilsystems detailliert beschrieben und durchgeführt. Testmethoden werden herausgesucht und angewandt.

Die erstellten Testfälle werden im Kontext der Automatisierung nochmal aufgegriffen. Eine Umgebung, in der diese Testfälle ohne Benutzerinteraktion ausgeführt werden können, soll erstellt, dokumentiert und getestet werden.

3 Grundlagen Softwaretests

3.1 Begriffsdefinition

Das grundlegende Verständnis für die Aufgabe des Softwaretestens ist zwingende Voraussetzung für alle Beteiligten, also alle Softwareentwickler und zumindest deren Vorgesetzte. Leider gibt es zwei Punkte, die dem erschwerend entgegen wirken:

- Vorlesungen im Bereich Softwaretests sind auch zum Zeitpunkt des Verfassens dieser Arbeit noch nicht selbstverständlich in Informatik-Studiengängen. Durch die auch weiterhin steigende Komplexität wird die Bedeutung zunehmend erkannt und entsprechende Angebote erstellt bzw. in bestehende Vorlesungen eingearbeitet. Doch lässt sich damit nicht das Defizit der jetzt berufstätigen Softwareentwickler so schnell aufholen.
- Die Begrifflichkeiten im Bereich Softwaretesten werden in verwirrend vielen Kombinationen benutzt. Das erschwert die Kommunikation erheblich. Bedeutungen fallen durcheinander oder gehen verloren, wenn es keinen gemeinsamen Wortschatz gibt. Der Autor hat sich entschieden die Begriffe von [Spillner u. Linz, 2005] zu verwenden. Diese werden hier im Folgenden erläutert, da sie zentraler Bestandteil des Themas sind.

3.1.1 Fehlerbegriffe

Ein Fehler liegt dann vor, wenn eine zuvor festgelegte Anforderung nicht erfüllt wurde. Man kann dies auch als Abweichung zwischen Soll- (Anforderungskatalog oder Spezifikation) und Ist-Zustand verstehen.

Beim Auftreten eines Fehlers nach o. g. Definition ist dieser bereits sichtbar bzw. messbar. Das nennt man Fehlerwirkung (DIN 66271, engl. failure), Fehlfunktion, äußerer Fehler oder Ausfall.

Die Fehlerwirkung jedoch muss von ihrer Ursache unterschieden werden. Zum Beispiel kann ein Programm abstürzen, weil ein Programmierer versucht hat durch 0 zu dividieren oder eine nicht vorhandene Anweisung auszuführen. Die Ursachen sind verschieden, das Ergebnis

gleich: Das Programm stürzt ab. Die Ursache wird Fehlerzustand (engl. fault), Defekt, innerer Fehler oder ganz häufig auch bug genannt.

Obwohl nicht oft verwendet, zeigen die Begriffe innerer und äußerer Fehler hier sehr schön die Trennung der Fehlerwirkung und des Fehlerzustandes.

Dabei muss ein innerer Fehler nicht immer zu einem äußeren führen. Das klassische Beispiel: Eine Berechnungsfunktion enthält einen Defekt. Sie wird jedoch aufgrund eines weiteren Defekts niemals ausgeführt. Erst wenn dieser behoben wurde, kann die Fehlerwirkung der Berechnungsfunktion auftreten. Dieses Verhalten wird Fehlermaskierung genannt. Ein Defekt A verhindert die Fehlerwirkung des Defektes B, womit dieser nicht festgestellt werden kann, solange Defekt A besteht.

Die Ursache eines Fehlerzustandes wiederum nennt man Fehlhandlung (engl. error). In der Regel hat der Programmierer eine Fehlhandlung begangen, indem er eine fehlerhafte Anweisung geschrieben hat. Aber auch äußere Einflüsse, die nicht durch die Spezifikation näher behandelt wurden, fallen in diese Kategorie. Zum Beispiel Strahlung, Magnetismus, vorsätzliche Manipulation der Umgebung wie Hardware oder Daten.

3.1.2 Testbegriffe

Testen bedeutet im wesentlichen die kontrollierte und stichprobenartige Ausführung eines Testobjekts. Dabei werden folgende Ziele verfolgt:

- Nachweisen der Fehlerwirkung, d. h. Fehler sichtbar zu machen
- Qualität bestimmen
- Vertrauen schaffen
- Analysieren um vorzubeugen

Erst wenn die Fehlerwirkung nachgewiesen wurde, kann das Debugging (engl., entwanzen) stattfinden. Es bezeichnet die Tätigkeit des Fehlerfindens. Wichtig ist dabei die Reproduzierbarkeit einer Fehlerwirkung.

Alle Tätigkeiten, die mit dem Testen zusammenhängen, gehören dem Testprozess an: Die Planung, das Ausführen mit Testdaten, das Auswerten der Ergebnisse sowie das Testmanagement seien hier aufgeführt. Der Testprozess sollte sinnvoll in den Entwicklungsprozess eingebettet werden. Darauf geht das Kapitel 3.2 gezielter ein.

Eines der Ziele des Testens ist die Qualität, was ein großes Schlagwort ist. Es bezeichnet die Güte eines Produktes. Vorrangig für ein Softwareprodukt ist die Fehlerfreiheit im alltäglichen Gebrauch. Die [ISO 9126] bezieht auch die Faktoren Funktionalität, Zuverlässigkeit, Wiederherstellbarkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit mit ein. Diese werden in Kapitel 3.1.4 detailliert beschrieben.

Über die zu erwartende Qualität eines Produktes können unterschiedliche Auffassungen entstehen. Damit alle Beteiligten ein einziges übereinstimmendes Qualitätsziel haben, sollten Qualitätsanforderungen vorab abgestimmt und definiert werden.

3.1.3 Testaufwand

Einem System Fehlerfreiheit nachzuweisen ist praktisch nicht möglich. Ein vollständiger Test müsste alle möglichen Situationen und alle möglichen Eingaben unter allen denkbaren Randbedingungen umfassen. Dies nennt man Testfallexplosion. Auch wenn dies im Einzelfall realisierbar wäre, so stiegen die Kosten möglicherweise ins Unermessliche.

Auf der anderen Seite, wenn zu wenig getestet wird, steigt das Risiko von verborgenen Defekten. Treten Fehlerzustände erst im Feld auf, können hohe Folgekosten bis hin zur Schädigung menschlicher Gesundheit bzw. Lebens auftreten.

Eine Balance zwischen Aufwand und Risiko muss gefunden werden. Anhand der Qualitätsanforderungen sowie Bewertung der Risiken und ihrer Eintrittswahrscheinlichkeiten kann ein ökonomisch sinnvolles Verhältnis gefunden werden.

3.1.4 Anforderungsklassifizierung

Die Anforderungen eines Softwareproduktes können mit Qualitätsmerkmalen versehen werden. Dies hilft bei der Beurteilung einer Anforderung, mit welcher Priorität sie umgesetzt und getestet werden sollte. Aber auch Zusammenhänge zu geeigneten oder ungeeigneten Tests lassen sich herstellen. Im Kapitel 4.4 werden diese Informationen weiterverwendet, um die richtige Teststrategie auszuwählen. Die [ISO 9126] führt sechs Hauptklassen und zugehörige Qualitätsmerkmale ein. Diese sind nachfolgend kurz beschrieben.

Funktionalität ist in der Regel das Hauptkriterium eines Produktes. Hiermit ist das korrekte Vorhandensein aller geforderter Funktionen gemeint. Unterpunkte sind:

- Angemessenheit (ob sich die Funktionen für die gestellten Aufgaben eignen)
- Richtigkeit
- Interoperabilität mit vorgegebenen externen Systemen
- Sicherheit gegen versehentlichen oder vorsätzlichen Dateneingriff

Zuverlässigkeit meint die Erhaltung der Leistungsfähigkeit über einen Zeitraum hinweg. Unterkategorien sind:

- Reife (je weniger Fehlerwirkungen und je geringer das Ausmaß, desto reifer ist eine Software)
- Fehlertoleranz (wie wirkt sich ein Fehlerzustand auf die Leistungsfähigkeit der Software aus?)
- Robustheit (wie geht das System mit unvorhergesehenen Ereignissen oder Eingaben um?)
- Wiederherstellbarkeit aus Fehlersituationen (Kriterien sind Dauer und Aufwand)

Benutzbarkeit wird vom Anwender wahrgenommen. Es ist seine Beurteilung der Software und den notwendigen Aufwand beim Einsatz der Software:

- Verständlichkeit
- Erlernbarkeit
- Bedienbarkeit
- Attraktivität (Anziehungskraft der Software)

Effizienz beschreibt das Leistungsniveau im Verhältnis zum Einsatz von Betriebsmitteln. Wesentliche Kriterien sind:

- Zeitverhalten (Antwort- und Verarbeitungszeiten, Durchsatz)
- Verbrauchsverhalten (Ressourcenverbrauch)

Änderbarkeit bezieht sich auf den Aufwand, der nötig wird, wenn ein System geändert werden muss.

- Analysierbarkeit
- Modifizierbarkeit
- Stabilität (Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen)
- Prüfbarkeit (Aufwand beim Testen)

Übertragbarkeit bezieht sich auf die Fähigkeit, ein System in eine andere Umgebung zu übertragen.

- Anpassbarkeit (Aufwand, um eine Software an eine neue Umgebung anzupassen)
- Installierbarkeit
- Koexistenz (paralleles Arbeiten mit ähnlichen Produkten)

- Austauschbarkeit (z. B. Ersetzen einer anderen Software)

3.2 Teststufen

Ganz nach dem Motto „Teile und Herrsche“ wird ein Projekt in mehrere Stufen zerlegt, um dessen Komplexität Herr zu werden. Die Beschreibungen zur Vorgehensweise und Aufteilung werden im Entwicklungsmodell festgehalten. Ein grundlegendes Modell ist das allgemeine V-Modell nach [Boehm, 1979]. Es integriert zu den jeweiligen Entwicklungsstufen auch passende Teststufen. Obwohl es einige nachfolgende Modelle, wie z. B. [V-Modell XT, 2006], als auch viele Alternativen, z. B. [Extreme Programming] gibt, eignet sich die vereinfachte Darstellung des V-Modells sehr gut zur Einordnung der jeweiligen Teststufen.

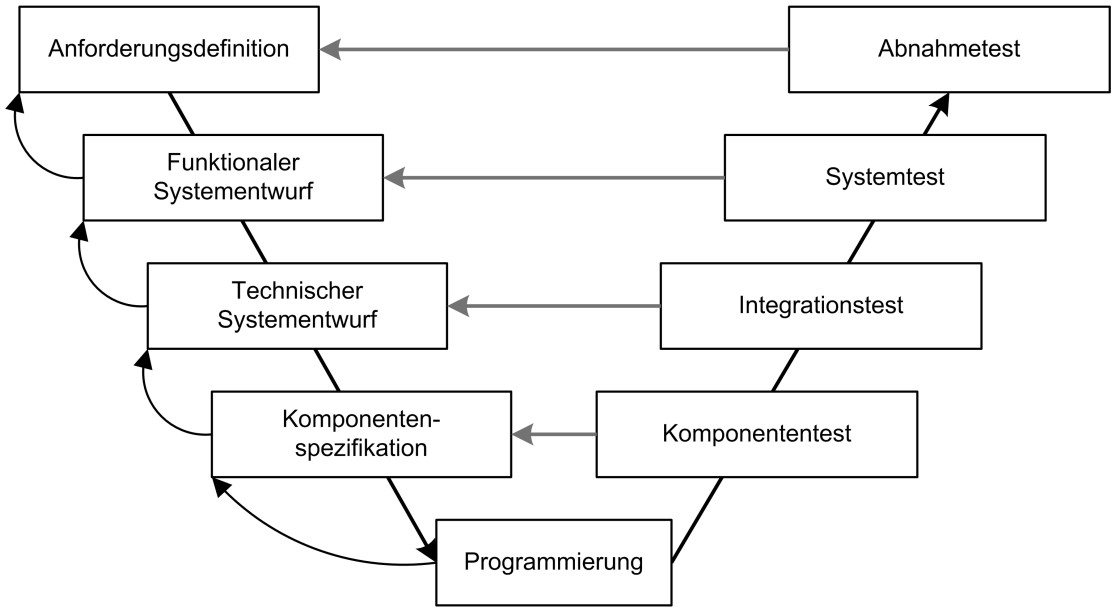


Abbildung 3.1: Allgemeines V-Modell

Bild 3.1 zeigt die konstruktiven Aktivitäten im linken Ast und die Prüf- und Integrationsarbeiten im rechten Ast. Die Ebenen verdeutlichen die Zusammenhänge zwischen den beiden Seiten. Konstruktive Aktivitäten werden auf gleicher Ebene mit passenden Testaktivitäten nachgeprüft.

So steht am Anfang jedes Projektes die *Anforderungsdefinition*, die die Kundenwünsche sammelt und spezifiziert. Der *funktionale Systementwurf* überträgt die Anforderungen auf Funktionen und Abläufe des Systems. Der *technische Systementwurf* benutzt diese wiederum als Vorgaben zur Realisierung und definiert Systemarchitektur, Schnittstellen zur Außen-

welt und Komponenten. Nach dem Aufteilen auf technischer Ebene muss für jedes Teilsystem eine *Komponentenspezifikation* erstellt werden. Diese legt Schnittstellen aus Sicht einer Komponente als auch ihre innere Architektur fest. Erst dann findet die eigentliche *Programmierung* statt.

Zu jeder dieser Aktivität gibt es nun eine Teststufe. Dabei kommt zugute, dass Fehler in einer Abstraktionsschicht auch auf derselben am einfachsten gefunden werden können. Die Komponentenspezifikation enthält alle Informationen, um einen Komponententest zu absolvieren. Ein Integrationstest wiederum prüft Gruppen von Funktionseinheiten, die im technischen Systementwurf spezifiziert wurden. Ein Systemtest betrachtet das System als Ganzes, wie es auch der funktionale Systementwurf darstellt. Und schließlich prüft ein Abnahmetest alle vom Kunden definierten Anforderungen, also auch aus der Sicht des Kunden.

Die vereinfachte Darstellung des V-Modells hat einen wesentlichen Nachteil. Die horizontale Achse wird oft als Zeitachse gedeutet. Tatsächlich ist im ursprünglichen V-Modell die Zeitachse an dieser Stelle eingezeichnet. Danach würden Testaktivitäten erst nach der Implementierung, also sehr spät angegangen werden. Dieses Vorgehen kann fatale Folgen für ein Projekt haben. Sollte sich ein Fehler beispielsweise in einer sehr frühen Phase, wie dem funktionalen Systementwurf, einschleichen, bliebe er möglicherweise bis zum Test dieser Phase unentdeckt. Alle Aktivitäten dazwischen könnten davon betroffen sein. Im schlimmsten Fall wären Arbeiten gar umsonst gemacht worden. In der Regel gilt, je später ein Fehler gefunden wird umso höher sind die Folgekosten. Wünschenswert ist die Einbeziehung aller Testaktivitäten von Beginn an. Während der Erstellung der Anforderungsdefinition wird ebenfalls der Abnahmetest definiert. In den beiden Phasen des Systementwurfs können bereits System- und Integrationstests definiert werden. Und Tests auf Komponenten- und Integrationsebene finden so früh wie möglich statt, also noch während der Programmierung. Aktuellere Modelle, wie z. B. das [V-Modell XT, 2006] oder das W-Modell von [Spillner u. a., 2006] bilden diese Thematik besser ab. Auch die in der Praxis oft üblichen Iterationen zwischen einzelnen Entwicklungsstufen finden sich darin wieder. In [Broekman u. Notenboom, 2003] ist etwa von „Multiple V's“ die Rede.

3.2.1 Komponententest

Eine Komponente abstrahiert kleinste Softwarebausteine, die je nach Programmiersprache z. B. Units, Klassen oder Module heißen. Der Komponententest hat zum Ziel, die enthaltene Codierung mit möglichst hoher Abdeckung zu testen. Der Fokus liegt dabei ausschließlich auf der einen zu testenden Komponente, weshalb die Ausführung möglichst isoliert stattfindet. Gefundene Fehler können damit zweifelsfrei dieser einen Komponente zugeordnet werden.

Da die Komponente isoliert ausgeführt werden soll, muss das Umfeld der Komponente insoweit aufbereitet oder nachempfunden werden, dass die Komponente ihre Arbeit aufnehmen kann. Eingaben müssen stimuliert und Ausgaben müssen verglichen werden. Oft müssen auch weitere Funktionen zumindest zum Schein vorgehalten werden, z. B. wenn die Komponente Ereignisse auslöst, auf deren Rückmeldung sie wartet.

Testframeworks helfen dabei, diese Aufgaben in bestimmten Umgebungen einfach und elegant umzusetzen. Dass das in Embedded-Systemen nicht immer ganz einfach ist, zeigt das Kapitel 4.5.9.

3.2.2 Integrationstest

Sobald die ersten Komponenten zusammengeführt werden können, findet der Integrationstest Anwendung. Wichtig ist, dass die Komponenten vorher schon erfolgreich einzeln getestet wurden. Das Ziel des Integrationstests ist es, Schnittstellenfehler aufzudecken. Sobald Komponenten zu einem größeren Verbund zusammengekoppelt werden, können Wechselwirkungen auftreten, die nicht mit dem vorhergehenden Komponententest ermittelt werden können. Dazu zählen u. a. unterschiedlich interpretierte Daten von miteinander kommunizierenden Komponenten, weil z. B. die Spezifikation von unterschiedlichen Entwicklern verschieden ausgelegt wurde.

Der Integrationstest hat auch Schwächen weshalb der Komponententest keinesfalls übersprungen werden sollte. So kann beim Auftreten von Fehlerwirkungen nicht mit Sicherheit die Ursache ermittelt werden, oder aber es ist sehr aufwändig dies zu tun. Zusätzlich dazu kann die Schnittstelle einer integrierten Umgebung kleiner sein als die einer einzelnen Komponente, was bedeutet, dass nicht alle Funktionen im getesteten Verbund auch tatsächlich stimuliert werden können.

3.2.3 Systemtest

Der entscheidende Schritt vom Integrationstest zum Systemtest bedeutet, dass das System komplett zusammengebaut vorliegen muss. Anders als bei den beiden vorhergehenden Stufen, wo Testtreiber einzelne Teilbereiche kontrolliert ausgeführt haben, kann das gesamte System nun wie spezifiziert selbständig in seiner Umgebung arbeiten. Die Testumgebung sollte dabei so ähnlich wie möglich zur späteren Produktionsumgebung sein, jedoch nicht die Produktionsumgebung selbst, da dies verschiedene Risiken bärge.

Der Systemtest verfolgt das Ziel, das gesamte System und seine Eigenschaften gegen die Anforderungen zu prüfen. Die Sichtweise ist dabei eher technischer Natur und die Durchführung erfolgt durch den Auftragnehmer. Auf dieser Ebene zeigt sich ob ein System an den Kunden auslieferungsfähig ist.

3.2.4 Abnahmetest

Abgekoppelt von der technischen Sicht prüft der Abnahmetest gegen die Anforderungsspezifikation. Dies ist die letzte Stufe des Testens und hat ihren Fokus daher auf den Kunden bzw. den Benutzer des Systems. Bei Auftragsentwicklungen ist es sogar üblich, dass der Auftraggeber den Abnahmetest durchführt, mindestens aber beteiligt wird. Ziel ist die Überprüfung der Akzeptanz des neuen Systems. Häufig wird auch von Akzeptanztests gesprochen. Je nach Produkt und dessen Risiken fällt der Umfang des Akzeptanztests sehr unterschiedlich aus. Eine speziell neu entwickelte Software, wie z. B. ein auf ein Unternehmen zugeschnittenes Verwaltungsprogramm, wird eher in einem gründlichen Feld- und Benutzertest auf Akzeptanz geprüft, bevor es produktiv eingesetzt wird. Ein Standardprodukt hingegen muss auf dieser Stufe nicht aufwändig getestet werden, da Erfahrungen in der Regel mit der Produktart schon vorliegen. Diese können schon in den vorherigen Teststufen eingebracht werden.

3.3 Testmanagement

Das Testmanagement ist verantwortlich dafür, dass die richtigen Tests zur richtigen Zeit ausgeführt werden (können). Die Testauswahl und Testplanung sowie Überwachung und Steuerung sind zentrale Bestandteile. Wesentliche Fragen sind:

- Was muss getestet werden?
- Was kann getestet werden?
- Wann wird getestet?
- Wer testet?
- Was wird zum Testen benötigt?
- Wie viel wird getestet?

Die frühestmögliche Einbeziehung aller Testaktivitäten, insbesondere des Testmanagements, steigert die Wahrscheinlichkeit, ein erfolgreiches Produkt fristgemäß abzuliefern. Kosten und Aufwand des Testens werden frühzeitig geschätzt und können Hinweise auf Problemfelder und Optimierungspotentiale geben. Eine entsprechende Rückmeldung an die konstruktiven Prozesse kann genutzt werden, um Alternativen zu entwerfen, die einfacher und damit kostengünstiger zu testen sind. Aber auch grobe Designfehler können bei gutem

Testmanagement früh erkannt werden. Wichtig ist die Erkenntnis, dass Testressourcen und Personal so früh wie möglich genutzt werden, um Tests zu planen, vorzubereiten, Reviews durchzuführen, und Testfälle zu erstellen und zu implementieren. Ein großer Teil aller Testaktivitäten kann schon abgearbeitet werden, bevor die eigentliche Implementierung fertig ist. Lediglich die tatsächliche Testdurchführung und Auswertung müssen auf diesen Arbeitsschritt warten.

Im Verlauf eines Projektes treten oft unvorhergesehene Dinge auf. Typisch ist z. B. eine wesentliche Änderung der Anforderungsspezifikation oder die Verkürzung der Entwicklungs- und Testzeit, weil der Liefertermins zu einem früheren Zeitpunkt geändert wurde. Das Testmanagement muss in solchen Fällen alle benötigten Informationen bereit haben, um abzuwägen, welche Schritte einzuleiten sind. Die Qualitätsvorgaben und Richtlinien der Auftraggeber und Verantwortlichen müssen mit den Prioritäten der geplanten Tests abgeglichen werden, weniger wichtige Tests können dann weggefallen. Die Verantwortlichen sollten über das gestiegene Risiko möglichst genau informiert werden. Natürlich sind diese Informationen und Maßnahmen nicht nur im Notfall wichtig, sondern sind auch Bestandteil ausgewogener Projektphasen.

3.4 Testkosten

In Kapitel 3.1.3 wurde schon die anzustrebende Balance zwischen Testaufwand und Risiken erwähnt. Um einzuschätzen, ob der Testaufwand noch gerechtfertigt ist, lohnt es sich, die Risiken und die Anforderungen detailliert zu betrachten.

Ein Risiko ist die negative Folge eines unterlassenen Tests und kann mit seiner Eintrittswahrscheinlichkeit und den Fehlerkosten[1] beziffert werden. Dadurch können Risikofälle gegeneinander abgewogen werden. Testfälle, die helfen, hohe Risiken zu minimieren, würden eine hohe Priorität erhalten.

In jedem Fall sollten die Anforderungen aufgrund ihrer Wichtigkeit bewertet werden. Weniger wichtige oder sogar optionale Anforderungen können in der Testplanung und im Testaufwand berücksichtigt werden. Dies kann helfen, knappe Testbudgets oder enge Zeitvorgaben umzusetzen.

In der Testplanung kann aufgrund dieser Einschätzungen eine Testmatrix (vgl. [Broekman u. Notenboom, 2003]) erstellt werden, die hilft, die richtigen Testfälle herauszusuchen.

Die Testaktivitäten müssen bzgl. ihres Aufwands ebenfalls geschätzt werden. Erst dann können Testkosten und Risiken gegeneinander gestellt werden. Anhand dieser Informationen

[1] Fehlerkosten sind die direkten oder indirekten Folgekosten einer Fehlerwirkung im produktiven Einsatz der Software als auch die Kosten für die Fehlerbehebung.

und mit dem nötigen Hintergrundwissen über das Umfeld des Produkts und Projekts kann es dem Testmanagement gelingen, die richtigen Entscheidungen zu fällen.

3.5 Testmethoden

Eine Testmethode beschreibt die Art, wie etwas getestet werden kann. In der Literatur wird dabei zwischen statischen und dynamischen Tests unterschieden. Der statische Test analysiert den Sourcecode oder die Dokumente. Dies kann werkzeuggestützt oder manuell erfolgen. Der dynamische Test kommt am „lebenden“ Testobjekt zum Einsatz. Testdaten und -funktionen werden kontrolliert ausgeführt. Im Folgenden werden die Methoden kurz vorgestellt, die in dieser Arbeit Verwendung finden. Detaillierte Erklärungen finden sich in den Kapiteln, in denen die Methoden eingesetzt werden.

3.5.1 Review

Unter einem Review versteht man die Begutachtung verschiedener Dokumente bzgl. verschiedener Kriterien. Damit gehört es zu den statischen Tests. [Spillner u. Linz, 2005] führen verschiedene Reviewtypen, Vorgehensweisen und Empfehlungen auf. Reviews können verschiedene Ziele haben. Während eines Projektes werden Produkte oder ihre Teile begutachtet, d. h. Spezifikation, Sourcecode, usw. Eine andere Ebene ist das Review des Entwicklungsprozesses selbst. Hierbei werden Aspekte des Projektmanagements und des tatsächlichen Projektverlaufs reflektiert.

3.5.2 Dynamische Tests

Testverfahren, die das Testobjekt nur von außen betrachten, gehören in die Kategorie der Blackbox-Methoden. Im Gegenteil dazu beziehen die Whitebox-Verfahren den Sourcecode des Testobjekts mit ein. Eine Mischung aus beiden Varianten ist denkbar und kommt in der Praxis mit der Bezeichnung Greybox vor.

Zur Familie der Blackbox-Verfahren gehören:

- Äquivalenzklassenbildung (Kap. 4.5.5)
- Grenzwertanalyse (Kap. 4.5.6)
- Zustandsbezogene Tests

Die genannten Kapitel erklären und vertiefen die Methoden am Beispiel. Die zustandsbezogenen Tests werden für diese Arbeit nicht benötigt. Der Vollständigkeit halber sind hier auch die Whitebox-Verfahren aufgeführt. Auch diese werden nicht verwendet.

- Anweisungsüberdeckung
- Zweigüberdeckung
- Test der Bedingungen
- Pfadüberdeckung

4 Testkonzept und Durchführung am Beispiel CAN-Treiber

Nachdem in Kapitel 2 die aktuelle Testsituation bei Garz & Fricke aufgezeigt wurde und eine Aufgabenstellung erarbeitet wurde, sowie Kapitel 3 benötigtes Basiswissen aufgearbeitet hat, beschäftigt sich dieses Kapitel mit den Aufgaben, die nötig sind, um die Theorie in die Praxis umzusetzen. Testplanung und Durchführung werden detailliert am Beispiel des CAN-Treibers erläutert.

Zuerst wird CAN allgemein erläutert und anschließend das Umfeld des Testobjektes beschrieben. Die benötigten Spezifikationen und Dokumente werden vorgestellt. Zum Verständnis und genauer Kenntnis des Testobjekts wird der innere Aufbau des CAN-Treibers beschrieben.

Dokumentierte und nachvollziehbare bzw. testbare Anforderungen existieren nicht. Es wird daher versucht, aus allen vorliegenden Informationen und Erfahrungen aus vergangenen Projekten die Anforderungen abzuleiten und testbar zu gestalten.

Nach der Spezifikation der Anforderungen werden geeignete Testmethoden ausgewählt und die Testdurchführung geplant. Die Implementierung benötigt evtl. noch weitere Werkzeuge, die ebenfalls ausgewählt und vorgestellt werden. Die Testumgebung und Implementierung wird ausführlich und mit Beispielen beschrieben.

4.1 CAN

Der CAN-Bus wird in Industrie- und Automobilbereich verbreitet eingesetzt. Die Vorteile sind einfache Zwei-Draht-Verbindungstechnik, Verbindungsgeschwindigkeiten bis zu 1 MBit/s und lange Übertragungsstrecken. Große und lange Kabelbäume können eingespart werden, indem CAN-Controller gekoppelt mit (Leistungs-) Schalteinheiten so nah wie möglich an die Aktoren gelegt werden. Die Steuersignale können dann über den CAN-Bus von verschiedenen, weit entfernten Stellen gesendet werden. Eine ausgeklügelte Bus-Arbitrierung sorgt für eine Multi-Master-Fähigkeit.

Die aktuelle Spezifikation [CAN 2.0, 1991] regelt sowohl die physikalische Verbindungsschicht als auch das Protokoll zur Nachrichtenübertragung. Die zuverlässige Fehlererkennung und -behandlung hilft bei der korrekten Datenübertragung und Busstabilität. Das starre Nachrichtenformat und garantierte Latenzzeiten ermöglichen die Realisierung von Echtzeitsystemen. Weitergehende Spezifikationen, wie z. B. [CANopen], regeln Nachrichtenformate und Inhalte für eine große Anzahl an Applikationen, Sensoren und Aktoren. Ein anderes Beispiel ist [SAE J1939], das zusätzlich noch die Ausführung der physikalischen Verbindung, die Baudrate uvm. detailliert festlegt.

Die Merkmale des CAN-Busses hier detailliert zu besprechen würde den Rahmen sprengen. Eine sehr ausführliche Beschreibung der CAN-Grundlagen findet sich in der Diplomarbeit [Schröder, 2005]. Für die vorliegende Arbeit ist lediglich das Verständnis des CAN-Busses aus Sicht der Programmierung zur zuverlässigen Nachrichtenübertragung wichtig.

Dazu sei noch die Einordnung in das ISO/OSI-Modell in Tabelle 4.1 dargestellt. Die gängigen CAN-Controller bringen alles Nötige mit, um die Spezifikation auch in den Punkten Fehlererkennung, Latenzzeiten uvm. zu erfüllen. Das heißt, die Software braucht sich im Idealfall nur auf das Senden und Empfangen sowie ihre eigene Anwendungslogik zu konzentrieren. Die Spalte „CAN“ deutet an, ob die Funktionalität in Hardware oder Software bereitgestellt wird. Weil CAN auch von sehr kleinen Systemen implementiert wird, sind die Schichten 3, 5 und 6 unnötiger Ballast, die nicht spezifiziert sind und in der Praxis auch nicht zur Anwendung kommen.

OSI Schicht		**Beschreibung**	**CAN**
7	Application	Anwendungslogik	Software
6	Presentation	Umsetzung von Kodierungen (Zeichensätze, Little/Big Endian)	-
5	Session	Verbindungsauf- und abbau	-
4	Transport	Datenübertragung zwischen zwei Prozessen inkl. Fehlerbehandlung	Hardware
3	Network	Routing der Nachrichten, wenn mehrere Netze/Wege vorhanden	-
2	Data Link	Zugriffskontrolle auf das Übertragungsmedium und Fehlererkennung	Hardware
1	Physical	Beschreibung des Übertragungsmediums	Hardware

Tabelle 4.1: CAN im ISO/OSI-Modell

Das Nachrichtenformat ist sehr eng definiert. Es enthält die Adresse, maximal acht Datenbytes und einige zusätzliche Informationen zur Steuerung und Fehlererkennung. Die Adresse ist bei CAN 2.0A 11 Bit breit. CAN 2.0B definiert dagegen 29 Bit Adressbreite. Im Folgenden

werden die Begriffe STANDARD (11 Bit) und EXTENDED (29 bit) für Adressen oder Nachrichten benutzt. Da der CAN-Bus nur eine logische Leitung hat (das Signal wird differentiell übertragen), auf der alle Teilnehmer ihre Kommunikation synchronisieren müssen, bietet er eine sog. Arbitrierung über die Adresse an. Teilnehmer prüfen zuerst, ob die Leitung frei ist, bevor sie mit dem Senden anfangen. Sollten zwei Teilnehmer gleichzeitig anfangen, setzt sich die kleinere Adresse auf dem Bus durch und die größere Adresse muss es später nochmal versuchen. Mit Adresse ist hier stets die Zieladresse gemeint. Eine Quelladresse ist im CAN-Protokoll nicht spezifiziert. Eine CAN-Nachricht enthält zusätzlich noch die Information über die Länge der enthaltenen Datenbytes, eine CRC-Prüfsumme[1] und andere Steuerinformationen, um die sich die Hardware selbständig kümmert. Die Software übergibt in der Regel nur die Adresse, die Länge und die Datenbytes selbst an den CAN-Controller. Wenn Fehler auftreten, kümmert sich ebenfalls der Controller darum, die anderen Teilnehmer mit einem Error-Frame zu benachrichtigen und notfalls sich selbst in einen passiven oder Bus-Off-Zustand zu versetzen.

4.2 Anforderungen

Die Einsatzgebiete des Jupiters mitsamt CAN-Option sind vielfältig. Ein Großteil der Kunden nutzt den CAN-Bus jedoch um direkt mit angekoppelten Buskomponenten zu sprechen. Beispiele sind Klimagerätesteuerungen oder dezentrale Mess- und Bedieneinheiten mit der Ankopplung einer zentralen Steuerung über CAN. Das Jupiter kann damit als Master und als Slave eingesetzt werden.

Üblicherweise verlangt ein Kunde die Erfüllung der [CAN 2.0, 1991] Spezifikation. Dies ist durch das Einsetzen eines CAN 2.0 kompatiblen Controllers schon weitestgehend erfüllt. Die verwendete Hardware, der SJA1000 von NXP (ehemals Philips), hält sich ebenfalls an diese Spezifikation. Der Treiber muss nun die Funktionalität dieses Bausteins der Applikation zur Verfügung stellen.

Darüber hinaus ist es einigen Kunden wichtig, die Priorität der CAN-Verarbeitung als besonders hoch anzusehen. Daraus lässt sich schließen, dass der Kunde zeitkritische Aufgaben realisieren möchte. Wünschenswert wäre hier eine weitere Beschreibung des Kundensystems und seiner Echtzeit-Eigenschaften. Aus Erfahrung heraus kommt es leider häufig vor, dass dieser Teil aus Sicherheitsgründen unter Verschluss gehalten wird, oder aber der Kunde hat selbst keine weiteren Vorstellungen bzgl. dieser Anforderungen, außer, dass es eben „schnell“ sein muss. Dieser Umstand wird in 4.2.3 und 4.6.5 aufgegriffen.

[1] Cyclic Redundancy check (CRC)

Einige Kunden kaufen zusätzliche CANopen-Bibliotheken, um auf gängige Aktoren und Sensoren zuzugreifen. Zugekaufte Komponenten, wie CANopen-Bibliotheken, sollen nicht Bestandteil des Lieferumfangs sein und werden damit auch nicht getestet.

4.2.1 Requirementstracing

Anforderungen müssen gewisse Eigenschaften haben, damit Testen sinnvoll möglich ist. In erster Linie müssen diese vorhanden und niedergeschrieben sein, was im Beispiel CAN schon mal nicht der Fall ist. Weiterhin müssen Anforderungen das Testen von vornherein berücksichtigen. Bei der Testdurchführung und danach wird sich immer die Frage stellen, ob gewisse Testfälle genügen, um das gewünschte Ergebnis oder Sicherheit zu erreichen oder ob es noch an einigen Stellen mangelt und wenn ja, an welchen. Dazu müssen Anforderungen formalisiert und feingliedrig aufgesplittet werden.

Jeder Anforderungspunkt wird zunächst mit einer ID versehen damit auch in den folgenden Kapiteln oder weiteren Dokumenten exakt auf bestimmte Anforderungen verwiesen werden kann. Weitere Metainformationen helfen bei der Strukturierung der Anforderungen in der Planungsphase sowie später bei Informationsverfolgung und Kontrolle während der Durchführung. Informationen zum Status und zur Priorität sind nützlich. Ganz wichtig ist neben der Anforderungsbeschreibung selbst auch die Beschreibung, wann die Anforderung erfüllt ist. Kriterien, Kennzahlen oder Szenarien sind typische Angaben dafür.

Große Projekte erfordern Werkzeugunterstützung zum Anforderungsmanagement. Ein verbreitetes Werkzeug ist [DOORS]. Es speichert alle Anforderungen in einer Datenbank ab. Zugehörige Informationen kann es daher leicht verknüpfen. Verschiedene Reports lassen sich mit einem einstellbaren Detaillierungsgrad als Word-Dokument erstellen. Seine Stärken kann die Software ausspielen, wenn es im Verbund mit Modellierungswerkzeugen eingesetzt wird. Der Hersteller Telelogic bietet eine Reihe solcher Werkzeuge an, das größte ist u. a. Rhapsody. Formalisierte Anforderungen werden in das Modellierungswerkzeug übernommen. Die Schritte und Modelle im Softwaredesign referenzieren dann die Anforderungen. Andersherum können Modellierungs- und Testwerkzeuge Rückmeldungen an das Anforderungsmanagement geben und Statusinformationen, Referenzen uvm. zurückspielen. Durch die starke Vernetzung wird das Auffinden von zugehörigen Informationen über den gesamten Entwicklungsprozess vereinfacht bzw. überhaupt erst ermöglicht.

Sowohl für diese Arbeit als auch für Garz & Fricke kommt DOORS aus Kostengründen nicht in Betracht. Das Requirementstracing soll aber trotzdem mit einfachen Mitteln stattfinden.

4.2.2 Anforderungsliste

Die Anforderungsliste führt alle bisher gesammelten Anforderungen formalisiert auf. Das Feld Ursprung verweist auf die Quelle, wo die Anforderung entstanden ist. Die Beschreibung bzw. Kriterien geben konkrete Anhaltspunkte, wann ein Test erfolgreich abgeschlossen wurde. Die Qualitätsmerkmale sind an die ISO 9126 angelehnt. Diese wurden in Abschnitt 3.1.4 eingeführt. Der Status dokumentiert den aktuellen Erfüllungsgrad der jeweiligen Anforderung und referenziert die benötigten Testfälle.

ID / Ursprung / Priorität:	A001 / System / Hoch
Anforderung:	Initialisierung der Hardware
Beschreibung / Kriterien:	Treiber kann von Applikation geöffnet und bedient werden.
Qualitätsmerkmal:	Funktionalität
ID / Ursprung / Priorität:	A002 / CAN 2.0 / Hoch
Anforderung:	Nachrichten senden
Beschreibung / Kriterien:	Erfolgreich, wenn eine gesendete Nachricht an einer Gegenstelle fehlerfrei ankommt.
Qualitätsmerkmal:	Funktionalität
ID / Ursprung / Priorität:	A003 / CAN 2.0 / Hoch
Anforderung:	Nachrichten empfangen
Beschreibung / Kriterien:	Erfolgreich, wenn eine Nachricht fehlerfrei empfangen werden kann, die von Gegenstelle gesendet wurde.
Qualitätsmerkmal:	Funktionalität
ID / Ursprung / Priorität:	A004 / CAN 2.0 / Hoch
Anforderung:	Nachrichtenformat sowohl STANDARD als auch EXTENDED
Beschreibung / Kriterien:	Nachrichten in diesen Formaten müssen gesendet und empfangen werden können.
Qualitätsmerkmal:	Funktionalität
ID / Ursprung / Priorität:	A005 / CAN 2.0 / Hoch
Anforderung:	Gängige Baudraten einstellen: 10, 20, 50, 100, 125, 250, 500, 1000 kBit/s
Beschreibung / Kriterien:	Erfolgreich, wenn der Controller auf die gewünschte Baudrate eingestellt werden kann und fehlerfrei mit jeder Baudrate mit Gegenstellen kommunizieren kann.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A006 / Kunde / Niedrig
Anforderung:	Kundenspezifische Baudraten einstellen
Beschreibung / Kriterien:	Erfolgreich, wenn die Parameter zur Einstellung der Baudrate direkt im Controller gesetzt werden können. Eine Überprüfung ob Kommunikation möglich, ist nicht nötig, da Realisierung aufwändig und Funktion sehr selten gefordert wird.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A007 / Kunde / Niedrig
Anforderung:	Stromsparmodus unterstützen
Beschreibung / Kriterien:	Der Stromsparmodus des SJA1000 soll ein- und ausgeschaltet werden können.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A008 / Kunde / Niedrig
Anforderung:	Aufwachen aus dem Stromsparmodus
Beschreibung / Kriterien:	Nach Erhalt einer CAN-Nachricht muss das Gerät aufwachen.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A009 / Kunde / Hoch
Anforderung:	Benachrichtigung der Applikation bei Ankunft neuer Nachrichten
Beschreibung / Kriterien:	Das Empfangen von Nachrichten muss ohne Busy-Waiting funktionieren.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A010 / Kunde / Hoch
Anforderung:	Benachrichtigung der Applikation bei Auftreten von Fehlern
Beschreibung / Kriterien:	Fehler müssen der Applikation sofort mitgeteilt werden können, ohne das die Applikation mittels Busy-Waiting darauf wartet.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A011 / Kunde / Mittel
Anforderung:	Einstellung der Empfangsfilter
Beschreibung / Kriterien:	Es werden nur Nachrichten in einem bestimmten Adressbereich empfangen.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A012 / Kunde / Mittel
Anforderung:	Echtzeitfähigkeit
Beschreibung / Kriterien:	Reaktionszeit beim Empfangen einer Nachricht vergleichbar mit anderen Teilen des Systems und der Systemperformance. Typ. unter 1 ms.
Qualitätsmerkmal:	Effizienz

ID / Ursprung / Priorität:	A013 / Kunde / Mittel
Anforderung:	Fehlertoleranz
Beschreibung / Kriterien:	Nachdem Busfehler aufgetreten sind, kann das System zurückgesetzt werden um anschließend weiter zuarbeiten.
Qualitätsmerkmal:	Zuverlässigkeit

ID / Ursprung / Priorität:	A014 / System / Hoch
Anforderung:	Stabile Initialisierung auch wenn SJA1000 nicht vorhanden.
Beschreibung / Kriterien:	Betriebssystem bootet stabil und genauso schnell auch auf der Jupiter-Variante ohne SJA1000.
Qualitätsmerkmal:	Funktionalität

ID / Ursprung / Priorität:	A015 / Kunde, Garz & Fricke / Mittel
Anforderung:	Verständlichkeit und Benutzbarkeit der API.
Beschreibung / Kriterien:	Kunde kann sich schnell und gezielt einarbeiten, ohne auf Support von Garz & Fricke angewiesen zu sein.
Qualitätsmerkmal:	Benutzbarkeit

ID / Ursprung / Priorität:	A016 / System / Niedrig
Anforderung:	Treiber soll auch auf andere Systeme portierbar und testbar sein.
Beschreibung / Kriterien:	Beim Portieren auf andere Systeme darf kein großer Aufwand entstehen.
Qualitätsmerkmal:	Änderbarkeit, Übertragbarkeit

4.2.3 Nicht-Funktionale Anforderungen

Funktionale Anforderungen können meistens einfach beurteilt werden: Funktion erfüllt oder nicht erfüllt. Ein Großteil der Anforderungen des CAN-Treibers stellt funktionale Anforderungen dar.

Nicht-Funktionale Anforderungen bedürfen genauerer Betrachtung und Abwägungsprozesse. Die Anforderungen A012, A013, A015 und A016 fallen in diese Kategorie. Das Erreichen einer nicht-funktionalen Anforderung kann oft nicht eindeutig bestimmt werden. Die Aufteilung in entsprechende Kategorien, wie sie in Kapitel 3.1.4 eingeführt wurden, ist daher hilfreich. Alle weiteren Überlegungen müssen im Kontext der jeweiligen Umgebung stattfinden. Das Thema Fehlertoleranz und die Bedeutung für den CAN-Treiber wird im Kapitel 4.6.4 erläutert. Die Effizienz des Systems wird in Kapitel 4.6.5 beleuchtet. Die Punkte Änderbarkeit und Übertragbarkeit werden im Kapitel 4.7 behandelt.

4.3 Aufbau

Der CAN-Treiber für Windows CE 5.0 ist als sogenannter Streaminterface-Treiber implementiert. Zur Laufzeit wird dieser Treiber in einen separaten Prozess für Treiber (device.exe) geladen. Dadurch wird ein Speicherschutz zur Anwendung hin realisiert: Die Anwendung kann bei üblichen Programmierfehlern (Zeigerfehler) nicht den Treiber beeinträchtigen. Andersherum kann der Treiber auch nicht die Daten der Anwendung manipulieren. Dieses Design hilft bei der Umsetzung der Anforderung A013 (Zuverlässigkeit, Seite 45). Wenn Fehler in diesen Systemteilen auftreten, beeinflussen sie in aller Regel nicht den Rest des Systems.

Das Streaminterface erlaubt es mithilfe des Betriebssystems Daten und Befehle der Anwendung an den Treiber zu übergeben und Ergebnisse wieder bereitzustellen. Zehn generische Funktionen muss ein Treiber dafür bereitstellen:

- Init, Open, Close und Deinit
- Read, Write und Seek
- IOControl
- PowerUp und PowerDown

Nicht alle Funktionen müssen mit Inhalt gefüllt sein. So nutzt der CAN-Treiber nicht die Funktionen Read, Write und Seek zur Übergabe der Daten, sondern die Funktion IOControl. Diese kann aufgrund eines zusätzlichen Parameters mehrere Aktionen unterscheiden. Die Übergabe der Daten erfolgt typenlos. Anwendung und Treiber müssen dabei dieselbe Sprache sprechen, d. h. gleiche Definitionen für den Aktionstyp und Datentypen. Für gewöhnlich wird diese Schnittstelle aufgrund ihrer etwas aufwändigeren Programmierweise und damit auch etwas höheren Fehleranfälligkeit mit eigenen Funktionen gekapselt. Diese Funktionen stellen dann die Schnittstelle für die Anwendung dar, die API. Sie haben einen passenden Namen und können die Datentypsicherheit besser gewährleisten als das generische Streaminterface.

Der CAN-Treiber bringt eine solche API mit, sie stellt alle Funktionen bereit, um den CAN-Baustein möglichst komfortabel zu bedienen. Üblicherweise, und so auch in diesem Fall, werden nur diese Funktionen dokumentiert. Die Schnittstelle zwischen API und Treiber ist eine interne, die die Anwendung nicht zu interessieren braucht.

Die CAN-API stellt folgende Funktionen zur Verfügung:

- CanCreateDevice
- CanCloseDevice
- CanTransmitMessage
- CanReceiveMessage
- CanGetBaudrate
- CanSetBaudrate

- CanGetBittiming
- CanSetBittiming
- CanGetAddressfilter
- CanSetAddressfilter
- CanReset
- CanGetStatus
- CanReadRegister
- CanWriteRegister

Die detaillierte Schnittstellenbeschreibung befindet sich als Inline-Dokumentation im Header der CAN-API (siehe Anhang B.1).

Die Abbildung 4.1 verdeutlicht den Aufbau und die Einbettung der CAN-API, des Treibers und der Hardware im Windows CE 5.0. Das Betriebssystem sorgt für die Trennung der beiden Prozesse Device.exe und Applikation.exe und stellt gleichzeitig Methoden zum Datenaustausch zwischen diesen beiden Komponenten zur Verfügung. Die API benutzt diese Methoden um die Applikationsanforderungen an den Treiber weiterzugeben. Der Treiber greift in der Regel direkt auf die Register der Hardware zu. Es sind komplexere Strukturen wie PCI[2], ISA[3] oder auch andere Busstrukturen denkbar, die zwischen Treiber und Zielhardware sitzen, doch auf dem Jupiter ist die Hardware „memory mapped“ in den physikalischen Adressraum der CPU eingebunden. Der Treiber ist verantwortlich dafür, dass er beim Initialisieren den richtigen Speicherbereich vom Betriebssystem anfordert. Zur Verifikation wird eine „Detect“ Routine verwendet, die anhand einiger Registerwerte der Hardware entscheidet, ob der rich-

[2] Peripheral Component Interconnect (PCI)
[3] Industry Standard Architecture (ISA)

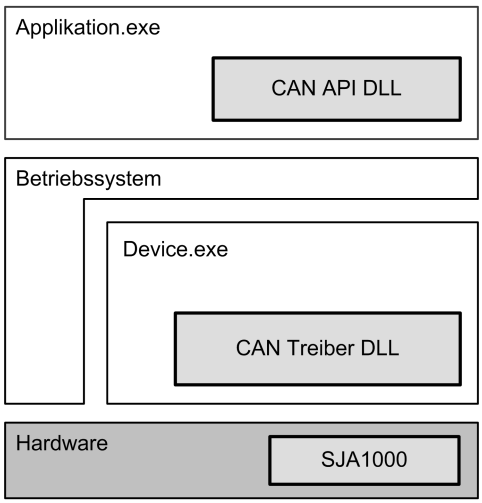


Abbildung 4.1: Struktur und Umgebung der CAN-Software

tige Hardwarebaustein gefunden wurde. Wenn dies fehlschlägt, wird der Treiber wieder entladen und kann von keiner Applikation mehr geöffnet werden. Auf einer Debug-Schnittstelle gibt er dabei eine Nachricht aus, ob der SJA1000 gefunden wurde. Dies wird später genutzt werden, um die Anforderungen A001 (Initialisierung, Seite 44) und A014 (Initialisierung ohne SJA1000, Seite 46) zu überprüfen.

4.4 Teststrategie

Bisher wurden alle Informationen zusammengetragen und erläutert, die zur Erstellung der Tests nötig sind. In diesem Abschnitt wird eine Strategie gesucht, die am geeignetsten für den CAN-Treiber erscheint. Es geht um die Auswahl passender Tests unter Berücksichtigung der Anforderungen und Wirtschaftlichkeit. Jeder Test kostet Geld, ob in der Planung oder Durchführung. Den Aufwand und damit auch die Kosten zu minimieren ist ein wirtschaftliches Interesse, um ein Produkt erfolgreich verkaufen zu können. Viele Testmethoden haben eine gewisse Überlappung ihrer Abdeckung und Testen andere Teile des Systems implizit mit. Eine gute Kombination aus Testfällen erlaubt es, einen guten Kompromiss zwischen Testaufwand und Produktrisiko (siehe Abschnitt 3.1.3) zu finden. Die Ermittlung der besten Strategie folgt der Vorgehensweise der Risk-based test strategy aus [Broekman u. Notenboom, 2003].

4.4.1 Anforderungsbewertung

Bereits im Kapitel 3.1.4 wurden die Anforderungen auf Qualitätsmerkmale abgebildet. Die Einteilung in abstrakte Klassen hilft dabei, eine Vielzahl von Anforderungen zu betrachten.

Die Anforderungsliste aus Kap. 4.2.2 umfasst 16 Punkte. Dies mag im Verhältnis zu anderen Projekten noch eine eher kurze Liste sein. Für den CAN-Treiber existieren entsprechend vorgefestigte Umgebungen und Dokumente ([CAN 2.0, 1991], CAN-API-Dokumentation im Anhang B), sodass diese im Zweifel immer herangezogen werden können. Dies ermöglicht eine knappe Anforderungsliste. Für Komponenten oder Projekte mit größerem Funktionsumfang oder weniger stark spezifiziertem Umfeld würde eine Anforderungsliste wesentlich länger ausfallen.

Um eine bessere Vorstellung von dem Endprodukt zu erhalten, können die Qualitätsmerkmale relativ zueinander gewichtet werden. Die Informationen sollten zu einem Teil aus der Anforderungsliste stammen. Anforderungspunkte mit einer hohen Priorität sollten sich in einer hohen Gewichtung ihrer Anforderungsklasse wiederfinden. Der andere Teil der Informationen zur Gewichtung sollte aus der Erfahrung und Fingerspitzengefühl der Verantwortlichen kommen. So gibt es durchaus Produkte, die in einem Bereich eine ganz hohe Anforderung

haben, wie z. B. sicherheitsrelevante Aspekte, die eine sehr kurze Reaktionszeit des CAN-Treibers erfordern würden. Wenn diese nicht erfüllt werden würden, so könnte auch das Produkt nicht eingesetzt bzw. verkauft werden. In einem solchen Fall sollten sich Testaktivitäten besonders auf diese wichtigen Merkmale konzentrieren.

Die Bewertungen der Qualitätsmerkmale des CAN-Treibers sind in der Tabelle 4.2 eingetragen. Sie werden im nächsten Kapitel dazu verwendet, den Fokus auf entsprechend geeignete Testmethoden bzw. Stufen zu legen. Die hohe Bedeutung der Funktionalität (40 %) ist nicht ungewöhnlich. Ein Softwareprodukt hängt in den meisten Fällen von seiner Funktionalität ab. Trotzdem mag es Ausnahmen geben, wie z. B. das Erstellen eines Prototypen, der nur auf Benutzbarkeit geprüft werden sollte. Die weitere Verteilung der nicht-funktionalen Klassen zeigt in diesem Fall, dass Zuverlässigkeit und Effizienz etwas wichtiger als die Benutzbarkeit eingeschätzt werden, da die späteren Benutzer der API Programmierer sind und in der Regel auch eine gewisse Erfahrung im Umgang mit diesen Systemen mitbringen. Die Benutzbarkeit dieser API ist nur sehr selten ein Kriterium, das beim Kauf herangezogen wird. Die Änderbarkeit und Übertragbarkeit haben eine sehr niedrige Priorität, sollten jedoch nicht völlig außer Acht gelassen werden, da dies in zukünftigen, hausinternen Projekten Kosten einsparen kann.

4.4.2 Strategie-Matrix

	Funktionalität	Zuverlässigkeit	Benutzbarkeit	Effizienz	Änderbarkeit	Übertragbarkeit
Relative Bedeutung (%)	*40*	*15*	*10*	*15*	*5*	*5*
Komponententest	++	+				
Review			++		++	+
Software-Integrationstest	+					
Hardware/Software-Integrationstest	+		+	++		
Systemtest	++	+		++		
Akzeptanztest	+			+		
Feldtest		+				

Tabelle 4.2: Bewertung der Anforderungskriterien und Testauswahl

Die Tabelle 4.2 listet zu den Qualitätskriterien alle Teststufen auf. Die Aktivität Integrationstest wird zusätzlich nochmal in Software- und Hardware-/Software-Integration aufgeteilt, um die

Abhängigkeiten zur Hardware deutlich zu machen. Besonders geeignete Tests werden mit ++ gekennzeichnet. Tests, die ein Qualitätsmerkmal implizit testen, haben ein einfaches + als Eintrag. Ein leerer Eintrag bedeutet, dass der Test ein Qualitätsmerkmal nicht ausreichend prüfen kann.

Zusammen mit der Gewichtung lässt sich ablesen welche Tests besonders wichtig sind und auf welche bei Bedarf auch verzichtet werden kann. Es wird in vielen anderen Projekten sehr ähnlich aussehen. Die Funktionalität lässt sich mit einer Reihe von Tests abprüfen, besonders geeignet erscheinen jedoch der Komponententest und der Systemtest. Der Komponententest prüft auf niedrigster Ebene die Funktion sehr detailliert. Implementierungsfehler fallen mit einer großen Wahrscheinlichkeit auf. Designfehler dagegen werden erst im Systemtest sichtbar, wenn wirklich alle Teile zusammengekoppelt in der Zielumgebung ihre Arbeit verrichten sollen. Grobe Designfehler können die Funktion eines Systems beeinträchtigen.

Als weitere sinnvolle Tests für die nicht-funktionalen Anforderungen erweisen sich in diesem Fall das Review und der Hardware/Software-Integrationstest. Die Anforderungen A015 und A016 (Seite 46) können nur sinnvoll mit einem Review geprüft werden. Die Benutzbarkeit einer API ist aus der Sicht eines Programmierers zu beurteilen, weshalb ein Review mit anderen Programmierern Verständlichkeitsprobleme in der Schnittstelle und Beschreibung schnell aufdecken kann. Die Anforderung A016 spricht Änderbarkeit und Übertragbarkeit gleichwertig an. Das Design und die Implementierung des Treibers sollten besonders auf Verständlichkeit hin geprüft werden, damit die Software später auch von anderen Programmieren gepflegt, geändert oder übertragen werden kann. Teile, die bei einer Portierung als aufwändig erkannt werden, sollten zwar dokumentiert und diskutiert werden, aber weitere Schritte sind aufgrund der niedrigen Priorität nicht nötig.

Die Anforderung der Effizienz lässt sich in den Phasen Hard- und Software-Integrationstest und Systemtest prüfen. Um die Performance und Reaktionszeit messen zu können, müssen in der Regel mehrere Komponenten zusammengeschaltet werden. In diesem Fall spielt die Hardware ebenfalls eine wesentliche Rolle.

Obwohl der Systemtest in den bisherigen Überlegungen als geeignetes Mittel erscheint, bleibt es eine Definitionsfrage, ob ein Systemtest für ein OEM-Produkt, wie es das Jupiter ist, durchführbar ist.

- Aus Sicht der CAN-Funktionalität gehört zu einem vollständigen System die Anwendung eines Kunden, die API, der Treiber, das Betriebssystem, der CAN-Controller, die weitere Hardware, die CAN-Verbindung und angeschlossene CAN-Geräte mit passender Anwendungsfunktionalität.
- Insbesondere die Anwendung liegt aber bei Garz & Fricke nicht vor. Das heißt, die erreichbare Teststufe bei Garz & Fricke kann nur der Integrationstest sein.

- Aus Sicht des Produktes Jupiter wiederum, würde das Zusammenarbeiten aller seiner Teile mit anderen Systemen schon als Systemtest angesehen werden können. Schließlich ist es dann schon ein Produkt, welches seine Anforderungen erfüllt und als Ganzes verkauft wird. Die Unterscheidung zum Integrationstest wäre allerdings, wenn überhaupt, minimal.

Hier wird sich für die erste Argumentationslinie entschieden. Ein Systemtest ist aus Sicht der CAN-Funktionalität erst möglich, wenn die Anwendung und ihre Anforderungen näher spezifiziert werden und die Umgebung (angeschlossene Geräte) vorgegeben wird. Der auszuwählende Test kann damit nur der Integrationstest sein.

Der zuletzt genannte Punkt hat auch einige wirtschaftliche Aspekte. Aus Sicht von Garz & Fricke kann es durchaus als positiv betrachtet werden, dass weitere Spezifikationen nicht vorliegen. Der Kunde muss diese erstellen, implementieren und auch testen. Da die Verantwortung für das Endprodukt beim Kunden liegt, kann Garz & Fricke den Testaufwand reduzieren und damit Geld sparen. Erst wenn der Kunde Fehler im CAN-Modul findet, wird er sich wieder an Garz & Fricke wenden und eine Lösung erwarten. In der Regel haben Kunden zu diesen Zeitpunkten eine sehr rigide Erwartungshaltung und werden Druck ausüben, da ihr Projekt möglicherweise in Verzug gerät. Garz & Fricke muss dann in der Lage sein, innerhalb kurzer Zeit eine Fehlerbehebung durchzuführen oder zumindest andere Lösungsvorschläge zu präsentieren. Das hat wiederum Einfluss auf die Zeitplanung bei Garz & Fricke, weshalb es im Nachhinein doch Kosten verursacht. Wenn Garz & Fricke nicht auf Kundenforderungen zu diesem Zeitpunkt reagiert, schlägt das in jedem Fall in Vertrauensverlust nieder oder kann im schlimmsten Fall sogar Kundenverlust bedeuten. Da das Jupiter ein OEM-Produkt mit Standardbetriebssystem und Standardschnittstellen ist, ist es in der Regel auch schnell austauschbar.

Diese Überlegungen zielen auf den Testaufwand ab. Für den konkreten Fall lässt sich eine Strategie ableiten. Fehlerwirkungen, die mit hoher Wahrscheinlichkeit großen Aufwand bei der Fehlerbehebung produzieren, sollten nicht mehr im Feld auftreten. Zu diesen Kategorien gehört das Senden und Empfangen der Nachrichten. Das Zeitverhalten sollte untersucht werden, um Kundenanforderung bzgl. glsEchtzeitanwendungen von vornherein richtig einschätzen zu können.

4.4.3 Testplanung

Die Vorgehensweise und Rahmenbedingungen der anstehenden Tests müssen geplant werden. Der Lebens- und Projektzyklus eines Produktes hat großen Einfluss darauf, wann welche Tests durchzuführen sind. Dabei wird beachtet, dass der CAN-Treiber in einer Produktlinie eingesetzt wird und jederzeit in neuere Projekte wie Produkte eingebettet werden kann. Tests müssen dann übertragen und wiederholt werden können.

Von wem die Planung und Durchführung der Tests vorgenommen wird, ist ebenfalls entscheidend und sollte in der Planung beachtet werden. Da einzelne Personen evtl. noch nicht zugeordnet werden können bedient man sich Rollen. Bei Garz & Fricke ist dieses zusätzlich eingeschränkt durch ein kleines Entwicklungsteam in diesem Bereich. Ausgewiesene Testspezialisten und hauptberufliche Tester haben sich noch nicht herausgebildet. Für dieses Beispiel werden die Rollen *Entwickler* (Autor einer Komponente oder Durchführender einer Implementierung), *erfahrener Entwickler* und *Projektleiter* ausreichen müssen.

Die strategischen Entscheidungen über die geeigneten Tests wurden im vorherigen Kapitel auf die Komponententests, Review und Integrationstests festgelegt. In diesem Abschnitt kommt noch die Forderung der Wiederholbarkeit, welche mit den Regressionstests in Angriff genommen wird. Diese werden im Kapitel 5.1 näher beschrieben.

- Die Komponententests prüfen in erster Linie die API-Funktionalität. Sie finden idealerweise während der Entwicklung statt. Im aktuellen Beispiel ist der CAN-Treiber bereits fertiggestellt. Es soll insbesondere die API auf Fehler geprüft werden. Nach jeder Änderung am Sourcecode müssen diese Tests wiederholt werden. Der Entwickler des CAN-Treibers ist für die Erstellung und Durchführung dieser Tests verantwortlich.
- Die Integrationstests prüfen die Kommunikation und andere CAN-Funktionalitäten im Zusammenspiel mit anderen angeschlossenen Teilnehmern. Ein exemplarisches CAN-Gerät ist als Gegenstelle zu definieren und aufzusetzen. Reaktionszeiten (A015) sind aufzunehmen und zu beurteilen. Die Durchführung erfolgt vom Entwickler der Komponente. Die Tests werden wiederholt, wenn sich wesentliche Parameter oder die Zusammenstellung des Betriebssystems oder der Hardwareplattform ändern.
- Ein Review wird mit dem Ziel durchgeführt, die Benutzbarkeit und Verständlichkeit der API für Kunden, sowie die Änderbarkeit und Übertragbarkeit des Treibers zu beurteilen. Der Zeitpunkt kann parallel zum Integrationstest sein, da schon etwas Erfahrung im Umgang mit der Schnittstelle zu einer besseren Beurteilung führen kann. Das Review sollte von einem Moderator, einem erfahrenen Entwickler und dem Entwickler des CAN-Treibers durchgeführt werden.
- Alle Tests sollten weitestgehend in einen automatischen Ablauf integriert werden. Dabei ist zu unterscheiden zwischen den Tests für die Entwicklung und den Tests für die Produktion. Alle Entwicklungstests sollen einen hohen Wiederverwendungsgrad haben, damit später bei den Hardwaretests der Produktion nicht doppelter Aufwand entsteht. Diese Schritte werden vom Komponentenentwickler in Zusammenarbeit mit dem Projektleiter durchgeführt. Im Falle der Hardwaretests ist der Testverantwortliche der Produktion mit einzubeziehen.

4.5 Komponententest

Der Komponententest wurde grundlegend in Kapitel 3.2.1 vorgestellt. In diesem Abschnitt erfolgt die Betrachtung des CAN-Treibers auf der Komponentenebene. Methoden zur Testfallerstellung werden vorgestellt und angewandt. Auch die benötigte Umgebung spielt eine große Rolle, da es zum ersten Mal um die Ausführung von Testfällen geht. Die Realisierung der Testfälle und aller Hilfsmittel wird in diesem Abschnitt detailliert beschrieben.

4.5.1 Testobjekt

Der Test des CAN-Treibers auf Komponentenebene konzentriert seine Testaktivitäten ausschließlich auf den Treiber. Das Testobjekt ist der gesamte CAN-Treiber.

An dieser Stelle sei angemerkt, dass dieser aus einer Treiber-DLL und einer API besteht. Jede für sich genommen stellt eine eigene Komponente dar und könnte separat getestet werden. Die Implementierung der API-DLL ist jedoch nur eine Kapselung der komplexeren Schnittstelle des Streaminterface-Treibers (vgl. 4.3). Sie hat keine eigene Intelligenz. Die meisten Funktionen der API rufen ihr Pendant im Streaminterface auf. Das Testen einer solchen Kapselungsschicht ist nicht besonders sinnvoll. Auf der anderen Seite, das Testen des CAN-Treibers am komplexeren Streaminterface verursacht einen höheren Aufwand und macht ebenfalls keinen Sinn, wenn eine angepasste Schnittstelle zur Verfügung steht. Hinzu kommt, dass die Schnittstelle zwischen API und Treiber-DLL nicht dokumentiert ist. Sie kann sich bei neueren Versionen ändern, ohne dass die API geändert werden muss. Das würde wiederum bedeuten, dass die Komponententests auf dieser Ebene ebenfalls nachgezogen werden müssen.

Aus diesen Gründen wird der gesamte CAN-Treiber als einzige Komponente und Testobjekt angesehen. Beide Teile werden somit gleichzeitig auf Fehler geprüft. Treten Fehler auf, so bedeutet das aber, dass stets ermittelt werden muss, in welcher Schicht die Ursache liegt. Die erste Prüfung bei einem Debug-Lauf sollte dann immer die Parameterübergabe zwischen diesen beiden Schichten im Auge haben. Erfahrungsgemäß passieren bei der Datenübergabe immer wieder kleinere Fehler, da die Typsicherheit an dieser Stelle fehlt.

4.5.2 Testumgebung

Die Ausführung der Tests findet auf dem Zielgerät statt. Anders als bei typischen Komponententests, z. B. in objektorientierten Umgebungen, wird die Komponente nicht isoliert ausgeführt. Damit ist ein wesentliches Merkmal eines Komponententests nicht erfüllt und die

Grenze zum Integrationstest ist verwischt. Es gibt jedoch gute Gründe diesen Weg einzuschlagen, welche im Folgenden erläutert werden.

Der CAN-Treiber bietet auf der einen Seite eine API, mit deren Hilfe die Hardware gesteuert werden kann. Zur Bedienung der API werden Testfälle erstellt, die den Treiber ähnlich bedienen, wie eine Applikation dies später tun würde. Auf der anderen Seite des Treibers steht die Hardware. Der Treiber schreibt auf Anforderung der Applikation bestimmte, umgesetzte Werte in die Hardware. Ein Testfall würde also eine API-Funktion ausführen und kontrollieren, ob die korrekten Werte vom Treiber geschrieben wurden. Würde man den Treiber isoliert ausführen, müsste die Hardwareumgebung zu einem großen Teil simuliert werden. Ohne dass die Hardwareregister ein bestimmtes Verhalten aufweisen, würde der Treiber noch nicht einmal starten. Das Simulieren der Hardwareeigenschaften ist sehr aufwändig und damit nicht durchführbar.

Denkbar wäre auch das Ausführen des Treibers in einer Cross-Testumgebung, also auf einem anderen System, z. B. auf einer anderen Hardware oder unter einem anderem Betriebssystem. Die Implementierung des Treibers ist jedoch so stark auf Windows CE zugeschnitten, dass eine Portierung mit viel Aufwand verbunden ist. Eine Hardware mit besseren Eigenschaften bzgl. Testbarkeit ist dem Autor ebenfalls nicht bekannt.

Der Treiber wird also in seiner gewöhnlichen Umgebung gestartet. In Bild 4.1 wurde dieser Aufbau bereits dargestellt. Die dargestellte Applikation.exe dient als Testtreiber und wird für die folgenden Tests den Namen testcan.exe erhalten. Der Vorteil ist, dass die Testfälle jederzeit auf das Zielgerät geladen werden können. Eine spezielle Testversion des Treibers oder gar Betriebssystems ist nicht nötig.

Nun ist es aber schwierig an die Schnittstelle Treiber-Hardware zu kommen. Es gibt zwar Debug-Funktionen, die erlauben Hardwareregister aus der Applikation auszulesen und zu beschreiben (CanReadRegister und CanWriteRegister), doch ist diese Methode nicht dazu geeignet, die Registerwerte eines Controllers abzuprüfen, da sich diese in der Regel sehr schnell ändern. Beispiel: Beim Senden wird von der Software ein Bit im Controller gesetzt. Sobald der Sendevorgang abgeschlossen ist, setzt der Controller dieses Bit wieder zurück. Eine Testfunktion müsste in dieser kurzen Zeit (wenige Microsekunden bei hohen Baudraten) prüfen, ob das Bit gesetzt wurde. Das kann nicht garantiert werden. Insgesamt ist es auch sehr aufwändig und fehleranfällig, die Eingangsaktionen, nämlich den Aufruf der API-Funktionen, dem Zustand der Hardwareregister zuzuordnen. Die Schnittstelle des Treibers zur Hardware kann auf diesem Testlevel daher nicht vollständig geprüft werden.

4.5.3 Testabdeckung

Es stellt sich die Frage, wie viel Funktionalität des CAN-Treibers im Komponententest abgedeckt werden kann. In erster Linie können alle von der API bereitgestellten Funktionen mit unterschiedlichen Parametern aufgerufen werden. In der Regel stellen diese Funktionen Rückgabewerte bereit, die Erfolg oder Misserfolg melden. Dies kann im Test überprüft werden.

Die API stellt an einigen Stellen Set- und Get-Funktionen bereit. Dies ist ein glücklicher Umstand, da ein mit Set gesetzter Wert mittels Get wieder ausgelesen und verglichen werden kann. Die Testabdeckung bei dieser Testmethode hängt von der internen Implementierung ab. Im CAN-Treiber wird ein Parameter der Hardware mittels einer Set-Funktion sofort in die Hardware geschrieben. Möglicherweise passiert noch eine Umsetzung des Wertes der Applikation in die Codierung der Hardware. Beim Aufrufen der Get-Funktion wird dieser Parameter wieder aus der Hardware herausgelesen und evtl. in die Darstellungsform für die Applikation umgesetzt. Das Testen der Set- und Get-Funktionen im Verbund erlaubt also das Prüfen dieser Implementierungspfade im CAN-Treiber. Das Aufspüren von Fehlern an diesen Stellen ist sehr wahrscheinlich, aber nicht garantiert. So kann ein Fehler in der Umsetzung sowohl die Get- als auch Set-Richtung betreffen, wenn z. B. ein Programmierer die Spezifikation der Hardware falsch ausgelegt hat. Diese und andere Fehlerklassen werden erst in der nächsten Stufe abgetestet.

Die Testfälle dieser Teststufe fokussieren sich auf diese Funktionen:

- CanCreateDevice
- CanCloseDevice
- CanGetBaudrate
- CanSetBaudrate
- CanGetBittiming
- CanSetBittiming
- CanGetAddressfilter
- CanSetAddressfilter

Die beiden Funktionen zum Öffnen und Schließen des Treibers (CanCreateDevice, CanCloseDevice) können in dieser Umgebung problemlos geprüft werden. CanCreateDevice gibt ein gültiges Handle[4] zurück. Wenn das erfolgreich ist, hat die Initialisierung des Treibers funktioniert. Dies ist eine Eigenschaft des Treiberladesystems unter Windows CE. Kann ein Treiber beim Systemstart oder später nicht erfolgreich initialisiert werden, kann er folglich von Applikationen nicht mehr geöffnet werden. CanCreateDevice nutzt diesen Mechanismus. Ob die Initialisierung der Hardware auch tatsächlich korrekt durchgeführt wurde, prüfen die weiteren Testfälle implizit mit. Damit sind die referenzierten Anforderungen:

[4] Ein Handle ist eine Art Zeiger, der unter Windows ein Betriebssystem-internes Objekt referenziert.

- A001 (Initialisierung, Seite 44)
- A005 (gängige Baudraten Seite 44)
- A006 (kundenspezifische Baudraten Seite 44)
- A011 (Empfangsfilter Seite 45)

Die verbleibenden Funktionen zur Nachrichtenübertragung, Statusabfrage und Reset können hier nicht sinnvoll getestet werden, da dazu eine Gegenstelle benötigt wird. Sie werden in der nächsthöheren Testschicht, dem Integrationstest aufgegriffen.

4.5.4 Testdaten

Die im vorherigen Kapitel aufgeführten Funktionen, die in diesem Test geprüft werden können, müssen mit geeigneten Testdaten ausgeführt werden. Ein Testtreiber muss jede Funktion mit entsprechenden Parametern aufrufen. Die Bestimmung der Parameter ist Thema dieses Kapitels.

Schon der Datentyp Integer hat auf einem 32-Bit-System 4294967296 mögliche Werte. Das Testen einer Funktion mit allen diesen Möglichkeiten ist sicherlich einfach zu realisieren. Die Ausführungsdauer würde jedoch sehr viel Rechenzeit beanspruchen: Angenommen, die zu testende Methode benötigt 1 ms Ausführungszeit, so kommen wir auf eine Gesamtzeit von knapp 50 Tagen. Schlimmer noch: Bei zwei Parametern multipliziert sich dieses wiederum mit 2^{32}. Siehe dazu auch Testfallexplosion in Kapitel 3.1.3.

Um sinnvolle Parameterwerte an die zu testenden Funktionen zu übergeben, gibt es mehrere Methoden zur Ermittlung. Ein Überblick über gängige Verfahren wurde in den Grundlagen (Kapitel 3.5.2) gegeben. Für die vorliegende Komponente eignen sich die Verfahren der Blackbox-Tests, speziell die Äquivalenzklassenbildung in Ergänzung mit der Grenzwertbildung, die in den nächsten beiden Kapiteln detailliert beschrieben und angewandt werden.

Die Whitebox-Verfahren haben einige Nachteile, die hier kurz aufgezählt aber nicht vertieft werden sollen. In erster Linie sind sie nur werkzeuggestützt durchführbar. Professionelle Werkzeuge, die dieses leisten können, sind allerdings sehr teuer. Der zweite Nachteil ist, dass einige Verfahren eine Instrumentarisierung im Zielgerät benötigen. Das bedeutet, das zur Überprüfung, ob eine Quellcodestelle durchlaufen wurde, ein Testcode in das Testobjekt eingeschleust wird. Dies wirkt sich negativ auf die Laufzeit aus und kann bei hardwarenaher Software Probleme verursachen.

4.5.5 Äquivalenzklassenbildung

Ein Parameter wird gemäß der Spezifikation und der API-Dokumentation in Äquivalenzklassen seiner Eingabewertebereiche unterteilt. Dabei wird nur ein Vertreter einer Klasse an die zu testende Methode übergeben. Es wird davon ausgegangen, dass sich das Testobjekt bei allen anderen Werten einer Klasse gleich verhält. Die verschiedenen Klassen müssen also so gewählt werden, dass Fallunterscheidungen im Verhalten und Ergebnis sichtbar werden. In erster Linie können gültige und ungültige Parameter unterschieden werden. Gültige Parameter können dabei meistens noch in weitere Klassen unterteilt werden.

Ein Beispiel der Unterteilung der Parameter zeigt die Tabelle 4.3 für einige Funktionen der CAN-API.

Funktion	**Parameter**	**Äquivalenzklasse**	**Vertreter**
CanCreateDevice	Nr	1	1
		2..9	2 [1]
		0, 9..MAX_UINT	10
		< 1	0
CanCloseDevice	Handle	gültiges Handle	* [2]
		ungültiges Handle	[3]
CanSetBaudrate	baud	10, 25, 50, 100, 125, 250, 500, 1000	alle
		alle anderen	1
CanSetFilter	code	0..MAX_UINT	0
	mask	0..MAX_UINT	0
CanSetBittiming	JumpWidth	0..3	0
		4..MAX_UCHAR	4
	PreScaler	0..0x3f	1
		0x40..MAX_UCHAR	0x40
	TSeg1	0..0xf	1
		0x10..MAX_UCHAR	0x10
	TSeg2	0..7	1
		8..MAX_UCHAR	8
	Sample3	0	0
		1	1
Get-Funktionen	Zeigervariablen	gültiger Zeiger	* [5]
		ungültiger Zeiger	0 [5]

Tabelle 4.3: Äquivalenzklassen

Nicht alle erlaubten oder möglichen Eingabewerte finden sich in der Spezifikation oder API-Dokumentation wieder. So musste für die Parameter der Funktion SetBittiming das Hardware-Manual des [SJA1000] Aufschluss darüber geben, welche Werte erlaubt sind. Diese Information gehört natürlich ebenfalls in die Dokumentation der API.

4.5.6 Grenzwertbildung

Die Äquivalenzklassenbildung ermittelte einen Satz von Testparametern. Dabei wurde darauf geachtet, dass unterschiedliche Verhaltensmuster des Testobjektes hervorgerufen werden. Die Grenzen der Äquivalenzklassen verdienen jedoch weitere Beachtung. Die Implementierung im Testobjekt beruht fast immer auf einem Vergleichsausdruck, wie z. B. `if(Nr>1 && Nr<9)`. Die Erfahrung zeigt, dass diese oder ähnliche Grenzabfragen nicht selten kleine Fehler enthalten. So auch in diesem Beispiel: Der zitierte Quellcodeausschnitt ist in der Funktion Open zu finden und müsste lt. Dokumentation eigentlich `if(Nr>=1 && Nr<=9)` heißen.

Die Überprüfung der Grenzen von Äquivalenzklassen erfordert die Ausweitung der Parametersätze um sinnvolle Werte am Rande, innerhalb und außerhalb der Äquivalenzklassen. Die Tabelle 4.4 erweitert das Beispiel aus Kapitel 4.5.5.

Die Tabelle enthält nur gültige Werte, d. h., denkbare Grenzwerte außerhalb des Wertebereichs der entsprechenden Variablen wurden weggelassen. So gibt es z. B. beim Parameter Sample3 nur die beiden Zustände 1 und 0, da es ein boolscher Typ ist.

Auf die Grenzwertbildung einiger Parameter wird verzichtet:

- Parametertypen wie Handle und Zeigervariablen (Datenflussrichtung out) haben keine sinnvollen Grenzwerte. Die Werte werden vom Compiler oder Betriebssystem dynamisch erzeugt. Bei Handles sorgt das Betriebssystem für eine Fehlerüberprüfung. Ungültige Zeiger (nicht null) in den Programmiersprachen C/C++ können nicht wirksam erkannt werden.
- Die Einstellung der Baudrate erlaubt nur fest vordefinierte Werte. Diese werden mittels einer „case“ Struktur implementiert. Zwischenwerte abzuprüfen macht keinen Sinn.

[1] Sofern nur eine einzige CAN-Treiber-Instanz mit Index 1 aktiv ist.
[2] Ein gültiges Handle wird von der Open-Funktion zurückgegeben.
[3] Wird nicht getestet, da kein Rückgabewert verfügbar. Auch bei weiteren Funktionen wird der Parameter vom Typ Handle nicht mit verschiedenen oder falschen Werten beaufschlagt. Es gibt für diesen Parameter nur einen gültigen Wert, der dynamisch ist. Intern wird dieser ohne weitere Verarbeitung an Betriebssystem-Funktionen übergeben. Man kann sich darauf verlassen, dass die Betriebssystemfunktionen falsche Handles erkennen.
[4] Ein gültiger Zeiger wird vom Compiler erzeugt.
[5] Ein ungültiger bzw. falscher Zeiger kann in C/C++ nicht erkannt werden. Es wird meistens die 0 überprüft.

Funktion	Parameter	u. Grenzwert [Äquivalenzklasse] o. Grenzwert
CanCreateDevice	Nr	0, [1], 2
		1, [2, 3, .., 8, 9], 10
CanSetFilter	code	[0, 1, .., MAX_UINT-1, MAX_UINT]
	mask	[0, 1, .., MAX_UINT-1, MAX_UINT]
CanSetBittiming	JumpWidth	[0, 1, 2, 3], 4
		3, [4, 5, .., MAX_UCHAR-1, MAX_UCHAR]
	PreScaler	[0, 1, .., 0x3e, 0x3f], 0x40
		0x3f, [0x40, 0x41, .., MAX_UCHAR-1, MAX_UCHAR]
	TSeg1	[0, 1, .., 0xe, 0xf], 0x10
		0xf, [0x10, 0x11, .., MAX_UCHAR-1, MAX_UCHAR]
	TSeg2	[0, 1, .., 6, 7], 8
		7, [8, 9 .., MAX_UCHAR-1, MAX_UCHAR]
	Sample3	[0]
		[1]

Tabelle 4.4: Grenzwertbildung

Bei strenger Betrachtung und Einbeziehung des Quellcodes muss angemerkt werden, dass die Parameter der Funktion SetBittiming intern mittels Bitmasken und Schiebeoperatoren verarbeitet werden. Auch dieses passt nicht zum o. g. Fehlerbild. Damit können nicht alle Fehler entdeckt werden. Die Prüfung der ermittelten Testparameter ist trotzdem sinnvoll, da immerhin Fehler wie z. B. „eine um ein Bit zu klein gewählte Bitmaske“ aufgespürt werden können.

4.5.7 Testfälle

Testfall	Nr	Rückgabewert	Fehlercode
0101	1	true	0
0102	2	false	55
0103	3	false	55
0104	8	false	55
0105	9	false	55
0106	10	false	87
0107	0	false	87

Tabelle 4.5: Testfälle der Funktion CanCreateDevice

Die Tabellen 4.5 und 4.6 zeigen die Testfälle exemplarisch für die Funktionen zum Öffnen

Testfall	pBittiming	JW	Scale	TSeg1	TSeg2	S3	Rückg.	Fehler
0301	gültig	0	0	0	0	1	true	0
0302	gültig	1	1	1	1	0	true	0
0303	gültig	2	0x3e	0xe	6	0	true	0
0304	gültig	3	0x3f	0xf	7	0	true	0
0305	gültig	4	1	1	1	0	false	87
0306	gültig	5	1	1	1	0	false	87
0307	gültig	0xfe	1	1	1	0	false	87
0308	gültig	0xff	1	1	1	0	false	87
0309	gültig	0	0x40	1	1	0	false	87
0310	gültig	0	0xff	1	1	0	false	87
0311	gültig	0	1	0x10	1	0	false	87
0312	gültig	0	1	0xff	1	0	false	87
0313	gültig	0	1	1	8	0	false	87
0314	gültig	0	1	1	0xff	0	false	87
0315	ungültig	0	1	1	1	0	false	87

Tabelle 4.6: Testfälle der Funktionen CanSetBittiming und CanGetBittiming

des Treibers, Einstellen und Abfragen des Bittimings. Die Reihenfolge der Tests wird so gewählt, dass Get- und Set-Funktionen paarweise aufgerufen werden. Ein mittels Set-Funktion gesetzter Wert wird nochmals abgerufen und überprüft. Zusätzlich wird der Rückgabe- und ein evtl. Fehlercode der Funktionen überprüft.

Die Testdaten (Parameter und Rückgabewerte) stammen aus den Überlegungen der Äquivalenzklassen- und Grenzwertbildung. Sie sind nun so eingesetzt, dass möglichst wenige Testfälle dabei herauskommen, d. h. die Ausführungsdauer minimiert wird. Jeder Wert eines Parameters muss dabei mindestens einmal vorkommen. Sind mehrere Parameter vorhanden, so müssen nicht alle Werte miteinander kombiniert werden.

Eine Implementierung wird beispielhaft für die Testfälle der Funktion CanCreateDevice in Listing 4.1 (Kap. 4.5.9) gezeigt.

4.5.8 Testendekriterium

Um den Testaufwand nicht unnötig in die Höhe zu treiben, sollte jederzeit klar sein, wann, wie viel, wie lange getestet werden muss. In erster Linie gibt die Anforderungsliste darüber Auskunft, wann welche Kriterien erreicht sind. Zusätzlich müssen an dieser Stelle der Rahmen des Komponententests und die besonderen Umgebungsbedingungen in diesem Fall

beachtet werden. Die Testfälle des Komponententests bearbeiten mit akribischer Genauigkeit einen Teil der API. Aus Sicht der Anforderung A015 (Fehlertoleranz, Seite 46) ist diese auch gerechtfertigt, da insbesondere unerwartete falsche Eingaben seitens der Applikation die API nicht zum Absturz bringen sollten. Die funktionalen Anforderungen können auf diesem Testlevel leider nicht vollständig umgesetzt werden, obwohl es an dieser Stelle wünschenswert wäre. So ist z. B. das Senden und Empfangen von Nachrichten erst auf einer weiteren Integrationsstufe testbar.

4.5.9 Framework

Bei der Implementierung von Testfällen auf Modulebene gibt es immer wiederkehrende Aufgaben und Muster, die nicht vielfach implementiert werden wollen. Das reine Ausführen von Testfällen, Hinzufügen neuer Testfälle, Überprüfen von Parametern, detailliertes Auflisten von Fehlerzuständen usw. sollen möglichst komfortabel benutzbar sein. So kann sich der Entwickler der Testimplementierung auf die Testfälle selbst konzentrieren ohne mit diesen Verwaltungsaufgaben konfrontiert zu werden.

Das bekannteste Framework seiner Art ist JUnit für die Programmiersprache Java. Auch für C und C++ wurden mittlerweile einige Testframeworks entwickelt. Eine gute Aufarbeitung und Vergleich der Komponenten findet sich in [Llopis, 2004] und hat auch für diese Arbeit als Vorauswahl gedient. Die folgenden Test-Frameworks stehen damit zur Auswahl:

- CppUnit
- Boost.Test
- CppUnitLite
- NanoCppUnit
- Unit++
- CxxTest

Der Vollständigkeit halber sei hier noch das CETK von Microsoft erwähnt, das im Kapitel 2.2.4 beschrieben wurde. Das enthaltene Framework nennt sich TUX.

Alle Testhelfer haben ihre jeweilige Spezialität, die meisten jedoch mindestens einen gravierenden Nachteil:

- Abhängigkeit von nicht vorhandenen Komponenten oder
- Einarbeitungsaufwand

So benötigen CppUnit, Boost.Test und Unit++ die Standard Template Library (STL) als Grundlage. STL ist unter Windows CE von Microsoft jedoch nicht vollständig implementiert.

CxxTest benötigt Perl zur Erzeugung des Testcodes. Die Integration von Perl und der Microsoft Buildumgebung ist zwar nicht unüberwindbar, fällt aber ebenfalls in des Kategorie eines großen Einarbeitungsaufwandes.

CppUnitLite und NanoCppUnit deuten schon aufgrund ihrer Bezeichnung eine einfache Realisierung an. Tatsächlich gibt es keine Abhängigkeiten, aber auch wenig Funktionsumfang. So fehlt z. B. die Umleitung und Anpassung der Ausgaben. Diese können in Windows CE nötig werden, da es einen separaten Kanal für Debug-Ausgaben hat. Meistens ist dieser auf eine serielle Schnittstelle gelegt. Ausgaben der Tests können damit in zeitlichen Zusammenhang mit Debug-Ausgaben der Testobjekte gebracht werden.

Mit TUX sind alle Tests von Microsoft im CETK realisiert. Die Schnittstelle zur Bedienung der enthaltenen Funktionalität ist umfangreich und aufwändig in der Benutzung. Ein Wizard in der Entwicklungsumgebung erzeugt schon für ein leeres Projekt fünf Dateien. Damit fällt das CETK ebenfalls in die Kategorie des hohen Einarbeitungsaufwandes.

Aufgrund dieser Betrachtung entschied sich der Autor, kein vorhandenes Framework einzusetzen. Beim Durcharbeiten und Ausprobieren der Frameworks zeigte es sich vielmehr, wie wenig Funktionalität tatsächlich gebraucht wird und wie einfach die Implementierung dafür ist, was später in Listing 4.2 erläutert wird.

Das Listing 4.1 zeigt die Implementierung der Testfälle aus Tabelle 4.5. Die lokale Struktur `TT` definiert Eingabe- und Ausgabetypen. Die nachfolgende Tabelle `testcase[]` enthält die Testfälle mit den Eingabedaten und zu erwartenden Ausgabedaten. Eine for-Schleife durchläuft pro Testfall den Aufruf der Funktion, die Überprüfung der Ausgabewerte und ein Aufräumen (in diesem Fall das Schließen des Treibers).

Listing 4.1: Testfälle für die Funktion CanCreateDevice auf Komponentenebene

```
1  void testAPI_CanCreateDevice()
2  {
3      struct TT {
4          DWORD param1;
5          bool expectedValidHandle;
6          DWORD expectedLastError;
7      } ;
8
9      TT testcase[] = {
10         {-1, false, ERROR_INVALID_PARAMETER},
11         { 0, false, ERROR_INVALID_PARAMETER},
12         { 1, true, 0},
13         { 2, false, ERROR_DEV_NOT_EXIST},
14         { 9, false, ERROR_DEV_NOT_EXIST},
15         {10, false, ERROR_INVALID_PARAMETER}
16     };
17
```

```
18      log(L"CanCreateDevice: %d testcases\r\n", sizeof testcase / sizeof
            TT);
19      for(int i=0; i<sizeof testcase / sizeof TT; i++) {
20          hCAN = CanCreateDevice(testcase[i].param1);
21          if(testcase[i].expectedValidHandle) {
22              CHECK_TRUE(hCAN, i);
23              CHECK_TRUE(hCAN != INVALID_HANDLE_VALUE, i);
24          } else {
25              CHECK_TRUE(!hCAN, i);
26          }
27          CHECK_COMPARE(GetLastError(), testcase[i].expectedLastError, i);
28          if(hCAN)
29          {
30              CanCloseDevice(hCAN);
31          }
32      }
33  }
```

Das Vergleichen der Ergebnisdaten mit den Solldaten gehört zu den am häufigsten wiederkehrenden Aufgaben. Schlägt eine Überprüfung fehl, so ist es sinnvoll, möglichst viele nützliche Informationen auszugeben, an welcher Stelle des Testablaufs ein Fehler passiert und welche Zustände die jeweiligen Variablen haben. Dies kann evtl. einen Debug-Lauf einsparen. Das Listing 4.2 zeigt die Makrofunktion `CHECK_COMPARE`, die die eigentliche Überprüfung durchführt. Erst wenn die Bedingung fehlschlägt werden alle Variablen in Text und Inhalt sowie Informationen über die Quellcodestelle (Datei, Funktion und Zeilennummer) an eine C-Funktion weitergereicht, die diese geordnet ausgibt.

Listing 4.2: Implementierung einer Assert Funktion zum Vergleich von Testdaten

```
1  #define CHECK_COMPARE(expr1, expr2, testcase) { \
2      if((expr1) != (expr2)) {\
3          check_compare_failed(TEXT(#expr1), TEXT(#expr2), expr1, expr2,
               testcase, TEXT(__FILE__), __LINE__, TEXT(__FUNCTION__)); \
4      } \
5  }
6
7  void check_compare_failed(LPCWSTR expr1, LPCWSTR expr2, DWORD is1, DWORD
       is2, int testcase, LPCWSTR file, int line, LPCWSTR func)
8  {
9      log(L"COMPARE FAILED case %i\r\n", testcase);
10     log(L"    expression 1: '%s' is %d (0x%x)\r\n", expr1, is1, is1);
11     log(L"    expression 2: '%s' is %d (0x%x)\r\n", expr2, is2, is2);
12     log(L"    file: %s\r\n", file);
13     log(L"    line: %d, function \"%s\"\r\n\r\n", line, func);
14     inc_error_count();
15 }
```

Dieses Listing verdeutlicht wie einfach es tatsächlich ist, Asserts auch unter C zu implementieren. Einziger Nachteil dieser Implementierung: Sie muss für jeden Datentypen, der geprüft werden soll, wiederholt werden. In der reinen objektorientierten Welt wäre dies mit Polymorphie- und Reflection-Mechanismen wahrscheinlich nicht nötig. Für die Testfälle des CAN-Treibers hat der Autor lediglich eine weitere Assert-Implementierung mit dem Typen Bool gebraucht. Das bestärkte die Entscheidung auch im Nachhinein, nicht auf vorhandene Frameworks gesetzt zu haben.

Das Builden der Applikation mit den Testfällen (testcan.exe) wird mit dem Platform Builder, der Entwicklungsumgebung von Windows CE 5.0, durchgeführt. Das Programm wird anschließend auf das Zielsystem über FTP[5] oder einer Speicherkarte übertragen. Das Gerät hat einen Bildschirm und stellt darauf auch eine Kommandozeile bereit, um die Tests zu starten. Es ist jedoch recht mühselig, die Kommandozeile auf einem 5,4 Zoll Bildschirm mit Stift und virtueller Tastatur zu bedienen. Per USB kann zwar eine Tastatur angeschlossen werden, doch streikte zum Zeitpunkt der Entwicklung der Testfälle der USB-Treiber. Eine weitere, bequeme Möglichkeit ist Telnet. Es stellt die Kommandozeile des Gerätes über das Netzwerk zur Verfügung. Telnet und FTP werden für die Automatisierung im Kapitel 5 noch eine tragende Rolle bekommen.

4.5.10 Ergebnisse

Die Überprüfung der CAN-API mittels des Komponententests konnte sehr einfach und schnell realisiert werden. Die Testfallerstellung mithilfe der Äquivalenzklassen- und Grenzwertbildung liefert eine solide Testbasis und überprüft im besten Fall auch gleich die Dokumentation der API mit. So ist z. B. aufgefallen, dass die Parameter der Funktion SetBitTiming nicht dokumentiert waren. Der Benutzer dieser Funktion hätte damit keine Chance gehabt, eine korrekte kundenspezifische Baudrate einzustellen. Die Dokumentation konnte mit dieser Erkenntnis schnell erweitert werden. Es wurden die gültigen Wertebereiche der einzelnen Parameter und ein Verweis auf das [SJA1000] Manual zwecks weiterer Details hinzugefügt. Damit die Fehlerbehebung sich nicht mit den Testaktivitäten vermischt, wurde zuerst ein Eintrag ins Fehlermanagementsystem gemacht. Erst nach Abschluss der Testaktivitäten wurde die Korrektur durchgeführt.

Bei der Ausführung der Tests konnte auch festgestellt werden, dass der Parameter `TSeg2` nicht auf falsche Eingabewerte hin überprüft wird. Das Übergeben von ungültigen Werten kann damit die Funktionalität negativ beeinflussen, z. B. eine falsche Baudrate einstellen. Bei ungültigen Werten sollte die API einen entsprechenden Fehlercode zurückgeben, damit der Programmierer einer Applikation sich nicht in falscher Sicherheit wiegt. Auch dieses wurde ins Fehlermanagementsystem aufgenommen und nach Abschluss des Tests korrigiert.

[5] File Transfer Protocol (FTP)

4.6 Integrationstest

Der Fokus dieser Teststufe richtet sich auf die Hard- und Softwareintegration der CAN-Funktionalität. Der Treiber konnte auf Komponentenebene aus genannten Gründen nicht auf seine gesamte Funktionalität überprüft werden. So wurde nicht versucht Nachrichten zu senden oder zu empfangen, was einen elementaren Funktionsumfang des Treibers darstellt. Dies wird auf dieser Stufe nachgeholt. Testobjekte sind der CAN-Treiber, die API und die Hardware. Die Umgebung wird auch Fremdgeräte und -software enthalten, weshalb eine Zuordnung der Fehlerursache nicht ohne weiteres möglich ist.

4.6.1 Testumgebung

Die Gegenstelle zur Nachrichtenübertragung wurde bisher nicht auf ein Gerät festgelegt. Im Sinne der Kundenorientierung könnte es sinnvoll sein, sich Geräte anzuschaffen, die auch beim Kunden zum Einsatz kommen. Integrationsprobleme könnten damit schon zu einem gewissen Teil bei Garz & Fricke festgestellt werden. Dagegen spricht allerdings meist eine starke Spezialisierung dieser Geräte. Die Kosten dieser Geräte sind höher und die Programmierung aufwändiger als von universell einsetzbaren Geräten. In einigen Beispielen gibt es auch anwendungsbedingte Einschränkungen, wie z. B. die fehlende Möglichkeit, den Adressfilter zu setzen. Ein frei programmierbarer CAN-Controller als Gegenstelle erlaubt dagegen das gesamte CAN-Funktionalitätsspektrum.

Der CAN-Adapter PCAN von Peak Systems ist ein solcher Controller. Er unterstützt die CAN 2.0 Spezifikation, das Host Interface ist der USB-Bus. Treiber liegen für Windows XP vor, was bei Garz & Fricke die Standardumgebung ist. Eine API erlaubt das Einbinden in eigene Programme. Ein kleines Monitor Programm erlaubt das Mitschneiden und Senden von Nachrichten auf dem CAN-Bus. Dieser Adapter wurde schon für vorhergehende Projekte angeschafft und steht somit zur Verfügung.

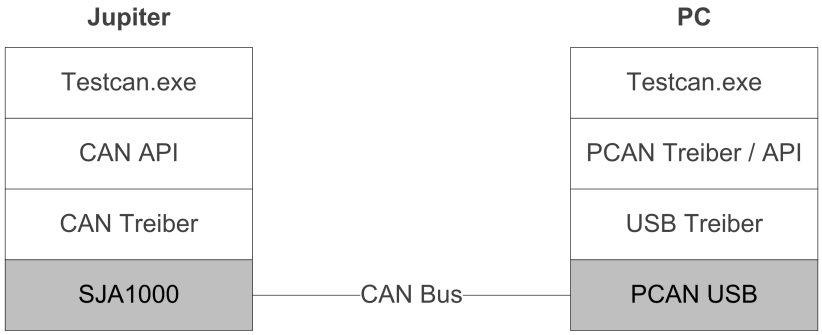


Abbildung 4.2: Umgebung des Integrationstests

Die Testsoftware wird auf zwei Geräten ausgeführt: Dem Jupiter und einem PC. Das Bild 4.2 stellt die beteiligten Komponenten schematisch dar. Bei der Implementierung der Testfälle muss nun auf ein sinnvolles Konzept geachtet werden, damit beide Seiten zur richtigen Zeit das Richtige tun, sich also synchronisieren.

4.6.2 Testkonzept

Die einfachste Möglichkeit, eine Synchronisation der beteiligten Instanzen zu implementieren, ist eine Master-Slave-Aufteilung. In anderen Bereichen könnte es Client-Server-Architektur heißen, doch im Bereich der Feldbusse ist Master-Slave gängiger.

Die Master-Seite enthält alle Testfälle. Sie instruiert die Slave-Seite mit vorher definierten Nachrichten. Die Slave-Seite enthält keinerlei intelligente Testlogik, sondern nur hilfreiche Funktionen, die während der Testausführung benötigt werden. Dies sind z. B. die Umschaltung der Baudrate, eine Sendefunktion mit einstellbarer Nachrichtenanzahl und eine Echo-Funktion. Die Master-Seite sendet Kommandos, die diese Funktionen ein- oder ausschalten.

Zu diesem Zeitpunkt lässt sich noch nicht sagen, ob der PC die Master- oder die Slave-Seite implementieren soll. Zur Fehlersuche kann es sogar hilfreich sein, die Rollen zu tauschen. Die Testsoftware trägt dem Rechnung, indem sie mithilfe von Objektorientierung die Testlogik in einer abstrakten, hardwareunabhängigen Klasse implementiert. Die CAN-APIs des Jupiter und des PCAN sind zwar ähnlich aber nicht identisch bzw. kompatibel. So muss z. B. für den PCAN die Baudrate schon beim Öffnen des Treibers angegeben werden.

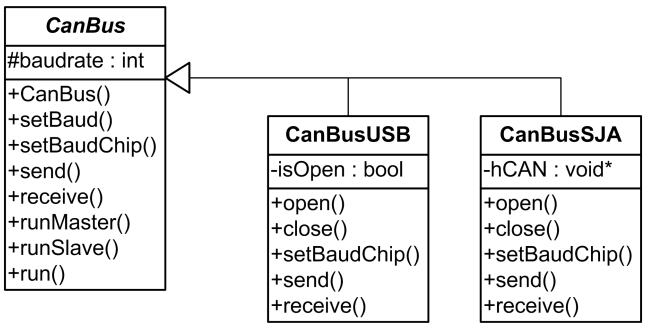


Abbildung 4.3: Klassendiagramm der Testimplementierung

Je nachdem, auf welchem System das Programm gestartet wird, wird eine Spezialisierung der Klasse mit der jeweiligen Anbindung an die auf dem System vorhandene API geladen. Abbildung 4.3 zeigt die Struktur in einem Klassendiagramm. Auf der abstrakten Ebene lassen sich nun Master- und Slave-Funktionalität entwickeln. Das Umschalten kann mittels Kommandozeilenparameter zur Laufzeit passieren.

Die Slave-Seite benötigt folgenden Funktionsumfang:

- Baudrate umschalten. Ist die Änderung der Baudrate nicht nötig, weil die angeforderte schon eingestellt ist, wird keine Programmierung der Baudrate vorgenommen, um Zeit zu sparen.
- Nachrichtenzähler auf Null setzen.
- Zählen eingegangener Testnachrichten. Kommandonachrichten werden nicht gezählt.
- Nachrichtenzähler abfragen. Es wird eine Antwort mit der Anzahl empfangener Nachrichten gesendet.
- Testnachrichten senden. Die Anzahl ist im Datenfeld enthalten.
- Echo-Funktion ein- und ausschalten.
- Slave beenden.

Zum Aufrufen dieser Funktionen werden Kommandos definiert, die je eine eigene CAN-Adresse erhalten. Dadurch ist eine einfache Unterscheidung möglich. Der Testablauf der Master-Seite wird im Folgenden aufgezählt:

- Sende 100 Nachrichten. (A002)
- Prüfe, ob 100 Nachrichten beim Slave angekommen sind. (A002, A003, A009)
- Empfange 100 Nachrichten. (A003, A009)
- Prüfe, ob 100 Nachrichten empfangen wurden. (A003, A009)
- Sende STANDARD Nachricht und vergleiche empfangene Nachricht (Echo). (A002, A003, A004, A009)
- Sende EXTENDED Nachricht und vergleiche empfangene Nachricht (Echo). (A002, A003, A004, A009)
- Wiederhole alle Punkte für alle gängigen Baudraten. (A005)
- Wiederhole alle Punkte mit einer großen Anzahl Nachrichten. (A005)

Der Vorteil dieser Art der Implementierung zeigt sich beim Hinzufügen neuer Testfälle. Ab einem gewissen Grad bleibt die Slave-Seite stabil und die Anpassung der Testlogik, das Ändern der Testdaten sowie das Hinzufügen von neuen Testfällen findet nur auf der Master-Seite statt.

4.6.3 Realisierung

Die Implementierung der Funktionalität für die Slave-Seite wird in Listing 4.3 gezeigt. In einer Schleife wird zunächst auf eine Nachricht gewartet (`receive`). Die `id` der Empfangenen Nachricht (Adresse) entscheidet dann über die Aktion. Die gelisteten Kommandos sind: Umschaltung der Baudrate, Anforderung zum Senden einer Anzahl Nachrichten und Schalten der Echo-Funktion. Die Parameter für die entsprechende Aktion stecken in den Datenbytes

der Nachricht, wie z. B. die Baudrate. Wenn nötig, werden die acht Bytes des Datenfeldes `data` in einen 64-Bit-Integer umgewandelt (cast).

Listing 4.3: Implementierung der Slave-Seite (Auszug)

```
1  void CanBus::runSlave()
2  {
3      BYTE echo = false;
4      int count = 0;
5
6      while(!stopSlave)
7      {
8          int id;
9          BYTE data[8];
10         int len;
11
12         receive(&id, data, &len);
13
14         switch(id)
15         {
16         case MSG_ID_BAUD:
17             unsigned __int64 data64;
18             data64 = *(unsigned __int64 *) data;
19             log(L"Slave: change baud to %d\r\n", (int)data64);
20             setBaud((int)data64);
21             break;
22
23         case MSG_ID_START_TX:
24             data64 = *(unsigned __int64 *) data;
25             log(L"Slave: start sending %d messages\r\n", (int) data64);
26             for(unsigned __int64 i=0; i<data64; i++)
27                 send(MSG_ID_DUMMY, true, i);
28             break;
29
30         case MSG_ID_ECHO:
31             echo = data[0];
32             log(echo ? L"Slave: echo on\r\n" : L"echo off\r\n");
33             suppress_print_message = echo != 0;
34             log(L"Slave: suppress message printing %s\r\n", suppress_print_message ? L"on"
                   : L"off");
35             break;
36
37         case MSG_ID_DUMMY:
38         default:
39             if(echo)
40                 send(MSG_ID_DUMMY, true, false, data, len);
41             count++;
42             break;
43
44         ...
45         }
46     }
47 }
```

Die tatsächliche Implementierung enthält noch weitere Codezeilen, die hier aus Gründen der besseren Lesbarkeit nicht abgedruckt wird. In der Praxis haben sich z. B. Mechanismen bewährt, die die Debug-Meldungen intelligent zu- und abschalten können. Für die Fehlersuche und zum Überprüfen des korrekten Ablaufs sind viele und detaillierte Debug-Meldungen

sinnvoll. In anderen Situationen können zu viele Meldungen jedoch die Übersicht verschlechtern oder gar Einfluss auf die Ausführungszeit haben, was bei späteren Zeitmessungen zum Tragen kommen wird.

Die Implementierung der Funktionalität für die Master-Seite wird in Listing 4.4 gezeigt. Ähnlich wie schon in den Testfällen im Komponententest (Kapitel 4.5.7, Listing 4.1) wird versucht die Testdaten zu trennen. Die Struktur `TT` und die Tabelle `testcase[]` definieren die Testdaten. Die anschließende for-Schleife iteriert über alle hinweg. Die Testlogik enthält drei Aktionstypen: `tx` zum Senden einer Anzahl von Nachrichten zur Slave-Seite und anschließendes Überprüfen, ob alle angekommen sind; `rx` zum Anstoßen des Slaves zum Senden einer Anzahl Nachrichten; `tx_verify` zum Senden und Empfangen einer Nachricht. Das dient dazu, die Sende- und Empfangsfunktionen zu prüfen. Die Testfälle sind nur ausschnittsweise abgedruckt. Die vollständige Tabelle enthält Testdaten für sehr viele Nachrichten unter verschiedenen Baudraten und Nachrichtenformaten.

Listing 4.4: Implementierung der Master-Seite (Auszug)

```
1  void CanBus::runMaster()
2  {
3      enum action_type { tx_verify, tx, rx };
4
5      struct TT {
6          bool quickModeTest;
7          action_type action;
8          bool extended;
9          DWORD count;
10         int baud;
11     } ;
12
13     TT testcase[] = {
14         {true,  tx,         true,   100, initial_baud},
15         {true,  rx,         true,   100, initial_baud},
16         {true,  tx_verify,  true,   100, initial_baud},
17         {false, tx,         true,   100, 1000},
18         {false, rx,         true,   100, 1000},
19         {false, tx_verify,  true,   100, 1000},
20         ...
21     };
22
23     int id, len;
24     BYTE data[8];
25     __int64 data64;
26
27     for(int i=0; i<sizeof testcase / sizeof TT; i++) {
28         if(onlyQuickModeTests && !testcase[i].quickModeTest)
29             continue;
30         log(L"****** starting test case %d *********\r\n", i);
31         setBaud(testcase[i].baud);
32
33         switch(testcase[i].action)
34         {
35         case tx_verify:
36             send(MSG_ID_ECHO, true, 1);
37             log(L"master: start sending %d messages with verify\r\n", testcase[i].count);
38             for(DWORD j=0; j<testcase[i].count; j++) {
```

```
39                  send(MSG_ID_DUMMY, testcase[i].extended, j);
40                  receive(&id, data, &len);
41                  data64 = *(__int64 *) &data;
42                  CHECK_COMPARE(id, MSG_ID_DUMMY, i);
43                  CHECK_COMPARE(len, 8, i);
44                  CHECK_COMPARE((int)data64, j, i);
45              }
46              send(MSG_ID_ECHO, true, 0);
47              break;
48
49          case tx:
50              send(MSG_ID_RESET_COUNT, true, 0);
51
52              for(DWORD j=0; j<testcase[i].count; j++)
53                  send(MSG_ID_DUMMY, true, j);
54
55              send(MSG_ID_GET_COUNT, true, 0);
56              receive(&id, data, &len);
57              data64 = *(__int64 *) &data;
58              CHECK_COMPARE((int)data64, testcase[i].count, i);
59              break;
60
61          case rx:
62              send(MSG_ID_START_TX, true, testcase[i].count);
63              for(DWORD j=0; j<testcase[i].count; j++) {
64                  receive(&id, data, &len);
65                  data64 = *(__int64 *) &data;
66                  CHECK_COMPARE(id, MSG_ID_DUMMY, i);
67                  CHECK_COMPARE(len, 8, i);
68                  CHECK_COMPARE((int)data64, j, i);
69              }
70              break;
71          }
72      }
73  }
```

Die Anzahl der zu sendenden Nachrichten wird an die Baudrate angepasst, damit die Ausführungsdauer nicht übermäßig lang wird. Die Testfälle werden in dieselbe Applikation implementiert, wie schon die Komponententests. Die Auswahl erfolgt über Kommandozeilenparameter. Die Aufrufsyntax der Testsoftware sieht damit so aus:

```
1       Syntax:  testcan.exe  [scenario] [options]
2           Scenarios:
3                       -x1     component tests
4                       -x3     integration tests
5                       -M      simple monitor mode
6           Options:
7                       -s      slave (only integration tests)
8                       -m      master (only integration tests)
9                       -b      baudrate (only monitor mode)
```

Das Programm wird sowohl für Windows CE als auch für den PC gebaut. Für Windows CE wird weiterhin die Umgebung des Platform Builder verwendet. Für den PC wird Visual

Studio 2005 verwendet. Der Sourcecode ist dabei jeweils derselbe. Lediglich die Compiler-Steuerung unterscheidet sich etwas: Unter Windows CE werden sog. SOURCES-Dateien verwendet, die eine Konfiguration für den Make-Prozess enthalten. Für den PC reicht ein einfaches Makefile aus. Es ist wichtig, dass für beide Umgebungen separate Kommandozeilenumgebungen mit den richtigen Umgebungsvariablen gestartet werden, damit Compiler und Bibliotheken gefunden werden. Damit das reibungslos funktioniert, liegt der Sourcecode in einem übergeordneten Ordner und die SOURCES und Makefiles in getrennten Unterordnern. Als Ergebnis erhalten wir zwei ausführbare Dateien, die jeweils nur auf dem vorgesehenen Zielsystem laufen.

Gestartet wird zuerst die Slave-Seite. Wenn die Angabe des Szenarios entfällt, werden alle enthaltenen Testfälle nacheinander durchgeführt. Wird das Programm auf dem PC gestartet, so entfallen natürlich die Komponententests, da sie nur gegen die CAN-API auf dem Jupiter funktionieren. Die Slave-Seite lädt die entsprechende Klasse zur Steuerung der API auf dem PC oder auf dem Jupiter und initialisiert die Hardware. Danach wartet sie auf ankommende Nachrichten.

Die Master-Seite fängt sofort nach dem Initialisieren der Hardware mit dem Testablauf an. Kommandos und Testnachrichten werden gesendet. Rückmeldungen der Slave-Seite und die Statusinformationen des CAN-Busses werden ausgewertet.

4.6.4 Fehlertoleranz

Das Auftreten von Fehlerzuständen in einem System muss nicht gleich bedeuten, dass es seinen Dienst auf alle Ewigkeiten versagt. Anforderung A013 (Seite 45) fordert ein Weiterarbeiten des Systems. Es ist wichtig, dass Fehlerzustände erkannt werden und die Applikation die Möglichkeit bekommt, passend darauf zu reagieren. Folgende Fehlermöglichkeiten sind denkbar und liegen im Verantwortungsbereich der CAN-Komponente:

1. Applikation übergibt falsche Parameter an die API.
2. Unerwartetes Verhalten des Controllers.
3. Fehler bei der Nachrichtenübertragung.

Punkt 1 liegt durchaus im Verantwortungsbereich der CAN-API. Wenn die Applikation ungültige Werte übergibt, so darf das in keinem Fall die Funktionalität der CAN-Komponente negativ beeinflussen. Damit der Applikationsprogrammierer sich nicht falscher Sicherheit wiegt, sollte die API auch einen entsprechenden Fehlercode zurückgeben. Dieses Verhalten wurde schon in den Testfällen der Komponentenebene im Kapitel 4.5.7 geprüft.

Der Punkt 2 ist etwas schwierig zu fassen. Unerwartetes Verhalten kann viele Ursachen und Ausprägungen haben. Entweder kann der Controller einen Defekt haben oder die Software behandelt eine selten auftretende Situation nicht oder nicht korrekt. Bei Hardwaredefekten ist

das Verhalten der Hardware nicht mehr vorhersagbar. Hier sollte der Programmierer soweit vorsorgen, dass entsprechende Debug-Nachrichten oder Fehlercodes ausgegeben werden, wenn unerwartete Situationen auftreten. Beide Fälle sind nur mit sehr hohem Aufwand zu testen. Die Punkte können aber durchaus für das Review in Kapitel 4.7 notiert werden.

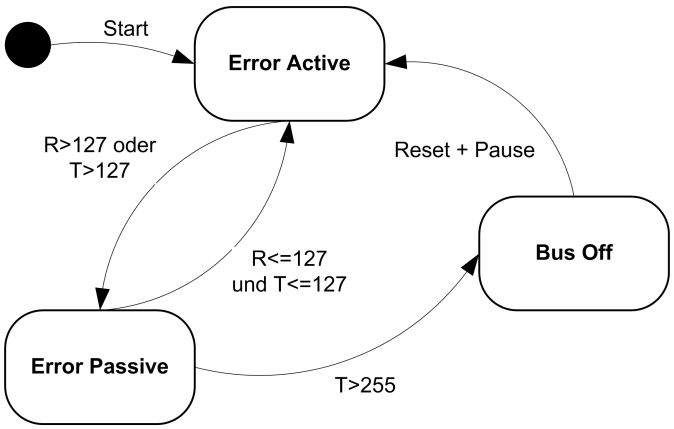


Abbildung 4.4: CAN-Bus-Fehlerzustände

Fehler auf dem CAN-Bus (Punkt 3) sind bereits gut in der CAN-Spezifikation ([CAN 2.0, 1991]) behandelt. Die Hardware stellt diese Funktionalität vollständig zur Verfügung. Der Treiber ist lediglich in der Pflicht, Fehler die von der Hardware erkannt und gemeldet werden, an die Applikation weiterzureichen. Die Fehlerzustände eines CAN-Teilnehmers zeigt das Bild 4.4. Durch entsprechende Maßnahmen sind CAN-Controller dazu in der Lage zu erkennen, ob eine Nachricht fehlerfrei bei allen Teilnehmern angekommen ist. Wenn nicht, versuchen sie das Senden erneut und erhöhen einen internen Fehlerzähler. Sobald ein Zähler den Wert 127 erreicht, schaltet ein CAN-Knoten in den Zustand „Error passive“, was bedeutet, dass er zwar noch Nachrichten senden und empfangen darf, jedoch bei den speziellen Fehlernachrichten des CAN-Busses nicht aktiv werden darf. Wenn der Fehlerzähler beim Senden 255 erreicht, geht der Knoten in den Zustand „Bus off“ über. Jetzt ist nur noch das Empfangen für diesen Knoten erlaubt, das Senden ist verboten. Beim SJA1000 müssen die Fehlerzähler zurückgesetzt werden, um wieder aktiv an der Kommunikation teilnehmen zu können, dazu ist die Funktion CanReset der API vorhanden. CanGetStatus dagegen liest die internen Fehlerzähler und -zustände aus. Sobald der Controller in den Zustand „Error passive“ oder „Bus off“ wechselt, generiert der Controller einen Interrupt, worauf der Treiber die Applikation benachrichtigen sollte.

Diese Mechanismen gilt es zu testen. Die Stimulierung eines Fehlers kann z. B. durch das Abziehen des Verbindungskabels erfolgen. Beim Sendeversuch wird ein CAN-Controller in den „Bus off“ Zustand wechseln. Diese Methode eignet sich jedoch nicht für den automatischen Test, da ein Bedienereingriff nötig wäre. Die Änderung der Baudrate auf nur einer Seite hat einen ähnlichen Effekt und lässt sich im Testfall programmieren. Die Teilnehmer

verstehen sich anschließend nicht mehr und der Sendende Knoten wird in den „Bus off“ Zustand wechseln.

Das Listing 4.5 zeigt die Implementierung eines entsprechenden Testfalls. Es werden bewusst unterschiedliche Baudraten für Master und Slave eingestellt. Danach reicht ein Versuch zum Senden einer Nachricht aus. Beim anschließenden Empfangen wird nicht eine Nachricht erwartet, sondern die Information, dass ein Fehlerzustand eingetreten ist: zuerst REASON_ERROR_PASSIVE_STATE, dann REASON_BUSOFF_STATE.

Listing 4.5: Implementierung eines Testfalls zur Fehlererkennung

```
1   setBaud(1000);      //Baudrate für beide Teilnehmer
2   setBaudChip(125);   //Eigene Baudrate "falsch" einstellen
3   send(MSG_ID_DUMMY, testcase[i].extended, j);
4   receive(&type, &id, data, &len);
5   CHECK_COMPARE(type, REASON_ERROR_PASSIVE_STATE);
6   getStatus(&state, &rx_failures, &tx_failures, &errorcode);
7   CHECK_COMPARE(state, REASON_ERROR_PASSIVE_STATE);
8   CHECK_COMPARE(tx_failures >= 255);
9   receive(&type, &id, data, &len);
10  CHECK_COMPARE(type, REASON_BUSOFF_STATE);
11  getStatus(&state, &rx_failures, &tx_failures, &errorcode);
12  CHECK_COMPARE(state, REASON_BUSOFF_STATE);
13  CHECK_COMPARE(tx_failures >= 255);
```

4.6.5 Performance

Die Anforderungen an die Performance des CAN-Systems müssen sehr genau betrachtet werden. Aufgrund eines aktuell fehlenden Kundenprojekts, das entsprechende quantifizierbare Anforderungen für die Geschwindigkeit und Reaktionszeit der Baugruppe vorsieht, müssen hier sinnvolle Anforderungen gefunden und ausbalanciert werden. Zu hohe Anforderungen würden Arbeit schaffen, die noch keiner bezahlt hat. Zu niedrige Anforderungen entlasten zwar den Aufwand an dieser Stelle, doch erschwert es die Positionierung des Produkts in einigen Bereichen. Im Automobilbereich etwa gelten eher hohe Hürden. Die Konsequenz aus dieser Situation ist, dass zumindest ermittelt werden sollte, was die jetzige Implementierung leistet und welche Leistung in welchen Zielgruppen erwartet wird. Wenn dazwischen starke Unterschiede auffallen, sollte beurteilt werden, ob mit entsprechenden Maßnahmen eine Steigerung der Leistungswerte zu erreichen ist, oder ob das System schon an seiner Leistungsgrenze arbeitet.

Die Beurteilung der Anforderungen von bestimmten Zielmärkten für das Jupiter-System fällt nicht ganz leicht. In Kapitel 4.2 wurden schon einige Einsatzgebiete beispielhaft genannt.

Aus keinem dieser Projekte sind allerdings messbare Anforderungen bekannt. Ein möglicher Weg wäre, ein bereits in einem CAN-Umfeld bestehendes Produkt von Garz & Fricke herzunehmen und den CAN-Treiber zu vermessen. Dieser Aufwand sollte jedoch vermieden werden, da hierfür dann zusätzlich Testfälle erstellt werden müssten. In einem vor kurzem geführten Akquisegespräch mit der Automobilindustrie konnte der Autor die Frage nach diesem Thema stellen. Danach sollte ein CAN-Teilnehmer in einem Fahrzeug innerhalb von wenigen Millisekunden reagiert und geantwortet haben. Ideal wäre die Reaktionszeit innerhalb von einer Millisekunde. Daran will sich diese Arbeit orientieren.

Der Begriff Performance enthält neben der Reaktionszeit auch den Durchsatz eines Systems, also z. B. Nachrichten pro Sekunde. In einem CAN-Netzwerk ist diese Größe allerdings nicht so wichtig, da in der Regel nur kleine Steuer- und Messgrößen transportiert werden.

Zur Messung der Reaktionszeit sind zwei Verfahren denkbar, die hier vorgestellt werden.

- Eine Messung der Round Trip Time (RTT) kann per Software erfolgen. Es wird die Laufzeit einer Nachricht gemessen, die vom Knoten A gesendet, vom Knoten B empfangen und zurückgesendet, und vom Knoten A wieder empfangen wird. Obwohl damit sehr viel mehr als die Reaktionszeit des CAN-Treibers gemessen wird, spiegelt sie den praktischen Einsatz des CAN-Busses recht gut wider. Zu beachten ist allerdings, dass die Gegenseite starken Einfluss auf die Messung hat. Der CAN-USB-Adapter hat möglicherweise durch die USB-Schicht weitere unerwünschte Verzögerungen. Besser ist die Messung zwischen zwei Jupiter-Systemen.
- Eine exakte Messung der Reaktionszeit kann nur sinnvoll mit einem Oszilloskop vorgenommen werden. Dabei kann die Zeit zwischen dem Eintreffen der Nachricht im CAN-Controller bis zur Signalisierung der Applikation gemessen werden. Dies spiegelt die Reaktionszeit des Controllers, der CPU, des Betriebssystems und des Treibers wider. In der Praxis wird in der Regel nur die Zeit vom Signalisieren eines Interrupts vom Controller bis zur Applikation gemessen, weil diese einfacher zu erfassen ist und die Verarbeitungszeit des Controllers im Verhältnis zur Verarbeitungszeit der Software meist sehr klein ist, daher auch nicht ins Gewicht fällt.

Die Messung der Round Trip Time (RTT) kann mit einfachen Mitteln in die Implementierung der Master-Seite aus Listing 4.4 eingepflegt werden. Windows CE stellt dazu die Funktionen QueryPerformanceCounter und QueryPerformanceFrequency bereit. Die Auflösung ist abhängig vom verwendeten System und kann daher dynamisch abgefragt werden. Auf dem Jupiter-ARM11-System beträgt die Frequenz 16,625 MHz was einer Auflösung von 60 ns entspricht[6]. In Listing 4.6 ist die Einbettung der Messung um die Funktionen `send` und

[6] Die hohe Auflösung von 60 ns darf nicht darüber hinwegtäuschen, dass der Aufruf dieser Funktionen ebenfalls Laufzeit kostet. Das Tool OSBench.exe ermittelt dafür 1,744 μs für das Jupiter-ARM11-System.

Listing 4.6: Zeitmessung mittels Peformance-Counter

```
1  LARGE_INTEGER g_liPerf1[5];
2  LARGE_INTEGER g_liPerf2[5];
3  LARGE_INTEGER g_liPerfFreq;
4  #define QPC_START(n)        {QueryPerformanceCounter(&g_liPerf1[n]);}
5  #define QPC_STOP(n)         {QueryPerformanceCounter(&g_liPerf2[n]);}
6  #define QPC_GET_DELTA(n)    ((DWORD)(((g_liPerf2[n].QuadPart - g_liPerf1
       [n].QuadPart) * 1000000000) / g_liPerfFreq.QuadPart))
7
8      ...
9      QPC_START(0);
10     send(MSG_ID_DUMMY, testcase[i].extended, j);
11     receive(&id, data, &len);
12     QPC_STOP(0);
13     ...
```

`receive` zu sehen. Zur besseren Lesbarkeit werden Makros verwendet, die die Berechnung kapseln und mithilfe der Arrays mehrere Messungen erlauben.

Leider ist beim Durchführen einiger Tests und Messungen ein Jupiter mit CAN beschädigt worden, so dass nur noch ein Gerät zur Verfügung stand. Die Messung wurde daher mit dem PCAN-USB-Adapter durchgeführt. Wie befürchtet, hat dieser einen sehr hohen Einfluss auf die RTT. Der gemessene Wert lag bei 100 ms, egal bei welcher Baudrate.

Eine Möglichkeit, die RTT mit nur einem Gerät ansatzweise zu ermitteln, könnte der Loopback-Mode des SJA1000 sein. In dieser Einstellung schickt sich der Controller die Nachrichten quasi selbst zu. Der Treiber unterstützt diesen Modus allerdings nicht. Mittels der Funktion CanWriteRegister könnte er zwar eingeschaltet werden, würde aber bei jeder weiteren Aktion des Treibers implizit wieder ausgeschaltet werden. Die Änderung dieses Verhaltens im Treiber ist aufwändig, und wird daher nicht durchgeführt.

Die Messung der Reaktionszeit mit dem Oszilloskop konnte dagegen erfolgreich durchgeführt werden. Dazu wurden mit dem PCAN-Gerät im 200 ms Takt Nachrichten geschickt. Das Oszilloskop wurde auf die fallende Flanke des Interruptsignals vom SJA1000 getriggert. Eine Applikation hat beim Empfangen der Nachrichten einen Impuls auf einen Ausgangspin gelegt, der ebenfalls im Oszilloskop aufgenommen wurde. Die Zeit dazwischen wurde vermessen. Bild 4.5 zeigt den Vorgang auf einer Zeitachse. Bei 1000 durchgeführten Messungen stellte sich ein Mittelwert von etwa 300 μs ein. Die maximale Reaktionszeit, die im Sinne der Echtzeitfähigkeit interessanter ist, hängt dagegen von einigen weiteren Faktoren ab. Der zuerst gemessene Wert lag bei 3,1 ms.

Dies liegt an den eingestellten Prioritäten im Windows CE System. Pro Thread lässt sich eine Prioritätszahl vergeben. Niedrige Prioritätszahlen verdrängen Threads mit höheren Prio-

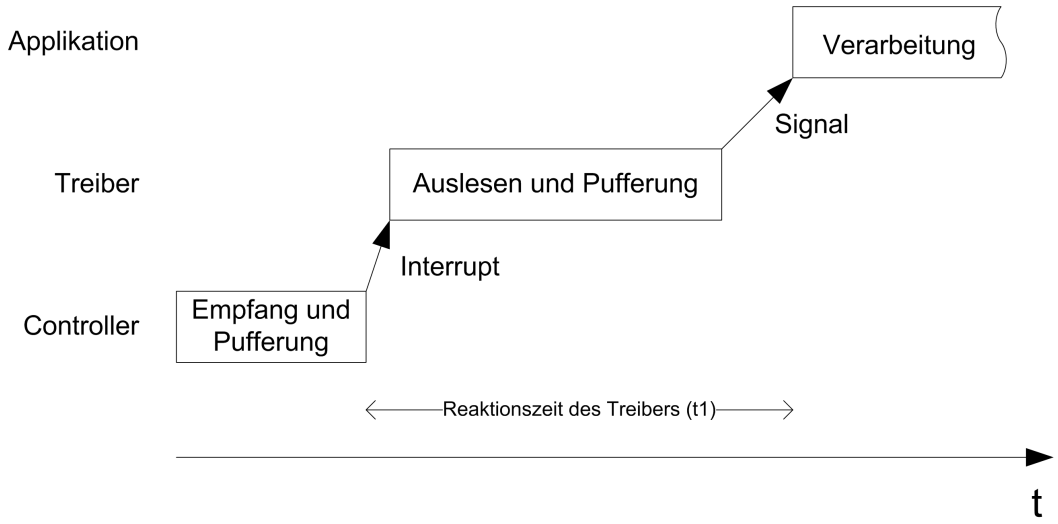


Abbildung 4.5: Verarbeitungszeiten einer CAN-Nachricht

ritätszahlen. Alle Treiber haben in der Regel einen Prioritätswert zwischen 80 und 100, Applikationen haben den Wert 249. Das bedeutet, dass zwischen der Verarbeitung der Nachricht im CAN-Treiber (80) und der Ankunft der Nachricht in der Applikation (249), alle anderen Treiber (80 $\sim$ 100) bei Bedarf Rechenzeit in Anspruch nehmen können. Je stärker das System auf den anderen Schnittstellen belastet würde, umso schlechter wäre die gemessene Reaktionszeit ausgefallen. Eine Stichprobenmessung mit einem Ethernet-Download im Hintergrund hat dies bestätigt.

Die Prioritäten des Treibers und der Applikation wurden testweise auf sehr niedrige Werte gestellt: 40 und 50. Damit sollten alle anderen im System aktiven Treiber und Komponenten verdrängt werden, sobald die Threads vom Treiber und Applikation lauffähig werden. Mit diesem Mittel lässt sich die maximale Reaktionszeit des CAN-Systems auf gemessene 370 μs begrenzen.

Die Methode hat jedoch einen Nachteil, weshalb im Einzelfall immer geprüft werden muss, welche Prioritäten tatsächlich vergeben werden können. Eine so starke Priorisierung eines Systemteils bremst andere Teile des Systems aus. Sofern es sich um Applikationsteile handelt, ist dies in der Regel nicht so schlimm. Andere Treiber hingegen können darauf empfindlich reagieren und zur Systeminstabilität führen. Wenn diese nicht innerhalb einer bestimmten Zeit zur Ausführung kommen, droht, je nach Funktionsweise der Hardware, Datenverlust oder Datenkorruption.

4.7 Review

Das Testkonzept in Kapitel 4.4 sieht ein Review für die Punkte Benutzbarkeit, Änderbarkeit und Übertragbarkeit der CAN-API respektive des Treibers vor. Hier kann die Erfahrung und Sichtweise anderer Entwickler helfen. Das Review muss zunächst vorbereitet werden, wozu auch das Aussuchen der Teilnehmer zählt. In der Regel sollte ein Review stets von einem Moderator begleitet werden, der bei Streitigkeiten die Diskussion wieder versachlichen kann. Aufgrund einer hohen Auslastung der Softwareentwickler wird jedoch oft darauf verzichtet. Lediglich ein weiterer Entwickler, der idealerweise die entsprechende Erfahrung mitbringt, wird in ein Review geladen. Diese Art wird auch informelles Review genannt. Dank einer guten Arbeitsatmosphäre kommen unsachliche Diskussionen nur sehr selten auf. Es gibt jedoch Situationen, in denen auf einen Moderator nicht verzichtet werden sollte, z. B. wenn sich Personen auf sehr unterschiedlicher Hierarchieebene begegnen.

Für das Review der CAN-API wird die informelle Review-Variante gewählt. Zur Vorbereitung des Reviews werden einem erfahrenem Entwickler alle benötigten Informationen einige Tage vorher übermittelt. Das sind:

- *Gegenstand des Reviews:* Der CAN-Treiber incl. API
- *Zu prüfende Merkmale:* Benutzbarkeit der API, Änderbarkeit und Übertragbarkeit des Treibers
- *Zu prüfende Aspekte*: siehe Text
- *Ort, Zeit und veranschlagte Dauer des Reviews:* 2 Stunden

Die Teilnehmer sollten sich vor Beginn zur angemessenen Vorbereitung mit der Thematik auseinandersetzen. Es wäre jedoch falsch, schon den gesamten Sourcecode vor dem Review zu untersuchen. Der Autor und die Teilnehmer gehen während des Reviews den Sourcecode zusammen durch, wobei der Autor Gelegenheit bekommt seine Entscheidungswege zu erklären. Alle von den Teilnehmern eingebrachten Punkte zu Missverständnissen und Verbesserungen werden notiert.

Die zu prüfenden Aspekte sollten genauer formuliert werden, als nur die Nennung der Anforderungsklassen. Wenn eine Checkliste erstellt werden kann, so sollte dies gemacht werden. Checklisten als alleiniges Prüfmittel sind jedoch mit Vorsicht zu genießen, da sie auch trügerische Sicherheit suggerieren können. Für das anstehende Review werden lediglich einige grobe Punkte beschrieben, die es abzuprüfen gilt:

- Die API sollte vollständig und verständlich dokumentiert sein.
- Gültige und ungültige Parameter sind anzugeben.
- Die Parameter- und Rückgabetypen sollten passend gewählt sein.
- Plattformabhängige Implementierungen sollten in Module ausgegliedert werden.
- Hardwareadressen und Interrupts sollten konfigurierbar sein.

- Der Sourcecode sollte strukturiert und gut verständlich aufgebaut sein.
- Angemessene Fehlerbehandlung, keine undefinierten Situationen. (siehe Kapitel 4.6.4 Punkt 2).

Alle weiteren Auffälligkeiten sollten mit Kreativität gesucht werden. Nach der Review-Sitzung standen folgende Ergebnisse fest:

- Plattformabhängige Initialisierungen (Interrupts, Pinkonfiguration) sind zwar getrennt implementiert, aber noch nicht in ein separates Modul ausgegliedert. Dies sollte nachgeholt werden.
- Das Review muss nicht wiederholt werden.

4.8 Bewertung

Die Durchführung der Tests an der Beispielkomponente CAN findet hier seinen Abschluss. Als große Hürde erwies sich das Fehlen von entsprechenden Anforderungen für diese Komponente. Diese mussten erst mühselig abgeleitet und erstellt werden. Da dies in der Vorbereitung und Planung der Arbeit nicht ausreichend berücksichtigt wurde, sind die einzelnen Anforderungspunkte auch nicht von besonders guter Qualität (Beispiel Fehlertoleranz). Es wurde trotzdem versucht möglichst viele Aspekte einzubeziehen, um die Testarbeiten zu unterstützen. Dabei ist für den Autor sehr deutlich geworden, dass fehlende oder unklare Anforderungen ein vernünftiges Testen fast unmöglich machen.

Die Erstellung der Testfälle auf Komponentenebene hat sich als sehr nützlich erwiesen. Die gefundenen Fehler waren allerdings eher im Detailbereich anzusiedeln. Bei enormen Zeitproblemen könnte dieser Test auch ausgelassen werden, wenn kleinere Fehler akzeptabel sind, wie in der Teststrategie beschrieben (vgl. Kap. 4.4).

Der Integrationstest ist dagegen unverzichtbar, da er die wesentlichen Funktionen des Treibers abprüft: die Kommunikation mit anderen Teilnehmern. In der Tabelle 4.7 werden alle Anforderungen aus Kapitel 4.2.2 mit den entsprechenden Testmaßnahmen gegenübergestellt. Diese abschließende Betrachtung in diesem Stadium ist natürlich wichtig, um zu erkennen, ob alle geforderten Punkte abgedeckt und erreicht wurden.

Eine formelle Übersicht über den Stand der Testarbeiten zeigt die Tabelle 4.7. In der Praxis würden diese Informationen in die Anforderungsliste mit eingearbeitet werden, wie im Requirementstracing vorgeschlagen.

Anforderung	erfüllt	Grund / Referenzen
A001	ja	Kap. 4.5.7 Testfälle 0101 - 0107 (Tab. 4.5)
A002	ja	Kap. 4.6.3 Master/Slave-Test
A003	ja	Kap. 4.6.3 Master/Slave-Test
A004	ja	Kap. 4.6.3 Master/Slave-Test
A005	ja	Kap. 4.5.7 Testfälle 0201 - 0210 (nicht abgebildet)
A006	ja	Kap. 4.5.7 Testfälle 0301 - 0315 (Tab. 4.6)
A007	nein	Implementierung fehlt (siehe Anm. 1)
A008	nein	Implementierung fehlt (siehe Anm. 1)
A009	ja	Kap. 4.6.3 Master/Slave-Test
A010	ja	Kap. 4.6.4 und 4.5.7
A011	ja	Kap. 4.5.7 Testfälle 0401 - 0415 (nicht abgebildet)
A012	ja	Kap. 4.6.5 (siehe Anm. 2 und 3)
A013	ja	Kap. 4.6.4 (siehe Anm. 4)
A014	ja	Kap. 5.4
A015	ja	Kap. 4.7 Review
A016	ja	Kap. 4.7 Review

Tabelle 4.7: Getestete Anforderungen

Anmerkungen:

1. Die Funktionen PowerDown und PowerUp des Streaminterfaces sind zu diesem Zeitpunkt nicht implementiert. Die Testfallerstellung sollte zusammen mit der Implementierung der Stromsparfunktion stattfinden.
2. Die Messung der RTT sollte noch mit zwei Jupiter-Systemen durchgeführt werden.
3. Die gemessene Reaktionszeit liegt im geforderten Bereich, sofern die Priorisierung zugunsten des CAN-Treibers eingestellt wird.
4. Im möglichen Zusammenspiel mit anderen Komponenten sollte zusätzlich noch ein Last- und Dauertest auf Systemebene durchgeführt werden. Genaue Anforderungen sollten aus realen Anwendungsfällen abgeleitet werden, die hier nicht vorgelegen haben.

5 Automatisierung

Die Tests im vorherigen Kapitel am Beispiel CAN-Treiber bringen eine solide Basis. Der Treiber wurde von vielen Seiten beleuchtet und unter vielen Gesichtspunkten getestet.

„Prima, abgehakt, nächste Komponente, nächstes Projekt!“ könnten forsche Kollegen jetzt rufen. Einige Monate später dann die Erkenntnis: In irgendeiner neuen Version, in irgendeiner neuen Zusammenstellung funktioniert der CAN-Treiber nicht wie er soll. So oder so ähnlich könnte die Geschichte lauten, wenn der Test nicht weiterverfolgt und weiterentwickelt wird, ja evtl. gar nicht mehr an neueren Versionen wiederholt wird.

Dieses Kapitel beschäftigt sich mit der Wiederholbarkeit von Tests. Ein Schwerpunkt ist dabei jeweils die Automatisierbarkeit von Vorgängen. Notwendigkeit, Vor- und Nachteile werden beleuchtet. Die Tests des CAN-Treibers werden in einen größeren Kontext gelegt und praktische Beispiele für die Situation bei Garz & Fricke werden entwickelt.

5.1 Regressionstest

Die Bedeutung des Regressionstest ist so simpel wie konsequent zugleich: Nach einer Änderung im System müssen die entsprechenden Tests wiederholt werden.

Der Lebenszyklus eines Softwareproduktes sieht selten einen totalen Stillstand vor. Erstrebenswert ist ein lebendiges Projekt, dass auf Rückmeldung der Benutzer reagiert und vom Hersteller gepflegt wird. Trotz ausführlichen Testens werden einige Fehler mit einer gewissen Wahrscheinlichkeit erst im Feld auftreten (siehe auch Tabelle 4.2). Das Beheben dieser Fehler bedeutet, Änderungen an der Software vorzunehmen. Doch auch kreative Rückmeldungen der Benutzer sollten nicht ungehört bleiben. Im Laufe der Zeit verändern sich Anforderungen an Software, neue Funktionen werden gewünscht oder einfachere Bedienung gefordert. Die Wahrscheinlichkeit, dass eine Software über einen Zeitraum hinweg Änderungen erfährt ist sogar dann noch groß, wenn dies anfangs gar nicht beabsichtigt worden war (z. B. Fehlerbehebung). Doch die Weiterentwicklung der 32-Bit Produkte bei Garz & Fricke wird von den Verantwortlichen kontinuierlich angestrebt und für einen gewissen Zeitraum geplant. Für die Jupiter-Reihe mit dem ARM11-Modul zeichnet sich eine Produktlieferbarkeit von 5 bis 10 Jahren ab. Es kann angenommen werden, dass die Softwareabteilung sich

noch viele weitere Jahre mit dem ARM11-Modul beschäftigt. Neben dem Jupiter sind auch andere Hardwareplattformen auf Basis des ARM11-Moduls gestartet worden (siehe Tabelle 2.1) oder im Gespräch. Sogar Nachfolger und andere Varianten des ARM11Moduls sind langfristig geplant (siehe i.MX32 und i.MX35 in der aktuellen Roadmap für Computermodule im Anhang D).

Im Fall des CAN-Treibers kann sich die Hardwarerevision des CAN-Controllers oder die des Baseboards (Jupiter) ändern. Am wahrscheinlichsten ist, dass der Treiber auf einer neuen Hardwareplattform mit dem gleichen Controller eingesetzt wird. In diesen Fällen muss sich die Software nicht unbedingt ändern, außer vielleicht einiger Konfigurationen, wie eine andere Interruptnummer oder Adressbereich. Die Änderung der Hardware bei Embedded-Systemen ist jedoch Grund genug, um Zweifel an der Funktionsweise und Stabilität der neuen Zusammenstellung aufkommen zu lassen. Schließlich ist das Embedded-System eine Einheit aus Hard- und Software die aufeinander abgestimmt sein muss. Schon eine weitere Revision eines Halbleiterbausteins kann ein verändertes Zeitverhalten der Hardware bedeuten.

Jede Änderung eines Systemteils birgt das Risiko einer eingeschleppten Nebenwirkung. Dies kann mitunter sehr tückisch sein, da die Nebenwirkung u. U. nur in gewissen Situationen auftritt. Der Programmierer, der die Änderung durchgeführt hat, wiegt sich in falscher Sicherheit, wenn er nur seine neue Implementierung testet und für gut befindet. Eine mögliche Fehlerwirkung kann nun im Zusammenspiel mit anderen Systemteilen, die gar nicht verändert wurden, auftreten. Typischerweise kommt dies erst in bestimmten Fällen zum Tragen, z. B. in Lastsituationen oder bei bestimmten aufeinander folgenden Aktionen.

Die Erfahrungen bei Garz & Fricke mit Windows CE zeigen, dass Fehler gehäuft auftreten wenn die Zusammenstellung des Betriebssystems grundlegend geändert wird. Aufgrund der sehr vielen Komponenten und deren Abhängigkeiten (CE 5.0 hat etwa 500 Komponenten) kann es passieren, dass das Hinzufügen oder Entfernen einer Komponente Einfluss auf andere hat. Dies sollte zwar durch Mechanismen zur Erkennung und Auflösung von Abhängigkeiten schon im Platform Builder erkannt werden, doch kann man sich auf diesen letztendlich nicht vollständig verlassen.

Der Regressionstest adressiert alle diese Fälle, indem die Wiederholung von allen oder einer Auswahl der Tests vorgeschrieben wird. Welche Tests wiederholt werden, sollte in erster Linie von der Art und Stelle der Änderungen abhängen. Alle beteiligten Komponenten sind potentielle Kandidaten. In der Praxis hängt die Auswahl der zu wiederholenden Tests jedoch viel mehr vom Testaufwand ab. Wenn Tests manuell und langwierig durchgeführt werden müssen, ist immer wieder eine Scheu bei Entwicklern und Testern zu beobachten. Bei zahlreicher Wiederholung manueller Tests schleicht sich auch schnell Betriebsblindheit ein, der Tester verliert die Sensibilität und führt die Tests nur noch stumpf aus. Die Gefahr des Durchrutschens offensichtlicher Fehler steigt.

Genau an diesem Punkt setzt die Automatisierung ein. Neben den genannten Gründen spart ein automatisierter Ablauf in der Regel auch sehr viel Ausführungszeit. Während der Tester nach Checklisten oder Anweisungsdokumenten Schritt für Schritt Testfälle startet und überprüft, ist ein automatischer Testlauf mit deutlich weniger Ressourcen anzusetzen. Lediglich die Erstellung automatischer Tests hat einen großen Erstaufwand. Bei hinreichender Wiederholung der Tests wird sich dieser allerdings schnell amortisieren. Für die Produktlinie der ARM11-Computermodule sollte dies in jedem Fall gelten, da es in verschiedenen Projekten und Produkten eingesetzt werden wird.

5.2 Konzept

Embedded-Systeme automatisiert zu testen ist eine Herausforderung. Anders als bei reinen Software-Produkten spielt die Hardware eine wesentliche Rolle. Die Software muss auf der Zielhardware ausgeführt werden. Das gesamte System ist als Testobjekt zu betrachten. Schnittstellen müssen soweit sinnvoll beschaltet, stimuliert und abgefragt werden, wie die Testfälle es erfordern. Dieser Punkt stellt den größten Aufwand in der Realisierung dar.

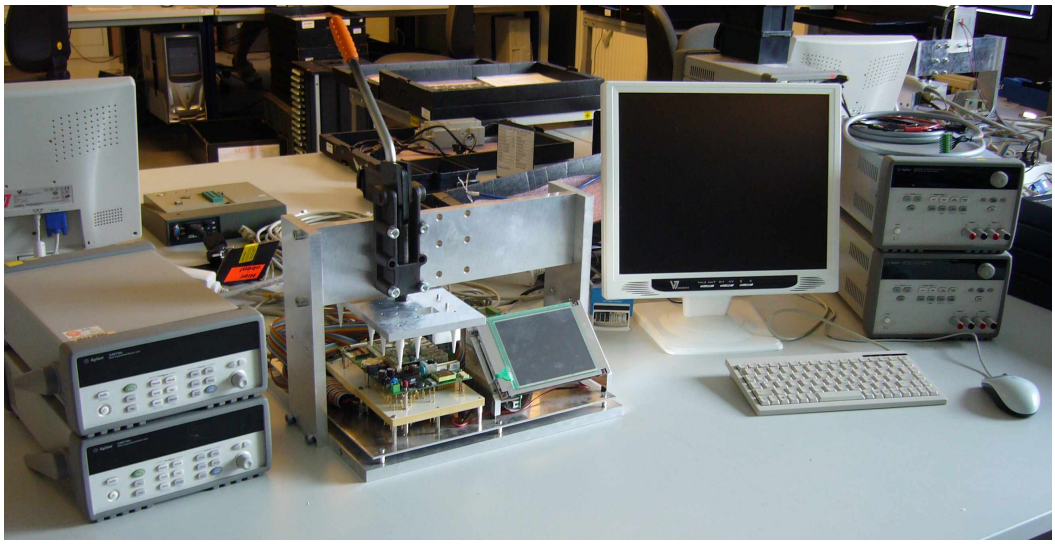


Abbildung 5.1: Teststand für den Serientest des Jupiter-ARM9-Systems

Das Aufbauen dieser Testumgebung kann sogar als ein eigenes Entwicklungsprojekt betrachtet werden. So werden für den Produktionsteststand bei Garz & Fricke für viele Produkte umfangreiche Nadeladapter-Platinen entwickelt. Je nach Baugruppengröße können schon mal über hundert Nadeln für die entsprechenden Testpunkte, Lötpunkte und Anschlüsse auf der Baugruppe auf einem Nadeladapter realisiert sein. Auf dem Bild 5.1 ist der Produktionsteststand des Jupiters zu sehen. Der Prüfling wird auf die Nadeln gelegt. Der Hebel drückt

den Prüfling mit passenden Abstandsstücken herunter, so dass die Nadeln die Testpunkte mit einer gewissen mechanischen Spannung kontaktieren können. Der Ablauf und die angeschlossenen Messgeräte werden von einem PC gesteuert.

Der Produktionsteststand ist zum Zwecke der Automatisierung gebaut worden. Alle externen Schnittstellen des Prüflings, aber auch interne Testpunkte, können gesteuert und abgefragt werden. Dieser Teststand würde sich gut für den automatischen Ablauf der Entwicklungstests eignen. Da die Produktion den Teststand jedoch mit einer hohen Auslastung benötigt, könnte es schwierig werden, diese Ressource zeitlich zu teilen. Aller Erfahrung nach steigt auch das Risiko, dass der Teststand Schaden nimmt oder verkonfiguriert wird, was unnötige Umstände in der Produktion schafft. Eine zweite Ausführung könnte Abhilfe schaffen, doch sind die Kosten einer redundanten Einrichtung ebenfalls zu beachten. Für die Automatisierbarkeit der Entwicklungstests kommt die Nutzung dieser Hardware daher nicht in Frage.

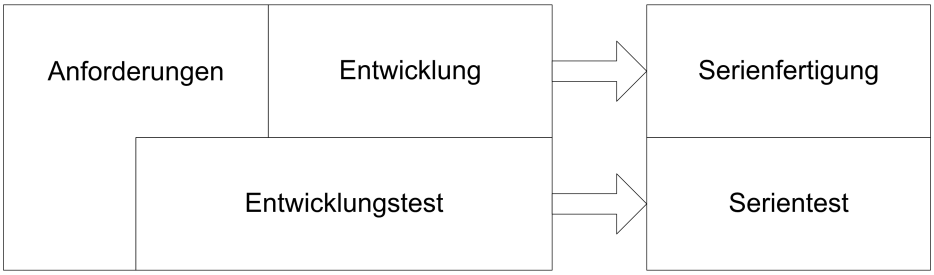


Abbildung 5.2: Übergang der Entwicklung in die Serienproduktion

Wünschenswert ist es jedoch, eine gewisse Wiederverwendung zu ermöglichen. Es lohnt sich daher, die Produktionsumgebung näher zu betrachten. Abbildung 5.2 deutet den Übergang einer Produktentwicklung in die Serienfertigung an. Zum Serienstart eines Produktes muss die Produktion vorbereitet werden. Neben der Generierung und Übergabe der Produktdaten (z. B. Bauteile- und Bestückungslisten, Firmware, etc.). können auch Tests zu einem gewissen Teil wiederverwendet werden. Die Serientests werden in verschiedenen Ausführungen am Prüfling angewandt (AOI, ICT, vgl. Kap. 2.3.2). Der interessanteste Test in diesem Zusammenhang ist der Funktionstest. Die Hardware wird hierbei auf ihre Funktion hin geprüft. Insbesondere die Schnittstellen werden nicht nur auf physische Verbindung geprüft, sondern zur Kontrolle der Eigenschaften werden auch Daten testweise übertragen. Diese oder eine ähnliche Implementierung findet sich zum Beispiel in Testfall xxxx des CAN-Treibers. Die Steuerung des Ganzen übernimmt eine Software, die Testfälle sowohl auf dem Test-PC als auch auf dem Prüfling ausführen und zusätzlich die Testumgebung ansteuern kann. Bei Garz & Fricke werden dafür die hauseigene Software TestServer und die zugekaufte Lösung TestStand von National Instruments eingesetzt.

Bisherige Windows CE Produkte werden mit TestServer angesteuert. Aufgrund von fehlender Erweiterbarkeit, gestiegener Komplexität und enormem Aufwand zur Pflege des TestServers

wurde mit TestStand[1] eine spezialisierte Umgebung angeschafft. Neue Produkte sollen mit TestStand realisiert werden. Die Umstellung kommt zu einem Zeitpunkt, der für die Jupiter-Produktlinie etwas ungeeignet scheint: Jupiter-ARM9 Produkte werden noch mit TestServer getestet. Jupiter-ARM11 Produkte sollen von der Stabilität und Erweiterbarkeit von TestStand profitieren. Das erschwert die Wiederverwendung der vorhandenen Lösungen der ARM9 Linie. Methoden zur Anbindung des Jupiters mit ARM11-Modul an TestStand müssen zu einem großen Teil neu entwickelt werden. Das Kapitel 5.3 wird sich mit den Details befassen.

Zuvor sollten die Aufgaben des Regressionstests beschrieben werden. Ein vollständiger automatisierter Ablauf ist in Tabelle 5.1 skizziert.

1	Quellcode holen
2	Build OS Image
3	Zielgerät einschalten
4	OS Image ins Zielgerät laden
5	OS starten
6	OS Debug-Nachrichten der Bootphase überprüfen
7	Tests ins Zielgerät laden
8	Tests starten (Zielgerät und Kontroll-PC)
9	Ergebnisse überprüfen und speichern
10	Nächste OS Konfiguration/Zusammenstellung

Tabelle 5.1: Ablauf eines automatischen Tests

Die beiden ersten Schritte zum Builden des OS Images sind nur für einige Situationen sinnvoll und daher optional. Weitere Erklärungen und Lösungen finden sich in Kap. 5.3.6. Stattdessen können auch vor dem Testdurchlauf schon fertige OS Images zur Verfügung gestellt werden. Diese müssen im Zielgerät zur Ausführung kommen. Die Debug-Nachrichten beim Starten des OS Images sollten in jedem Fall auf Folgendes geprüft werden:

- Sind die benötigten Treiber geladen und gestartet?
- Sind Exceptions aufgetreten?
- Sind evtl. bekannte Fehler aufgetreten?

Für gewöhnlich geben Treiber beim Initialisieren einige Debug-Nachrichten aus, die Erfolg oder Misserfolg anzeigen. Diese Strings sind schon vorher bekannt, so dass man die Nachrichten daraufhin durchsuchen kann. Beliebige Fehler können sich in Exceptions äußern. Auch der Aufbau einer Exception Nachricht ist bekannt. Danach sollte nicht nur beim Starten, sondern auch im späteren Verlauf der Tests gesucht werden.

[1] Achtung: Der feine Unterschied Teststand (Stand zum Testen) und TestStand (Softwareprodukt von National Instruments) ist zu beachten.

Alle Dateien zur Ausführung der Testfälle müssen übertragen werden. Beim Starten der Testfälle müssen evtl. noch Parameter wie IP-Adresse, Portnummer etc. übergeben werden. Außerdem muss die Reihenfolge beim Starten eingehalten werden, beim CAN-Beispiel etwa erst der Slave, dann der Master. Der Slave muss im Hintergrund laufen können. Die Ergebnisse müssen automatisch überprüfbar sein. Mindestens muss erkennbar sein, ob ein Test Erfolg hatte. Wenn nicht, muss ein Test als „Fehlgeschlagen“ markiert werden. Alle Ausgaben, auch die von erfolgreichen Tests, sollten gespeichert werden. Dies kann sich im Zweifel bei einer Fehlersuche schon mal als nützlich erweisen, wenn erfolgreiche und nicht-erfolgreiche Ausgaben verglichen werden können.

Nach einem Durchlauf von allen Tests können Variationen ausprobiert werden. Dazu gehören sowohl Variationen in der Hardware als auch in der Software. Die Hardware Jupiter bietet einige Optionen an:

- Displaytyp und Auflösung: 320 x 240, 640 x 480 Bildpunkte
- „Light“ Variante: Nur Real Time Clock (RTC) , Ethernet, RS-232 und Universal Serial Bus (USB)
- „Full“ Variante: Audio, Digital IO, Analog IO, Serial Peripheral Interface (SPI) und Inter-Integrated Circuit (IIC)
- In beiden Ausstattungsvarianten optional: CAN und Bluetooth

Weitere Details dazu finden sich im Data-Brief des Jupiters ([JUPITER-brief]). Die Hardware kann natürlich nicht automatisch ausgetauscht werden (auch wenn dies mit entsprechend aufwändigen Mitteln denkbar wäre). Es reicht an dieser Stelle, dass die Hardware manuell nach einem Testdurchlauf durchgetauscht wird. Für die Zukunft ist auch eine Parallelisierung denkbar, d. h. ein Aufbau müsste pro Hardwarevariante vorhanden sein.

Die Variationen der Software sind wiederum etwas einfacher zu durchlaufen. Es genügt den Testablauf mit einem neuem OS Image zu starten. Hierbei können verschiedene Konfigurationen von Windows CE geprüft werden (vgl. Kap. 5.3.6). Auch das Verändern von Konfigurationen auf dem Gerät ohne das OS auszutauschen ist hier denkbar. Dies rechtfertigt allerdings nur selten den erneuten Durchlauf aller Testfälle und sollte deshalb in eigenen Testfällen mit einem Neustart des Gerätes gekapselt werden.

Das Implementieren des gesamten automatischen Tests für das Produkt Jupiter mit ARM11-Modul kann in dieser Arbeit nicht geleistet werden, da dazu auch alle benötigten Testfälle der vielen anderen Komponenten von Garz & Fricke (IIC, SPI, RTC, etc.) gehören. Diese Arbeit wird sich der Implementierung des Testkonzepts und der Testumgebung beschäftigen. Es geht darum eine Grundlage zu schaffen, auf der weitere Testarbeiten leicht aufsetzen können. Dies gilt sowohl im Produktionsbereich für den Hardwaretest als auch für die Testfälle der Softwareentwicklung. Die entwickelten Testroutinen eigen sich als Beispiel zur Veranschaulichung der hier entwickelten Lösungen.

5.3 Umgebung

Die Realisierung der Umgebung für die Ausführung automatischer Tests darf nicht unterschätzt werden. Ein gewisser Aufwand zahlt sich jedoch durch die vielen Wiederverwendungsmöglichkeiten, auch für zukünftige Softwareprojekte auf Basis des ARM11 Moduls mit Windows CE, schnell aus.

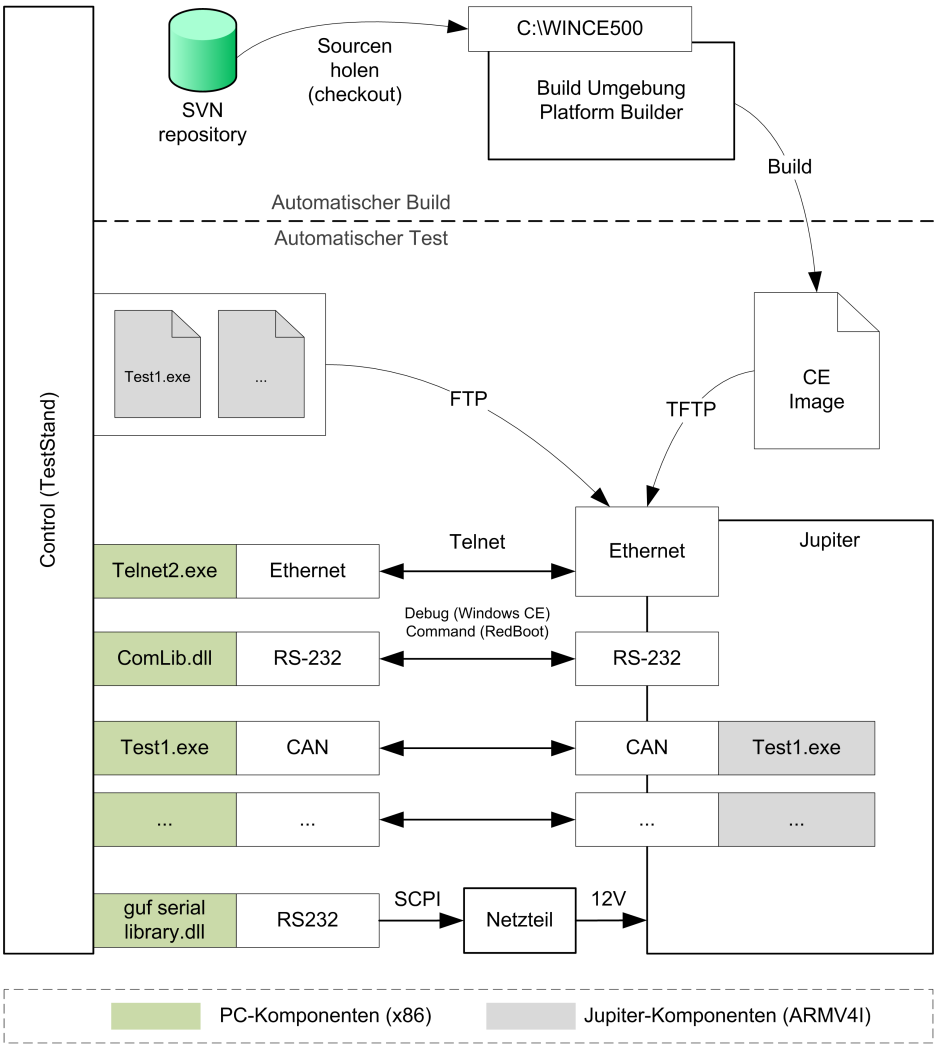


Abbildung 5.3: Interaktion aller Komponenten beim automatischen Test

Die Abbildung 5.3 zeigt das Zusammenspiel aller Komponenten in einer Art Kollaborationsdiagramm. Hard- und Softwareebene sind dabei gemischt vertreten. Die Aktionen 1 und 2 aus Tabelle 5.1 finden sich im oberen Teil wieder. Der untere Teil bildet die Verbindungen und Komponenten zwischen Prüfling und Steuerungs-PC ab. Die Details der einzelnen An-

bindungen werden in den folgenden Abschnitten besprochen. Die gesamte Steuerung wird die Software TestStand übernehmen. Der Prüfling bietet einige Möglichkeiten zur Kommunikation und Steuerung an. Je nach Ausführungszustand können diese sehr verschieden sein.

- Wenn sich auf dem Computermodul noch keine Software befindet, kommt Joint Test Action Group (JTAG) zum Einsatz. Dies ist nur in der Produktion relevant. Für die Tests in der Entwicklung gehen wir davon aus, dass die Erstinbetriebnahme schon durchlaufen wurde.
- Der Bootloader RedBoot lässt sich über die serielle Schnittstelle oder Ethernet ansprechen. Der Download des Betriebssystems erfolgt mit Trivial File Transfer Protocol (TFTP). Danach kann das Image auf den Flash-Speicher gespeichert und / oder gestartet werden.
- Das Betriebssystem Windows CE ist der normale Ausführungszustand. In der Regel sind die Geräte so konfiguriert, dass das Betriebssystem automatisch gestartet wird. Hier werden über den Ethernet-Anschluss Telnet und File Transfer Protocol (FTP) zur Verfügung gestellt. Auch andere Dienste und Protokolle, wie z. B. Hypertext Transfer Protocol (HTTP), sind ebenfalls in Windows CE vorhanden. Debug-Ausgaben werden auf die serielle Schnittstelle umgeleitet.

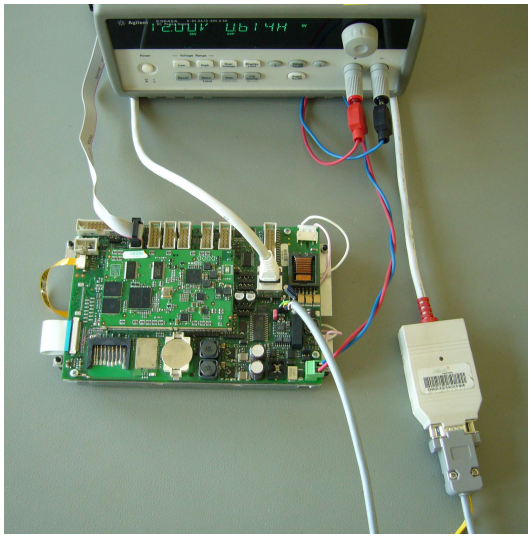


Abbildung 5.4: Prototypischer Aufbau eines Testsystems für den CAN-Treiber

Das Bild 5.4 zeigt ein Foto des Aufbaus, der hier zur Durchführung benutzt wird und schon in ähnlicher Weise für die Integrationstests im Kapitel 4.6 benutzt wurde. Das Jupiter-System ist in der Mitte zu sehen, rechts unten der PCAN-USB-Adapter, oben ein programmierbares Netzteil.

5.3.1 NI TestStand

Ein Werkzeug zur automatischen Ausführung von Testaktionen ist ab einem gewissem Umfang unerlässlich. Kleinere Aufgaben können noch sehr effektiv mit wenigen Aufrufen in Batch- oder anderen Skriptsprachen erledigt werden. Doch die immer wiederkehrenden Aufgaben zum Starten von Testfällen in verschiedensten Umgebungen, das Überprüfen von Ergebnissen und Erstellen von Berichten werden mit dem Umfang der Aufgabe zusehends mühseliger. Die Eigenentwicklung TestServer erwies sich im Produktionsumfeld ebenfalls als sehr hinderlich und wenig anpassbar, im Entwicklungsbereich wurde sie gar nicht erst eingesetzt. Im Vorfeld dieser Arbeit wurde von dem Autor in Zusammenarbeit mit der Produktionsabteilung eine ausgewachsene Lösung gesucht die mit TestStand von National Instruments (NI) gefunden wurde.

TestStand bietet eine grafische Umgebung zur Erstellung und Ausführung von Testabläufen. Die Software erlaubt eine Adaptierung an sehr viele Programmiersprachen und -konstrukte. Eine Testaktion kann u. a. ein Aufruf eines Programms oder einer Batch-Datei sein, ein Aufruf einer einzelnen Funktion in einer DLL oder die objektorientierte Anbindung von Klassen, Objekten und Methoden in C++ DLLs oder .NET Assemblies. Die Vielseitigkeit der Anbindungsmöglichkeiten ist eine Stärke von TestStand. Testdateien können mittels File Transfer Protocol (FTP) auf andere Rechner oder den Prüfling übertragen werden. Für jede Testaktion kann festgelegt werden, ob der Rückgabewert ausgewertet und bei Nichtübereinstimmung der Test angehalten werden soll. Während oder am Ende des Tests lassen sich Ergebnisse in eine Datenbank ablegen.

Eine weitere Stärke liegt in den Möglichkeiten zur Parallelisierung. Beliebige Aktionen oder ganze Abschnitte können zur parallelen Ausführung konfiguriert werden. Das erleichtert die Erstellung von Testabläufen in etlichen Situationen.

- Parallelisierung von Testschritten wird im Serientest mit dem Ziel der Zeitersparnis verfolgt. Bei mittleren und großen Serien zahlt es sich aus die Ausführungsdauer des Testablaufs zu optimieren, da damit auch die Stückkosten gesenkt werden können. Einzelne Schritte können zeitgleich ausgeführt werden, sofern der Prüfling das zulässt. Dies wird meistens auf Aktionen angewandt, die sich nicht gegenseitig behindern, wie z. B. gleichartige Schnittstellen. Alternativ oder zusätzlich können mehrere Prüflinge gleichzeitig bedient werden.
- Für die Tests der Entwicklung ist die Parallelisierung von Testaktionen auch von Vorteil. Im Vordergrund steht hier nicht die Ausführungsdauer, sondern die Belastung eines Systems mit verschiedenen Aktionen. Zur Realisierung von Lasttests eignet sich die Parallelisierung von Testaktionen sehr gut.

TestStand erlaubt das asynchrone Ausführen von Testschritten oder -abläufen. Das Programmiermodell ähnelt dabei der üblichen Thread-Programmierung. Die Software stellt sehr umfangreiche Synchronisierungsmechanismen wie Rendezvous, kritische Abschnitte oder Semaphoren zur Verfügung, um gemeinsame Ressourcen (z. B. Messgeräte) effizient zu nutzen.

5.3.2 Steuerung RedBoot

Der Bootloader RedBoot bietet seine Funktionen über ein Kommandozeileninterface an. Darauf kann entweder über den ersten RS-232 Port oder wahlweise per Telnet zugegriffen werden. Telnet scheidet als dauerhafte Lösung aus, da die IP Adresse des Prüflings konfiguriert und dem Steuer-PC bekannt sein muss. Wenn im Netzwerk mittels DHCP dynamische Adressen verwendet werden, müsste ein weiterer Mechanismus gefunden werden, der das richtige Gerät im Netzwerk findet. Bei der Erstinbetriebnahme in der Produktion liegt möglicherweise sogar keine gültige IP Adresse vor. Die Kommunikation mit RedBoot über die RS-232 Schnittstelle ist daher vorzuziehen.

Zur Realisierung der Kommunikation zwischen TestStand und RedBoot muss eine Glue-Logic gefunden werden. Gängige Telnet Implementierungen sind zu stark mit Ethernet verzahnt. Die „GuF Serial Library.dll“ ist nur zur Ansteuerung von SCPI Geräten zu gebrauchen. Streng genommen sollte eine Namensänderung dieser Bibliothek angestrebt werden, doch zieht das für gewöhnlich das Ändern sehr vieler Skripte nach sich. Da bisher noch kein RedBoot Produkt mit TestStand angesteuert werden musste, existiert hier noch keine Lösung. Es muss also eine Hilfsbibliothek, mit der Möglichkeit Kommandos über die serielle Schnittstelle zu schicken und auf deren Ende bzw. Rückmeldung zu warten, implementiert werden. Am schnellsten sollte dies mit .NET und C# gelingen, da einerseits die Programmierumgebung sehr viel Komfort bietet (.NET 2.0 bringt eine brauchbare Kapselung der seriellen Schnittstelle mit) und andererseits auch die Einbindung von .NET Assemblies in TestStand sehr komfortabel und umfangreich ist. Implementiert wird die Klasse ComLib mit den Methoden:

```
1 public void Write(string s)
2 public void WriteLine(string s)
3 public bool WaitForString(string tofind, int timeout)
4 public bool WriteLineAndWaitForString(string toWrite, string toWait, int
      timeout)
```

Die Methoden `Write` und `WriteLine` nehmen einen String entgegen, der an den seriellen Port geschickt wird. Die `WaitForString` Methode nimmt einen String entgegen, auf den gewartet wird. `WriteLineAndWaitForString` ist eine Kombination aus beiden, also Senden und Warten. Um nicht ewig zu warten, sollte ein Timeout-Parameter übergeben

werden und der Rückgabewert signalisieren, ob der String gefunden wurde oder Timeout zugeschlagen hat. Auf eine ressourcenschonende Programmierung sollte geachtet werden, d. h. die Bibliothek sollte nur reagieren, wenn auch Daten empfangen wurden und nicht mittels Busy-Waiting darauf warten. Die Arbeitsplätze in der Produktion sind in der Regel eher mit niedrigen Leistungsdaten ausgestattet und TestStand und .NET sind stark ressourcenverbrauchende Umgebungen. Die exakte Implementierung findet sich im Anhang C.

Die Eingabe der Testaufrufe in TestStand erfolgt grafisch und wird im XML Format in einer Datei abgelegt. Das Bild 5.5 zeigt die Eingabemaske mit einigen Testfällen und Einstellungen für CAN. Diese Darstellung eignet sich allerdings nicht für weitere Erläuterungen, weshalb im Folgenden eine C# ähnliche Schreibweise benutzt wird.

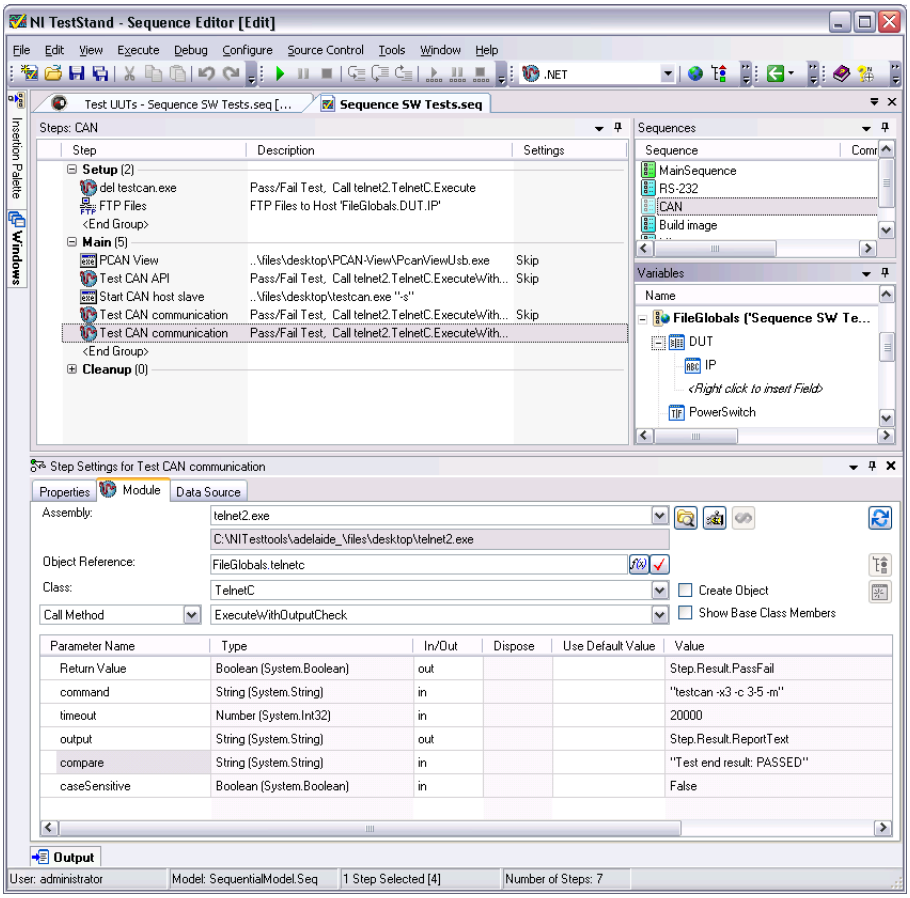


Abbildung 5.5: Eingabemaske von TestStand

Der Aufruf von TestStand aus, zum Laden und Starten des Betriebssystems wird in Listing 5.1 gezeigt.

Ein Objekt comlib der Klasse ComLib wird instanziiert. In TestStand musste die entsprechende Assembly, also die .NET DLL, die die Klasse ComLib enthält, angegeben werden, was

Listing 5.1: Auszug aus dem Testablauf zur Ansteuerung von RedBoot

```
1 FileGlobals.comlib = new ComLib().Open("COM1", 11520);
2 Step.Result.PassFailed = comlib.WriteLineAndWaitForString(
3         toWrite = "load -v -r -b 0x80100000 -h "+StationGlobals.IP+" \""
              +FileGlobals.Device.OSname+"\"",
4         toWait = "RedBoot >",
5         timeout = 90000);
6 Step.Result.PassFailed = comlib.WriteLineAndWaitForString(
7         toWrite = "exec",
8         toWait = "Windows CE Kernel for ARM",
9         timeout = 30000);
```

im Listing nicht zu sehen ist. Die nachfolgenden Funktionsaufrufe enthalten zusätzlich die Parameternamen, um die Bedeutung der Übergabewerte deutlich zu machen. Der Parameter `toWrite` ist das Kommando, das an RedBoot übergeben werden soll. Die Variablen `StationGlobals.IP` und `FileGlobals.Device.OSname` werden von TestStand erzeugt und sollten mit sinnvollen Werten belegt werden. Die beiden weiteren Parameter `toWait` und `timout` im ersten Fall geben an, maximal 90 Sekunden auf den wiederkehrenden Prompt von RedBoot zu warten. Im zweiten Fall wird auf eine Nachricht des Windows CE gewartet, um sicher zu sein, dass es gestartet wurde. Der Rückgabewert wird benutzt, um das Testskript im Fehlerfall zu beenden oder den Fehler nur anzuzeigen und fortzufahren.

Das erste Kommando („`load ...`“) bringt RedBoot dazu eine Datei zu laden. Dazu benutzt RedBoot das geläufige Protokoll Trivial File Transfer Protocol (TFTP). Auf dem Steuer-PC (oder dem Rechner mit der IP in `StationGlobals.IP`) muss ein TFTP-Server gestartet sein, der die benötigte Datei in seinem Wurzelverzeichnis vorhält. Hier wird die freie Variante „Solarwinds TFTP Server“ eingesetzt. Aber auch andere Implementierungen sind im Internet erhältlich. Praktischerweise kann TestStand den TFTP-Server zuvor schon starten.

Nachdem die Datei, also das OS Image, auf das Zielgerät geladen wurde, wird es mit „`exec`“ gestartet. Das OS und die startenden Treiber geben dabei eine Menge Nachrichten über die Debug-Schnittstelle aus. Der CAN-Treiber gibt an, ob er seine Hardware, den SJA1000 gefunden hat. Diese Nachricht kann benutzt werden, um den Test abzubrechen, wenn die Hardware nicht gefunden wurde. Dazu werden in der ComLib weitere nützliche Methoden implementiert, die es erlauben den Empfangsbuffer nach bestimmten Nachrichten zu durchsuchen:

```
1 public string GetStringFromBuffer(string regexPreString, string
      regexValue, bool ignoreCase, bool rightToLeft, int occurenceNumber)
2 public uint   GetUIntFromBuffer(string regexPreString, string regexValue
      , bool ignoreCase, bool rightToLeft, int occurenceNumber)
```

```
3 public bool   IsStringInBuffer(string regex, bool ignoreCase)
```

Im konkreten Fall wird `IsStringInBuffer` verwendet um die CAN-Nachricht abzufragen. Die anderen beiden Funktionen ermöglichen es mithilfe von regulären Ausdrücken beliebige Strings zu suchen, zurückzugeben oder zu parsen[2]. Dies wird u. a. benötigt, um beim automatischen Einrichten des Gerätes die Flash-Speicher- und RAM-Größe zu ermitteln.

Für die praktische Arbeit mit TestStand, also beim Erstellen neuer Testabläufe, ist es hilfreich, wenn alle gesendeten und empfangenen Strings zur Fehleranalyse und Debugging sichtbar werden. Die ComLib schreibt deshalb eine Logdatei.

5.3.3 Steuerung Windows CE

Das Betriebssystem Windows CE bringt ebenfalls Interfaces mit, die sich für die automatische Steuerung eignen. Leider gibt es einige Unterschiede zu RedBoot oder anderen Standardbetriebssystemen, so dass auch hier wieder Hand angelegt werden muss.

Die ComLib aus dem vorherigen Kapitel könnte auch hier verwendet werden, wenn Windows CE eine Kommandozeile auf der seriellen Schnittstelle bereitstellen könnte. Laut Dokumentation sollte dies in der Version 5.0 mithilfe von Registry Einträgen möglich sein. Es wird eine Umleitung der cmd.exe auf die serielle Schnittstelle konfiguriert. Baudrate und andere Verbindungsparameter sind vorher festzulegen. Der Autor scheiterte jedoch daran, dieses für das Jupiter-System einzustellen. Während die Ausgabe der cmd.exe auf der Leitung zu sehen war, wurden Eingaben nicht verarbeitet. Da die cmd.exe für sich genommen funktioniert, muss die Ursache in der Umleitung liegen. Bis zum Schluss dieser Arbeit lies sich allerdings nicht herausfinden, an welcher Stelle es hakt. Die Verwendung dieser Methode hätte auch einen Nachteil gehabt, weshalb die Fehlersuche auch nicht besonders priorisiert wurde. Für gewöhnlich möchte ein Kunde den seriellen Port für seine Applikation nutzen. Ist er durch die cmd.exe belegt, muss er dies vorher abschalten, was unnötigen Aufwand für ihn darstellt. Wenn diese Einstellung wiederum nicht in der Standardkonfiguration enthalten ist, hat der Test ein Anfangsproblem: Wie kann die cmd.exe auf den seriellen Port umgeleitet werden?

Die Schnittstelle der Wahl ist Telnet über Ethernet, welches vom Windows CE bereitgestellt wird. Zur Ankopplung von Telnet an TestStand muss wiederum eine Glue-Logic gefunden oder geschrieben werden, da TestStand diese nicht mitbringt.

Eine Recherche im Internet fördert viele kommerzielle Telnet-Bibliotheken zutage. Die freie und offene Implementierung von [Pinch, 2007] überzeugte dagegen durch Beispielcode und

[2] parsen: Umwandlung einer Zahl aus einem String in eine Ganz- oder Fließkommazahl

die Tatsache, dass sie in C# / .NET implementiert ist. Direkt nutzen kann man das vorhandene Interface allerdings nicht. Es muss noch an die Anforderungen von TestStand angepasst werden. Dazu werden ähnliche Aufrufe wie schon in der ComLib implementiert:

```
1 public bool   Execute(string command, bool waitForPrompt, int timeout)
2 public string ExecuteWithOutput(string command, int timeout)
3 public bool   ExecuteWithOutputCheck(string command, int timeout, out
      string output, string compare, bool caseSensitive)
```

Aufgrund der etwas anderen Semantik als in der RedBoot-Umgebung wurden hier andere Funktionsnamen gewählt. `Execute` sendet ein Kommando, das per String an die Funktion übergeben wird. Die zusätzlichen Parameter `wait` und `timeout` geben an, ob und wie lange gewartet werden soll. Wenn der Timeout zuschlägt, kehrt die Funktion mit `false` zurück. In der Regel wird durch ein Kommando per Telnet auf dem entfernten System ein Prozess gestartet. Nachdem der Prozess sich beendet hat, wird mit einem Prompt („>“) Eingabebereitschaft signalisiert. Genau darauf wartet die Funktion und kehrt mit `true` zurück (sofern der Timeout noch nicht abgelaufen ist). Das bedeutet keinesfalls, dass das Kommando erfolgreich war, lediglich dass es beendet wurde (oder wegen eines Fehlers gar nicht erst durchgeführt werden konnte). Mithilfe der beiden weiteren Funktionen `ExecuteWithOutput` und `ExecuteWithOutputCheck` kann die Ausgabe auf den gewünschten Inhalt überprüft werden. Erstere gibt den Ausgabestrom einfach zurück und der Aufrufer, in diesem Fall TestStand, muss sich um die Auswertung kümmern. Letztere übernimmt die Suche nach einem gewünschten String selbst.

Im Nachhinein, bei der Wiederverwendung dieser Adapterkomponenten in einem anderen Projekt, hat sich gezeigt, dass die unterschiedlichen Interfaces der Telnet und der seriellen Bibliothek hinderlich sind. Besser wäre es gewesen, ein und dasselbe Interface für beide Kommunikationskanäle zu implementieren. Dadurch könnte die Verbindungsart (Telnet oder seriell) abstrahiert werden und darauf aufbauende Software etwas weniger komplex und einfacher zu implementieren sein. Für diesen Teil der Arbeit kann dieser Makel jedoch so belassen werden.

Listing 5.2: Auszug aus dem Testablauf zur Ansteuerung von Windows CE

```
1 FTP.TransferFile(FileGlobals.Device.IP, testcan.exe);
2 FileGlobals.telnet2 = new Telnet2().Connect(FileGlobals.Device.IP);
3 ExecuteLocal.StartAsynchronously("testcan.exe -s");
4 Step.Result.PassFailed = telnet2.ExecuteWithOutputCheck(
5         command = "testcan.exe -m",
6         timeout = 30000,
7         output = ref out Step.Result.ReportText,
8         compare = "Test end result: PASSED",
9         caseSensitive = false);
```

Die Nutzung des Telnet Adapters zeigt das Beispiel in Listing 5.2. Zuvor werden benötigte Dateien zum Ausführen des Tests auf das Zielgerät mittels FTP heruntergeladen. Diese Funktionalität bringt TestStand schon mit. Anschließend wird eine Telnet-Verbindung zum Zielgerät aufgebaut. Die IP kann in TestStand entweder statisch eingestellt, vom User abgefragt oder in den vorherigen Testschritten aus den Ausgaben auf dem seriellen Debug-Port automatisch entnommen werden. Im Beispiel wird der Test des CAN-Treibers gestartet. Dieser besteht aus einem Slave und einem Master-Teil (siehe Kap. 4.6.2). Der Slave wird auf dem lokalen PC gestartet. Die Ausführung erfolgt im Hintergrund, damit der Testablauf fortfahren kann. Das Starten des Masters erfolgt auf dem Zielgerät. Die Ausgaben des Testprogramms werden an die Variable `Step.Result.ReportText` übergeben. Der Inhalt wird später in einem HTML-Bericht zu sehen sein. Mithilfe des Parameters `compare` wird die Prüfung des Ausgabestroms erreicht. Wenn der Test erfolgreich ist, wird „Test end result: PASSED“ erwartet. Wenn der Test fehlschlägt, wird dieser String nicht ausgegeben und die Methode gibt `false` zurück, sobald der Prompt wieder in der Telnet-Umgebung auftaucht. Eine andere Möglichkeit ist der `timeout`, der ebenfalls `false` zurückgibt.

5.3.4 Prüfung der Debug-Ausgaben

Um eine Prüfung der Debug-Ausgaben zu ermöglichen, müssen vorher die verschiedenen Kommunikationskanäle Telnet und Seriell näher untersucht werden. Folgende Eigenschaften ergeben sich:

- Verschiedene Windows CE Funktionen bringen die Ausgaben an verschiedene Stellen. Die Standardausgabe mittels printf ist nur auf dem Telnet-Kanal zu sehen. Debug-Nachrichten werden mit DEBUGMSG oder RETAILMSG auf die erste serielle Schnittstelle, welche als Debug-Kanal konfiguriert ist, gesendet. Der Autor hat sich angewöhnt - wegen möglicher Verwirrung - immer beide Kanäle zu benutzen.
- Die Nachrichten der Telnet-Umgebung enthalten nur die Ausgaben des aktuell laufenden Prozesses. Andere Nachrichten sind nicht zu sehen.
- Die Nachrichten auf dem Debug-Port (serielle Schnittstelle) enthalten die Initialisierungsmeldungen beim Bootvorgang.
- Der Debug-Port stellt Nachrichten aller Prozesse dar. Dadurch ist eine zeitliche Zuordnung der Nachrichten untereinander möglich.
- Nicht behandelte Exceptions sind nur auf dem Debug-Port sichtbar.

Auf Seite 85 wurden schon die verschiedenen Arten von Debug-Ausgaben aufgeführt. Hier wird beispielhaft der CAN-Treiber auf erfolgreiches Starten geprüft und die Erkennung von allgemeinen Exceptions gezeigt. Zum Mitschneiden des Nachrichtenstroms kann die Comlib

aus Kapitel 5.3.2 verwendet werden. Nachdem das Betriebssystem gestartet wurde, wird die Funktion `IsStringInBuffer` benutzt, um die Erfolgsmeldung der CAN-Initialisierung zu suchen.

```
1 Step.Result.PassFailed = comlib.IsStringInBuffer(
2         regex = "SJA1000 detected",
3         ignoreCase = false);
```

Die Erkennung von Exceptions funktioniert analog dazu und basiert auf dem Format einer Exception-Nachricht, die das Betriebssystem ausgibt, sobald eine Exception nicht abgefangen wurde.

```
Data Abort: Thread=99198b6c Proc=96ce9598 'device.exe'
AKY=00020001 PC=00011474 RA=01f52ec4 BVA=24000000 FSR=00000007
```

Diese Art von Exceptions deutet immer auf Programmierfehler hin und sollte sehr ernst genommen werden. Die Überprüfung darauf kann mit der Funktion `IsStringInBuffer` realisiert werden. Der Suchstring sollte die festen und veränderlichen Teile der Nachricht berücksichtigen. Die Methode akzeptiert dazu reguläre Ausdrücke, wie den Folgenden:

```
"AKY=.* PC=.* RA=.* BVA=.* FSR=.*"
```

Um die Ursachen besser abgrenzen zu können, wird diese Überprüfung in regelmäßigen Abständen in den Testablauf eingebaut, etwa nach dem Testen jeder Komponente.

5.3.5 Netzteil

Ein steuerbares Netzteil ist in der Produktion nötig, um bei kritischen Fehlern wie z. B. zu hoher Stromverbrauch eine Notausschaltung durchführen zu können oder um den Prüfling für gewisse Testaktionen stromlos zu schalten. Der zuletzt genannte Grund ist auch für den automatischen Entwicklungstest interessant. Vorstellbar ist auch, dass Fehlersituationen im Entwicklungstest nur durch das Abschalten der Spannungsversorgung und Neustarten des Gerätes zu beheben sind.

Das DC-Netzteil E3642A von Agilent bietet eine RS-232 und eine GPIB[3] Schnittstelle. Für GPIB werden Erweiterungskarten im PC benötigt, weshalb die einfachere Schnittstelle RS-232 verwendet wird. Die Ansteuerung erfolgt mit einem ASCII basiertem Protokoll, dem Standard Commands for Programmable Instruments (SCPI).

Listing 5.3 zeigt die Befehle beispielhaft. Zuerst wird das Gerät in den Remote-Modus geschaltet. Danach erfolgt die Programmierung der Spannungsversorgung, sowie das an- oder ausschalten.

[3] General Purpose Instrumentation Bus (GPIB)

Listing 5.3: Kommandos zur Ansteuerung des Netzteils E3640A

```
1 SYSTEM:REMOTE
2 VOLT:RANGE P20V;:VOLT 12.0;:CURR 1.2
3 OUTPUT ON
4 OUTPUT OFF
```

Abweichungen bei der Ausstattung der Netzteile müssen evtl. in anderen Aufbauten berücksichtigt werden. So hat z. B. das E3632A zwei Ausgangskanäle. Im Befehl muss dann zusätzlich die Kanalnummer mit angegeben werden:

```
1 INSTRUMENT OUT1;:VOLT:RANGE P35V;:VOLT 12.0;:CURR 1.2
```

Die Übermittlung der Kommandos erfolgt mit der „GuF Serial Library.dll“. Diese ist im Rahmen der Einführung von TestStand in der Produktion, um die Netzgeräte und Messgeräte von Agilent anzusteuern, entstanden. Neben der Übertragung von einfachen Kommandos hilft sie bei der Abfrage von Messwerten.

5.3.6 Nightly Builds

Das automatische Kompilieren des Quelltextes ist nicht zwingend erforderlich. Während der Entwicklung und spätestens an strategischen Zeitpunkten des Projektes müssen sowohl interne Zwischenversionen als auch Versionen für Kunden erstellt werden. Diese Versionen können dann an den Test übergeben werden. Das Generieren einer solchen Version kann jedoch zeitaufwändig sein, da evtl. mehrere Konfigurationen erstellt werden müssen, oder der Sourcecode eines Mitarbeiters mit dem eines anderen aktualisiert werden müssen. Dieses führt dazu, dass während der Entwicklung auf Zwischenversionen verzichtet wird.

Automatisch generierte Versionen haben den Vorteil, dass sie praktisch jederzeit ohne Aufwand erzeugt werden können. Vor jedem Testlauf wird der aktuellste Sourcecode übersetzt und geprüft. Damit kann ein automatischer Testlauf viel häufiger durchlaufen werden und somit können Fehler früher entdeckt werden. Angenommen, ein Build und Testvorgang wird jede Nacht durchgeführt, dann könnten die Entwickler schon am nächsten Morgen die Testergebnisse ihrer Arbeit vom Vortag erhalten. Und das, ohne jeden Abend eine Version zusammenstellen zu müssen. Integrationsprobleme unterschiedlicher Komponenten verschiedener Mitarbeiter könnten damit auch schneller sichtbar werden.

Auch bietet sich ein System zur automatischen Übersetzung als Referenzsystem an, wenn es auf einem speziell dafür reservierten PC installiert wird. Da die Entwicklungsumgebung

von Windows CE sehr umfangreich und damit komplex ist, ist es in der Vergangenheit vorgekommen, dass unterschiedliche Arbeitsplätze beim Kompilieren unterschiedliche Ergebnisse geliefert haben.

Das Aufsetzen eines automatischen Builds ist in der Regel nicht besonders aufwändig, weshalb die Vorteile schnell überwiegen können. Das Versionierungssystem Subversion liefert Kommandozeilenwerkzeuge, die sich problemlos automatisieren lassen.

Anders sieht es bei der Entwicklungsumgebung für Windows CE aus. In der Regel wird eine grafische Entwicklungsumgebung benutzt. Erst aus dieser heraus lassen sich dann Kommandozeilenfenster starten, die alle benötigten Umgebungsvariablen für den Buildvorgang enthalten. Diese Umgebungsvariablen können mit einem speziellen Tool von Microsoft aus der Projektdatei gelesen und anschließend gesetzt werden. Der Buildvorgang kann dann ebenfalls gestartet werden. Das Listing 5.4 zeigt den Ablauf stark vereinfacht.

Listing 5.4: Automatisches Übersetzen des Windows CE 5.0

```
1 svn checkout
2 pbxmlutils .... > setenv.bat
3 setenv.bat
4 blddemo -q
```

Beim Aufsetzen und Aufräumen des Quellcode-Baums sollte achtsam vorgegangen werden. Windows CE ist eine sehr große Sammlung von bereits übersetzten, statischen Bibliotheken, Komponenten, die als Sourcecode vorliegen und vielen Einstellungsdateien. Und das für vier Prozessorarchitekturen. Nach einer typischen Installation bei Garz & Fricke für 2 Zielsysteme liegen etwa 1,5 GB Entwicklungsdateien vor. Dazu kommen einige Hundert MB Sourcecode von Garz & Fricke. Beim Buildvorgang können etwa weitere 500 MB anfallen.

Später können auch viele unterschiedliche Konfigurationen automatisch kompiliert werden. Da Windows CE ein sehr modulares System mit vielfältigen Einstellungsmöglichkeiten ist, besteht immer das Risiko, dass eine Komponente nicht mit einer bestimmten Konfiguration funktioniert. Die typischen Konfigurationen können somit die Verträglichkeit der Komponenten testen, noch bevor eine Konfiguration auch tatsächlich gebraucht wird. z. B. ist es in der Vergangenheit vorgekommen, dass ein Stück Software, das sowohl in einem grafischen Umfeld als auch auf einem Gerät ohne Display funktionieren sollte, aber unabsichtlich Abhängigkeiten zu grafischen Komponenten enthielt. Solche Fehler werden üblicherweise erst bemerkt, wenn diese Komponente tatsächlich in der abgespeckten Konfiguration benötigt wird. Wenn dies erst sehr spät passiert, muss sich der Entwickler neu einarbeiten, um den Fehler zu beheben. Wenn der Entwickler allerdings zeitnah auf das Problem hingewiesen wird, kann diese Einarbeitungszeit entfallen.

5.4 Testzyklus

Im Abschnitt 5.3 wurde die Umgebung geschaffen und anhand des CAN-Treiber-Beispiels illustriert. In der Praxis wird der automatische Test insoweit vervollständigt werden, bis die Testabdeckung ausreichend ist (siehe auch Tabelle 2.2 den von Garz & Fricke entwickelten Komponenten). Auch die Testfälle des CETK können zu einem großen Teil ohne Benutzerinteraktion ablaufen und sollten daher in den Ablauf mit eingebunden werden.

Während der Entwicklung wird sich zwangsläufig die Frage stellen, wann welche Tests durchlaufen werden müssen. In der Regel werden bei Erreichen eines Meilensteins alle Testfälle solange wiederholt, bis keine Fehler mehr gemeldet werden. Die Automatisierung ist in diesem Fall sehr hilfreich, da das Ergebnis nach Aufsetzen und Durchlaufen des Tests schon nach kurzer Zeit zur Verfügung steht. Es sollte an dieser Stelle jedoch nicht vergessen werden, dass bereits existierende Tests die Änderungen einer Software nicht überprüfen. Es können lediglich Nebenwirkungen in Softwareteilen erkannt werden, die durch die Tests abgedeckt werden. Neue oder veränderte Softwarekomponenten sind daher mitsamt ihrer Tests genauer zu überprüfen. Dazu ist die Kenntnis aller Testfälle und Änderungen in der Software notwendig. Die Änderungen der Software gehen aus den „Relase Notes“ hervor, die bei Erreichen eines Meilensteins und Lieferung einer Softwareversion erstellt werden. Die vorhandenen Testfälle sollten ebenfalls dokumentiert sein. Es wird hier eine Struktur vorgeschlagen, die es ermöglicht, eine Übersicht über alle Testfälle zu erstellen. Beispielhaft werden die Tests für die CAN-Komponente und ein CETK-Testfall eingetragen. Letzterer wird hier nicht weiter behandelt, dient aber als Beispiel für Einbindung externer Tests.

Komponente / ID:	CAN-01
Hardware / Variante:	Jupiter-ARM11 mit CAN-Option
Beschreibung:	Funktionen der CAN-API werden überprüft
Dateien:	testcan.exe (ARMV4I)
Aufruf:	testcan.exe -x1
Hilfsmittel:	-
Weitere Dokumente:	Diplomarbeit von Christoph Kutzera, Kap. 4.5
Sourcecode:	/trunk/Tests/WINCE500/Common/CAN

Komponente / ID:	CAN-02
Hardware / Variante:	Jupiter-ARM11 mit CAN-Option
Beschreibung:	Überprüft die Kommunikation des CAN-Treibers
Dateien:	testcan.exe (ARMV4I und x86 Varianten)
Aufruf:	Slave: testcan.exe -x3 -s / Master: testcan.exe -x3 -m
Hilfsmittel:	PCAN-USB-Adapter
Weitere Dokumente:	Diplomarbeit von Christoph Kutzera, Kap. 4.6
Sourcecode:	/trunk/Tests/WINCE500/Common/CAN

Komponente / ID:	CAN-03
Hardware / Variante:	Jupiter-ARM11 ohne CAN-Option
Beschreibung:	Überprüfung der Bootausgaben, ob der CAN-Treiber die Initialisierung mit entsprechender Meldung abbricht.
Dateien:	-
Aufruf:	IsStringInBuffer(SSJA1000: not detected")
Hilfsmittel:	-
Weitere Dokumente:	Diplomarbeit von Christoph Kutzera, Kap. 5.3.4
Sourcecode:	-

Komponente / ID:	OAL-01
Hardware / Variante:	Jupiter-ARM11
Beschreibung:	Überprüfung der OAL/HAL Library mit CETK Testfällen
Dateien:	tux.exe kato.dll tooltalk.dll ioctltest.dll
Aufruf:	"tux -o -d Ioctltest -f kato.log | type kato.log"
Hilfsmittel:	-
Weitere Dokumente:	ms-help://MS.WindowsCE.500/wcedebug5/html/wce50conOALIOCTLTest.htm
Sourcecode:	WINCE500\PRIVATE\TEST\PQOAL\IOCTLS

Mit dieser Dokumentation wird eine gute Hilfestellung für Testentwickler und Tester gegeben. Falls Probleme auftreten, kann ein Test anhand dieser Informationen auch manuell, ohne die Umgebung von TestStand, ausgeführt werden. Weiterführende Dokumente sollten möglichst verlinkt werden. Dazu gehören in erster Linie die detaillierte Dokumentation, aber auch die entsprechenden Anforderungsdokumente. Die benötigten Dateien und der Aufruf der Testfälle sollte ebenfalls enthalten sein. Verweise auf den Sourcecode der Testfälle runden das Ganze ab. In der Praxis läuft diese Liste ständig Gefahr dem aktuellen Stand hinterher zuhinken. Eine andere Möglichkeit wäre, diese Informationen in TestStand selbst zu pflegen, was mit entsprechenden Kommentarfeldern zu den Testaktionen grundsätzlich möglich ist. Doch würde es sich zum Nachteil der Übersicht in der Eingabemaske von TestStand auswirken.

Die Durchführung der Regressionstests sollte stets einhergehen mit der kreativen Prüfung dieser Testliste und den zuletzt vorgenommenen Änderungen in der Software.

5.5 Ergebnisse

Der automatische Ablauf von Testaktionen konnte erfolgreich implementiert werden. Durch das Ausnutzen von vorhandenen Ressourcen (TestStand) konnten Kosten eingespart werden und gleichzeitig eine hohe Wiederverwendung erreicht werden. Die realisierte Umgebung wurde schon zum Zeitpunkt der Erstellung dieser Arbeit in die Produktionsumgebung

übernommen und wird dort ebenfalls erfolgreich eingesetzt. Wünschenswert wäre es gewesen, noch mehr bestehende Tests, wie z. B. die des CETK einzubinden, um die Testabdeckung des Jupiter-Systems schrittweise mit vorhandenen Mitteln zu erhöhen. Dies kann im Rahmen dieser Arbeit allerdings nicht geleistet werden und ist auch nicht Teil der Aufgabenstellung.

6 Zusammenfassung und Ausblick

Insgesamt bietet diese Diplomarbeit einen umfangreichen Rundumblick über den Bereich der Softwaretests. Die Kernbereiche dieser Arbeit, automatisierte Komponenten- und Integrationstests, konnten detailliert vorbereitet und durchgeführt werden. Dem besonderen Umfeld der Embedded-Systeme und der starken Abhängigkeit und Kopplung zur Hardware wurde stets Rechnung getragen. Nach der Analyse der Situation und der Aufarbeitung der Grundlagen wurden Tests am exemplarischen Beispiel des CAN-Treibers gezeigt. Alle erstellten Tests wurden im Kapitel der Automatisierung erprobt. Hierbei zeigte sich, dass die Schaffung einer Umgebung für einen automatischen Test durchaus aufwändig sein kann.

Alle realisierten Testfälle und die für die Umgebung benötigten Implementierungen fließen in die Entwicklung der ARM11-Module ein und finden somit praktische Verwendung.

Der aktuelle Entwicklungsprozess und die prozess-unterstützenden Werkzeuge bei Garz & Fricke wurden beschrieben und verwendet. Zur Verbesserung der Qualität im Testprozess wurden in dieser Arbeit weitere Methoden aufgezeigt, die bei Planung, Implementierung, Ausführung und Überwachung von Tests notwendig sind.

6.1 Offene Punkte

Die Anforderungen an den CAN-Treiber wurden nicht vollständig abgedeckt. Für das Thema Powermanagement etwa, mangelt es an der Implementierung im Treiber selbst. Testfälle sollten anschließend erstellt werden, wobei drauf geachtet werden sollte, dass schon ähnliche Testfälle für andere Treiber im CETK enthalten sein könnten.

Die Anforderungen bzgl. Performance werden wieder eine Rolle spielen, wenn Projekte mit entsprechendem Fokus auf Echtzeit mit CAN umgesetzt werden sollen. Dann sind die Anforderungen von Kunden- bzw. Applikationsseite einzufordern und die Testfälle ggf. anzupassen bzw. zu erweitern.

Der automatische Test ist natürlich ebenfalls voranzutreiben. Praktisch gesehen ist dies sogar die wichtigste Aufgabe für die nächsten Monate. Derzeit können nur etwa 3 von 15 Komponenten automatisch getestet werden. Ziel sollte es sein, alle Komponenten im Test abzudecken. Dazu sollten bestehende Tests in hohem Maße wiederverwendet werden. Auch

das CETK bietet eine Reihe von Testfällen, die für andere Komponenten benutzt werden können.

6.2 Ausblick

Für die weitere Arbeit bei Garz & Fricke sind im Verlauf dieser Diplomarbeit noch einige Punkte aufgefallen, die hier besondere Erwähnung verdienen. Ideen, erste Konzepte und weitere Werkzeuge werden kurz vorgestellt, um weitere Verbesserungen einzuleiten.

6.2.1 Testpolitik

In der Analyse wurde schon das gute technische Verständnis der Geschäftsführung erwähnt. Im Teil der Anforderungsanalyse für den CAN-Treiber wurde jedoch schnell klar, dass die Vorgaben der nicht-funktionalen Anforderungen nicht ausreichen. Das gute Verhältnis zur Geschäftsführung sollte genutzt werden, um dieses Defizit zukünftig abzubauen. Anforderungen sollten so früh wie möglich formuliert werden. Die Geschäftsführung sollte bei neuen Entwicklungen stets eine Richtung für die gewünschten Qualitätsmerkmale eines Produktes vorgeben. Die Projektverantwortlichen sollten den Testaufwand einschätzen und an die Geschäftsführung zurückmelden. In Gesprächen und Interviews, auch mit den Kunden, sollte dann eine gemeinsame Vorstellung vom benötigten Aufwand erarbeitet werden.

6.2.2 Rückkopplung in den Entwicklungsprozess

Die Erfahrungen dieser Arbeit können grundlegend für die Verbesserung des weiteren Entwicklungsprozesses sein. Dafür muss zunächst das Wissen über die hier angewandten Testmethoden weitergegeben werden. Gerade in der täglichen Praxis können die hier erarbeiteten Beispiele und Erfahrungen einen erheblichen Zeitgewinn mit sich bringen, da sich nicht erneut über Strukturen und Umgebung Gedanken gemacht werden muss. So ist z. B. die Art der Testfallprogrammierung und Datentrennung eine durchaus hilfreiche und Übersichtliche Vorgehensweise und sollte auch in den zukünftigen Testimplementierungen der anderen Komponenten Anwendung finden. Auch der Entwurf mittels Master/Slave-Aufteilung kann natürlich bei anderen Busstrukturen ebenfalls nützlich sein. Diese und andere Punkte sollten in einer Art Workshop an die Kollegen weitergetragen werden.

Auch ist TestStand der Entwicklungsabteilung noch nicht besonders vertraut. Alle Entwickler sollten die Potenziale dieser Software kennen, da sie beim automatischen Test zum Tragen

kommen. Wenn Entwickler Testfälle erstellen, sollte zukünftig darauf geachtet werden, dass diese einfach in den automatischen Ablauf integriert werden können.

Die hier aufgeführten Punkte sollten nicht alle zusammen in einer Sitzung behandelt werden. Vielmehr ist eine Aufteilung der Themen in eine Workshop-Reihe zu empfehlen. Die Erfahrungen und Empfehlung zur konkreten Implementierung von Testfällen sollte eine Einheit bilden, die Schulung für die Software TestStand eine andere. Und auch die Betrachtung des Entwicklungsprozesses und die Einordnung der Testaktivitäten sollte separat, am besten mit allen Projektleitern, besprochen werden. Hier muss besonders hervorgehoben werden, dass erfolgreiches Testen nur mit einem guten Anforderungsmanagement einhergeht.

6.2.3 Einbettung der Testaktivitäten in Trac

Das Trac-System wurde in Kapitel 2.4.3 vorgestellt. Während dieser Arbeit wurde es genutzt, um gefundene Fehlerwirkungen zu dokumentieren und später abzuarbeiten. Der Zustand `intest` im Workflow des Trac-Systems (Abbildung 2.3) verdient aus Sicht des Testens besondere Beachtung.

Bevor ein Ticket geschlossen werden kann, muss es zwingend den Zustand `intest` durchlaufen. Dieser Zustand ist neu und so nicht im Standard-Workflow von Trac vorhanden. Im Garz & Fricke System wurde dieser Zustand bereits eingepflegt, doch noch nicht besonders genutzt bzw. beachtet. Der Zustand `intest` sollte stärker in den Testprozess eingebunden werden. Erst wenn eine Änderung überprüft wurde, darf dieser Zustand verlassen werden. Die Schwierigkeit zeigt sich an dieser Stelle in der Auswahl der Tests, die für eine Änderung durchlaufen werden müssen. Der Durchführende muss Testfälle aus einem bestehenden Pool auswählen und ausführen oder neue Testfälle implementieren. Auch eine reine statische Prüfung, wie z. B. das Review, ist zum Testen einer Änderung denkbar.

Weitere interessante Möglichkeiten des Trac-Systems zeigt das [TestCaseManagementPlugin]. Damit können die Testfälle eines Projektes verwaltet werden. Testfallgruppen und -abhängigkeiten können definiert werden, um die tatsächliche Ausführung dann an Benutzer zuzuweisen. Ob diese Möglichkeiten in den Prozess von Garz & Fricke passen, muss aber erst evaluiert werden.

6.2.4 Requirementstracing

Das Requirementstracing, wie es in Kapitel 4.2.1 gezeigt ist, hat sich während der Arbeit als notwendige Maßnahme herausgestellt, um Anforderungen effizient zu verwalten. In jedem

Fall sollten Ansätze zur regelmäßigen Nutzung dieses Instruments bei Garz & Fricke diskutiert werden. Schon jetzt ist absehbar, dass dafür keine teuren Werkzeuge wie [DOORS] angeschafft werden können.

Eine Idee, die während dieser Arbeit entstanden ist, ist die Einbettung des Requirementstracing in Trac. Mit seinem Ticketsystem bietet es grundsätzlich die Möglichkeit, referenzierbare Einträge einzustellen. Wenn auch, wie im vorherigen Kapitel vorgeschlagen, die Testfälle in Trac verwaltet würden, könnten sich starke Synergien bilden: Das Querreferenzieren von Anforderungen, Implementierungen, Testfällen und Fehlereinträgen erreicht in einem Online-System wie Trac einen sehr hohen Bedienkomfort bei der Navigation.

6.2.5 Weitere Testwerkzeuge

Bei der Recherche nach Testwerkzeugen sind sehr viele Tools aufgefallen, die in anderen Projekten nützlich sein könnten. So können auch grafische Programme werkzeuggestützt mit sog. „Capture and Replay“ Tools geprüft werden. Der [TestExplorer] zum Beispiel eignet sich für Desktop Programme.

Ein sehr interessantes Konzept hat VNC Robot zu bieten. Es stellt mittels VNC den Bildschirminhalt eines Windows CE auf Desktop System dar. Die „Capture and Replay“ Technik kann dann an dieser Stelle angewandt werden. Bei Garz & Fricke existieren zwar nicht viele grafische Tools und diese sind dann auch nicht besonders missionskritisch (z. B. Versionsanzeigetool für Debug-Zwecke). Doch könnte dies von großem Interesse für die Windows CE Kunden, speziell für die Applikationsprogrammierer sein. Da Garz & Fricke hier auch beratend wirkt und Einsteigerschulungen anbietet, sollte dieses Thema nochmal genauer betrachtet werden.

Für die Suche nach weiteren Werkzeugen zur Unterstützung von Softwaretests lohnt es sich in jedem Fall die folgenden Übersichtsseiten aufzusuchen. Insgesamt werden weit über hundert verschiedene Werkzeuge gelistet.

- [Opensource testing]
- [Automated test tools]
- [Testing FAQ]
- [imbus Toollist]
- [APTest]

Literaturverzeichnis

ADELAIDE-brief
Computermodule ADELAIDE Data brief. http://www.garz-fricke.com/media/Embedded/SoMs/pdfs/ADELAIDE/AdelaideDataBrief.pdf, Abruf: Mai 2008

ADELAIDE-manual
Computermodule ADELAIDE Hardware manual. http://www.garz-fricke.com/media/Embedded/SoMs/pdfs/ADELAIDE/AdelaideV1.1HardwareManualRev1-0.pdf, Abruf: Mai 2008

APTest
APTest. http://www.aptest.com/resources.html, Abruf: März 2008

Automated test tools
Automated test tools. http://sqa-test.com/toolpage.html, Abruf: März 2008

Boehm 1979
BOEHM, B. W.: *Guidelines for verifying and validating software requirements and design specification*. 1979. – ISBN 0–321–15986–1

Broekman u. Notenboom 2003
BROEKMAN, Bart ; NOTENBOOM, Edwin: *Testing Embedded Software*. Addison Wesly, 2003. – ISBN 0–321–15986–1

CAN 2.0 1991
CAN Specification Version 2.0. http://www.semiconductors.bosch.de/pdf/can2spec.pdf. Version: 1991, Abruf: März 2008

CANopen
CANopen (EN 50325-4). http://www.can-cia.org, Abruf: April 2008

DOORS
DOORS - Qualitätsverbesserung mit Anforderungsmanagement und Nachverfolgbarkeit. http://www.telelogic.de/products/doors, Abruf: Mai 2008

Extreme Programming
Extreme Programming. http://de.wikipedia.org/wiki/Extreme_Programming, Abruf: Mai 2008

Fish
FISH, Shlomi: *Better SCM Initiative: Comparison*. http://better-scm.berlios.de/comparison/comparison.html, Abruf: März 2008

imbus Toollist
imbus Toollist. http://www.imbus.de/tool-list.shtml, Abruf: April 2008

ISO 9126
ISO/IEC: *Software engineering - -Product quality - Part 1: Quality model (identisch mit [DIN 6672, 94], Bewerten von Softwareprodukten, Qualitätsmerkmale und Leitfaden zur ihrer Verwendung).*

JUPITER-brief
Touchpanel JUPITER-MX31 Data brief. http://www.garz-fricke.com/media/Embedded/DoMs/pdfs/JUPITERDataBrief.pdf, Abruf: Mai 2008

JUPITER-manual
Touchpanel JUPITER-MX31 User Manual. http://www.garz-fricke.com/media/Embedded/DoMs/pdfs/JUPITER-MX31UserManualr06.pdf, Abruf: Mai 2008

Llopis 2004
LLOPIS, Noel: *Exploring the C++ Unit Testing Framework Jungle*. http://www.gamesfromwithin.com/articles/0412/000061.html. Version: 2004, Abruf: Mai 2008

Opensource testing
Opensource testing. http://opensourcetesting.org/functional.php, Abruf: Mai 2008

Pinch 2007
PINCH, David: *Telnet Class Library for .NET (version 0.2)*. http://www.thoughtproject.com/Libraries/Telnet/. Version: 2007, Abruf: März 2008

SAE J1939
SAE J1939: Solide Kommunikation im Nutzfahrzeug. http://www.vector-worldwide.com/vi_j1939_de.html, Abruf: April 2008

Schröder 2005
SCHRÖDER, Jan: *Kopplung von Sensornetzwerken*. HAW Hamburg, 2005. – (Diplomarbeit)

SJA1000
SJA1000 data sheet. http://www.nxp.com/acrobat_download/datasheets/SJA1000_3.pdf, Abruf: Mai 2008

Spillner u. Linz 2005
SPILLNER, Andreas ; LINZ, Tilo: *Basiswissen Softwaretest*. dpunkt.verlag, 2005. – ISBN 3–89864–358–1

Spillner u. a. 2006
SPILLNER, Andreas ; ROSSNER, Thomas ; WINTER, Mario ; LINZ, Tilo: *Praxiswissen Softwaretest - Testmanagement*. dpunkt.verlag, 2006. – ISBN 3–89864–275–5

Subversion
Subversion - Version control system. http://subversion.tigris.org/, Abruf: Juni 2008

TestCaseManagementPlugin
TestCaseManagementPlugin. http://trac-hacks.org/wiki/TestCaseManagementPlugin, Abruf: Juni 2008

TestExplorer
TestExplorer. http://www.sirius-sqa.com/products_testexplorer_details.html, Abruf: Mai 2008

Testing FAQ
Testing FAQ. http://testingfaqs.org, Abruf: März 2008

Trac
Trac - Integrated SCM & project management. http://trac.edgewall.org/, Abruf: Juni 2008

V-Modell XT 2006
V-Modell XT. http://www.v-modell-xt.de/. Version: 2006, Abruf: Mai 2008

A Projektplan

Der Projektplan wurde mit Microsoft Project 2007 erstellt. Die Abbildung A.1 zeigt die Auflistung aller geplanten Tätigkeiten und deren Reihenfolge bzw. Abhängigkeiten in einem Gantt-Diagramm.

Die Statusanzeige ist in Prozent ausgedrückt. Der Plan wurde im Laufe der Diplomarbeit regelmäßig angepasst. Neue Punkte sind hinzugekommen, andere Punkte wurden verfeinert, bzw. neu geschätzt.

Der Schnappschuss wurde am 5. Mai 2008 aufgenommen. Es ist deutlich zu sehen, dass für dieses Datum noch zu wenig Punkte abgeschlossen wurden. Vor allem im Bereich Ist-Situation und Integrationstest zeigten sich Verzögerungen. Diese konnten im Verlauf der Arbeit, u. a. durch bessere Verknüpfung der Inhalte untereinander, wieder aufgeholt werden.

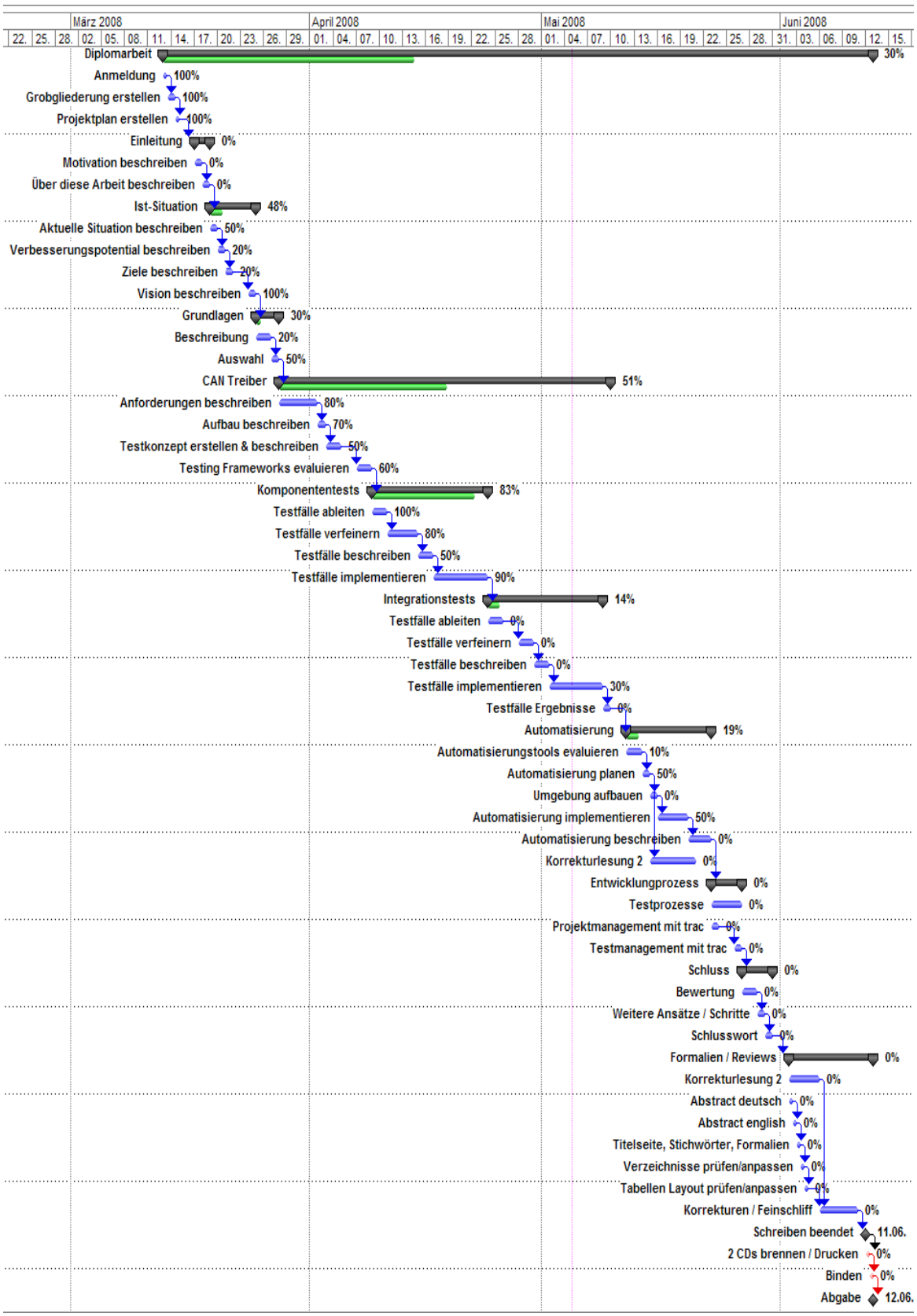


Abbildung A.1: Projektplan

B CAN-API: Header und Dokumentation

Listing B.1: CAN-API: Header und Dokumentation

```
/*****************************************************************************/
/* Copyright (C) 2008 Garz&Fricke Industrieautomation GmbH           */
/*      No use or disclosure of this information in any form without     */
/*      the written permission of the author                             */
/*****************************************************************************/

/** @file       CAN_Api.h
*   @brief  Definitions for the CAN API exports for Windows CE.
*           SVN: $Revision: 1353 $
*   @mainpage   CAN_API
*/
#ifndef GFCANAPICE_H
#define GFCANAPICE_H

// The following ifdef block is the standard way of creating macros which make exporting
// from a DLL simpler. All files within this DLL are compiled with the GFCAN32CE_EXPORTS
// symbol defined on the command line. this symbol should not be defined on any project
// that uses this DLL. This way any other project whose source files include this file see
// GFCAN_API functions as being imported from a DLL, wheras this DLL sees symbols
// defined with this macro as being exported.
#ifdef GFCAN32CE_EXPORTS
#define GFCAN_API __declspec(dllexport) /*!< Define for exporting functions. */
#else
#define GFCAN_API __declspec(dllimport) /*!< Define for importing functions. */
#endif


#define GF_CAN_API_VERSION 3


/*****************************************************************************/
/*  Defines                                                                  */
/*****************************************************************************/

/** Written into descriptor register of CAN controller during driver installation,
*   driver deinstallation and message place deinstallation (short descriptor).
*/
#define CAN_NO_IDENTIFIER       (USHORT)0xFFE0

/** Written into descriptor register of CAN controller during driver installation,
*   driver deinstallation and message place deinstallation (extended descriptor).
*/
#define CAN_NO_IDENTIFIER_EXT   (ULONG)0xFFFFFFF8

#define CAN_MSG_STANDARD    0x00        /*!< Use standart (11 bit id) frame types. */
#define CAN_MSG_EXTENDED    0x01        /*!< Use extended (29 bit id) frame types. */

#define REASON_TRANSMIT_SUCCESS         0   /*!< Code for the Message Queue. */
#define REASON_RECEIVE_SUCCESS          1   /*!< Code for the Message Queue. */
#define REASON_BUSOFF_STATE             2   /*!< Code for the Message Queue. */
#define REASON_WARNING_BELOW_OVERFLOW   3   /*!< Code for the Message Queue. */
#define REASON_NEW_MESSAGE              4   /*!< Code for the Message Queue. */
#define REASON_ERROR_PASSIVE_STATE      5   /*!< Code for the Message Queue. */
#define REASON_MESSAGE_LOST             6   /*!< Code for the Message Queue. */


/**
 *  Error codes.
 */
/*@{*/
/** Define for error if controller status is RWARNING. */
#define CAN_E_RWARNING                  ((NTSTATUS)0x40070020L)

/** Define for error if controller status is RPASSIVE. */
#define CAN_E_RPASSIVE                  ((NTSTATUS)0x40070021L)

/** Define for error if controller status is TWARNING. */
#define CAN_E_TWARNING                  ((NTSTATUS)0x40070022L)

/** Define for error if controller status is TPASSIVE. */
#define CAN_E_TPASSIVE                  ((NTSTATUS)0x40070023L)

/** Define for error if controller status is BUSOFF. */
```

```
74  #define CAN_E_BUSOFF                    ((NTSTATUS)0x40070025L)
75
76  /** Define for error if message is still sent from message place. */
77  #define CAN_DRV_E_TRA_FULL              ((NTSTATUS)0xC0070038L)
78  /*@}*/
79
80
81  /*****************************************************************************/
82  /*  Structures                                                               */
83  /*****************************************************************************/
84
85
86  /** Structure for extended frame types (29 Bit adress-range).
87  *
88  *   Set "frametype" to "STANDARD" or "EXTENDED".
89  *   The RTR (Remote-Frame) flag and data length code are now in seperate
90  *   variables for better hardware independence!
91  */
92  typedef struct _CAN_MESSAGE {
93      int     frametype;          /*!< STANDARD or EXTENDED. */
94      ULONG   id;                 /*!< Message Identifier (11 or 29 Bit). */
95      ULONG   rtr;                /*!< True: Remote Frame (ignore length). */
96      USHORT  length;             /*!< Message length (0-7). */
97      USHORT  place;              /*!< Message place. */
98      ULONG   timestamp;          /*!< (0 if not supported). */
99      ULONG   ErrorStatus;        /*!< != 0  -> this is an error frame. */
100     UCHAR   data[8];            /*!< Message data. */
101 } CAN_MESSAGE, *PCAN_MESSAGE;
102
103
104 typedef struct _CAN_STATUS {
105     int cbLen;      /*!< Length of structure in bytes */
106     BOOL bError;    /*!< Flag is set when error in hardware or bus. Details are device specific and can be found in data
            union. */
107     union {
108         struct {
109             ULONG   BusStatus;              /*!< Zero means bus is ok. */
110             ULONG   ControllerStatus;       /*!< SJA1000 status register. See SJA1000 manual for further details. */
111             BOOL    OverflowFlag;           /*!<  */
112             ULONG   ArbitrationLostCapture; /*!< SJA1000 arbitration lost (ALC) register. Contains the bit number
113                                                 where the arbitration was lost. Only the lower 5 bits are valid.
114                                                 See SJA1000 manual for further details.*/
115             ULONG   RX_ErrorCounter;        /*!< SJA1000 RX error counter register. See SJA1000 manual for further
                    details.*/
116             ULONG   TX_ErrorCounter;        /*!< SJA1000 TX error counter register. See SJA1000 manual for further
                    details.*/
117             ULONG   ErrorCode;              /*!< SJA1000 status register. This register contains information about
118                                                 the type and location of errors on the bus.
119                                                 See SJA1000 manual for further details. */
120         } SJA1000;  /*!< Philips SJA1000 and SJA1000T */
121         struct {
122             int DeviceError;
123             int ControllerStatus;
124             int TransmitError;
125             int ReceiveError;
126             int MonitorActiveFlag;
127             int DeviceActiveFlag;
128             int REC;
129             int TEC;
130             int TCEC;
131         } SAE81C9x;
132         struct {
133             BYTE    ucCanStatusReg;     /*!< CAN Status Register */
134             BYTE    ucCanOptionReg;     /*!< CAN Optional Register*/
135             BYTE    ucCanTECReg;        /*!< CAN Transmit Error Counter Register */
136             BYTE    ucCanRECReg;        /*!< CAN Receive Error Counter Register */
137             BYTE    ucCanControlReg;    /*!< CAN Control Register */
138         } OKI;
139     } data;
140 } CAN_STATUS, *PCAN_STATUS;
141
142
143 /*! \struct CAN_BITTIMING_STRUCT
144  * Struct for the Bittiming configuration.
145  */
146 typedef struct
147 {
148     WORD    wBaudratePreScaler;         /*!< The Baud Rate Prescaler can be extended to values up to 1023 */
149     BYTE    ucSynchronisationJumpWidth; /*!< 0..3 Time Units are possible */
150     BYTE    ucTSeg1;                    /*!< The time segment before the sample point 0..15 Time Units are possible  */
151     BYTE    ucTSeg2;                    /*!< The time segment after the sample point 0..7 Time Units are possible */
152     BOOL    bSampleTriple;          /*!< The time segment after the sample point 0..7 Time Units are possible */
153 }CAN_BITTIMING_STRUCT, *PCAN_BITTIMING_STRUCT;
154
155
```

```
156 /**
157  * @brief   Struct for the Message Queue.
158  */
159 typedef struct
160 {
161     DWORD           dwMessageType;      /*!< Meassage Type */
162     CAN_MESSAGE     Msg;                /*!< Can Message */
163 }MESSAGE_QUEUE_STRUCT, *PMESSAGE_QUEUE_STRUCT;
164
165
166
167 /*********************************************************************/
168 /*  Functions                                                        */
169 /*********************************************************************/
170 #ifdef __cplusplus
171   extern "C" {
172 #endif
173
174 /** Prototypes of functions exported by the CAN API for Windows CE. */
175 /*@{*/
176
177
178
179 /**
180  * @brief   Creates a new instance of the CANPort controller structure
181  *          in the library and opens the port of the driver.
182  *          This function must be executed before any other function is used.
183  *          Its return value has to be passed as first parameter to all following functions.
184  *
185  * @param   szDeviceName    Name of the CAN port that should be opened.
186  *                          The name consist of "CAN" and a consecutive number from 1 to 9.
187  *                          <br>For example "CAN1:" for the first CAN port of the system.
188  *
189  * @return  If successful, a handle of the new created CAN device will be returned.
190  *          This value is passed as first parameter to all other functions and must not be
191  *          changed by the application. If errors occur, INVALID_HANDLE_VALUE will be returned
192  *          and informations about the error are available in the following call of GetLastError().
193  *
194  * @remarks The driver may not support multiple access. When tried to open a second time, the
195  *          error code ERROR_SHARING_VIOLATION may occur.
196 */
197 GFCAN_API HANDLE _stdcall CanCreateDevice(DWORD dwInstanceNumber);
198
199
200
201 /**
202  * @brief   Closes the device opened by the CanCreateDevice() function
203  *
204  * @param   hDevice Handle of the CAN device, that is created in the library.
205  *                  Will be returned via CanCreateDevice().
206  */
207 GFCAN_API BOOL   _stdcall CanCloseDevice(HANDLE hDevice);
208
209
210
211 /**
212  * @brief   Reads the baudrate of the selected CAN interface
213  *
214  * @param   hDevice Handle of the CAN device, that is created in the library. Will be returned via CanCreateDevice().
215  *
216  * @param   pBaud   Address of an integer variable. The driver assigns the current used Baudrate to this variable.
217  *              The value is the Baudrate in kBaud. CanSetBaudrate() sets a new Baudrate.
218  *
219  * @return  If successful, the returned value is TRUE. If errors have occurred, FALSE will be returned.
220  *          Informations about the error can be retrieved by a following call of GetLastError().
221  */
222 GFCAN_API BOOL   _stdcall CanGetBaudrate(HANDLE hDevice, OUT int *pBaud);
223
224
225
226 /**
227  * @brief   Reads all driver-internal status- and error-variables into an integer array
228  *
229  * @param   hDevice     Handle of the CAN device, that is created in the library.
230  *                      Will be returned via CanCreateDevice().
231  *
232  * @param   pStatus
233  *
234  * @return  If successful, the returned value is non-zero.
235  *          If errors have occurred, zero will be returned.
236  *          Informations about the error can be retrieved by a following call of GetLastError().
237  */
238 GFCAN_API BOOL   _stdcall CanGetStatus(HANDLE hDevice, OUT PCAN_STATUS pStatus);
239
240
```

```
241
242 /**
243  * @brief   Reads the value of a set ID-filter
244  *
245  * @param   hDevice     Handle of the CAN device, that is created in the library.
246  *                      Will be returned via CanCreateDevice().
247  *
248  * @param   pAcceptanceCode     Address of a 32-bit variable, where the desired descriptor is stored into.
249  *
250  * @param   pAcceptanceMask     Address of a 32-bit variable, where the desired mask is stored into.
251  *
252  * @return  If successful, the returned value is non-zero.
253  *          If errors have occurred, zero will be returned.
254  *          Informations about the error can be retrieved by a following call of GetLastError().
255  */
256 GFCAN_API BOOL   _stdcall CanGetAddressFilter(HANDLE hDevice, OUT ULONG *pAcceptanceCode, OUT ULONG *pAcceptanceMask);
257
258
259
260 /**
261  * @brief   Reads a register of the CAN controller
262  *
263  * @param   hDevice     Handle of the CAN device, that is created in the library.
264  *                      Will be returned via CanCreateDevice().
265  *
266  * @param   ucRegAddr       Register address. See controller manual for register addressing
267  *
268  * @param   pucRegVal       Address of an integer variable. Here the driver inscribes the value
269  *                      read from the controller register.
270  *
271  * @return  If successful, the returned value is non-zero.
272  *          If errors have occurred, zero will be returned.
273  *          Informations about the error can be retrieved by a following call of GetLastError().
274  *
275  * @remarks This function is designed only for testing purposes.
276  *          The register-programming of the CAN-controller is not included in this documentation.
277  *          Therefore the manual of the CAN-controller can be considered.
278  *          During normal operations, all accesses to registers are executed by driver and library.
279  */
280 GFCAN_API BOOL   _stdcall CanReadRegister(HANDLE hDevice, BYTE ucRegAddr , OUT BYTE* pucRegVal);
281
282
283
284 /**
285  * @brief   Resets the CAN-controller and the driver to the default settings.
286  *          The last used baudrate is kept and all buffers are flushed.
287  *
288  * @param   hDevice     Handle of the CAN device, that is created in the library.
289  *                      Will be returned via CanCreateDevice().
290  *
291  * @return  If successful, the returned value is non-zero.
292  *          If errors have occurred, zero will be returned.
293  *          Informations about the error can be retrieved by a following call of GetLastError().
294  *
295  * @remarks CanReset is executed during start of the driver and after CanSetBaudrate().
296  *          In user applications this function is required to reset drivers and bus to a
297  *          defined state after exceptions or other profoundly errors.
298  *          All buffers are flushed and messages in buffers get lost.
299  *          The last used baudrate and monitoring mode are untouched by this reset.
300  *          After booting of the system a baudrate of 125kbaud is set and monitoring mode is activated.
301  */
302 GFCAN_API BOOL   _stdcall CanReset(HANDLE hDevice);
303
304
305
306 /**
307  * @brief   Sets a new baudrate in the controller
308  *
309  * @param   hDevice Handle of the CAN device, that is created in the library.
310  *                  Will be returned via CanCreateDevice().
311  *
312  * @param   baud        The new baudrate in kbit / sec.
313  *              Valid baudrates are 10, 20, 50, 100, 125, 250, 500 and 1000kBaud.
314  *
315  * @return  If successful, the returned value is non-zero.
316  *          If errors have occurred, zero will be returned.
317  *          Informations about the error can be retrieved by a following call of GetLastError().
318  */
319 GFCAN_API BOOL   _stdcall CanSetBaudrate(HANDLE hDevice, int baud);
320
321
322
323
324 /**
325  * @brief   Transmits a message on the CAN-bus
```

```
326  *
327  * @param   hDevice     Handle of the CAN device, that is created in the library.
328  *                      Will be returned via CanCreateDevice().
329  *
330  * @param   msg         Pointer to a CAN_MESSAGE-structure.
331  *                      Into this structure the currently oldest message is read out of the driver.
332  *
333  * @return  If successful, the returned value is non-zero.
334  *          If errors have occurred, zero will be returned.
335  *          Informations about the error can be retrieved by a following call of GetLastError().
336  *
337  */
338 GFCAN_API BOOL   _stdcall CanTransmitMessage(HANDLE hDevice, IN PCAN_MESSAGE msg );
339
340
341
342 /**
343  * @brief   Wait for a messages and read it out of the message queue.
344  *
345  * @param   hDevice     Handle of the CAN device, that is created in the library.
346  *                      Will be returned via CanCreateDevice().
347  *
348  * @param   pMsgStruct  PMESSAGE_QUEUE_STRUCT
349  *
350  * @param   dwTimeout       BlockingTime in ms, use INFINITE for waiting for a new message forever.
351  *
352  * @return  If successful, the returned value is non-zero.
353  *          If errors have occurred, zero will be returned.
354  *          Informations about the error can be retrieved by a following call of CanGetLastError().
355  *
356  */
357 GFCAN_API BOOL   _stdcall CanReceiveMessage(HANDLE hDevice, PMESSAGE_QUEUE_STRUCT pMsgStruct, DWORD dwTimeout);
358
359
360
361 /**
362  * @brief   Placeholder in a userdefined callback-function for event-controlled message receiving
363  *
364  * @param   hDevice     Handle of the CAN device, that is created in the library.
365  *                      Will be returned via CanCreateDevice().
366  *
367  * @param   place       Number of the messageplace that should be read. Numbers are counted from 0 to 15.
368  *
369  * @param   descr       Address of a 32-bit variable, where the desired descriptor is stored into.
370  *
371  * @param   mask        Address of a 32-bit variable, where the desired mask is stored into.
372  *
373  * @return  If successful, the returned value is non-zero.
374  *          If errors have occurred, zero will be returned.
375  *          Informations about the error can be retrieved by a following call of GetLastError().
376  *
377  */
378 GFCAN_API BOOL   _stdcall CanSetAddressFilter(HANDLE hDevice, ULONG acceptanceCode, ULONG acceptanceMask);
379
380
381
382 /**
383  * @brief   Determines the baudrate of the driver and returns it as kbaud-value
384  *
385  * @param   hDevice     Handle of the CAN device, that is created in the library.
386  *                      Will be returned via CanCreateDevice().
387  *
388  * @param   ucRegAdr        Address of the register that is to be new set.
389  *                      The addressing of registers is described in the manual of the controller.
390  *                      The advises there are to be considered.
391  *
392  * @param   ucRegVal        Value to be written into the register. Considered is only the LSB of this integer value.
393  *
394  * @return  If successful, the returned value is non-zero.
395  *          If errors have occurred, zero will be returned.
396  *          Informations about the error can be retrieved by a following call of GetLastError().
397  *
398  * @remarks This function is designed for testing purposes only and should be used with great care,
399  *          because wrong write accesses to the controller register may harm the function of the driver.
400  *          <br>
401  *          In any case, before manipulating single registers, the manual of the controller has to be considered in
       detail,
402  *          to understand the register-model of the controller and its functionality.
403  */
404 GFCAN_API BOOL   _stdcall CanWriteRegister(HANDLE hDevice, BYTE ucRegAdr, BYTE ucRegVal);
405
406
407
408 /**
409  * @brief   Defines the individual bit-timing on the CAN-bus.
```

```
410  *
411  * @param   hDevice         Handle of the CAN device, that is created in the library.
412  *                          Will be returned via CanCreateDevice().
413  *
414  * @param   pBitTiming      Bittiming struct
415  *
416  * @return  If successful, the returned value is non-zero.
417  *          If errors have occurred, zero will be returned.
418  *          Informations about the error can be retrieved by a following call of GetLastError().
419  *
420  * @remarks For additional informations about the single meanings of the values, see the CAN-controllers manual.
421  *          <br>
422  *          Settings of the bit-timing are persistent and kept over on a reset.
423  *          Changes take effect by calling this function with different values or calling the function CanSetBaudrate().
424  */
425 GFCAN_API BOOL   _stdcall CanSetBitTiming(HANDLE hDevice, IN PCAN_BITTIMING_STRUCT pBitTiming);
426
427
428
429 /**
430  * @brief   Reads the bus timing registers of the CAN controller
431  *
432  * @param   hDevice         Handle of the CAN device, that is created in the library.
433  *                          Will be returned via CanCreateDevice().
434  *
435  * @param   pBitTiming      Bittiming struct
436  *
437  * @return  If successful, the returned value is non-zero.
438  *          If errors have occurred, zero will be returned.
439  *          Informations about the error can be retrieved by a following call of GetLastError().
440  *
441  * @remarks The meaning of the registers values is illustrated in the controllers manual.
442  */
443 GFCAN_API BOOL   _stdcall CanGetBitTiming(HANDLE hDevice, OUT PCAN_BITTIMING_STRUCT pBitTiming);
444
445 /*@}*/
446 #ifdef __cplusplus
447   } //extern "C"
448 #endif
449
450 #endif
```

C ComLib

Listing C.1: ComLib: Adapter zur Benutzung in TestStand für die Ansteuerung von Kommandozeilenartigen Interfaces über RS-232 wie RedBoot

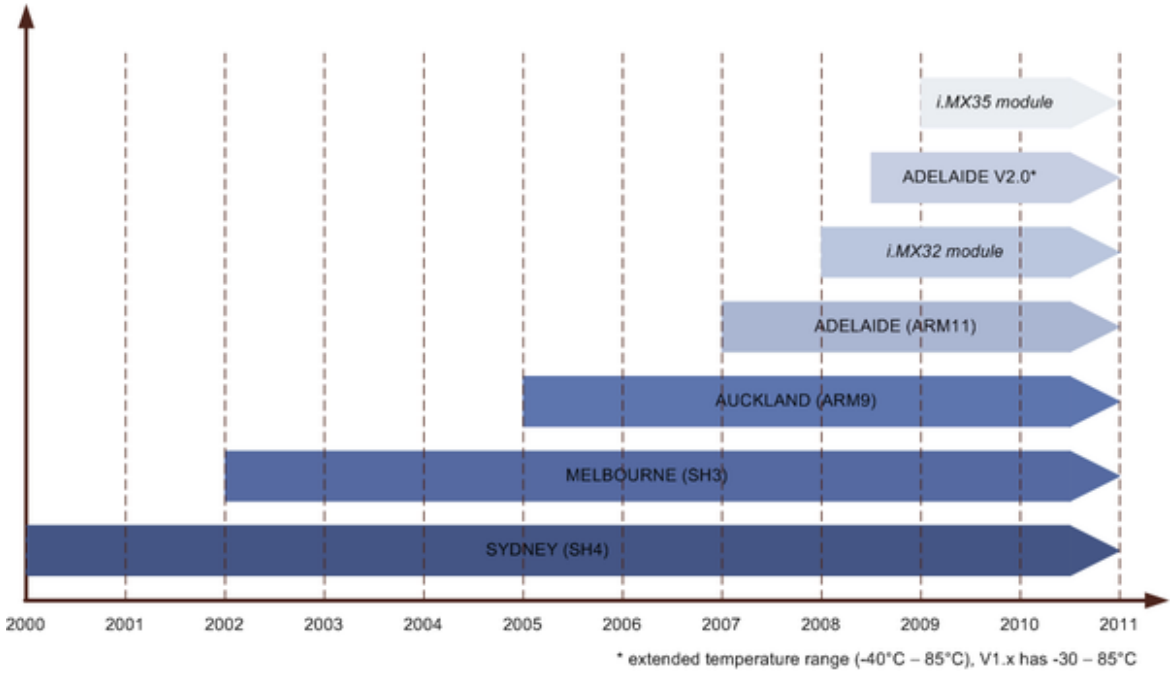
```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO.Ports;
using System.IO;
using System.Text.RegularExpressions;

namespace comlib
{
    public class Com : IDisposable
    {
        public Com()
        {

        }

        SerialPort serialPort = new SerialPort();
        StringBuilder globalBuffer = new StringBuilder();
        StringBuilder tempBuffer;
        TextWriter debugFile;

        public bool Open(string port, int baudrate, string debugFileName)
        {
            if(debugFileName != String.Empty)
            {
                debugFile = new StreamWriter(debugFileName);
                debugFile.WriteLine();
                debugFile.WriteLine("-- open port " + port + ", " + DateTime.Now.ToString());
                debugFile.WriteLine();
                debugFile.Flush();
            }

            serialPort.PortName = port;
            serialPort.BaudRate = baudrate;
            serialPort.Open();
            return true;
        }

        public void Close()
        {
            serialPort.Close();
            if (debugFile != null)
            {
                debugFile.WriteLine(Environment.NewLine + "-- close port");
                debugFile.Close();
            }
        }

        public void Dispose()
        {
            serialPort.Dispose();
            debugFile.Dispose();
        }

        public void Write(string s)
        {
            serialPort.Write(s);
            if (debugFile != null)
            {
                debugFile.WriteLine(Environment.NewLine + "-- WRITE:");
                debugFile.Write(s);
                debugFile.Flush();
            }
        }

        public void WriteLine(string s)
        {
            serialPort.WriteLine(s);
            if (debugFile != null)
            {
                debugFile.WriteLine(Environment.NewLine + "-- WRITE:");
```

```
72                  debugFile.WriteLine(s);
73                  debugFile.Flush();
74              }
75          }
76
77          public void BackspaceAndWriteLine(int backcount, string s)
78          {
79              byte[] bs = new byte[]{ 0x08 };
80              for(int i=0; i<backcount; i++)
81                  serialPort.Write(bs, 0, 1);
82              WriteLine(s);
83          }
84
85          private bool breakRedBootScript;
86          public bool BreakRedBootScript
87          {
88              get { return breakRedBootScript; }
89              set { breakRedBootScript = value; }
90          }
91
92          public bool WaitForString(string tofind, int timeout)
93          {
94              serialPort.ReadTimeout = timeout;
95              try
96              {
97                  tempBuffer = new StringBuilder();
98                  byte[] buff = new byte[512];
99                  bool sendBreak = breakRedBootScript;
100
101                 while(true)
102                 {
103                     int len = serialPort.Read(buff, 0, buff.Length);
104                     string buff_s = Encoding.ASCII.GetString(buff, 0, len);
105                     tempBuffer.Append(buff_s);
106                     globalBuffer.Append(buff_s);
107
108                     if (debugFile != null)
109                     {
110                         debugFile.Write(buff_s);
111                         debugFile.Flush();
112                     }
113                     //== Executing boot script in 1.000 seconds - enter ^C to abort
114                     string s = tempBuffer.ToString();
115                     if (sendBreak && s.Contains("== Executing boot script"))
116                     {
117                         Write("\x03");
118                         sendBreak = false;
119                     }
120                     if (s.Contains(tofind))
121                     {
122                         return true;
123                     }
124                 }
125             }
126             catch (TimeoutException)
127             {
128                 return false;
129             }
130         }
131
132         public bool WriteLineAndWaitForString(string toWrite, string toWait, int timeout)
133         {
134             WriteLine(toWrite);
135             return WaitForString(toWait, timeout);
136         }
137
138         public string GetStringFromBuffer(string regexPreString, string regexValue, bool ignoreCase, bool rightToLeft,
                int occurenceNumber)
139         {
140             string s = globalBuffer.ToString();
141             RegexOptions ro1 = RegexOptions.None;
142             RegexOptions ro2 = RegexOptions.None;
143             int startat = 0;
144             if (ignoreCase)
145             {
146                 ro1 |= RegexOptions.IgnoreCase;
147                 ro2 |= RegexOptions.IgnoreCase;
148             }
149             if (rightToLeft)
150             {
151                 ro1 |= RegexOptions.RightToLeft;
152                 startat = s.Length;
153             }
154             Regex r1 = new Regex(regexPreString, ro1);
155             Regex r2 = new Regex(regexValue, ro2);
```

```
156             Match m1 = null;
157             Match m2 = null;
158             for (int i = 0; i < occurenceNumber; i++)
159             {
160                 m1 = r1.Match(s, startat);
161                 m2 = r2.Match(s, m1.Index + m1.Length);
162                 startat += m1.Index + m1.Length;
163             }
164             return m2.Value;
165         }
166 
167         public uint GetUIntFromBuffer(string regexPreString, string regexValue, bool ignoreCase, bool rightToLeft, int
                occurenceNumber)
168         {
169             string s = GetStringFromBuffer(regexPreString, regexValue, ignoreCase, rightToLeft, occurenceNumber);
170             return uint.Parse(s);
171         }
172 
173         public bool IsStringInBuffer(string regex, bool ignoreCase)
174         {
175             string s = globalBuffer.ToString();
176             RegexOptions ro1 = RegexOptions.None;
177             int startat = 0;
178             if (ignoreCase)
179             {
180                 ro1 |= RegexOptions.IgnoreCase;
181             }
182             Regex r1 = new Regex(regex, ro1);
183             return r1.Match(s).Success;
184         }
185     }
186 }
```

D Roadmap

Die Firma Garz & Fricke entwickelt, produziert und vertreibt seit 2000 32-Bit-Systeme in Form von Computermodulen. Die Baureihe „Downunder“ hat eine gemeinsame Bauform im Schekkarten-Format und eine, soweit möglich, pinkompatible Steckerbelegung. Ein Beispiel eines Computermoduls aus dieser Baureihe ist in Abbildung 2.1 gezeigt. Auf dieser Seite ist der zeitliche Zusammenhang der „Downunder“-Modelle dargestellt. Quelle: Garz & Fricke Marketingmaterial.



Glossar

Automatische Optische Inspektion (AOI)

Eine oder mehrere zu prüfende Leiterkarten werden mit einem beweglichen Kamerasystem abgefahren. Mithilfe von Bildverarbeitungsverfahren wird versucht zu erkennen, ob Bauteile richtig bestückt wurden, Kurzschlüsse vorhanden sind und die Qualität der Lötstelle ausreichend ist. 22, 84

Application Programming Interface (API)

Schnittstelle einer Softwarekomponente zur Anbindung an eine weitere Softwarekomponente. 18, 22, 47, 51, 53, 59

Assert

Bezeichnung einer Softwarefunktion, die für Debug- oder Testzwecke bestimmte Programmbedingen und -zustände überprüft. 9, 64, 120

Binary Input Output System (BIOS)

Ein proprietäres Ladeprogramm zur ersten Initialisierung eines x86 Computersystems, welches auch das Betriebssystem lädt und dieses startet. 19

Bootloader

Ein Ladeprogramm zur ersten Initialisierung eines Computersystems und das Laden und Starten des Betriebssystems. 19, 88, 90, 120

Controller Area Network (CAN)

Ein Feldbus, der von der Robert Bosch GmbH entwickelt wurde und bei dem die Datenübertragung asynchron und seriell erfolgt. CAN wird hauptsächlich in Industrie und Automobilbereich eingesetzt. 16, 28, 40, 41, 47, 48, 86, 95

Windows CE Test Kit (CETK)

Test Framework und Sammlung von Testfällen für Standardkomponenten von Microsoft für Windows CE. 20, 22, 62, 63, 98, 99, 102

Complex Programmable Logic Device (CPLD)

Ein IC mit der Möglichkeit, die interne Logik aus UND- / ODER-Gattern und Speicherzellen immer wieder neu zu programmieren. 16, 19

Cyclic Redundancy check (CRC)

Algorithmus zur Erstellung einer Prüfsumme für die Erkennung von Datenfehlern. 41

Double Data Rate RAM (DDRRAM)

wie SDRAM, jedoch erfolgt die Taktung auf beiden Flanken. 16

Echtzeit

Eigenschaft eines Systems, mit der die Vorhersagbarkeit des Zeitverhaltens beschrieben wird. Das System muss dabei zuverlässig (immer) innerhalb einer vorher definierten Zeitspanne reagieren bzw. seine Funktion erfüllen. 40, 42, 45, 76, 102, 120

Flash-Speicher

Ein spezielles IC zur persisten (nicht-flüchtigen) Speicherung von Daten. Flash-Speicher werden aufgrund ihrer kleinen Bauform sehr häufig in Embedded-Systemen eingesetzt. 18, 21, 88, 93, 120

Field Programmable Gate Array (FPGA)

Ein IC mit der Möglichkeit, die internen Logikbausteine feinkörnig zu rekonfigurieren und dadurch komplexe Schaltungsentwürfe bis hin zu Designs von Mikrocontrollern zu realisieren. 16

File Transfer Protocol (FTP)

Protokoll zur Übertragung von Dateien. Es benutzt TCP als Transportschicht. 65, 88, 89, 94

Gateway

Eine Verbindung oder Brücke zwischen zwei oder mehreren meist verschiedenen Netzwerken. 120

Glue-Logic

Bezeichnung für eine Softwarekomponente, die als Vermittler oder Übersetzer zwischen Softwaresystemen dient. 90, 93, 120

General Purpose Instrumentation Bus (GPIB)

Ein paralleler Datenbus, der vorrangig zur Verbindung von Messgeräten und Peripheriegeräten, wie Plottern und Druckern, mit einem Computer eingesetzt wird. Andere Bezeichnungen sind IEC-625-Bus und IEEE-488. 96

Hardware Abstraction Layer (HAL)

Teile der Software, die zur direkten Ansteuerung der Hardware konzipiert sind. Beim Portieren eines System auf eine neue Hardware muss nur dieser Teil angepasst werden, die darauf aufbauende Logik bleibt erhalten. 19

Hypertext Transfer Protocol (HTTP)

Ein Protokoll zur Datenübertragung, basierend auf TCP. Es findet hauptsächlich im World Wide Web (www) zur Übertragung von Webseiten Verwendung. 88

In Circuit Test (ICT)

Ein Prüfverfahren elektronischer Baugruppen und bestückter Leiterplatten in der Elektronikfertigung. Der Prüfling wird mittels Nadeladaptern auf Bauteilefehler und Kurzschlüsse hin untersucht. 22, 84

Inter-Integrated Circuit (IIC)

IIC ist ein serieller Bus zur Datenkommunikation zwischen Elektronikkomponenten. Andere geläufige Namen sind I2C, I^2C oder TWI (Two Wire Interface). 15, 86

Industry Standard Architecture (ISA)

Ein früher weit verbreiteter Busstandard zur Anbindung von Peripheriegeräten in einem IBM-kompatiblem PC. 48

Joint Test Action Group (JTAG)

JTAG ist eine Schnittstelle (IEEE-Standard 1149.1), die viele ICs zur Verfügung stellen. Ursprünglich entwickelt zum Testen der inneren Logik von ICs, wird es heute u. a. auch für Debugging von Software verwendet oder Aufspielen der Firmware bei Erstinbetriebnahme. 88

Merge

Bezeichnet den Vorgang des Zusammenführens verschiedener Quelltextänderungen in eine Datei oder Entwicklungslinie. Dies ist die grundlegende Arbeitsweise von Versionsverwaltungen die differenzbasierend arbeiten, wie z. B. Subversion. 24, 120

Memory Management Unit (MMU)

Einheit zur Verwaltung von Speichertabellen in einer CPU. 18

Nadeladapter

Ein mit vielen Nadeln bestückter Adapter, der auf den Prüfling (PCB) aufgedrückt wird. Die Nadeln kontaktieren Bauteilanschlüsse und speziell zu diesem Zeck vorhandene Testpunkte. Eine Prüf- und Messelektronik kann darüber Punkte mit Spannung und Strömen stimulieren und zum Qualitätsnachweis messen (Siehe auch ICT). 22, 83, 120

OEM Adaption Layer (OAL)

Microsoft bezeichnet damit alle Änderungen, die ein OEM im Kernel von Windows CE vornimmt. Neben der HAL können dies auch herstellerspezifische Einstellungen oder besondere Implementierungen, die z. B. der Sicherheit des Gerätes dienen, sein. 19

Original Equipment Manufacturer (OEM)

Hersteller von Originalausrüstungen. In der Regel werden OEM-Produkte von einem anderen Unternehmen vertrieben. Dieses übernimmt dann Garantie, Gewährleistung und Support. 14, 19

Peripheral Component Interconnect (PCI)

Ein stark verbreitetes Bussystem zur Verbindung von Peripheriegeräten und Einsteckkarten mit dem CPU-System. 48

Personal Computer Memory Card International Association (PCMCIA)

Organisation, die Standards für Einsteckkarten für mobile Geräte entwickelt und publiziert. Wesentliche Eigenschaften sind Hot-Plug-Fähigkeit und Plug-and-Play durch die enthaltenen Konfigurationsparameter auf der Karte selbst. Der Name PCMCIA wird auch für den Standard selbst verwendet. 16

RS-232

Standard für eine serielle, asynchrone Schnittstelle (siehe auch UART). RS-232 spezifiziert die physikalischen Verbindungseigenschaften, wie Anschlussbelegung und Spannungspegel. 120

Real Time Clock (RTC)

Echtzeituhr. Baustein zur Speicherung und Ermittlung der Systemzeit. Eine Batterie- oder Kondensatorpufferung sorgt für Unabhängigkeit von der Hauptstromversorgung, so dass die Uhr ständig weiterlaufen kann. 17, 86

Round Trip Time (RTT)

Die Rundreisezeit gibt die benötigte Zeit eines Datenpaketes an, die es benötigt vom Sender zum Empfänger und wieder zurück zu gelangen. 75, 76

Single Board Computer (SBC)

Einplatinen-Computer beinhalten alle zum Betrieb benötigten Bauteile auf einer Leiterkarte. 15

Standard Commands for Programmable Instruments (SCPI)

Ein Protokoll zur Kommunikation mit Laborgeräten, wie Messgeräte, Netzteile, etc. 96

Synchronous Dynamic RAM (SDRAM)

Synchron getaktetes dynamisches RAM. Wird als Arbeitsspeicher verwendet. 15

System on a Chip (SoC)

Alle grundlegenden Teile eines Computersystems, wie Prozessor, Schnittstellen- und Peripherie-Controller, werden in ein IC integriert. Die Funktionseinheiten sind über einen internen Bus miteinander verbunden. In einigen Ausführungen sind auch Flash- und RAM-Speicher enthalten. Bei entsprechender Massenproduktion lässt sich eine hohe Kostenreduktion erreichen. 15, 16

System on Module (SoM)

Erweiterung des Begriffs SoC. Bezeichnet ein Computersystem als Modullösung. Das Modul liegt oft in Form einer Steckkarte vor, die alle hauptsächlichen Bauteile eines Computersystems enthält. Dazu gehören meist ein SoC mit CPU und Controllern, sowie Flash, RAM und Verbindungslogik. 15

Serial Peripheral Interface (SPI)

Schnelle, serielle und synchrone Datenübertragungsschnittstelle. Wird oft eingesetzt um Hardware-Komponenten auf einer oder mehreren Leiterkarten zu verbinden. 15, 86

Startup-Code

Die ersten grundlegenden Instruktionen eines Bootloaders zur Initialisierung der CPU und des Arbeitsspeichers. 19, 120

Standard Template Library (STL)

Erweiterungen der C++ Standard-Bibliothek. 62

Streaminterface

Eine generische Schnittstelle für Treiber in Windows CE. 18, 46, 47, 54, 120

Telnet

Telnet ist ein Protokoll, das die Bereitstellung einer ASCII basierten Umgebung in einem Terminal erlaubt. Neben der einfachen Übertragung der Kommandos und Rückmeldungen gibt es verschiedene Modi, die die Kommunikationspartner miteinander aushandeln können, um erweiterte Möglichkeiten zu schaffen. Die einfachste Protokollimplementierung ist VT100. In sicherheitsrelevanten Umgebungen ist Secure Shell (SSH) weit verbreitet. 65, 88, 90, 93, 94, 120

Trivial File Transfer Protocol (TFTP)

Sehr einfaches Protokoll zur Dateiübertragung. Es basiert auf UDP und kann einfacher implementiert werden als FTP. Wird oft verwendet, wenn nur eingeschränkte Ressourcen zur Verfügung stehen, wie z. B. in Bootloadern. 88, 92

Treiber

Eine Softwarekomponente, die die Kommunikation zu einem bestimmten Hardware-Baustein übernimmt. Sie ist Teil der HAL (aus Sicht des Betriebssystems) und erlaubt damit das Programmieren von hardware-unabhängigen Applikationen. 12, 18, 40, 46, 120

Universal Asynchronous Receiver Transmitter (UART)

UARTs werden zur Übertragung eines asynchronen, seriellen Datenstroms mit einem festem Rahmen eingesetzt. 15

Universal Serial Bus (USB)

Ein Bussystem zur Verbindung verschiedener Geräte, wie Speichermedien, Maus, Tastatur, Kamera, usw. an ein Computersystem. 15, 86

Vector Floating Point Unit (VFP)

Einheit zur Berechnung von Fließkommazahlen. Die Bezeichnung Vektor deutet auf Möglichkeiten zum Optimieren und Parallelisieren von Berechnungsoperationen hin. 16

Index

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Juni 2008

Ort, Datum

Unterschrift