Universität Stuttgart
Institut für Automatisierungs- und Softwaretechnik
Prof. Dr.-Ing. Dr. h. c. P. Göhner

15.12.2006

Studienarbeit

Thomas Ruschival 2105 SA

Untersuchung von Angriffs-möglichkeiten auf Feldgeräte von Automatisierungssystemen

Betreuer: **Prof. Dr.-Ing. Dr. h. c. Peter Göhner**
Prof. Dr.-Ing. Vicente Lucena
Dipl.-Ing. Felix Gutbrodt
Dipl.-Ing. Thomas Stiedl

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANSI	**A**merican **N**ational **S**tandards **I**nstitute
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
SPS	**S**peicher **P**rogrammierbare **S**teuerung
ROM	**R**ead **O**nly **M**emory
IT	**I**nformation **T**echnology
IC	**I**ntegrated **C**ircuit
OTP	**O**ne **T**ime **P**rogrammable
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
PROM	**P**rogrammable ROM
EPROM	**E**rasable **P**rogrammable ROM
EEPROM	**E**lectrically **E**rasable **P**rogrammable ROM
RAM	**R**andom **A**ccess **M**emory
SRAM	**S**tatic RAM
DRAM	**D**ynamic RAM
NV-RAM	**N**on **V**olatile RAM
ISP	**I**n **S**ystem **P**rogramming
SPI	**S**erial **P**rogramming **I**nterface
JTAG	**J**oint **T**est **A**ction **G**roup
MISO	**M**aster **I**n **S**lave **O**ut
MOSI	**M**aster **O**ut **S**lave **I**n
BSDL	**B**oundary **S**can **D**escrition **L**anguage
ELF	**E**xecutable and **L**inkable **F**ormat
MMU	**M**emory **M**anagement **U**nit
TLB	**T**ranslation **L**ookaside **B**uffer

ASLR	**A**ddress **S**pace **L**ayout **R**andomization
ITLB	**I**nstruction **T**ranslation **L**ookaside **B**uffer
DTLB	**D**ata **T**ranslation **L**ookaside **B**uffer
GNU	**G**nu is **N**ot **U**nix
GCC	**G**NU **C**ompiler **C**ollection

Begriffsverzeichnis

Pufferüberlauf oder Buffer-Overflow	Es werden mehr Daten als vom Programmierer vorgesehen in einen Speicherbereich geschrieben, das führt zum überschreiben anderer Daten
OPCodes	Elementare Prozessorbefehle aus denen Programe bestehen.
Master	Moderator einer Kommunikation. Instanz, die die Kommunikation steuert.
Slave	Kommunikationsteilnehmer, meist passiv, nimmt nur nach Aufforderung durch den Master an der Kommunikation aktiv teil.
Toolchain	Werkzeugkette, mehrere kleine Werkzeuge die im häufig in einer festen Reihenfolge hintereinander auf Daten Angewandt werden. Beispiel hierfür wäre: Präprozessor-Compiler-Linker
Compiler	Übersetzungsprogramm das Programme in höheren Programmiersprachen in Maschinencode übersetzt.
Präprozessor	Parser für Programmcode der Makros expandiert und Quellcode modifiziert, vor dieser kompiliert wird
Linker	Programm das Objectcodedateien miteinander und mit Bibliotheken zu einem Lauffähigen Programm verarbeitet.
Stack	Temporärer Speicher nach dem LIFO Prinzip der für Variablen und Argumente von Funktionen benutzt wird.
Heap	Dynamischer Speicher, der vom Programmierer zur Laufzeit verwaltet wird

Zusammenfassung

Die Verbreitung von Automatisierungstechnik hat in den letzten Jahren stark zugenommen. Hinzu kommt ein stetig wachsender Funktionsumfang und stärkere Vernetzung und Verteilung der Systeme. Die Verteilung verringert die Komplexität der Einzelkomponenten, der Feldgeräte, bringt aber im gleichen Zug neue Gefahren bezüglich der Angreifbarkeit des Gesamtsystems mit sich. Schon sehr früh wurde in der Automatisierungstechnik das Kriterium „Safety“, der Sicherung gegen Gefahren, die das System für die Umwelt darstellt, berücksichtigt. Nun muss auch der Gesichtspunkt „Security“, also auf welche Weise das System von außen bedroht ist, beim Entwurf beachtet werden. Deshalb werden in dieser Arbeit Security-Probleme von Software und Systemen erläutert. Danach wird untersucht, wie sich diese in möglichen Angriffen auf Feldgeräte von Automatisierungssystemen auswirken. Als mögliche Antwort auf diese Gefahren werden Technologien, die Hard- und Softwaresysteme sicher machen sollen vorgestellt. In einem abschließenden Teil wird die Installation und Konfiguration eines Embedded Systems mit Linux beschrieben. Anhand dieses Prototyps wurden die beschriebenen Angriffsmöglichkeiten auf Feldgeräte evaluiert.

Abstract

The use of automation systems has grown significantly within the last years. Additionally there has been a continuous growth range of functionality a stronger interconnection and distribution of systems. While the distribution helps to control the complexity of each component it also introduces new threats regarding the vulnerability of the whole system. Very early on the aspect of safety, the precautions against dangers the system could bring its environment, has been regarded in design. Now, also the aspect of safety, how the system is threatened from the outside, has to bee regarded in system engineering. Therefore this work will explain general security problems of software and systems. Then deduced how these problems result in possible attacks for product and industrial automation systems. As possible answer for these security threats a range of technologies both in hard- and software, which are supposed to make systems more secure, are presented in this paper. The last section contains instructions for installation and configuration of an embedded system based on Linux. With this prototype, the security problems, which have been described in this thesis, have been evaluated.

Einleitung

Das Ziel dieser Arbeit war es, die Gefährdung von Feldgeräten in Automatisierungstechnik zu untersuchen. Um die Sicherheitsprobleme und Angriffsvektoren zu erarbeiten, wurden generelle Probleme aus dem Bereich der Softwaresicherheit auf die Automatisierungstechnik übertragen.

Im ersten Teil werden die Grundlagen für das Verständnis erarbeitet. Dazu ist zunächst die Erklärung von Begriffen und dem Aufbau von Automatisierungssystemen aus dem ersten Kapitel wichtig. Kapitel zwei geht auf die verwendeten Speichertechnologien und Programmierschnittstellen ein, da diese für Angriffe eine entscheidende Rolle spielen. Kapitel drei beschäftigt sich damit, welche Softwaretechnologien in Automatisierungssystemen verwendet werden und wie Speicher verwaltet wird. Kapitel vier bietet einen kurzen Überblick über die in Automatisierungsprojekten verwendeten Programmiersprachen und ihre Sicherheitsprobleme.

Der zweite Teil der Arbeit beschäftigt sich mit möglichen Angriffen und den bestehenden Schutzmechanismen. In Kapitel fünf wird die Motivation von Angriffen auf verschiedene Systeme vorgestellt, Angreifer, Angriffe und Systeme werden klassifiziert. Kapitel sechs stellt detailliert den Ablauf von Angriffen mittels Pufferüberläufen und die dadurch entstehende Gefährdung von Systemen dar. Im siebten Kapitel liegt der Schwerpunkt auf möglichen Schutzmechanismen gegen den Diebstahl geistigen Eigentums, während das achte Kapitel Mechanismen zur Abwehr und Erkennung der vorgestellten Angriffe mittels Pufferüberläufen vorstellt.

Der dritte, praktische Teil der Arbeit beschäftigt sich mit der Evaluierung der identifizierten Gefährdungen anhand eines eingebetteten Feldgeräts sowie der Installation und Inbetriebnahme dieses Embedded Systems mit dem Open-Source-Betriebsystem GNU/Linux. Dieser Vorgang ist in Kapitel neun beschrieben.

1 Begriffe

Um das Problem von Angriffen auf Feldgeräten in Automatisierungssystemen besser erläutern zu können ist es wichtig, Begriffe zu erklären und vor allem klar gegeneinander abzugrenzen. In diesem Kapitel werden Begriffe, Automatisierungssysteme und ihr Aufbau beschrieben, um die Security-Probleme und Angriffsvektoren zu unterscheiden und verständlich zu machen.

1.1 Definition von Safety und Security

Wenn im Deutschen von „Sicherheit“ gesprochen wird so ist dies ein unscharf umrissener Begriff. Die genaueste Definition, die in der Technik relevant ist, wird in DIN EN ISO 14971 verwendet. Dort definiert Sicherheit „die Freiheit von nicht akzeptablen Risiken“.

Im Englischen gibt es für die Sicherheit zwei unterschiedliche Begriffe, Security und Safety. Mit Safety ist dabei die Verhinderung von Gefahren, die von einem Prozessautomatisierungssystem ausgehen, gemeint. Ist ein System „save“ bedeutet die, dass zu keinem Zeitpunkt Menschen, Umwelt oder andere Systeme gefährdet sind.

Hier ist im Fehlerfall immer ein „fail-safe“ Verhalten gefordert. Das System muss in einem sicheren Zustand verharren. Bei vielen Automatisierungssystemen ist oft ein absoluter Stillstand ein sicherer Zustand. Dies kann jedoch nicht verallgemeinert werden, nicht in jedem Fall bedeutet „fail-safe“ ein Stillstand des Systems. Zum Beispiel darf ein Flugzeuges während des Fluges wegen eines Fehlers nicht zu Stillstand kommen. Für ein Kraftfahrzeug, bei dem ein Fehler in der Steuerung der Bremsanlage erkannt wird, hingegen, ist der Stillstand ein sicherer Zustand.

Mit Security ist der Schutz gegen gezielte Manipulationen und die Verhinderung von Gefahren, die sich gegen die Funktionsfähigkeit des Prozessautomatisierungssystems richten gemeint. Es darf kein Dritter unbemerkt unautorisiert Programme oder Daten des Systems ändern oder lesen.

Mittels Sicherheitskonzepten werden die 5 Ziele verfolgt [FeGu05]:

- Vertraulichkeit/Diskretion: Schutz vor unbefugtem Lesen und Schreiben von Daten und Programmen.
- Integrität: Unberührtheit von Daten oder deren eindeutigen Ursprung feststellen.
- Verfügbarkeit: Die Funktion eines Dienstes aufrecht zu erhalten.
- Ergonomie: Die Sicherheitsmaßnahmen dürfen die Funktion des Systems nicht zu sehr einschränken.
- Nachweisbarkeit: Sollte ein Verstoß gegen die Sicherheit erfolgt sein, so sollte dieses auch bemerkt werden, dazu können Sicherheitsprotokolle analysiert werden.

Natürlich gehören beide Begriffe eng zusammen und sie beeinflussen sich gegenseitig. In der Automatisierungstechnik stand bisher aber „Safety“ im Vordergrund. Erst in jüngster Zeit entwickelt sich mit der Vernetzung von Systemen über das Internet und sich häufenden Fälle von Patentrechtsverstößen das Bewusstsein für die Bedeutung von „Security“ in Automatisierungssystemen unter Ingenieuren.

1.2 Automatisierungssysteme

Ein Automat ist ein selbsttätiges, arbeitendes, technisches System[LaGö1999]. Somit steht der Begriff Automatisierung ganz allgemein für Maschinen, Geräte oder Anlagen, die mehr oder weniger selbsttätig arbeiten. Notwendig oder wünschenswert wurden sie, um dem Menschen stupide und monotone oder sehr komplexe Arbeitsvorgänge abzunehmen.

Unter einem Automatisierungssystem versteht man nun die Summe aller notwendigen technischen Einrichtungen, die für die Automatisierung erforderlich sind. Des Weiteren ist ein Prozessautomatisierungssystem ein Automatisierungssystem, in dem ein technischer Prozess, also ein Vorgang der in einem technischen System abläuft und den Zustand von Materie, Energie oder Informationen ändert, abläuft.

1.2.1 Produktautomatisierung und Anlagenautomatisierung.

Es wird zwischen mehreren speziellen Prozessautomatisierungssystemen unterschieden, in dieser Arbeit ist vor allem die Unterscheidung von Produkt- und Anlagenautomatisierungssystemen von Bedeutung, da sich diese Arten durch unterschiedliche Angriffsmotive und Angriffsvektoren charakterisieren.

Bei so genannten Produktautomatisierungssystemen läuft der technische Prozess in einem Produkt (z.B.: Mobiltelefon, Waschmaschine) ab. Charakteristisch für Produktautomatisierungssysteme ist die hohe Stückzahl in der sie produziert werden und dass die Geräte selbst meist klein sind im Gegensatz zu den Anlagenautomatisierungssystemen.

Bei Anlagenautomatisierungssystemen wird der technische Prozess in einem großen Raum ausgeführt, man teilt den technischen Prozess auch in mehrere Teilprozesse auf. Anlagenautomatisierungssysteme sind meist Einzelstücke (z. B.: Kraftwerke, Raffinerien oder Gepäcktransportsysteme an Flughäfen).

1.2.2 Komponenten von Automatisierungssystemen

Ein Automatisierungssystem ist eine Einheit die mit der Außenwelt, dem technischen Prozess interagiert. Dazu ist die Erfassung von physikalischen Größen die den Prozess beeinflussen notwendig. Diese Aufgabe übernehmen Sensoren. Sensoren sind spezielle Geräte, die eine physikalische Größe erfassen und meist in eine elektrische Größe umwandeln.

Heutzutage sind Sensoren oft in komplexere Einheiten eingebettet, die die erfassten Daten aufbereiten und über ein Kommunikationssystem zum Empfänger weiterleiten.

Um den technischen Prozess zu beeinflussen, sind Aktoren erforderlich, die gemäß den Informationen aus der Steuerung eine physikalische Größe des Prozesses einstellen. Auch diese Steller sind meist dezentral und mit einer Kommunikationseinheit an einen Nachrichtenkanal angegliedert.

In komplexen verteilten Systemen ist auch die Steuerung nicht zentralisiert. Die Berechnung und Verarbeitung von Informationen wird in kleinere, mehr oder weniger autonome, abgekapselte Systeme unterteilt, die Vorort die Steuerung der physikalischen Größen übernehmen. Solche so genannte Feldgeräte sind über Kommunikationssysteme zur Verwaltung und wechselseitigen Benachrichtigung zum gesamten Automatisierungssystem zusammengefasst.

Prinzipiell lässt sich die Hierarchie von Automatisierungssystemen in drei Ebenen, nach Aufgaben und Rechenleistung gliedern:

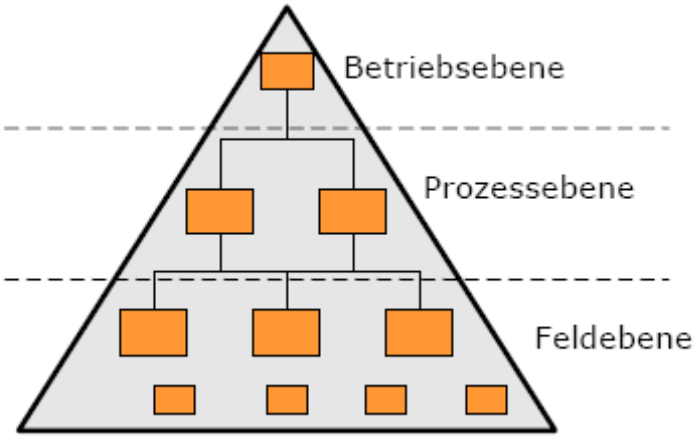

Abbildung 1: Schichten von Automatisierungssystemen [GöPA1]

Je nach Art des Automatisierungssystems und Komplexität werden unterschiedliche Techniken eingesetzt. Als Hardware können Mikrocontroller, SPS (Speicher Programmierbare Steuerungen), Industrie-PCs oder Prozessleitsysteme zum Einsatz kommen. Die Ausführungen der Feldgeräte unterscheiden sich sehr je nach Einsatzgebiet und Verwendungszweck. Prinzipiell sind alle Feldgeräte, egal ob "SPS", "Steuergerät" oder "Embedded System", aus einem Mikroprozessor, Speicher und Kommunikationsschnittstellen aufgebaut, und funktionieren prinzipiell gleich wie ein handelsübliches Personal Computer.

1.2.3 Mikroprozessoren

Wird von einem Mikroprozessor gesprochen ist damit die digitale Recheneinheit auf einem Chip mit Registern und eventuell Cache und Speichercontroller gemeint. Mikroprozessoren haben

heutzutage eine Wortbreite von 8 bis 64 Bit, in manchen speziellen Systemen sogar bis zu 128 Bit.

Die Prozessoren für PCs sollen möglichst alle unterschiedlichen Aufgaben, wie zum Beispiel Textverarbeitung, Spiele oder Multimediananwendungen möglichst gleichzeitig und gleichmäßig schnell abarbeiten können. Im Bereich der Feldgeräte und Automatisierungstechnik hingegen, lassen sich durch sinnvolle Auswahl der Prozessoren leicht Energie, Entwicklungszeit und auch Kosten des Endprodukts einsparen. Da ein Feldgerät genau für eine spezifische Aufgabe entwickelt wird ist es möglich einen Prozessor für genau diesen Zweck auszuwählen. So muss zum Beispiel ein Prozessor einer Regelung der Klimaanlage im Auto keine komplexen schnellen Fliesskommarechnungen anstellen, diese Arbeit kann von einem general-purpose-Mikroprozessor zusammen mit anderen Aufgaben übernommen werden. Hingegen ist für das Encoding eines MPEG-Streams mit der Kamera eines Mobiltelefons ein spezieller Prozessor nötig der das Encoding mit Hardwareeinheiten unterstützt. Ein general-purpose-Mikroprozessor, der MPEG-Encoding in Echtzeit abarbeiten könnte, wäre in Mobiltelefonen undenkbar, da ein solcher Chip zu teuer ist und viel zuviel Energie benötigt.

1.2.4 Embedded System.

Oft wird in der Technik von "Embedded System" gesprochen. Hiermit ist ein komplexeres Gerät mit Speicher, Mikroprozessor und Kommunikationsschnittstellen gemeint, das auf einer einzigen Platine integriert ist. Solche Geräte werden für den Massenmarkt produziert. Ein Embedded System zeichnet sich durch die Kosten begrenzten Ressourcen bezüglich Speicher und Rechenleistung aus. Wieder ist hier nicht die Eignung für beliebige Aufgaben im Vordergrund sondern die Erfüllung einer besonderen bestimmten Aufgabe. Das System muss seine Aufgabe zuverlässig über eine lange Zeit erfüllen. Die Software von Embedded Systems unterliegt sehr oft harten Echtzeitbedingungen und muss von hoher Qualität sein. Embedded Systems sind also Feldgeräte im weiteren Sinne. Die meisten erfordern keine oder nur minimale Benutzerinteraktion und werden oft vom Benutzer gar nicht bemerkt. Ein Beispiel hierfür wäre ein Steuergerät eines Kraftfahrzeuges, aber auch ein Mobiltelefon oder eine Videospielkonsole. Der Benutzer braucht von der Existenz und Software des Kraftfahrzeuges nichts zu wissen und interagiert damit nur indirekt. Mit einer Spielkonsole interagiert er direkt mit der Software aber das Betriebssystem bleibt ihm verborgen. Die hohe Zahl an ausgelieferten Produkten und die für den Endbenutzer unzugänglichen Programmierschnittstellen machen es außerordentlich schwierig und kostenintensiv Fehler in der Software durch Aktualisierung zu beheben.

1.2.5 Mikrocontroller

Mit zunehmender Integration der Halbleitertechnologie wurde es möglich die Größe von Embedded Systems stark zu reduzieren und immer mehr Funktionalität bei gleicher Leistung auf einem einzigen Chip zuintegrieren. Ein Mikrocontroller ist ein Computer auf einem Chip mit Programm-, Datenspeicher und Kommunikationsschnittstellen. Viele Mikrocontroller haben

selbst einen Oszillator, mehrere Analog-/Digitalwandler, Timer und Zähler auf dem Chip integriert. Mikrocontroller sind sehr flexibel in ihrem Einsatz.

Mikrocontroller werden in allen erdenklichen Geräten eingesetzt und werden in Massen produziert, was sich positiv auf die Kosten auswirkt. Einfache Chips sind schon für weniger als 1 Euro auf dem Markt erhältlich.

Es existiert eine unüberschaubare Menge an unterschiedlichen Controllern mit verschieden Vor- und Nachteilen. Allerdings lassen sich Mikrocontroller in Familien gleicher Architekturen gliedern, innerhalb der sich die Chips nur durch ihre Leistungsfähigkeit unterscheiden. Manche Hersteller lizenzieren für ihre Mikrokontrollerarchitekturen auch Prozessorkerne anderer Hersteller, zum Beispiel existieren Chips von Texas Instruments und Phillips die beide auf dem Prozessorkern der englischen Firma ARM basieren. Dies wiederum erleichtert den Toolsupport und die Portierung von Software. Die Entscheidung bei der Auswahl wird oft durch Bekanntheit unter den Entwicklern und den verfügbaren Entwicklungsumgebungen und Compilern bestimmt.

2 Hardware in Automatisierungssystemen

Während im vorangegangen Kapitel bereits Begriffe für Hardware erläutert wurden ist vielleicht der Eindruck entstanden es handle sich um andere Technologien als im klassichen Personal Computer. Die in der Automatisierungstechnik verwendeten Hardwarekomponenten unterscheiden sich in ihrer prinzipiellen Funktionsweise und Technologie nicht von Komponenten in Standardrechnern. Jedoch sind die verwendeten Bausteine viel mehr unter dem Gesichtspunkt der optimalen Lösunge einer speziellen Aufgabe ausgesucht. Es können somit kleinere und günstigere Bauteile verwendet werden. Dieses Kapitel geht tiefer auf die Komponenten Speicher und Kommunikation ein.

2.1 Programmspeichertechnologien.

Je nach Verwendungszweck werden in Feldgeräten unterschiedliche Speichertechnologien als Programmspeicher eingesetzt. Prinzipiell werden ROM (Read Only Memory) als Programmspeicher eingesetzt da diese Technologie die Daten auch ohne Spannungsversorgung dauerhaft speichert. ROM bedeutet allerdings nicht, dass der Chip nicht geschrieben werden kann, zum Schreiben der Daten ist ein verhältnismäßig hoher Zeit- und Energieaufwand nötig, so dass es während der Programmausführung nicht wahlfrei geschrieben und gelesen werden kann.

Oft sind auf einem Chip für verschiedene Speicher angeordnet. So wird zum Beispiel bei Atmel AtmegaXX Mikrocontroller Familie die Software in ein Flash-ROM geschrieben, dem Programmierer steht aber zur Speicherung von Daten die während der Programmausführung dauerhaft gespeichert werden sollen, ein EEPROM zur Verfügung.

Die prominentesten ROM-Technologien heutzutage sind (UV)EPROM, Mask ROM, EEPROM, FLASH und Static RAM.

	OTP EPROM	**Mask ROM**	**Flash EEPROM**	**UV EPROM**	**EEPROM**	**Static RAM**
Lesezeit	mittel ca. 50 ns	schnell ca. 5 ns	schnell ca. 20 ns	mittel ca. 50 ns	mittel ca. 50 ns	Schnell ca. 10 ns
Schreibzeit	langsam ca. 10 ms	-	mittel ca. 10 ms	langsam ca. 10 ms	langsam ca. 1 ms	Schnell ca. 10 ns
Datensicherheit	> 10 Jahre	-	> 100 Jahre	> 10 Jahre	> 40 Jahre	> 5 Jahre abhängig von Batterie
Schreibzyklen	1	-	10^4-10^6	100	10^3-10^6	-
Kosten	mittel	niedrig	niedrig	Hoch	Mittel	Hoch

Tabelle 1 : Speichertechnologien [SeSk05]

2.1.1 Statische Speicher

Generell kann man zwei unterschiedliche Klassen unterscheiden. Zum einem gibt es Chips, die sich nur ein einziges Mal schreiben lassen, so genannte OPT (One Time Programmable) Chips und andere die mehrmals beschrieben werden können. Eine weitere Unterscheidung lässt sich danach treffen, ob die Speicher elektronisch im System geschrieben und gelöscht werden können. Mask ROM, PROM und EPROM sind nicht für die Programmierung im System geeignet, diese Speicherbausteine müssen mit speziellen Programmiergeräten geschrieben und eventuell gelöscht werden. EEPROM, Static RAM und Flash EEPROM lassen sich elektronisch wiederbeschreiben.

Mask ROM ist die einfachste und billigste Alternative falls kein generischer Mikrocontroller eingesetzt werden soll und die Software nicht aktualisiert werden muss. Wird ein ASIC (Application Specific Integrated Circuit) für eine spezielle Anwendung entworfen, so kann ein Teil der Chipfläche als Speicher ausgelegt werden. Die Daten sind die Halbleiterstruktur selbst, das Programm wird also „fest verdrahtet“ in den Chip geschrieben. Der Nachteil, dass die Daten nicht geändert werden können, ist allerdings so gravierend, dass diese Technik für große Programme ungeeignet ist. Mask ROM Speicher werden beispielsweise in RFID Chips eingesetzt.

PROM (Programmable ROM) können nur ein einziges Mal programmiert werden, soll die Software ausgetauscht werden, muss der PROM Chip ausgetauscht werden. Ein PROM ist ein Spezialfall einer programmierbaren logischen Anordnung. Prinzipiell nichts anderes als ein Array von Dioden, so genannte "Fuses". Beim Programmiervorgang werden mit einer erhöhten Spannung gezielt einzelnen Fuses durchgebrannt, mit anderen Worten zerstört. Daraus ergibt sich, wie im Falle einer Maske, eine Struktur aus leitenden und nicht leitenden Stellen, die ein Bitmuster darstellen. Das PROM ist nach einem fehlerhaften Schreibvorgang zerstört. Die Halbleitertechnik ist aber heutzutage weiter ausgereift und diese ursprüngliche Technologie wird kaum mehr verwendet, sie wurde praktisch von UV-EPROMS abgelöst.

EPROM (Erasable Programmable ROM) ist eine weitere nichtflüchtige Speichertechnologie. Die Halbleiterstuktur besteht aus einer Anordnung besonderer MOSFETs, bei denen die Gateelektrode isoliert ist. Jeder Transistor kann ein Bit speichern. Für das Schreiben eines EPROMS ist wiederum eine erhöhte Spannung nötig. Die Schreibspannung wird über längere Zeit an das isolierte Gate angelegt, durch den Tunneleffekt wird Ladung auf das Gate gebracht, das danach leitend bleibt, da es sich nicht mehr entladen kann. Die Lesespannung ist deutlich unterhalb der Schreibspannung und beeinflusst die Ladung auf dem Gate nicht. Die Daten können nun beliebig oft ausgelesen werden.

Das Gehäuse des Chips hat ein Fenster aus Glas. Zum Löschen der EPROMS ist UV-Licht nötig. Durch die Energiereiche Strahlung wird die Halbleiterstruktur ionisiert und das Floating Gate wird entladen. Der Löschvorgang ist sehr langwierig, mit konstantem UV-Licht dauert es 10 bis 30 Minuten bis das ROM gelöscht ist. Der Schreib-Löschvorgang kann ca. 100-200 Mal wiederholt werden. Soll das ROM nicht gelöscht werden muss das Fenster abgeklebt sein, da

sonst das ROM durch das Tageslicht innerhalb von 3 Monaten gelöscht wird. Der langwierige Löschvorgang verursacht verhältnismäßig hohe Kosten und macht das EPROM für die Massenfertigung ungeeignet, EPROMS werden mehr in Anlagenautomatisierung eingesetzt, wo nur bei wenigen Geräten die Software aktualisiert werden muss.

EEPROM (Electrically Erasable Programmable ROM), auch E²PROM geschrieben, ist eine Weiterentwicklung des EPROM die elektronisch gelöscht werden kann. Prinzipiell ist die Technik die gleich wie bei UV-EPROMs, Die Daten werden wiederum in eine Speichermatrix aus Feldeffektransistoren gespeichert. Zum Schreiben ist auch hier eine erhöhte Spannung nötig, die die isolierten Gateelektroden lädt. Der Speicher wird immer Byteweise beschrieben. Die Transistormatrix kann durch Pulse höherer Spannung aber auch wieder gelöscht werden. Die Spannung erzeugt ein elektrisches Feld das die Gateelektroden wieder entlädt. Der Vorgang ist deutlich schneller als das Löschen mit UV-Licht und es muss nicht der ganze Chip gelöscht werden, es können gezielt einzelne Bytes beschreiben und gelöscht werden. Ein weiterer Vorteil ist die wesentlich höhere Lebensdauer von ca. 100.000 Schreibzyklen.

Die nötige Programmier- und Löschspannung wird durch Halbleiterschaltungen direkt auf dem Chip erzeugt, so dass keine Programmiergeräte oder weitere Versorgungsspannung nötig sind und der Speicher direkt auf dem Mikrocontroller selbst integriert werden kann. Dennoch sind EEPROMS verhältnismäßig langsam und eignen sich nur zur Speicherung von Daten, die nicht sehr oft und schnell geändert werden müssen. EEPROMS werden hauptsächlich eingesetzt, um Konfigurationen oder Daten des technischen Prozesses dauerhaft zu speichern.

Flash-EEPROM ist ein Speicher mit hoher Speicherdichte, die wesentlich schneller große Mengen von Daten speichern. Der Speichervorgang lädt durch quantenmechanische Effekte (Fowler-Nordheim-Tunneleffekt) wiederum ein isoliertes Gate der FET-Matrix. Beim Schreiben werden allerdings immer ganze Blöcke von Daten auf einmal geschrieben. Der Löschvorgang funktioniert auch über eine erhöhte Spannung und löscht ganze Sektoren der Speicherstruktur auf einmal. Soll also ein Wort im Speicher geändert werden muss zunächst der gesamte Block gelesen werden, das entsprechende Wort kann im RAM geändert werden, danach muss das Flash gelöscht werden und der Block kann als Gesamtes wieder geschrieben werden. Diese Eigenschaft macht das Flash EEPROM nur für Programmspeicher oder Speicher für Daten, die Blockweise verarbeitet werden sollen interessant.

SRAM (Static RAM) ist ein anderer Ansatz zum Speichern von Daten. Hierbei handelt es sich um flüchtigen Speicher der eine Betriebsspannung zum Erhalt der Daten benötigt. Im Gegensatz zu dynamischen RAM (DRAM) wird aber keine Energie für Refreshzyklen benötigt. SRAM ist sehr schnell und sehr kurze Zugriffszeiten. Es eignet sich aber nur für Feldgeräte mit eingebauter Batteriespannungsversorgung die Stromausfälle puffern kann.

NV-RAM (Non Volatile RAM) ist ein nicht flüchtiger Speicher der die Vorteile von RAM, Geschwindigkeit und Zugriffszeit mit dem Vorteil von EEPROM, Energieunabhängigkeit vereint. Ein NVRAM Zelle besteht aus einem EEPROM mit SRAM kombiniert mit einer Ansteuerlogik. Beim Start werden die Daten aus dem EEPROM ins SRAM geschrieben. Fällt die Versorgungsspannung aus bleiben die Daten permanent im EEPROM gespeichert. Durch die

hohe Komplexität sind die Speicherbausteine sehr teuer und haben eine geringe Kapazität pro Fläche. Sie werden daher nur für geringe Datenmengen eingesetzt.

2.2 Hardwareschnittstellen

Zur Programmierung von Mikrocontrollern bestehen neben der Möglichkeit ROMs extern zu beschreiben auch Techniken die Programme im System selbst zu debuggen und zu aktualisieren. Für das so genannte ISP (In System Programming) gibt es neben proprietären herstellerspezifischen Schnittstellen auch weitgehend standardisierte. Die bekanntesten Schnittstellen zum Programmupload sind SPI (Serial Peripheral Interface) und Standard Test Access Port and Boundary-Scan Architecture IEEE 1149.1, das von der JTAG (Joint Test Action Group) entworfen wurde, daher ist die allgemeine Abkürzung für das Interface JTAG.

2.2.1 SPI – Serial Peripheral Interface

SPI ist ein einfaches Interface nach einem Standard von Motorola, das zur Kommunikation zwischen beliebigen digitalen Bausteinen entworfen wurde. Das Interface eignet sich für einfache Kommunikation und arbeitet im Full-Duplex-Mode, das heißt es werden gleichzeitig Daten empfangen und gesendet. Jedoch ist kein hardwareseitiges Kommunikationsprotokoll für diesen Bus spezifiziert, es wird keine Prüfsumme zur Verifikation der Daten erzeugt, ferner existiert keine Flusskontrolle des Datenstroms [DaKa2002]. Wird über SPI kommuniziert so müssen die Daten mit Softwareprotokollen kontrolliert werden. Die einfachste Weise ist das das gesendete Byte vom Slave wieder an den Master zurückgesendet wird, der es gegen das Original prüft.

Zur Kommunikation sind lediglich 3 Leitungen erforderlich: MISO (Master In Slave Out), MOSI (Master Out Slave In) und die Taktleitung SCK (Serial Clock), die den Takt zur seriellen Kommunikation vorgibt, dieser Takt wird immer vom Master vorgegeben. Abbildung 2 ist eine schematische Darstellung einer SPI Kommunikation mit nur einem Slave.

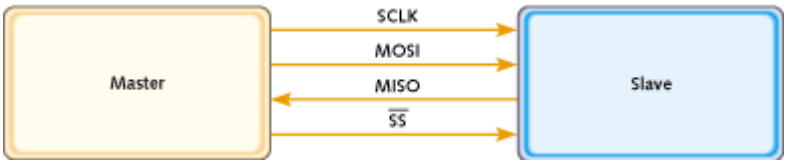

Abbildung 2: SPI Kommunikation Master/Slave [DaKa2002]

SPI wird von vielen Chipherstellern aber auch als Möglichkeit zum Lesen und Schreiben auf die, im Controller eingebauten, Speicher bereitgestellt. SPI ist einfach und benötigt für die Logik nur wenig Chipfläche. Zusätzliche Leitungen sind anwendungsabhängig, so können mehrere SS (Slave Select) Leitungen eingesetzt werden wenn über dieselben Leitungen mehrere Slaves zu einem Bus zusammen gefügt werden. Atmel benutzt zur Programmierung ihrer

Mikrocontrollerfamilien zusätzlich eine Reset Leitung und einen gemeinsamen Ground [AtSPI2000].

Das Programmiergerät ist beim Schreiben von Chips immer der Master. Dem Master muss bekannt sein welches Slavegerät programmiert werden soll. Dafür sind von Controllerherstellern spezielle Opcodes vorgesehen. Andere Opcodes ermöglichen die Auswahl verschiedener Speicher, -Adressen und Kontrollregister um Code zu lesen oder zu schreiben. Der Inhalt der Kontrollregister kontrolliert zum Beispiel ob über SPI der Speicher gelesen oder beschrieben werden darf oder ob das Interface völlig deaktiviert ist. Die Schreib- /Leseschutz kann danach nur durch einen vollständigen Löschvorgang des gesamten Chips aufgehoben werden.

2.2.2 JTAG – IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture

Die Standard Test Access Port and Boundary-Scan Architecture wurde 1990 mit dem Ziel entwickelt auf Platinen und ICs Software und Hardware zu debuggen. Es bietet somit eine komfortable Möglichkeit die Mehrheit der internen Speicher auszuwerten. 1994 wurde der Standard um die BSDL (Boundary Scan Description Language) erweitert, die den Logikumfang der Boundary Scans beschreibt. Nahezu alle Chiphersteller haben diese Standards implementiert.

Das JTAG Interface ist ein serieller Bus mit 4 bis 5 Leitungen an dem mehrere Endgeräte zum Debugging verbunden sein können. TDI (Test Data In), TDO (Test Data Out), TCK (Test ClocK), TMS (Test Mode Select) und das optionale TRST (Test ReSeT).

2.2.3 Höhere Programmierschnittstellen

Mit den beschriebenen Methoden lässt sich der Chip als ganzes programmieren. SPI oder JTAG bieten vollen Hardwarezugriff und können den gesamten Speicher löschen und beschreiben. Sie erfordern allerdings direkten Zugriff auf den Chip oder die Platine, beide Methoden sind elektrisch nicht zur Fernwartung ausgelegt. Soll die Software einfach von Wartungspersonal oder dem Endbenutzer aktualisiert werden können oder die Software über ein Netzwerk gewartet werden muss eine andere Technik angewandt werden.

Es bietet sich an ein weiters Programm zu Beginn auf dem Mikrocontroller zu installieren, das den Schreibvorgang übernimmt. Ein solches Programm wird manchmal auch als Bootloader für Mikrocontroller bezeichnet. Es kommuniziert über die Standardkommunikationskanäle des Systems und kann von außen zum Beispiel über einen Interrupt gestartet werden. Soll also eine neue Version der Software auf das Feldgerät geladen werden, so wird zunächst die Programmieranforderung an das Gerät gesendet. Der Controller führt daraufhin den Bootloader aus, welches Daten vom Master empfängt und gemäß den Anforderungen ein neues Programm oder eine andere Konfiguration in den gewählten Speicher schreibt. Ist der Vorgang beendet, beendet sich der Bootloader und veranlasst den Prozessor den neuen Code auszuführen. Dieses

Vorgehen hat offensichtlich erhebliche Vorteile gegenüber der direkten Programmierung. Neben der Fernwartung ermöglicht es auch beliebig komplexe Kommunikationsprotokolle mit Verschlüsselung und Authentifizierung. Ferner können nach dem erstmaligen beschreiben des Controllers mit dem Ladeprogramm die direkten Schnittstellen JTAG oder SPI abgeschaltet werden, wodurch es unmöglich wird den Chip direkt auszulesen oder zu überschreiben.

3 Software in Automatisierungssytemen

Dadurch, dass Hardware nach optimaler Erfüllung der Aufgabe bei geringstmöglichen Kosten ausgesucht wird, muss die Software noch viel stärker optimiert sein. Im gleichen Zuge aber müssen hohe Anforderungen and die Safety erfüllt werden. Damit unterscheiden sich die Software und die eingesetzten Programmiertechniken der Automatisierungssysteme von denen im klassischen IT-Bereich stark. Dieses Kapitel geht auf einige, für diese Arbeit wichtige Aspekte der Programmierung und Softwareumgebung in Automatisierungssystemen ein.

3.1 Programmumgebung

Je nach Anwendung werden Programme zur Bearbeitung des eigentlichen Automatisierungsprozesses in unterschiedlichen Umgebungen ausgeführt. Der Entwickler muss entscheiden ob ein Programm als Stand-alone in einer Runtime der auf einem Betriebssystem läuft.

3.1.1 Stand Alone

Sind Prozessoren exklusiv für eine bestimmte Aufgabe reserviert, läuft meist nur ein Ausführungsthread auf dem Prozessor, in diesem Falle verzichtet man auf ein teures Echtzeitbetriebssystem. Das Programm läuft entweder in einer Endlosschleife oder wird bei Bedarf von einem Interrupt gestartet und beendet sich nach erfolgreicher Ausführung wieder. Dies hat den Vorteil dass der Mikrocontroller in einen Energiesparmodus versetzt werden kann. Ohne Betriebssystem hat der Programmierer vollen direkten Hardwarezugriff, alle I/O Ports und Kontrollregister des Systems sind direkt sichtbar und können beliebig angesteuert werden. Auch muss die gesamte Speicherverwaltung vom Programmierer selbst übernommen werden. Diese Art der Entwicklung ist sehr teuer und fehleranfällig, da die gesamte Funktionalität neu geschrieben und getestet werden muss. Die Entwicklung wird erst einfacher wenn bereits Standardbibliotheken auf die Zielarchitektur portiert sind.

3.1.2 Runtimeumgebung

Eine Runtimeumgebung ist eine Programmbibliothek die dem Programmierer viele Aufgaben abnimmt. Gegen diese Bibliothek kann dynamisch gelinkt werden, wenn sie auf dem Zielsystem von mehreren Programmen benutzt wird, oder statisch in das Programm gelinkt werden wenn es exklusiv auf dem Prozessor läuft. Beispiele für Programme die in einer Runtimeumgebung ausgeführt werden sind Applikationen auf einem Mobiltelefon in Java. Die Java-Runtime ist fest auf dem Mobiltelefon installiert und die Benutzerprogramme benutzen die Funktionalität der Runtime, diese Technik spart erheblich Speicheplatz ein. Je nach Bedarf kann eine Runtime um weitere Funktionen erweitert werden. Eine Runtime übernimmt oft auch Aufgaben eines

Betriebssystems wie Speicherverwaltung oder (Echtzeit-) Scheduling, eine bekannte Runtimebibliothek für Echtzeitanwendungen ist Ada. Jedoch nicht immer ersetzt eine Runtime ein Betriebssystem.

3.1.3 Betriebsystem

Unter einem Betriebssystem versteht man eine Software, die die Hardware abstrahiert und vordefinierte Schnittstellen zum Zugriff auf die Hardware bereitstellt. Es bietet weiterhin Möglichkeiten zur Prozesskommunikation, Prozessscheduling und übernimmt die Speicherverwaltung.

Die Abstraktion der Hardware bedeutet nicht nur, dass ein Programm nicht mehr direkt auf die Hardware zugreifen kann. Jeder Prozess hat seinen eigenen Speicheradressraum. Prozesse auf einem Betriebsystem können ebenfalls über Runtimebibliotheken mit dem Betriebsystem kommunizieren oder direkt Systemcalls ausführen. Runtimebibliotheken auf Betriebsystemen liegen meist als Shared Library vor. Der ausführbare Code der Bibliothek liegt nur ein Mal im Speicher, jedoch werden die Verwaltungsdaten, die die Runtime für jeden Prozess benötigt, separat in dessen Speicheradressraum gespeichert. Die Bibliothek wird mittels Virtual Memory und Linker in den Adressraum des Prozesses eingeblendet. Runtime und Betriebsystem ergänzen sich meist gegenseitig. Für den Programmierer ist es einfacher über Bibliotheken mit dem System zu kommunizieren. Ein bekanntes Beispiel hierfür ist die Standard C Library „glibc“, die benutzerfreundliche Funktionen für Systemcalls bereitstellt.

Der Ansatz eines Betriebsystems eignet sich vor allem auf Prozessleitebene oder in komplexeren Produkten. In der Automatisierungstechnik sind hauptsächlich Echtzeitbetriebsysteme von Bedeutung. Ein Betriebsystem erfordert meist leistungsstärkere Prozessoren, hat einen relativ großen Speicherbedarf und nicht unerheblichen Verwaltungsoverhead.

Abbildung 3 illustriert Kapselung der Hardware in einem parallelen System mit Betriebsystem (a) und mit einer Runtime die einen Taskscheduler bereitstellt (b).

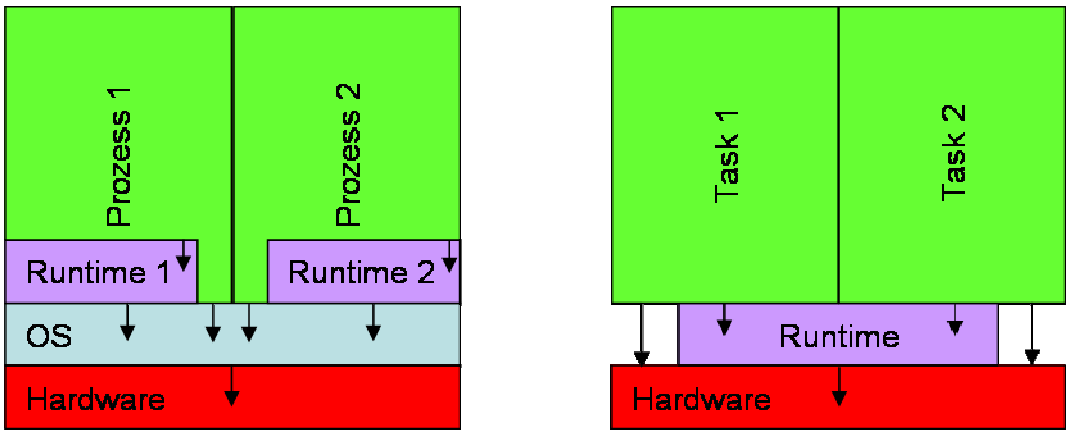

Abbildung 3: parallele Prozesse in Betriebsystem (a) in Runtime mit Taskscheduler (b)

3.2 Speicherorganisation

Wird keine Runtime und kein Betriebsystem eingesetzt, so muss der Programmierer den gesamten physikalischen Speicher selbst verwalten. Dies ist bei einfachen Programmen oder bei Single-Threaded-Applikationen nicht besonders schwer, insbesondere wenn kein Speicher dynamisch alloziert wird und alle Daten in statischen Variablen gespeichert werden. Das ausschließliche Verwenden von statischen Variablen macht, im Falle dass ein Programm exklusiv auf einem Prozessor läuft, durchaus Sinn, der Speicher wird dann implizit durch das zuweisen von Variablen zur Compilezeit verwaltet.

Sind parallele Prozesse zur Bewältigung der Aufgaben eines Feldgerätes nötig so wird meist eine Runtime mit Echtzeitscheduler oder ein Betriebssystem eingesetzt, welches die Speicherverwaltung übernimmt. Läuft ein Prozess in einer Runtimeumgebung oder auf einem Betriebsystem ist es nicht mehr dem Programmierer überlassen den physikalischen Speicher selbst zu organisieren. Nun ist es möglich den einzelnen Tasks dynamisch aus einem gemeinsamen Pool Speicherbereiche je nach Bedarf zusätzlich zu bereits belegten statischen Speicher zuzuweisen. Die Programme können, wenn sie ihn nicht mehr benötigenden, Speicherbereich wieder ans das Betriebsystem oder die Runtime zurückgeben.

Die nachfolgenden Abschnitte beziehen sich nur noch auf die Programme die in einem Betriebsystem laufen.

3.2.1 ELF – Executable and Linkable Format

Auf modernen Betriebsystemen ist das Laden und Ausführen eines Programms kein trivialer Vorgang mehr. Es ist dazu eine Reihe von Informationen nötig. Die grundlegenden Informationen, die das System benötigt um ein Prozessabbild im Speicher zu erstellen, sind in der Binärdatei selbst gespeichert. Beispiele für solche Container mit Metainformationen sind das inzwischen überholte „a.out“, „COFF“ oder das heutzutage weit verbreitete „ELF“ (Executable and Linkable Format) [TIS2000]

ELF wurde von den Unix System Laboratories (USL) entwickelt und 1995 in den TIS (Tool Interface Standards) veröffentlicht, um die Nachteile der Portabilität von „a.out“ abzuschaffen und ein Dateiformat zu entwickeln, das alle notwendige Metadaten für das Ausführen oder Binden enthält. Es ist das Standardformat unter Sun Solaris, GNU/Linux, BSD und SGI Irix und ein flexibles Dateiformat für Objektcode, Bibliotheken, sogenannete Libraries und Ausführbare Dateien, Executables. Für die weitere Betrachtung dieser Arbeit ist nur das Format der Executable von Bedeutung.

Eine Ausführbare Datei enthält nicht nur den eigenen Maschinencode sondern meistens auch Informationen über dynamische Bibliotheken die zur Laufzeit geladen werden müssen.

Jede ELF-Datei beginnt mit dem ELF-Header der die Organisation der Binärdatei beschreibt. Darauf folgt bei der Executable der Program-Header und meist mehrere so genannte Segmente,

die den Programmcode und Daten enthalten. Die Position der Segmente und die des Programheaders sind nicht vorgeschrieben, nur der ELF-Header muss am Dateianfang stehen.

Der Program-Header einer Executable ist ein Array von Strukturen, die jede ein Segment oder andere Informationen beschreiben, die nötig sind, um ein Programm auszuführen. Jedes Segment enthält eine oder mehrere Sektionen. Eine Executable Datei in Sektionen zu zerlegen und mit zusätzlichen Metainformationen auszustatten hat große Vorteile, insbesondere auf Maschinen mit einer MMU (Memory Management Unit) die Virtual Memory Paging unterstützen. Die typische Sektionsgröße bei einem Binary für 32 Bit Intel x86 Architektur ist 4096 Byte, was genau der Seitengröße des Virtuellen Speichers der MMU entspricht.

Die wichtigsten Sektionen in einer ELF Datei sind:

- .text : hier ist der ausführbare Maschinencode gespeichert
- .rodata: Konstanten die zur Compilezeit initialisiert wurden und nicht geschrieben werden können.
- .data: Zur Compilezeit initialisierte Variablen
- .bss: Informationen über die nicht initialisierten Variablen. Hier sind lediglich Informationen gespeichert wie viel Speicher beim Ausführen für statische Variablen benötigt wird.
- .symtab: Symboltabelle, in dieser Sektion werden Symbole also Sprungmarken auf Zieladressen umgesetzt
- .dynamic: Informationen für dynamische Linker, hier sind die Namen der zur Laufzeit benötigten externen Bibliotheken.
- .plt: Procedure Linkage Table. Diese Sektion verbindet positionsunabhängige Adressen für Funktionsaufrufe. Sie enthält kleine Stücke Maschinencode zur Dereferenzierung von Einträgen der GOT, die PLT wird im schreibgeschützten bereich abgelegt.
- .got: Global Offset Table, enthält Informationen über absolute Adressen die zur Laufzeit errechnet werden.

Auf Systemen mit Virtuellem Adressraum sind Sprungziele zur Compilezeit nicht bekannt. Die tatsächlichen Adressen müssen zur Laufzeit vom Linker errechnet werden. Dazu muss die Executable positionsunabhängig kompiliert sein. Benötigt ein Programm Zugriff auf die absolute Adresse eines Symbols, so hat dieses Symbol einen Eintrag in der GOT. Die dynamisch geladene Bibliothek liegt nur einmal im Speicher vor, kann aber parallel von verschiedenen Programmen benutzt werden. Der Code der Bibliothek wird in den Adressraum des Prozesses eingeblendet. Da ein Programm nur Sprungadressen kennt werden diese vor dem Start in vom Linker in der GOT eingetragen und beim Aufruf über die PLT doppelt dereferenziert, das Programm springt letztendlich an die Adresse die als Ziel in der GOT eingetragen ist. Es sind im ELF Standard noch weitere Sektionen und Informationen spezifiziert, der gesamte Standard ist in [TIS2000] erläutert.

3.2.2 Speicherabbild eines Prozesses

Eine elementare Ressource die ein Prozess belegt ist Speicher. Auf einem System mit Virtuellem Speichermodell ist dies ein Adressraum, der von 0 bis zu einer systemabhängigen maximalen Obergrenze linear wächst. Die virtuellen Adressen werden für jeden Prozess von der MMU aufgelöst und auf physikalische Speicheradressen abgebildet. Dieser Speicherraum wird dem Prozess beim Start exklusiv vom Betriebsystem zugesichert, versucht ein anderer Prozess auf eine Adresse zuzugreifen, die nicht in seinem Adressraum ist, ist dies ein schwerwiegender Fehler und der Prozess wird vom Betriebssystem beendet.

Beim Laden und Starten des Programms wird zunächst ein Stück zusammenhängender Speicher gemäß für den Programmcode alloziert, allgemein wird dieser Speicherbereich als .text Sektion bezeichnet. Ebenso wird für die anderen Sektionen .rodata, .data und .bss Speicher alloziert. Mittels Memory-Mapped-Files-Mechanismus werden die Sektionen der Datei auf die Seiten des virtuellen Speichers abgebildet, erst beim ersten Zugriff werden diese in das physikalische RAM geladen. Dies hat neben der Ladegeschwindigkeit weiter den Vorteil, dass kein Speicher für noch nicht benötigten Programmcode oder Daten benutzt wird. Die Sektionen .text und .rodata sind als „read-only“ markiert und können nicht mehr verändert werden. Sollte ein Programm versuchen sich selbst zu verändern ist das ein fataler Programmierfehler und das Betriebsystem beendet den Prozess. Danach wird die .got-Sektion modifiziert, alle absoluten Adressen von Funktionsaufrufen und Symbolen werden auf die momentan gültigen Adressen der externen Bibliotheken im Speicher umgeschrieben.

Der Stack ist ein einfacher LIFO-Speicher für lokale Variablen, Argumente für Funktionen und Rücksprungadressen. Zur Verwaltung ist der Stack in so genannte Stackframes unterteilt, ein Stackframe ist die Menge der für eine Funktion auf dem Stack abgelegten Daten. Auf Intel x86 Architekturen stehen zwei Register zur Verwaltung des Stacks zur Verfügung. Der so genannte „Basepointer“ %ebp, der immer auf den Anfang des Stackframes der Funktion zeigt und der eigentliche Stackpointer %esp, der auf den letzten belegten Speicherplatz, die niedrigste Adresse des Stack zeigt. Der Speicherplatz für den Stack wird vom Kernel verwaltet und wächst, je nach Bedarf, bei Funktionsaufrufen dynamisch, kann allerdings nicht mehr verkleinert werden. Abbildung 4zeigt einen typischen Stackframe, Abbildung 5 veranschaulicht die Organisation von Speicher auf einer 16 Bit von-Neumann Maschine

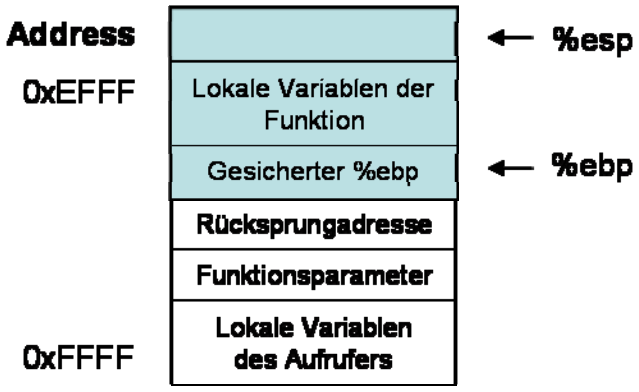

Abbildung 4: Stackframe auf Linux Intel x86

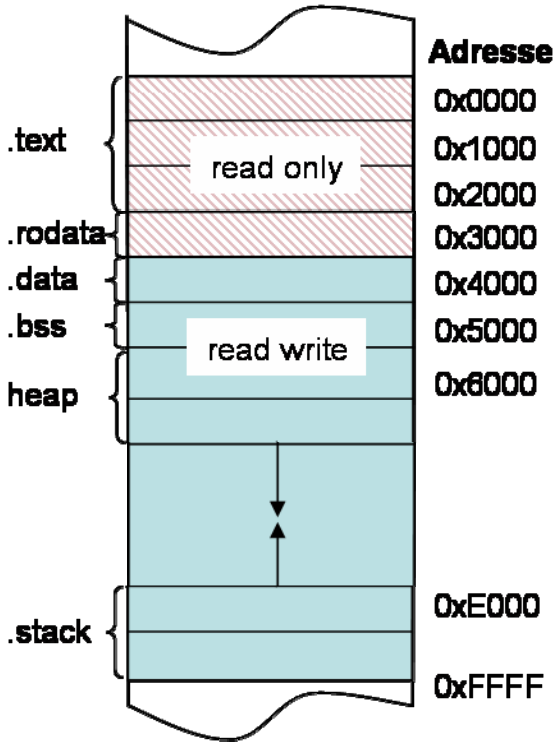

Abbildung 5: Speicherorganisation eines Prozesses

3.2.3 Dynamische Speicherverwaltung

Auf dem Heap wird erst bei Bedarf Speicher für dynamische Objekte, hier als allgemeine Speicherstruktur ohne Bezug auf objektorientierte Programmierung, reserviert. Nach dem der Speicher nicht mehr gebraucht wird, soll der Platz wieder freigegeben und dem Betriebsystem zur Verwaltung zurückgegeben werden. Die Speicheranforderung an das Betriebsystem wird durch den `sbrk()` Systemcall ausgeführt, diese Methode ist allerdings unflexibel und unkomfortabel deshalb stellen Bibliotheken Funktionen zur Speicherverwaltung bereit. C selbst verfügt nicht über die Möglichkeiten Speicher dynamisch zu allozieren, der ANSI-Standard definiert jedoch verschiedenen Funktionen für die dynamische Speicherverwaltung, diese sind in Runtimebibliotheken wie „glibc“ implementiert. ANSI-C schreibt folgende Funktionen vor:

- `void *calloc(size_t nmemb, size_t size)` belegt Speicher für ein Array von `nmemb` Elementen von jeweils `size` Byte und gibt einen Zeiger auf den Speicherbereich zurück. Der Speicher wird auf Null gesetzt.
- `void *malloc(size_t size)` belegt size Byte und gibt einen Zeiger auf den belegten Speicherbereich zurück. Der Speicher wird nicht mit Nullen beschrieben.
- `void free(void *ptr)` gibt den Speicher frei, auf den `ptr` zeigt, welcher von einem früheren Aufruf von `malloc()`, `calloc()` oder `realloc()` zurückgegeben worden sein muss. Wenn `ptr` NULL ist, wird keine Operation ausgeführt.
- `void* realloc(void * ptr, size_t size)` ändert die Größe des Speicherblocks, auf den `ptr` zeigt, auf `size` Byte. Der Inhalt bleibt unverändert bis zum Minimum von alter und neuer Größe; neu angeforderter Speicher bleibt uninitialisiert. Wenn `ptr` NULL ist, ist der Aufruf äquivalent zu `malloc(size);` wenn die Größe `size` gleich 0 ist, ist der Aufruf äquivalent zu `free(ptr).` Wenn `ptr` nicht NULL ist, muss er von einem früheren Aufruf von `malloc(), calloc()` oder `realloc()` zurückgegeben worden sein.

Der zugesicherte Speicher eines Programms wird von Runtimebibliotheken verwaltet, meist wird eine doppelt verkettete Liste mit Einträgen über unbenutzte Speicherblöcke, die so genannte „Freelist“ verwendet. Beim allozieren eines Blockes wird mit einer bestimmten Strategie, entweder „First-Fit“ oder „Best-Fit“, ein passender Block ausgesucht der größer oder gleich dem angeforderten Speicher ist. Der Block wird aus der Liste genommen und die Kette wird mit dem eventuell übrig gebliebenen Rest des Speicherblockes oder dem Nachfolger in der Liste wieder geschlossen. Wurde kein ausreichend großer Block gefunden wird über den `sbrk()` Systemcall mehr Speicher vom System angefordert. Wird der Speicher über `free()` wieder freigegeben, wird der Block an das Ende der Freelist angehängt.

Diese ursprüngliche Methode der dynamischen Speicherverwaltung hat den entscheidenden Nachteil dass der Speicher sehr stark fragmentiert. Das heißt das sehr viele kleine Blöcke in der Freelist stehen die nicht mehr verwendet werden können, der Speicher ist verschwendet. Durch die große Zahl von kleinen Blöcken in der Liste dauert auch das Suchen von passenden Blöcken immer länger, da die Zahl der Listenelemente wächst. In der Literatur ist dieses Problem ausführlich dokumentiert.

Deshalb verwenden Laufzeitbibliotheken wie Java einen Garbage Collector der neben der Aufgabe, Speicherplatz von nicht dereferenzierbaren Objekten wieder freizugeben, auch die Aufgabe hat den Speicher, defragmentieren.

Es ist nicht zwingend ein Garbage Collector zur Verhinderung von Speicherfragmentierung nötig, unter Linux implementiert die „glibc“ einen nicht fragmentierenden Heap. Der dort verwendete Heapverwaltungsalgorithmus ist eine Variante des Algorithmus von Doug Lea [DoLe2000].

Zur Zuweisung von Speicher benutzt das Verfahren eine Best-Fit-Strategie, jedoch für Speicheranfragen von weniger als 256 Bytes weicht das Verfahren ab, falls kein exakt passendes

Stück freier Speicher gefunden wurde und benutzt Speicher der an das Stück angrenzt, das für die letzte Anfrage an kleinen Speicher benutzt wurde. Für große Speicherbereiche, größer als 256 kB verwendet `malloc()`, falls es das Betriebssystem ermöglicht, Memory-Mapping. Die Best-Fit Methode ist prinzipbedingt aufwändig. Daher werden die freien Speicherblöcke in Klassen, so genannte „Bins“, sortiert nach Größe unterteilt. Diese unterschiedlichen Verfahren helfen die Fragmentierung gering zu halten, können das Auftreten aber nicht verhindern.

Bei jedem Aufruf von `free()` wird überprüft, ob die beiden physikalisch angrenzenden Blöcke des Speichers frei sind. Ist einer der beiden Blöcke frei werden sie zu einem zusammengefasst und in die Liste der Bins eingetragen. Alle Speicherblöcke sind mindestens 8-Byte-aligned und haben einen Overhead von 8-16 Byte zur Verwaltung. Die Komplexität des Algorithmus ist O(1). [DoLe2000]

4 Sprachen in der Automatisierungstechnik

Die am weitesten verbreitete Programmiersprache in der Automatisierungstechnik ist nach wie vor C. Der Grund dafür ist der Bekanntheitsgrad unter den Entwicklern und der Toolsupport der Hersteller. Das erste was für einen neuen Chip erscheint ist ein C Compiler, da es für den Hersteller wichtig ist, bestehende Programme auf neue Hardware portierbar zu machen, außerdem muss nur das Codegenerator-Backend der alten Compiler angepasst werden. Zu Anfang werden in diesem Teil oft auch nur die Opcodes geändert, die Optimierung für den Chip wird erst später in einer zweiten Version durchgeführt. Time-to-Market ist hier essentiell, da kein Kunde Prozessoren kauft, für die keine Compiler existieren.

Die Verwendung von reinem Assembler geht auch in der Automatisierungstechnik zurück. Assembler wird meist nur noch „inline“ in C-Code eingebettet. Inline-Assembler wird verwendet wenn Hardwareregister direkt angesprochen werden müssen, beispielsweise wenn der Stackpointer verändert werden soll, oder für die ersten Schritte des Bootvorgangs. Für große Programme ist reiner Assembler allerdings viel zu fehleranfällig und zu teuer in der Wartung. Die Compilertechnologie ist auch so weit fortgeschritten, dass automatisch generierter Code meist schneller ist als handgeschriebener Assembler.

Beide Sprachen haben allerdings den großen Nachteil der Safety. Es existiert keinerlei Typsicherheit oder Überprüfung der Grenzen für Arrays. Die Sprachen bieten dem Programmierer absolute Freiheit mit der Hardware zu kommunizieren zum Preis, dass der Programmierer für alles verantwortlich ist.

Die Sprache Java ist eine Umgebung die teilweise Typsicherheit bietet, ist aber wegen ihrem Threadmodell und Garbage Collector nicht für Echtzeitanwendungen geeignet, und ist keine sichere Sprache. Selbst Hersteller wie Microsoft, Apple oder IBM raten in ihrem Lizenzbestimmungen davon ab Java für sicherheitskritische Anwendungen zu benutzen [MS2000].

Die fehlende Safety der existierenden Sprachen veranlasste das US-Verteidigungsministerium dazu, die Entwicklung einer neuen sicheren Sprache in Auftrag zu geben. Es wurde Ada aus einer Reihe von Konzepten ausgewählt, sie gilt als die sicherste Programmiersprache und wird in vielen sicherheitskritischen Systemen eingesetzt. Ferner definiert Ada eine Runtime die einen Echtzeitscheduler implementiert. Die Entwicklungszeit und die Kosten sind mit Ada sogar geringer und die Codequalität ist ein Vielfaches besser als mit C oder Assembler [StZe1995]. Ada wird jedoch weniger eingesetzt, da nicht für alle Chips Compiler existieren und nur wenige Entwickler Erfahrung mit Ada haben.

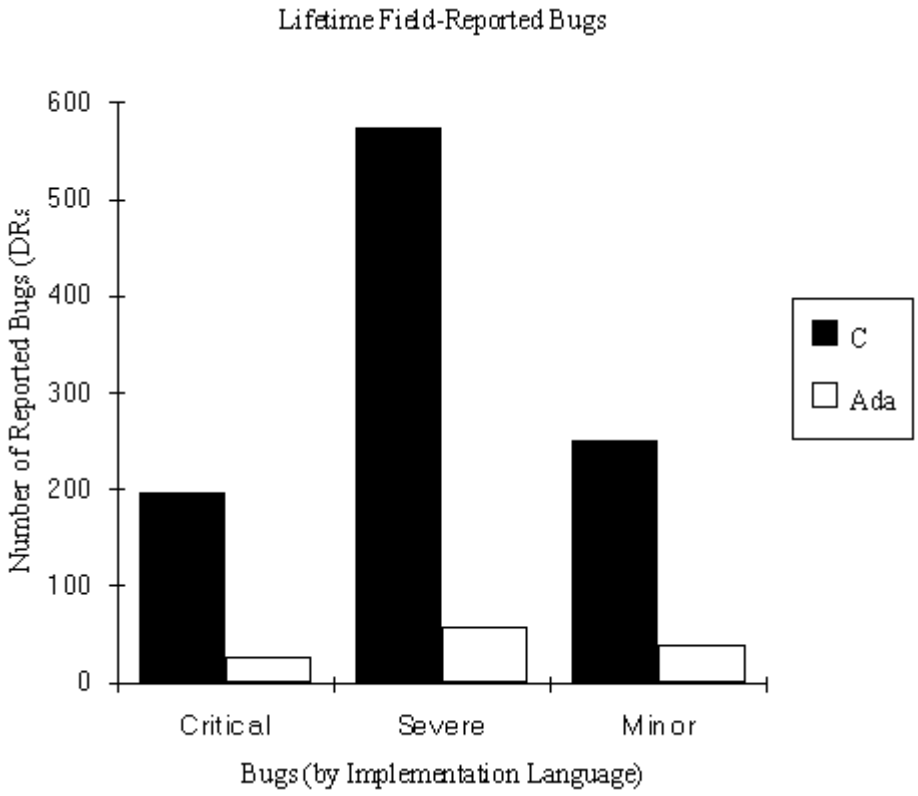

Abbildung 6: Vergleich der Fehler in Ada mit C [StZe1995]

5 Angriffe auf Automatisierungsgeräte

Je nach Art des Automatisierungssystems existieren unterschiedliche Gefahren und Ziele. Es sind verschiedene Angriffsmotivationen und Angreiferklassen, die die Sicherheit eines Systems bedrohen. Dieses Kapitel führt nach den Angriffsmotiven Standards zur Klassifizierung von Angreifern, ihren Angriffen und den Angegriffenen Systemen ein.

5.1 Angriffsmotivation

Je nach Art des Automatisierungssystems existieren für verschiedene Gruppen unterschiedliche Angriffsmotivationen und Ziele. Der Faktor durch „Malware“ berühmt zu werden, spielt anders als in der klassischen IT weniger eine Rolle. So genannte "Script- Kiddies", die ohne tief greifende technische Kenntnisse durch einfache Skripts, wie seinerzeit der "I-love-you-Virus", großen wirtschaftlichen Schaden anrichten, existieren in der Automatisierungstechnik bisher glücklicherweise noch nicht. Mehr als der bloße Spaß am Anrichten von Schaden und um zweifelhaften Ruhm zu erreichen, spielen hier handfeste wirtschaftliche Motive eine Rolle. Angriffe auf Automatisierungsanlagen können erhebliche finanzielle Schäden nach sich ziehen, wenn dadurch Menschen, die Umwelt oder andere Anlagen beschädigt werden. Ein Ziel das sich daraus ergibt ist Sabotage.

Der Urheber kann aber im Gegensatz zu einfacher Sabotage durch physikalische Beschädigung nur sehr schwer gefunden werden. Die fehlerhafte Software kann schon Monate früher eingeschleust worden sein, der Saboteur braucht zum Zeitpunkt des Auslösens nicht vor Ort zu sein. Darüber hinaus kann die schadhafte Software flüchtig sein, das heißt sie ist nach dem Schaden aus dem Speicher verschwunden. In diesem Fall ist es nahezu unmöglich die genaue Ursache des Schadens festzustellen.

Neben Sabotage ist auch in der Anlagenautomatisierung Spionage eine durchaus lohnende Methode, sich unerlaubterweise Wettbewerbsvorteile zu verschaffen. Angreifer können wieder ohne zum Zeitpunkt des Abhörens vor Ort zu sein Daten über technische Prozesse sammeln, speichern und zu einem anderen Zeitpunkt abrufen. Der Aspekt der Spionage, beziehungsweise Piraterie ist auch für Produktautomatisierungssoftware eine direkte Bedrohung.

Nicht nur Daten des technischen Prozesses sondern auch die Software der Feldgeräte, bzw. der Produkte sind für Konkurrenten interessant. Es ist in Firmen üblich Produkte der Konkurrenz zu untersuchen und Lösungsansätze zu kopieren. Rechtlich ist dieses Vorgehen in einer Grauzone, abhängig von den nationalen Patentrechten. Die Grenze zwischen legalem Reverse-Engineering und Piraterie ist schwer zu ziehen. Die Firma, die die Software entwickelt hat, versucht das Kopieren zu verhindern, um ihr geistiges Eigentum zu schützen.

5.2 Angriffsarten

Eine grobe Einteilung der Angriffsarten kann nach dem funktionalen Ziel der Angriffe in drei Gruppen gemacht werden.

- Angriffe auf die Vertraulichkeit, Privacy Attacks: Das Ziel dieses Angriffes ist es, Kenntnis über vertrauliche Daten zu erlangen, die im System verarbeitet, gespeichert oder mit der Peripherie ausgetauscht werden.
- Angriffe auf die Integrität. Der Angreifer versucht, Daten oder Programmcode im System zu manipulieren.
- Angriffe auf die Verfügbarkeit: Diese Art von Angriff versucht die normale Funktion des Systems durch Missbrauch der Ressourcen zu stören.

Ein anderer Unterscheidungsaskekt der Angriffe auf Embedded Systems, liegt darin, wie die Angriffe durchgeführt werden und welche Mittel dazu benutzt werden.

- Software-Angriffe sind Angriffe die mit Hilfe eingeschleuster Software, z.B. trojanische Pferde oder Viren durchgeführt werden. Diese Art von Angriffen ist weit verbreitet, da keine Spezielle Hardware oder Kenntnis des Systems vorausgesetzt ist. Es werden bestehende Eingabeströme auf bekannte Schwachstelle wie Buffer-Overflows überprüft. Diese Angriffe können aber auch relativ kostengünstig durch sichere Software verhindert werden.
- Physikalische Angriffe: Alle Angriffe auf physikalischer Ebene auf das System. Sie sind wesentlich aufwändiger als Softwareangriffe, da sie mehr Kenntnis über das System, wesentlich höheres Wissen und Erfahrung des Angreifers als und Ausrüstung voraussetzen. Beispiele für physikalische Angriffe sind Öffnen des Chipgehäuses für Microprobing oder das Abhören der Buskommunikation im System.
- Sidechannel-Angriffe beobachten sekundäre Größen während das System bestimmte Aktionen wie kryptographische Algorithmen durchführt. Es wird zum Beispiel Ausführungszeit, Energieverbrauch, Spannungsschwankungen, oder das Verhalten im Fehlerfall beobachtet. Sie sind wie physikalische Angriffe sehr aufwändig und erfordern viel Erfahrung und Zeit.

Angriffe können entweder passiv, im Sinne, dass sie die Funktionsfähigkeit des Systems nicht stören, sondern lediglich beobachten sein, oder sie können aktiv das Verhalten des Systems beeinflussen. Der Angreifer wird meist mehrere Arten von Angriffen parallel ausführen, um seine Chancen das System zu kompromittieren zu erhöhen. So kann zum Beispiel gezeigt werden, dass über Sidechannel-Attacken, die den Energieverbrauch analysieren, während ein Mikrocontroller eine Softwareimplementierung des DES-Algorithmus ausführt, der Schlüsselraum von 2^{56} auf 2^{40} verkleinert werden kann [SrArSc2004].

5.3 Klassifizierung von Angreifern

Abhängig vom finanziellen Wert und der Art des geistigen Eigentums muss es gegen verschiedene Gruppen von Angreifern geschützt werden. [Abr91] teilt die Angreifer in drei Klassen ein:

- Klasse I: Clevere Außenstehende. Sie sind oft sehr intelligent, haben aber nur wenig Wissen über den Aufbau des Systems. Sie haben nur Zugang zu wenig aufwändiger Ausrüstung. Sie nutzen oft existierende Schwächen des Systems aus, anstelle neue zu schaffen. Beispiele sind Studenten oder Hobbyelektroniker.
- Klasse II: Erfahrene Insider. Sie haben eine fundierte Ausbildung und Erfahrung mit der Materie. Sie haben unterschiedlich gute Kenntnis über die Komponenten Systems, haben aber möglicherweise Zugang zu den meisten davon. Oft haben sie auch Zugang zu hoch entwickelter Spezialausrüstung zur Untersuchung von elektronischen Systemen. Beispiele hierfür sind ehemalige Mitarbeiter oder Mitarbeiter des Herstellers.
- Klasse III: Zahlungskräftige Organisationen. Sie sind in der Lage, Spezialistenteams zusammenzustellen, die mit großen finanziellen Mitteln ausgestattet sind. Sie können eine detaillierte Analyse der Systeme durchführen, und ausgereifte Angriffe entwickeln. Klasse-III-Angreifer haben Zugang zu den aufwändigsten modernsten Laborgeräten wie (Chiptestarbeitsplätze, Elektronenstrahlmikroskope, oder Laser). Sie können eventuell Klasse-II-Angreifer als Teammitglieder beschäftigen. Beispiele für Klasse-III-Angreifer sind die Labors von Regierungsbehörden oder großen Firmen.

5.4 Sicherheitslevel eines Systems

Um über Sicherheitsstufen zu sprechen, muss ein System in verschiedene Sicherheitsklassen eingeteilt werden. Der Vorschlag von [Abr91] wird von anderen Autoren übernommen und hat sich als sinnvoll erwiesen. Danach wird ein System in 6 Klassen eingeteilt.

- ZERO: Keine speziellen Sicherheitsmassnahmen ergriffen. Beispiel Standard PC in einem nicht abgeschlossenen Raum oder Mikrocontroller mit externem Speicher
- LOW Es wurden Sicherheitsmassnahmen ergriffen die aber, mit minimalem Toolaufwand wie Zange, Lötkolben und Mikroskop überwunden werden können. Der Angriff erfordert Zeit aber die Kosten sind gering. Beispiele hierfür wären Systeme nach dem Prinzip „Security by Obscurity“.
- MODL: Sicherheitsmaßnahmen gegen einfache Angriffe. Es sind höher entwickelte Werkzeuge und Ausrüstung, sowie gewisses Spezialwissen nötig. Die Kosten für Ausrüstung liegen bei 500-5000 US$. Ein Beispiel dieser Klasse sind Mikrocontroller mit Lockbits, oder Entkoppelkondensatoren die eine Leistungsanalyse erschweren.
- MOD: Spezielle Ausrüstung, eine fundierte Ausbildung und Kenntnis über Chips ist nötig. Die Werkzeuge kosten bis 50000 US$ und der Angriff ist zeitaufwändig aber letztendlich erfolgreich.

- MODH: Es existiert Ausrüstung aber diese ist sehr teuer zu beschaffen und zu verwenden. Die Kosten liegen im 200000 US$ Bereich. Es wird spezielle Erfahrung, Können und Wissen für die Bedienung der Geräte vorausgesetzt. Es wird mehr als eine Operation benötig, dass der Angriff gelingt und es sind Expertenteams verschiedener, sich ergänzender Disziplinen nötig. Der Angriff ist unter umständen dennoch erfolglos. Beim Design des Systems wurde besonderer Wert auf die Sicherheit gelegt. Beispiele hierfür sind Mikrocontroller und ASICs mit speziellen Sicherheitseinrichtungen wie sie in modernen Smartcards und Zahlungssystemen eingesetzt werden.
- HIGH: Alle bisher bekannten Angriffe waren erfolglos. Ein Forschungsteam mit hohem Budget ist notwendig, das Tools und Anlagen speziell für den Angriff entwickelt. Die Gesamtkosten betragen über 1 Mio. US$ und der Erfolg des Angriffs ist ungewiss.

Diese Einteilung ist zwar hilfreich aber nicht standardisiert und nur schwer überprüfbar da sich die Einteilung nach dem Aufwand des Angriffs richtet. US und Kanadische Regierungsbehörden haben in den FIPS 140-1 [FIPS] detaillierte Standards für die Zertifizierung sicherer Systeme mit kryptographischen Verfahren beschrieben. Hier werden Anforderungen an die Schutzmechanismen gestellt aber keinerlei Aussagen über Kosten und Aufwand eines eventuellen Angriffs gemacht. Daher eignet sich diese Standardisierung für die Zertifizierung neuer Produkte.

- Sicherheitslevel 1: Es werden nur Anforderungen an die Software und die verwendeten Verschlüsselungsalgorithmen gestellt. Die Hardware wird nicht physikalisch geschützt, es sind keine Anforderungen an das Gehäuse oder den Installationsort des Systems gestellt.
- Sicherheitslevel 2: Das System muss mit einem Siegel oder einem Schloss gesichert sein, so dass im Nachhinein ein einfacher Manipulationsversuch festgestellt werden kann.
- Sicherheitslevel 3: Der Versuch einer Manipulation soll schon während sie stattfindet erkannt werden, und nicht nur im Nachhinein optisch sichtbar sein. Das System schützt sich selbst dadurch, dass alle sensiblen Daten bei einem bemerkten Eingriff gelöscht werden.
- Sicherheitslevel 4 verlangt einen umfassenden Schutz gegen Zugang zum kryptographischen Modul. Wie in Level 3 muss das System alle empfindlichen Daten löschen. Die Sensoren müssen ferner nicht nur das Gehäuse sondern auch die Umgebung auf Unregelmäßigkeiten beobachten. So dürfen keine starken Temperatur-, Spannungs- oder Kapazitätsschwankungen auftreten, die auf einen physikalischen Angriff hindeuten können.

Dass ein System als sicher gilt müssen vier wichtige Kriterien erfüllt werden. Zunächst gilt es einen Angriff so weit wie möglich zu verhindern, dies schließt Schutz gegen physikalische Angriffe und Sidechannel Angriffe ein. Ein zweites Kriterium ist, dass ein Angriff vom System selbst bemerkt werden muss. Hierbei ist die Zeitspanne zwischen Angriffszeitpunkt und Alarm, bzw. Reaktion von entscheidender Bedeutung. In dieser Zeitspanne hat der Angreifer Kontrolle über das System. Das dritte Kriterium ist, welche Gegenmaßnahmen bei einem registrierten Angriff ergriffen werden, das System muss in der Lage sein autonom angemessen auf einen Angriff zu reagieren, und den Schaden so gering wie möglich halten. Die vierte und letzte Stufe ist die Beweissicherung für den Systembetreiber, dieser muss in der Lage sein nach einem

Angriff oder Angriffsversuch aufgrund der gespeicherten Daten den Angriff nachzuvollziehen um zukünftige Angriffe gleicher oder ähnlicher Art zu verhindern. [SrArSc2004]

6 Security-Probleme in Software

Viel häufiger als professionelles Reverse-Engineering und physikalische Angriffe auf Automatisierungssysteme sind Softwareangriffe. Diese sind in der Regel viel kostengünstiger, da sie weniger Hardware und Fachkenntnis erfordern. Die Schwachstellen sind sehr ähnlich und der Ablauf eines Angriffs ist nahezu immer der Selbe. Die Schwachstelle, die am meisten für Angriffe ausgenutzt wird ist ein so genannter Buffer-Overflow. Bei einem Buffer-Overflow schleust der Angreifer eigenen Code über die Kommunikationsschnittstellen des Programms ein und verändert den Programmablauf so, dass sein eigener Code ausgeführt wird. Nachdem der Angreifer Kontrolle über den Programmfluss hat, kann er beliebige Aktionen, wie Auslesen des geschützten geheimen Programmcodes oder Schlüsseldaten oder Angriffe auf benachbarte Systeme durchführen. Gegen solche Angriffe nutzt auch ein sicherer Mikrocontroller nicht.

Bei einem Puffer handelt es sich um ein zusammenhängendes Stück Speicher das in der Regel für temporäre Daten benutzt wird. Meist dient ein Puffer dazu, die Ein- und Ausgabeoperationen zu beschleunigen. Daten können Blockweise in Puffer geschrieben werden um von dort im RAM dann Byte für Byte verarbeitet werden, dies ist meist um ein Vielfaches schneller als byteweise einen Ein- oder Ausgabestrom zu bearbeiten.

Das Problem eines Pufferüberlaufs besteht, wenn mehr Daten in den Puffer geschrieben werden als dieser aufnehmen kann. Dies geschieht, da viele Programmiersprachen keinerlei Überprüfung der Grenzen durchführen. In der Programmiersprache C sind auch Strings nur Arrays von Character Daten. Ein String hat keine variable Länge, sondern es muss vorher der maximale Speicherbedarf vom Programmierer abgeschätzt und reserviert werden. Das Ende eines Strings wird durch das ASCII Zeichen NUL definiert. Liegt im Eingabestrom keine ASCII NUL vor, so erkennen die Standardbibliotheken von C kein Ende des Strings und lesen weiter Daten ein. Im Folgenden wird nun beschrieben wie ein Angriff mittels Pufferüberlauf auf dem Stack erfolgt.

6.1 Buffer-Overflow auf dem Stack

Der einfachste und häufigste Bufferoverflow geschieht auf dem Stack. Oft wird auch von einem „Stackoverflow“ in diesem Zusammenhang gesprochen, dies ist aber ein falscher Ausdruck da nicht der Stack überläuft sondern nur ein Puffer auf dem Stack. Um zu verstehen, wie ein Angreifer Code einschleusen kann, muss man zunächst den normalen Programmablauf beim Aufruf einer Funktion analysieren. Folgendes Programm in ANSI C wird vom GNU C Compiler unter Cygwin auf einer Intel x86 Architektur übersetzt. Die Funktion `func1()` enthält einen typischen Programmierfehler, der angegriffen werden kann.

```
#include <stdio.h>
int global = 4;
int func1(int,int);
```

```
int func2(int,int);

int main(int argc, char **argv){
      int a=1,b=7,retval=0;
      retval = func1(a,b);
      printf("addr: %p val %d \n",&retval ,retval);
      return retval;
}

/* Funktion mit Bufferoverflow */
int func1(int par1_1, int par2_1){
      int retval_1;
      char buf[12];
      // zweiter Funktionsaufruf
      retval = func2(par1_1,par2_1);

/* Lesen eines Eingabestrings in den Puffer, die Länge
* wird nicht überprüft!!!!
*/
      gets(buf);
      return retval_1;
}

/* Normale Funktion zur Demonstration des Anlegen eines Stackframes*/
int func2(int par1_2,int par2_2){
   int retval_2;
   retval = par1_2+par2_2-global;
   return retval_2;
}
```

Abbildung 7: C-Code Beispiel

Wird ein Eingabestring innerhalb der Grenzen übergeben, so wird, vom Programm die Zahl 4 als Wert für `retval` zurückgegeben. Selbst wenn ein doppelt so langer String eingegeben wird führt dies nicht zu einem falschen Wert für `retval` oder dem Absturz des Programms, da der Compiler einen großzügigen Stackframe anlegt der mehr Daten als nötig aufnehmen kann, dieses Verhalten ist aber nicht für jeden Compiler garantiert. Erst bei einer Eingabelänge von 30 Zeichen wird der Speicher von `retval` überschrieben, das Programm stürzt aber immer noch nicht ab. Ab einer Länge von 40 Zeichen wird der gesicherte %ebp überschrieben, was zu einem Segmentation Fault SIGSEV, nach der Rückkehr aus der `main()` Funktion führt. Im Disassembler wird deutlich was bei einem Funktionsaufruf geschieht.

```
00401050 <_main>:
      push   %ebp                          // Funktionsprolog
      mov    %esp,%ebp
      sub    $0x28,%esp
      and    $0xfffffff0,%esp
      mov    $0x0,%eax
      add    $0xf,%eax
      add    $0xf,%eax
      shr    $0x4,%eax
      shl    $0x4,%eax
      mov    %eax,0xfffffff0(%ebp)
      mov    0xfffffff0(%ebp),%eax
      call   401110 <___chkstk>           // Cygwin spezifisch unter Linux
      call   4011a0 <___main>            // nicht vorhanden
                                          // Anlegen der lokalen Variablen
```

```
                                          // auf dem Stack mit positivem
                                          // Offset zum %ebp
      movl   $0x1,0xfffffffc(%ebp)        // variable a
      movl   $0x7,0xfffffff8(%ebp)        // variable b
      movl   $0x0,0xfffffff4(%ebp)        // retval
                                          // Lokale Variablen auf den Stack
                                          // als Parameter für func1 ablegen
      mov    0xfffffff8(%ebp),%eax        // b laden
      mov    %eax,0x4(%esp)               // b->par2_1
      mov    0xfffffffc(%ebp),%eax        // a laden
      mov    %eax,(%esp)                  // a->par1_1
      call   4010c3 <_func1>              // Func1 aufrufen
      mov    %eax,0xfffffff4(%ebp)        // Rückgabewert einer Funtion auf
                                          // i386 liegt im Akkumulator
                                          // Akkumulator in lokale Variable
                                          // schreiben
      mov    0xfffffff4(%ebp),%eax        // Variablen für printf() auf
      mov    %eax,0x8(%esp)               // den Stack ablegen
      lea    0xfffffff4(%ebp),%eax
      mov    %eax,0x4(%esp)
      movl   $0x403000,(%esp)
      call   4011c0 <_printf>
      mov    0xfffffff4(%ebp),%eax        // Rückgabewert von main()
      leave
      ret
004010c3 <_func1>:
      push   %ebp                         // Funktionsprolog
      mov    %esp,%ebp
      sub    $0x38,%esp                   // Stackpointer dekrementiern
                                          // für lokalen Speicher
                                          // gcc4.01 unter Linux reserviert
                                          // nur 0x28
      movl   $0x5,0xfffffff4(%ebp)        // retval = 5
      mov    0xc(%ebp),%eax               // par1_1 vom Stack lesen
      mov    %eax,0x4(%esp)               // par1_2 für func 2 auf den Stack
                                          // ein Wort unterhalb %esp ablegen
      mov    0x8(%ebp),%eax               // par2_1 vom Stack lesen
      mov    %eax,(%esp)                  // par2_2 auf den Stack ablegen
      call   4010f5 <_func2>              // func2 aufrufen
      mov    %eax,0xfffffff4(%ebp)        // Rückgabewert in
                                          //  lokalen Speicher kopieren
      lea    0xffffffd8(%ebp),%eax        // Adresse von buf laden und für
      mov    %eax,(%esp)                  // gets() auf den Stack legen
      call   4011b0 <_gets>
      mov    0xfffffff4(%ebp),%eax        // retval_1 in Akkumulator
      leave
      ret
004010f5 <_func2>:
                                          // Funktionsprolog
      push   %ebp                         // Framebasepointer sichern
      mov    %esp,%ebp                    // Stackpointer wird Framebasepointer
      sub    $0x4,%esp                    // Speicherplatz für retval_2
      mov    0xc(%ebp),%eax               // par1_2 vom Stack
      add    0x8(%ebp),%eax               // par2_2 vom Stack
      sub    0x402000,%eax                // global subtrahieren
      mov    %eax,0xfffffffc(%ebp)        // Ergebnis in retval_2 speichern
      mov    0xfffffffc(%ebp),%eax        // retval_2 in Akkumulator
      leave
      ret
```

Abbildung 8: Disassembler zum Beispiel

Jede Funktion, auch die `main()` Funktion beginnt mit dem Funktionsprolog, indem der Stackframe initialisiert wird. Der alte Framebasepointer %ebp wird auf den Stack abgelegt, der alte Stackpointer %esp wird zum neuen Framebasepointer %ebp. Auf 32-Bit Maschinen wird der Stackpointer bei jeder PUSH-/POP-Operation um vier dekrementiert, bzw. inkrementiert, der Prozessor kann nur auf Speicherworte von vier Byte zugreifen, das heißt dass eine Variable auf dem Stack immer mindestens vier Byte in Anspruch nimmt. Für Jede lokale Variable wird der Stackpointer um vier dekrementiert. Hat ein Array eine nicht durch vier teilbare Größe, so muss aufgerundet werden, ein Array mit einer Größe von beispielsweise neun Byte nimmt drei Speicherworte, zwölf Byte in Anspruch. Vor dem Funktionsaufruf werden die Parameter für die Funktion auf dem Stack, relativ zum Framebasepointer, abgelegt. `0x4(%ebp)` bedeutet ein positives Displacement von zwei Speicherworten zum Inhalt von Framebasepointer %ebp. Die Anweisung CALL legt den Instructionpointer %eip auf den Stack und springt zur Zieladresse. Abbildung 4 illustriert einen typischen Stackframe. Die Übergabeparameter an eine Funktion haben also ein positives Displacement zum Framebasepointer, die lokalen Variablen ein negatives. Durch diese Technik mit einem zweiten Zeiger, dem Framebasepointer ist es leichter möglich auf variable Parameterzahl beim Funktionsaufruf zu reagieren, außerdem kann der Stack so auch vom Programmierer innerhalb der Funktion benutzt werden ohne dass sich das Displacement bei der Referenzierung ändert. Die Variablen und Parameter können immer über das gleiche konstante Displacement vom Framebasepointer aus angesprochen werden.

Nach einem Funktionsaufruf liegen also Verwaltungsdaten für den Programmfluss, die Rücksprungadresse und der Stackpointer, und lokale Daten auf dem Stack. Zusammen mit fehlender Überprüfung der Grenzen von Arrays ergibt sich daraus das eigentliche Problem das zu Exploits eines Buffer-Overflows führt. Der Array der lokal angelegt ist beginnt bei einer niedereren Startadresse, wird über den reservierten Speicherbereich hinaus geschrieben wird der Inhalt der höheren Adressen, wie lokale Variablen, der gesicherte %ebp und die Rücksprungadresse überschrieben. Das Problem wird vom Programmierer und im Test nicht erkannt, da sich das Programm innerhalb der Spezifikation normal verhält. Selbst wenn ein Test versucht mehr Daten in einen Array zu schreiben als dieser aufnehmen kann, ist dies zwar ein Buffer-Overflow, der aber nicht zwangsläufig auffallen muss. Es ist leicht möglich dass der Buffer in den Speicherbereich anderer lokaler Variablen wächst, die aber danach nicht mehr verwendet werden.

Läuft das Programm in einer Betriebsystemumgebung, so stürzt es in der Mehrheit der Fälle, wenn die Rücksprungadresse überschrieben wurde, nach Rückkehr aus der Funktion einfach ab. Beim Zugriff auf die ungültige Rücksprungadresse, wird von der MMU ein Page-Fault-Interrupt ausgelöst, das Betriebsystem registriert diesen und prüft, ob die Seite im Adressraum des Prozesses liegt und in den virtuellen Speicher geladen werden muss. Dies ist meist nicht der Fall und der Prozess wird mit einem Segmentation Fault SIGSEV beendet. Problematisch wird es erst, wenn die Rücksprungadresse eingültige Adresse des Programmcodes ist. Dies kann vom Betriebsystem nicht bemerkt werden, da es ein zulässiger Vorgang ist, und das Programm führt eine zufällige Codesequenz aus. Dieses Verhalten tritt ohne böswilliges Zutun eines Angreifers auf.

Interessant ist das Einschleusen von fremdem Code über einen solchen Pufferüberlauf auf den Stack. Wie beschrieben kann die Rücksprungadresse beliebig verändert werden. Der Puffer kann den Code aufnehmen, der ausgeführt werden soll. Allerdings besteht für einen erfolgreichen Angriff mit fremdem Code eine weitere Reihe von Problemen.

Erstens muss die Anfangsadresse des Puffers bekannt sein, um an die richtige Stelle zu springen. Zweitens muss der Angreifer wissen wie viele Bytes er in den Buffer schreiben muss bis dieser alle lokalen Variablen, den gesicherten %ebp und die Rücksprungadresse überschreibt. Drittens muss der Puffer groß genug sein um den Code aufzunehmen. Der Angreifer weiß nicht, wo, an welcher Adresse auf dem Stack sein Code beginnt, um die Rücksprungadresse dementsprechend zu verändern, dass sie auf den Anfang des fremden Programmcodes zeigt. Der fremde Code muss außerdem so geschrieben sein, dass er mit relativer Adressierung auskommt. Der Angriff muss erfolgreich sein ohne den Quellcode zu kennen. Der Angreifer kann keine Aussage über den „Function-Call Graph“ machen, er weis nicht in welcher Schachtelungstiefe sich das Programm befindet und wie viele Argumente jeweils bei jedem Funktionsaufruf auf dem Stack abgelegt wurden. Dies ist eine Reihe von Hindernissen die für einen erfolgreichen Angriff überwunden oder umgangen werden müssen.

Zunächst kann der Angreifer davon ausgehen dass der Stack, auf der jeweiligen Zielarchitektur und dem gleichen Compiler, bei jedem Programmaufruf an der gleichen Adresse, meist dem Ende des Speichers, beginnt. Durch viele Iterationen kann er den nötigen Offset zum Anfang seines Codes ermitteln und die Rücksprungadresse dementsprechend anpassen. Dieses Vorgehen ist sehr zeitaufwendig, es sind mindestens wenige hundert Ausführungen des Programms nötig. Bei einem Produktionssystem ist diese Art von Angriff nicht möglich, da das Zielprogramm bei fast jedem fehlgeschlagenen Angriff abstürzt oder willkürliche Codesequenzen ausführt. Der Angreifer hat gar nicht die Möglichkeit das Programm mit den gleichen Startbedingungen erneut anzugreifen. Angriffe dieser Art sind nur möglich wenn der Angriff an einem Produktautomatisierungssystem, das der Angreifer besitzt, ausgeführt wird oder der Angreifer über den Quellcode des Systems verfügt.

Viel einfacher ist es, vorausgesetzt der Puffer ist groß genug, den Anfang des eingeschleusten Codes mit Null-Operationen, NOPs, zu füllen. Gelingt es irgendwo in den Block mit NOPs zu springen wird nach wenigen Instruktionen der eigentliche Code ausgeführt. Am Ende des Codes, der ausgeführt werden soll stehen eine Reihe von Adressen des angenommen Anfangs des Puffers.

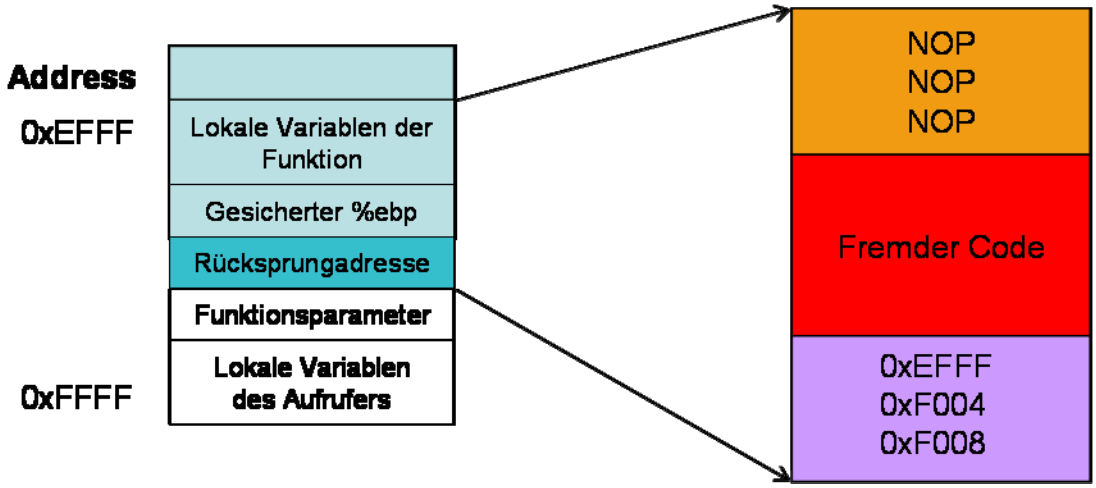

Abbildung 9: Stackframe mit eingeschleustem Code

Damit werden zwei Probleme auf einmal gelöst. Es ist nicht mehr nötig die genaue Anfangsadresse und Länge des Puffers zu kennen. Der Angriff wird nach wenigen Operationen erfolgreich sein. Der Vollständigkeit halber, muss an dieser Stelle noch erwähnt werden, dass der fremde Code bei einem Angriff über unsichere Stringfunktionen wie `gets()` keine ASCII-NUL enthalten darf, da dies als Terminatorcharakter erkannt wird. Mit ein paar Tricks lassen sich aber NULL Zeichen vermeiden, siehe dazu auch [AlOn96]. Das dritte Problem, ein zu kleiner Puffer, lässt sich nicht generisch lösen.

Allerdings sind andere Angriffstypen möglich. Läuft der Prozess der angegriffen wird in einer Betriebsystemumgebung so kann ein return-to-libc-Angriff erfolgreich sein. Bei dieser Art Angriff werden wenige Datenworte als Funktionsparameter auf den Stack abgelegt und die Rücksprungadresse mit einer gültigen Adresse einer Bibliotheksfunktion der libc beispielsweise `system()` überschrieben. Die aufgerufene Funktion nimmt ihre Argumente vom Stack, die vom Angreifer dort abgelegt wurden. Diese Art von Angriff ist besonders interessant, wenn der angegriffene Prozess mit Administratorrechten läuft, dies ermöglicht das Ausführen von beliebigen Betriebsystembefehlen mit Administratorrechten. Auf Feldgeräteebene wäre zum Beispiel denkbar dass der Angreifer ein Wartungs- oder Debugprogramm startet und so beliebige Aktionen durchführen kann.

6.2 Bufferoverflow auf dem Heap

Auf dem Heap wird Speicher dynamisch alloziert. Während der Laufzeit des Programms werden Speicherbereiche für das Programm von Systembibliotheken reserviert. Ein reservierter Speicherbereich auf dem Heap enthält ebenfalls Verwaltungsdaten wie Informationen über die Größe und Zeiger auf angrenzende Blöcke.

Das Einschleusen von fremden Code über einen Pufferüberlauf auf dem Heap ist wesentlich schwieriger, da hier keine Verwaltungsdaten des Programmflusses wie Rücksprungadressen und Framepointer in der Nähe der Daten liegen. Für erfolgreiche Angriffe ist eine sehr genaue

Kenntnis des Heaplayouts und der verwendeten Bibliotheken erforderlich, um Code der dort eingeschleust wurde auch auszuführen. Dies ist möglich wenn der Angreifer die Art der Objekte auf dem Heap kennt. Wird mit Funktionszeigern in dynamischen Speicherobjekten gearbeitet, wie dies der Fall bei objektorientierten Programmiersprachen ist, so kann der Angreifer diese Funktionszeiger so überschreiben, dass diese wiederum, nicht auf den ausführbaren Code der Klasse im .text Segment sondern auf den eingeschleusten Code auf dem Heap zeigen [jp85].

Ein Überlauf auf dem Heap ist aber dennoch nicht ungefährlich. Wird mehr in einen Block auf dem Heap geschrieben als dieser aufnehmen kann so werden zunächst die Verwaltungsdaten der nachfolgenden Speicherblöcke überschrieben. Dadurch wird die Heapstruktur zerstört, die Systembibliothek kann nicht mehr entscheiden ob ein Block frei oder belegt ist, oder wo der Block beginnt und wo er endet. Dies führt früher oder später zu einem Programmabsturz, durch einen Aufruf von `free()` auf einem ungültigen Speicherblock. Bis das Programm aber abstürzt wird es mit fehlerhaften Daten Arbeiten was unkontrolliertes und gefährliches Programmverhalten hervorruft.

7 Schutz geistigen Eigentums in Embedded Systems

Dieses Kapitel befasst sich mit den Möglichkeiten die Daten und den Programmcode direkt aus dem Speicher des Systems auszulesen. Der Angreifer hat direkten physikalischen Zugriff auf die Geräte. Diese Art von Spionage oder Produktpiraterie ist in Produktautomatisierungs- oder Kommunikationssystemen interessant. Der Entwickler muss sich vor Augen halten, dass der Aufwand ein System zu schützen bzw. ein System zu kompromittieren immer in Relation zu dem zu schützenden Gut stehen muss. Security ist ein nicht vernachlässigbarer Kostenfaktor. Der Schutz muss für das gesamte System bestehen. Es darf keine Schwachstellen geben, der Angreifer wird diese finden. Viele Firmen, insbesondere kleinere, sind heutzutage immer noch in der Auffassung das "Security by Obscurity" ausreichend ist. Die trügerische Annahme dass sich niemand die Arbeit des Reverse-Engineering macht und dass geheime Algorithmen zur Sicherheit ausreichend sind, ist leider weit verbreitet.

7.1 Einfache Schutzvorkehrungen

Die Programmierung der Programmspeicher eines Systems über standardisierte Programmierschnittstellen wurde bereits beschrieben. Die Schnittstellen erlauben außerdem auch das Lesen aller Speicherbereiche. Nachdem ein System ausgeliefert wurde soll aber kein Dritter, in Produktautomatisierungssystemen auch nicht der Kunde in der Lage sein die Software auszulesen.

Nahezu alle Mikrocontroller unterstützen daher so genannte Lockbits, die die Programmierschnittstellen deaktivieren können. Dabei handelt es sich meist um einen Bitvektor in EEPROM Technologie, der neben dem Schreib-/Leseschutz des Speichers noch andere Eigenschaften wie Abwärtskompatibilität oder Aktivierung andere Chipfunktionalitäten, wie Watchdog-Timer beeinflusst. Sind die Lockbits für den Speicher gesetzt kann, so kann der Controller nur nach einem kompletten Chip-Reset, wieder neu beschrieben werden, dieser Reset löscht auch alle Speicherbereiche. Das Verfahren eignet zum Schutz der Software gegen Angriffe von unerfahrenen Klasse-I-Angreifern. Es stellt allerdings keinen Schutz gegen physikalische Angriffe dar.

Die Lockbits können mit wenig Aufwand durch physikalische Angriffe auch von Klasse-I Angreifern deaktiviert werden. Mit rauchender Salpetersäure (HNO_3 >98%) wird das Chipgehäuse aufgelöst und das Epoxydharz mit Aceton entfernt. Die Metallisierungsschicht, die Bondingdrähte und das Silizium werden von der Säure nicht angegriffen. Danach wird mit einem lichtundurchlässigen Stift der Speicher abgedeckt, um den Inhalt des EEPROM oder Flashspeichers zu schützen. Der Speicherbereich für die Lockbits befindet sich meist and einer anderen Stelle auf dem Die im einfachsten Fall in der Logik des Programmierinterfaces. Für den

Angreifer ist der Ort uninteressant, er setzt den freigelegten Chip längere Zeit starkem UV-Licht aus. Dadurch wird das Floating Gate des EEPROM Speichers entladen und die Lockbits sind gelöscht. Schwieriger für die Angreifer wird es wenn die Lockbits in der Nähe des Speichers angeordnet sind, der Angreifer muss dann den Ort kennen. Manche Hersteller versuchen solche Angriffe dadurch zu erschweren, dass die Lockbits „active-low“ sind oder durch eine Metallisierungsschicht abgedeckt werden die UV-Licht undurchlässig ist. Dies kann durch komplettes Abtrennen der Lockbits oder Angriff mit Microprobes umgangen werden, dazu sind aber in der Regel nur Klasse-II Angreifer in der Lage [SeSk2005].

Wird ernsthaft von Schutz von Daten gesprochen ist der erste Gedanke Kryptographie. Ein System das den Programmcode und die Daten nur verschlüsselt speichert und auch die Kommunikation auf den Bussystemen durch kryptographische Verfahren absichert erscheint jedem zunächst als sicher, insbesondere wenn erprobte kryptographische Algorithmen wie Triple-DES, AES oder RSA mit langen Schlüssellängen eingesetzt werden. Diese Sicherheit ist allerdings ebenfalls trügerisch. Sie ist vergleichbar mit einem Haus mit abgeschlossenen Türen und Fenstern, wobei der Schlüssel aber unter der Fußmatte liegt. Kein Einbrecher wird sich die Mühe machen die Tür aufzubrechen, wenn er den Schlüssel des Besitzers einfach stehlen kann. Genauso wenig macht sich niemand die Arbeit die Verschlüsselung der Daten zu knacken wenn er den originalen Schlüssel auf einfache Weise auslesen kann. Die Sicherheit des Systems steht und fällt also damit, wo und wie der Schlüssel gespeichert wird. Der Schlüssel selbst muss in einem Speicher im Klartext gespeichert sein, da sonst der Prozessor selbst die Daten nicht verarbeiten kann. Speicher für die Schlüssel außerhalb des Chips aus denen der Prozessor den Schlüssel beim Bootvorgang ausliest, bieten keinerlei Sicherheit. Der Angreifer braucht nicht mehr als ein einfaches Oszilloskop um den Bus zu beobachten und den Bitstrom aufzuzeichnen. Innerhalb kurzer Zeit verfügt er dann über Kenntnis des Schlüssels und kann die geheimen Daten dechiffrieren. Selbst Klasse-I-Angreifer können dieses Verfahren anwenden.

Wird der Schlüssel in einem Speicherbereich auf dem Chip gespeichert bietet das einen erhöhten Schutz, ist aber auch gegen Klasse-I-Angreifer nicht gesichert. Mit einfachen Chemikalien, lässt sich wie beschrieben, das Chipgehäuse auflösen. Da Speicher eine auffällige Struktur haben lassen sich mit einem einfachen Lichtmikroskop der Ort der Speicherzellen und der Busleitungen bestimmen. Der Angreifer benötigt weiterhin Microprobes, Nadeln mit einer Spitze von ca. 5 µm, und kann wiederum mit einem einfachen Vorverstärker und Logikanalysator oder Oszilloskop den Inhalt der Buskommunikation abhören und in den Besitz des Schlüssels gelangen.

Es ist also nötig das System hermetisch gegen Angriffe abzuschotten, jedoch ist es schwierig ganze Platinen zu schützen. Wird das ganze gesamte System zum Beispiel in Epoxydharz eingegossen und mit Sensoren, die das Eindringen registrieren, versehen ist dies ein sehr großer technischer Aufwand und ist oft nicht möglich da die Hitze nicht richtig abgeführt werden kann. Ein besserer Ansatz ist nur den Prozessor zu in ein sicheres Gehäuse zu verpacken und mit einem kleinen nicht flüchtigen Speicher zu versehen, der die geheimen Schlüssel enthält. Der Prozessor kann dann mit externen Komponenten wie RAM oder Programm-ROM verschlüsselt kommunizieren. Gelangt ein Angreifer in Besitz des Programmspeichers oder hört die

Nachrichten auf dem Speicherbus ab, sind die Informationen für ihn, ohne den geheimen Schlüssel, wertlos.

7.1.1 Sichere Mikrocontroller

Da Embedded Systems auch für Finanztransaktionen oder Wahlen eingesetzt werden und sich dadurch eine äußerst verlockende Motivation für Betrugsversuche ergibt, müssen Systeme die Anforderungen der FIPS-1 Level 4 erfüllen. Schon Anfang der 90er Jahre wurden erste sichere Mikrocontroller von Herstellern vertrieben. Beispiele für Chips die soweit als sicher gelten sind der DS5002FP und der Nachfolger DS5240 der Firma Dallas Semiconductors. Viele große Hersteller bieten heute sichere Mikrocontroller zu relativ geringem Preis an. Im Folgenden sollen nun die verschiedenen Schutzmechanismen eines solchen als sicher geltenden Chips erläutert werden.

Beide Mikrocontroller sind kompatibel zur 8051 Familie der Firma Intel. Es sind 8-Bit Rechner der klassischen Harvard-Architektur mit getrennten 16-Bit Adressräumen für Daten- und Programmspeicher, beim DS5240 steht ein erweiterter Adressraum bis zu 8 MB externen RAM zur Verfügung. Für den Anschluss von nicht sicherheitskritischer Peripherie besitzt der Chip vier bidirektionale 8-Bit Ports, für den Speicherbus ist ein separates Interface vorgesehen.

Interessant sind die Schlüsselspeicher des Controllers. Dabei handelt es sich um 64 Bit SRAM Zellen die von einer externen Batterie mit Spannung versorgt werden müssen. SRAM benötig allerdings so wenig Strom, dass eine kleine Lithium Batterie den Speicher für 10 Jahre mit dem benötigten Strom versorgen kann. Auch der externe Programmspeicher ist als SRAM ausgeführt und kann innerhalb von 100ns gelöscht werden. Die Prozessoren verfügen über spezielle SDI (Self Destruct Inputs). Wenn an diesen Pins eine Spannung von 5 Volt anliegt werden die externen Speicher und die internen Schlüssel innerhalb von 100ns von einer speziellen Hardwarelogik überschrieben. An diese Pins kann beispielsweise eine Alarmeinrichtung einer Sicherheitseinrichtung des Gehäuses angeschlossen werden [DS2003].

Das Schlüsselregister enthält den Schlüssel K, der von einem Adressbusverschlüssler EA, dem Datenverschlüssler ED und dem Datenbusentschlüssler ED^{-1} gemeinsam genutzt wird. Die Kryptographischen Algorithmen sind DES oder 3 DES die von einer speziellen Hardwareeinheit, einem 4096 Bit MAA (Modulo Arithmetic Accelerator) unterstützt wird. Der Ciphertext des ED hängt zusätzlich vom Schlüssel auch noch von der Adresse ab auf die zugegriffen wird. ED ist eine 8-Bit Blockcypher, EA arbeitet mit einer Wortgröße von 15 Bit. Wenn der CPU-Kern einen Bytewert *d* an die Adresse *a* schreiben möchte, so wird im externen SRAM der Wert $d' = ED_{K,a}(d)$ in der Speicherzelle mit der Adresse $a' = EA_K(a)$ abgelegt. Bei einem Lesezugriff der CPU auf die Adresse *a* wird aus der externen Speicherzelle $a' = EA_K(a)$ der Wert d´ geladen und nach der Entschlüsselung wird das Ergebnis $d = ED^{-1}_{K,a}(d')$ an den CPU-Kern zurückgegeben. Da sowohl *EA* als auch *ED* bijektive Funktionen sind, ist aus der Sicht des Programmierers kein Unterschied zu einem normalen unverschlüsselten RAM festzustellen. Anschaulich gesprochen sind alle Bytes im externen RAM untereinander durch die Adressverschlüsselung vertauscht worden und zusätzlich wird durch die Datenverschlüsselung

jeder Bytewert einzeln, mit einer eigenen adressabhängigen 256-elementigen Permutationstabelle verändert [MaKu1998].

Zusätzlich zu diesen Mechanismen der Verschlüsselung der echten Buszugriffe erzeugt beim DS5002FP ein Pseudozufallszahlen Generator, immer wenn die CPU nicht auf den Speicherbus zugreift einen Scheinzugriff mit Dummy-Daten auf eine beliebige Dummy-Adresse, um dem Angreifer das abhören und entschlüsseln mittels Plaintextattacken zu vereiteln. Des Weiteren enthält der DS5002FP einen echten Zufallszahlengenerator, dies Zufallszahlen können für user-implementierte kryptographische Algorithmen verwendet werden.

Gegen physikalische Angriffe sind auf dem Mikrocontroller mehrere Sensoren installiert die laufend Umweltparameter wie Temperatur überwachen. Die Variante DS5002FPM, erfüllt die Spezifikation MODH. Dabei ist der Die von einem Netz aus feinsten Metalldrähten umgeben, der Widerstand wird von einer speziellen Sensorschaltung laufend überwacht. Ändert sich der Widerstand, spricht der Alarmmechanismus an und die Schlüsselregister und die Speicher werden sofort gelöscht. [DS2003][DS5240]

Ein Spezieller Pin des Prozessors ist dazu vorgesehen den Firmwaremonitor zu aktivieren der einen neuen Schlüssel generiert, die Software kann dann per seriellem Interface, SPI, in die CPU geladen werden, dort wird sie mit dem Schlüssel des Firmwaremonitors verschlüsselt und im externen SRAM Programmspeicher abgelegt. Mit einem speziellen Firmwarekommando wird der Firmwaremonitor deaktiviert, nachdem die Software in den Speicher geschrieben ist. Das einzige Firmwarekommando das noch zur Verfügung steht dient dem Chipreset, das ermöglicht den gesamten Inhalt zu löschen und neu zu programmieren. Die Grundidee dahinter ist das niemand, nicht einmal der Hersteller dazu in der Lage ist den Speicherinhalt zu entschlüsseln, da der Schlüssel nur im sicheren Bereich der CPU existiert. Jedoch ist es möglich mit der Software schreibend auf den Programmspeicher zuzugreifen, das heißt die das Programm darf sich selbst verändern, dies ermöglicht einfache Softwareupdates, wie oben beschrieben, wenn das System im Feld eingesetzt wird.

Obwohl dieser Mikrocontroller als weitestgehend sicher betrachtet wird stellt Markus Kuhn in [MaKu1998] erfolgreiche Angriffe auf die Busverschlüsselung des DS5002FP vor. Dennoch wird diese Familie von Prozessoren vom Hersteller Dallas Semiconductors/Maxim als sicher verkauft [DS5240].

8 Sicherheitsmassnahmen gegen das Ausführen von Daten

Nachdem im letzten Kapitel detailliert die verschiedenen Angriffsmöglichkeiten auf Systeme vorgestellt wurden soll nun auf die Schadens- oder Angriffsabwehr eingegangen werden. Angriffe mit eingeschleustem Code sind ein schwerwiegendes Security-Problem. Jedoch wurden im Lauf der letzten zehn Jahre einige Mechanismen zum Schutz gegen Angriffe entwickelt. Je nach Betriebssystem und Hardwarearchitektur stehen unterschiedliche Möglichkeiten zur Verfügung. Selbst ohne Betriebsystem und spezielle Hardwareregister auf dem Prozessor ist mit geringem Softwareaufwand ein bedingter Schutz möglich.

8.1 Sicherheitsmechanismen moderner Betriebssysteme

Ein einfacher Ansatz zum Schutz gegen Ausführen von eingeschleusten Daten ist vorher festzulegen welche Speicherbereiche ausführbar sind und welche Speicherbereiche als Daten interpretiert werden sollen. Diesen Ansatz verfolgen mehrere Technologien, manche mit Unterstützung von Hardwarefunktionen. Diese Methoden arbeiten alle völlig transparent für bestehende Programme, das heißt es muss kein Programm neu kompiliert werden.

8.1.1 NX-Bit

Das Prinzip des NX-Bit wurde auf mehreren Prozessorarchitekturen, schon früher bereitgestellt, meist waren dies Server oder Workstation-RISC-Prozessoren sie Sun SPARC, Digital ALPHA oder IBM PowerPC. Der Ausdruck NX-Bit wurde von AMD mit der 64-Bit Athlon Prozessorfamilie eingeführt. Intel implementiert seit dem Pentium 4 „Prescott“ diese Technologie und nennt sie XD, eXecution Disable. Beim NX-Bit handelt es sich um das 63. Bit des Eintrags einer Speicherseite in der Seitentabelle. Ist dieses Bit zu 0 gesetzt kann der Inhalt der Seite ausgeführt werden. Ist das Bit 1 so wird der Inhalt als Datum interpretiert und der Prozessor führt den Inhalt nicht aus.

Allerdings muss das Betriebsystem in der Lage sein diesen Hardwaremechanismus zu nutzen. Eines der ersten Betriebsysteme die dieses Bit nutzten war OpenBSD 3.3, das am 1. Mai 2003 veröffentlicht wurde. OpenBSD implementiert die so genannte W^X-Technologie. W^X steht für Write XOR Execute. Seiten des virtuellen Speichers können entweder geschrieben oder ausgeführt werden. Bemerkenswert ist, dass falls der Prozessor keine Hardwareregister dafür bereitstellt wird dies vom Betriebsystem mit minimalem Mehraufwand emuliert. Ist kein Hardwarebit vorhanden so nutzt OpenBSD für W^X die Eigenschaft des Prozessimages aus, dass ausführbarer Code im niederen Adressraum liegt. Es wird eine Grenze im Adressraum definiert oberhalb derer Speicherinhalt als Daten interpretiert wird und nicht ausführbar ist.

Linux unterstützt den NX-Bit Mechanismus auf 32-Bit CPUs seit Kernelversion 2.6.8 standardmäßig über Ingo Molnars ExecShield-Implementierung.

Windows nutzt das NX-Bit seit WindowsXP Service Pack 2. Allerdings existiert keine Emulation wenn der Prozessor das NX-Bit nicht bereitstellt.

Die Verwendung von Sicherheitsmechanismen auf Hardware und Betriebsystemebene ist besonders sicher, da der Angreifer keinen Zugriff auf die Betriebsystem Verwaltungsdaten der Speicherseiten hat. [WikiNX]

8.1.2 PaX mit Linux

PaX existiert als Patch für Linux Kernel seit Oktober 2000 wurde aber bisher nicht in den Source-Tree übernommen. PaX bietet verschiedene Sicherheitsmethoden, die erste war PAGEEXEC. Diese Sicherheitsmethode beruht darauf Seiten des Virtuellen Speichers als ausführbar zu markieren. PaX kann das NX-Bit nutzen falls dies die Zielarchitektur unterstützt, dies geschieht dann ohne Performanceeinbußen. Ist kein NX-Bit vorhanden, so emulierte PAGEEXEC dieses Verhalten. Auf x86-Prozessoren, auf denen kein NX-Bit existiert wird das Supervisor-Bit dafür verwendet. Auf IA-32-Architekturen hält die MMU typischerweise zwei verschiedene TLB (Translation-Lookaside-Buffer) zur Dereferenzierung von Daten DTLB und Instruktionen, ITLB. Wird auf eine Seite zugegriffen, die noch nicht im ITLB liegt wird ein "Protection Fault" von der MMU an den Kernel gemeldet, der entscheidet dann ob die Ausführung der Seite erlaubt ist und lädt sie, oder ob es sich um eine unzulässige Ausführung einer Datenseite handelt, in diesem Fall wird der Prozess beendet. Ohne NX-Bit ist die PAGEEXEC Methode ein nicht vernachlässigbarer Verwaltungsoverhead. Mit dem `mprotect()` Systemcall kann die Ausführberechtigung einer Speicherseite verändert werden. PaX stell sicher dass eine Seite entweder ausführbar oder beschreibbar ist.

Eine andere Methode ist SEGMEXEC, diese halbiert den Adressraum eines Prozesses, sind Seiten im Adressraum `0-0x5fffffff` sind ausführbar und Seiten von `0x60000000-0xbfffffff` werden nichtausführbar als Datenseiten interpretiert. (`0xc0000000-0xffffffff` sind für den Kernel reserviert). Dieses Vorgehen ist sehr effizient auch wenn kein NX-Bit auf der Zielarchitektur existiert, der Nachteil ist das nur der halbe Adressraum zur Verfügung steht, für Feldgeräte ist dies aber irrelevant, da hier meist keine 3 GB Speicher benötigt werden.

2001 wurde außerdem ASLR (Address Space Layout Randomization) eingeführt. Diese Technik nimmt zufällige Adressen für den Anfang des Stacks und des Heaps beim Start eines Programms, dadurch wird es für Angreifer extrem schwierig Rücksprungadressen zu erraten und diese auf dem Stack abzulegen. [wikiPaX][PAX]

8.1.3 Exec Shield für Linux

Exec Shield bietet Schutz gegen Stack- und Heapbasierte Buffer-Overflow Angriffe und andere Arten von Exploits, die darauf basieren, Speicherstrukturen mit ausführbarem Code zu überschreiben und später dorthin zu springen.

ExecShield arbeitet mit der Eigenschaft des Linux Kernels das Mapping der ausführbaren Seiten im virtuellen Speicher zu verfolgen. Wiederum wird hier eine maximale Speicheradresse des ausführbaren Codes verwendet, das so genannte „exec-limit“. Der Scheduler benutzt bereits dieses exec-limit bei jedem Context-Switch um das Programmcode-Segment zu aktualisieren. Da jeder Prozess ein anderes exec-limit besitzt muss der Scheduler das exec-limit für jeden Prozess aktualisieren. Der Kernel hält die Code-Segment Beschreibungen in einem Cache so dass der Overhead, der dadurch entsteht maximal zwei bis drei Instruktionen lang ist. [InMo2003]

8.2 Sicherheitsmechanismen ohne Betriebsystem

Alle bisher vorgestellten Mechanismen des Schutzes gegen Ausführung von eingeschleustem Code sind Abhängig von Betriebsystemen und Hardwarefunktionen. In Embedded Systems in denen kein Betriebsystem vorhanden ist besteht dennoch die Möglichkeit das Einschleusen von Code erheblich zu erschweren. Im Folgenden werden Compilertechniken vorgestellt die den Stack schützen. Allerdings ist keine der Techniken absolut sicher, es existieren bereits Modelle die erfolgreich die Schutzmechanismen außer Kraft setzen. [GeRi2002]

Ein Problem hierbei ist dass der Quellcode neu kompiliert werden muss, es ist nicht möglich fremde Binärdateien nachträglich zu mit dem Sicherheitsmechanismus auszustatten.

Weiterhin muss erwähnt werden, dass diese Mechanismen zwar mehr oder weniger gut gegen das ausführen von fremdem Code funktionieren, lokale Daten die nach dem übergelaufenen Buffer dennoch verändert werden können und so Integrität kompromittiert wird ohne, dass ein Programmabsturz auftritt.

8.2.1 StackGuard

StackGuard wurde 1997 als Patch für den GNU-C-Compiler entwickelt. Die Idee von StackGuard liegt darin eine Sicherheitsbarriere, ein so genannter Carnary, innerhalb des Stackframes abzulegen. Beim Kompilieren des Quelltextes erstellt der Compiler einen Funktionsprolog der den Carnary direkt vor der gesicherten Rücksprungadresse ablegt. Dazu wird ein passender Funktionsepilog erstellt der den Carnary auf dem Stack gegen ein Original geprüft. Sind die beiden Carnarywerte ungleich so wird das Programm abgebrochen.

Verschiedene Versionen von StackGuard hatten unterschiedliche Ansätze für die Art des Carnary. Version 2.0.1 legte die Konstante `0x000aff0d`, die ASCII-NUL, LF, 255 und CR auf den Stack. Der Gedanke dabei ist dass die Bibliotheksfunktionen, die über Formatstringattacken

angreifbar sind mindestens eines der Zeichen als Terminator erkennen und die Eingabe abbrechen, das heißt kein der Angreifer kann diesen Carnary nicht in seinen Eingabestring legen und danach noch ein Rücksprungadresse ablegen. In früheren Versionen beruhte der Carnary auf einem Zufallswert. Der Zufallswert wird bei jedem Funktionsaufruf neu generiert und im .data-Segment gesichert. Der Angreifer kann den Carnary nicht erraten, da er erst zu Laufzeit generiert wird und sich laufend ändert.

Der Laufzeit-Overhead der durch StackGuard entsteht ist minimal. Lediglich ein Maschinenbefehl im Prolog und zwei Anweisungen im Epilog.

8.2.2 Stackshield

StackShield verfolgt eine andere Idee die Rücksprungadresse zu validieren. Es wird vom Compiler ein zweiter separater Stack in einem andern Speichersegment angelegt. Auf diesem Stack wird im Funktionsprolog eine Kopie der Returnadresse abgelegt. Beim Rücksprung aus der Funktion wird die Adresse auf dem Stack gegen die gesicherte Version geprüft.

Vom Compiler wird standardmäßig eine Array mit 256 Elementen für die gesicherten Rücksprungadressen angelegt, die Größe kann beim kompilieren allerdings eingestellt werden. Ist kein Platz mehr im Array, zum Beispiel wenn die Rekursionstiefe zu groß ist dann wird keine Kopie abgelegt, der Stackpointer dieses Stacks aber dennoch inkrementiert, so dass festgestellt werden kann ab wann die Rücksprungadressen wieder geprüft werden sollen.

StackShield unterstützt aber auch andere Sicherheitsmechanismen, die auf der Aufteilung des Speichers des Prozessimages beruhen. So kann sichergestellt werden, dass Rücksprungadressen immer unterhalb einer maximalen Adressgrenze liegen. Da der Stack meist bei der höchsten Adresse beginnt kann somit verhindert werden dass in den Stack gesprungen wird und dort Daten als Programmcode ausgeführt werden. Auch bei StackShield ist der Laufzeit-Overhead vernachlässigbar gering.

8.2.3 SSP Stack Smashing Protector früher Propolice

Stackshield und Stackguard sichern lediglich die Rücksprungadresse vor einem überschreiben, Funktionsparameter bleiben ungeschützt. Das Hauptproblem wenn lediglich mit Carnaries gearbeitet wird besteht darin dass ein Angriff erst bei der Rückkehr aus einer Funktion bemerkt wird, die Funktion selbst arbeitet nach dem Angriff allerdings mit falschen Daten. Bei IBM wurde von Hioaki Etoh eine Compilertechnik entwickelt die den Framepointer, die lokalen variablen, die Funktionsparameter und die Rücksprungadresse schützt. Diese Technik ist als Patch für gcc 3.4 vorhanden und ist in gcc Version 4.1 implementiert.

Ähnlich zu StackGuard wird hier wiederum wird hier ein zufälliger Carnarywert eingeführt der zu Beginn der Programmlaufzeit erstellt wird. Wurde der Wert verändert wird eine Funktion aufgerufen die Logdaten schreibt und das Programm abgebrochen. Eine zweite Sicherheitsmassnahme ist das umsortieren von Variablen. Lokale Variablen werden so auf dem

Stack angelegt, dass sie vor den Arrays und Strukturen, die Arrays enthalten liegen. Dieses Vorgehen stellt sicher, dass einfache Variablen nicht überschrieben werden können, wird ein Array überflutet, so werden nur die nachfolgenden Puffer überschrieben. Die Dritte Eigenschaft von SSP ist, dass Funktionsparameter kopiert werden. Dieses Kopieren erfolgt vor dem Umsortieren der Variablen, das bedeutet, dass die Argumente der Funktion ebenso geschützt werden. Es werden niemals die originalen Übergabeparameter benutzt sondern nur die Kopien. Dadurch ist die Funktion weitestgehend geschützt. In Abbildung 10 ist die Funktion func1() des Beispielprogrames aus Abbildung 7 dargestellt, die mit der StackGuard Funktion von gcc Version 4.1 kompiliert wurde.

```
08048445 <func1>:
            push   %ebp
            mov    %esp,%ebp
            sub    $0x28,%esp
            mov    %gs:0x14,%eax
            mov    %eax,0xfffffffc(%ebp)
            xor    %eax,%eax
            movl   $0x5,0xffffffec(%ebp)
            mov    0xc(%ebp),%eax
            mov    %eax,0x4(%esp)
            mov    0x8(%ebp),%eax
            mov    %eax,(%esp)
            call   8048493 <func2>
            mov    %eax,0xffffffec(%ebp)
            lea    0xfffffff0(%ebp),%eax
            mov    %eax,(%esp)
            call   80482ec <gets@plt>
            mov    0xffffffec(%ebp),%eax
            mov    0xfffffffc(%ebp),%edx
            xor    %gs:0x14,%edx                  // Vergleich mit Carnary
            je     8048491 <func1+0x4c>           // Korrekter Inhalt
            call   80482fc <__stack_chk_fail@plt> // Wird bei verändretem
                                                  //Carnary erreicht
            leave
            ret
```

Abbildung 10: Funktion func1() mit StackGuard

8.3 Fazit zu Softwareschutzmechanismen

Es existiert leider kein Schutz der absolut vor Angriffe auf Softwareebene schützt. Eine Kombination der vorgestellten Methoden ist sinnvoll, wenn es die Zielarchitektur zulässt. Dadurch wird es für den Angreifer erheblich erschwert die Security und somit auch die Safety des Systems zu kompromittieren. Für die Entwicklung eines sicheren Systems sind regelmäßige Code-Audits des Entwicklerteams unerlässlich. Zusätzlich helfen moderne Compiler mit Warnungen beim übersetzen des Quelltextes. Darüber hinaus sollten Tools die Code automatisch auf Sicherheitslücken überprüfen, wie z.B. „Flawfinder“ [Flaw] oder „Splint“ [Splint] eingesetzt werden um automatisiert Schwachstellen zu finden. Diese Tools sind allerdings keine Garantie für sicheren Code. Generell sollte, so weit möglich, eine sichere Programmiersprache wie Ada eingesetzt werden, um zumindest die Schwachstellen der Sprache zu eliminieren. Fehlerhafte Programmierung wird aber auch dadurch nicht verhindert, diese

Gefahr kann nur durch gute Entwickler, präzise Spezifikation und regelmäßige Kontrolle minimiert werden.

9 Installations- und Benutzungsanleitung des Prototypen

Dieses Kapitel beschreibt den praktischen Teil der Arbeit. Die Installation und Inbetriebnahme von Linux auf der Evaluierungsplatine STK8XXL mit der Minimodulplatine TQM823L der Firma TQComponents. Im ersten Teil wird die Hardware beschrieben, Teil zwei beschreibt die Anforderungen und die Installation des Hostrechners der zu Entwicklung dient. Der dritte Teil erklärt das Vorgehen für die Installation und Konfiguration einer neuen Firmware mit Bootloader auf dem Embedded System. Dieser Teil ist sehr wichtig, da bei der fehlerhaften Installation der Firmware das System beschädigt werden kann, und der Schaden nur vom Hersteller behoben werden kann. Teil vier beschreibt die Installation von Linux in einer Ramdisk des Flashspeichers und den Bootvorgang.

9.1 Hardware

9.1.1 STK 8XXL Entwicklungsplatine

Die Evaluierungsplatine STK8XXL führt lediglich die vom Minimodul bereitgestellten Schnittstellen nach außen und ermöglicht somit die Entwicklung von Software parallel zur Entwicklung der Hardware des späteren Zielsystems. In dieser Arbeit wurde lediglich die Funktionalität folgender Schnittstellen betrachtet.

Die Platine verfügt über zwei RS232 COM Anschlüsse X18 und X19, die primäre (X18) wird als Anschluss für eine Terminalkonsole benutzt und einen Ethernetanschluss für das Laden von Software. Die Eingesetzte Version .400 der Platine verfügt darüber hinaus über einen USB-Anschluss, allerdings ist es nicht gelungen, in der vorhandenen Kernelversion die Treiber für die USB Unterstützung in Linux erfolgreich zu kompilieren.Die genaue Beschreibung der Platine und aller Schnittstellen ist in [TQSTK03] zu finden.

9.1.2 Das Minimodul TQM850L

Das Minimodul TQM850L der Firma TQ-Components ist ein komplettes Embedded System mit Freescale PowerPC Prozessor, Speicher und integrierten Kommunikationsschnittstellen.Die eingesetzte Version TQM850L.300 arbeitet mit dem Prozessor PPC823 mit einer Taktfrequenz von 50 MHz, ist mit 16 MB DRAM und 8 MB Flash ausgestattet. Die Kommunikation ist direkt in den Prozessor integriert. Der Kommunikationsprozessor unterstützt CAN, Ethernet und RS232, welche im Weiteren vor allem bei der Konfiguration eines Kernels wichtig sein werden.

Die Standardeinstellung des primären RS232-Ports (X18), der für die Bootkonsole benutzt wird ist 115200/8N1 für das Kommunikationsprotokoll (115200 bps ohne Parity Bit, 1 Stopbit kein Handshake) Weitere Informationen sind dem Datenblatt in [TQM850L] zu entnehmen.

9.2 Der Hostrechner

Für die Entwicklung von Software und die Bereitstellung der Boot-Images für die Platine, ist ein Rechner mit RS232-Schnittstelle und Ethernetanschluss nötig. Als Betriebsystem wurde Ubuntu Linux [ubuntu] Version 6.06 mit Kernel 2.6.17 eingesetzt. Im Folgenden wird nun beschrieben, welche Pakete installiert sein müssen und wie diese konfiguriert werden.

9.2.1 Beschreibung der Beispielumgebung:

- Hostname des Embedded System "Minilinux"
- IP Adresse 192.168.2.127
- Netmask 255.255.255.0
- Hostrechner 192.168.2.55

9.2.2 Installation des ELDK

Unbedingt erforderlich ist der Embedded-Linux-Development-Kit, im Folgenden ELDK der Firma Denx [ELDK]. Diese Umgebung stellt Crosscompiler und Tools für die Erstellung von Binaries für eine Reihe von Zielsystemen auf einer x86-Hostarchitektur bereit. Hier in der Beispielumgebung soll diese in `/opt/eldk3.1` installiert werden. Zur Installation sind folgende Schritte nötig:

Auf der Installations CD ist ein Skript `'install'` das fast alle nötigen Schritte vornimmt. Es wird empfohlen dieses Skript als `'user'` und nicht als `'root'` auszuführen. Bei der Installation kann der Zielordner mit der Option `'-d'` angegeben werden. Ohne Angabe einer Zielarchitektur werden alle verfügbaren Architekturen installiert. In diesem Beispielfall wird nur ein PowerPC823 benötigt es wird somit die Zielarchitektur `'ppc_8xx'` angegeben.

```
user# ~/> /cdrom/install -d /opt/eldk3.1 ppc_8xx
```

Abbildung 11: Installation ELDK

Achtung: Sicherstellen dass die CDROM mit der `'exec'` Option gemounted ist, sonst ist ein ausführen des Skripts nicht möglich. Viele Linuxdistributionen mounten CD-ROMs standardmäßig mit `'noexec'`

In `/opt/eldk3.1/ppc_8xx` befindet sich nun ein komplettes Root-Dateisystem für das Embedded System. Um dieses als Root-Dateisystem über NFS benutzen zu können müssen noch die Device-Nodes in `/opt/eldk3.1/ppc_8xx/dev` angelegt werden. Dies erfordert Superuserrechte auf dem Hostsystem. Der ELDK bietet für das erstellen der Device-Nodes ein

Skript 'ELDK_MAKEDEV'. Dies muss als 'root' in /opt/eldk3.1/ppc_8xx ausgeführt werden und hat keine Optionen.

```
user# /opt/eldk3.1/ppc_8xx/dev> sudo /cdrom/ELDK_MAKEDEV
```

Abbildung 12: Device-Nodes erstellen

Danach müssen noch Dateiberechtigungen angepasst werden, da manche Werkzeuge des ELDK das SUID-Bit benötigen. Dies geschieht mit dem Skript 'ELDK_FIXOWNER', das wiederum in /opt/eldk3.1/ppc_8xx ausgeführt werden muss. Um die Rechte und Besitzer zu korrigieren und Programme mit SUID-Bit korrekt auszuführen müssen /bin, /sbin und /proc 'root' gehören

```
user# /opt/eldk3.1/ppc_8xx/dev> sudo /cdrom/ELDK_FIXOWNER
user# /opt/eldk3.1/ppc_8xx/ > sudo chown –R root:root /bin
user# /opt/eldk3.1/ppc_8xx/ > sudo chown –R root:root /sbin
user# /opt/eldk3.1/ppc_8xx/ > sudo chown –R root:root /proc
user# /opt/eldk3.1/ppc_8xx/ > sudo chown –R root:root /etc
```

Abbildung 13: Dateirechte anpassen

Zudem ist es nützlich die Dateistruktur des Hostrechners im Root-Verzeichnis des Zielssystems zu replizieren, um beim kompilieren festgelegte Dateipfade benutzen zu können. Dies ist insbesondere für Kernelmodule wichtig.

```
user# /opt/eldk3.1/ppc_8xx/ > mkdir –p ./opt/eldk3.1/ppc_8xx/usr/
user# /opt/eldk3.1/ppc_8xx/ > ln –s /usr/src  \
/opt/eldk3.1/ppc_8xx/usr/src
```

Abbildung 14: Dateistruktur replizieren

Um den Crosscompiler und die Tools zur Erstellung von Programmen auf dem Hostsystem benutzen zu können, muss der User vor der Benutzung einige Umgebungsvariablen anpassen. Die Variable 'CROSS_COMPILE' gibt das Zielsystem an, im erweiterten Pfad sind die Tools für das Hostsystem zu finden. Eine bequeme Variante ist in ~/.profile oder ~/.bashrc diese Zeilen hinzuzufügen.

```
export CROSS_COMPILE=ppc_8xx-
export PATH=$PATH:/opt/eldk3.1/bin:/opt/eldk3.1/usr/bin
```

Abbildung 15: Umgebungsvariablen

Unbedingt erforderlich ist eine Terminalemulation für den Hostrechner. Im Beispielfall wurde das Paket „Kermit“ unter Linux benutzt, für Microsoft-Windows-Hostrechner wird „Teraterm“ [Teraterm] empfohlen. Kermit bietet die Möglichkeit Konfigurationen für unterschiedliche Zielhosts abzuspeichern und besitzt einen reichhaltigen Funktionsumfang.

Beispielkonfiguration für Kermit zum Verbinden über das serielle Device-Node /dev/ttyUSB0. In Anderen Installationen muss /dev/ttyUSB0 durch das entsprechende

Device ersetzt werden, meist ist dies `/dev/ttyS0`. Folgende Einträge müssen in `~/.kermrc` vorgenommen werden:

```
set line /dev/ttyUSB0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
```

Abbildung 16: ~/.kermrc

9.2.3 Installation einer Netzwerkbootarchitektur

Soll das Embedded System über ein Netzwerk gebootet werden, so müssen alle Informationen von einem Hostrechner bereitgestellt werden. Dies geschieht mit den bekannten Verfahren DHCP, TFTP und NFS. DHCP das Dynamic Host Configuration Protokoll ist in [RFC 2131] beschrieben es stellt Informationen über die Netzwerkarchitektur bereit.

TFTP das Trivial File Transfer Protocol ist in den RFCs [RFC1350] und [RFC2347] beschrieben und dient dazu Bootimages des Kernels für das Embedded System bereitzustellen. NFS, das Network-File-System stellt dem Kernel das Root-Dateisystem über ein TCP/IP Netzwerk bereit. Im vorliegenden Beispielfall wurde NFS Version 3 eingesetzt, das in [RFC1813] beschrieben wird.

Für die Installation sind auf Ubuntu Linux sind folgende Pakete erforderlich:

- nfs-common
- nfs-user-server
- tftpd-hpa
- dhcp3-server

9.2.3.1 Konfiguration des TFTP Servers

Die Konfiguration des TFTP-Servers befindet sich in `/etc/default/tftpd-hpa` und enthält folgende Informationen:

```
RUN_DAEMON="yes"
OPTIONS="-l -u root -s /opt/eldk3.1/ppc_8xx/images/"
```

Abbildung 17: /etc/default/tftpd-hpa

`'RUN_DEAMON'` startet den Dienst beim Systemstart, `'OPTION'` sind Argumente für den Server. Interessant ist hier lediglich der Pfad, wo die Kernel-Images zu finden sind, dies wird mit der Option `'-s'` angegeben. Weitere mögliche Optionen finden sich in der Manpage für `tftpd`.

9.2.3.2 Konfiguration des NFS Servers

Der NFS-Server wird über `/etc/exports` konfiguriert.

```
/opt/eldk3.1/ppc_8xx/ 192.168.2.0/255.255.255.0  (rw,no_root_squash,sync)
```

Abbildung 18: /etc/exports

Mit diesem Eintrag wird das Rootdateisystem für das Subnetz `192.168.2.0` exportiert, die Optionen sind auf lesen und schreiben gesetzt, `'rw'`. Die Option `'no_root_squash'` muss gesetzt sein, so dass `'root'` auf dem Zielsystem auch Dateien mit Administratorrechten auf dem Hostsystem ändern kann. Dies stellt kein Sicherheitsrisiko für das Hostsystem dar, da der Superuser des Zielsystems nur innerhalb freigegebenen Ordners operieren kann.

9.2.3.3 DHCP Server

Soll das Zielsystem völlig dynamisch ohne serielle Konsole über das Netzwerk konfiguriert werden bietet der DHCP Server alle nötigen Informationen an. Unter Ubuntu Linux reicht ein Eintrag einer Zone in die Datei `/etc/dhcp/dhcpd.conf.`

```
subnet 192.168.2.0 netmask 255.255.255.0 {
       option routers          192.168.2.55;
       option subnet-mask      255.255.255.0;
       option domain-name      "embedded.sys";

         host PLATINE {
             hardware ethernet       00:D0:93:07:D6:27;
            fixed-address           192.168.2.127;
             option root-path        "/opt/eldk3.1/ppc_8xx";
             option host-name        "Minilinux";
             next-server             192.168.2.55;
             filename                "uImage";
      }
}
```

Abbildung 19: /etc/dhcp/dhcpd.conf

Mit dem Eintrag für dieses Subnetz wird eine feste IP Adresse für die MAC-Adresse des Embedded Systems reserviert und alle Informationen für das Booten bereitgestellt. Im Beispielfall sind `'next-server'`, der TFTP-Server der das Image bereitstellt und Router dieselben, es können auch unterschiedliche Server gewählt werden.

Das Embedded System versucht die Datei `'uImage'` über TFTP vom Server zu laden, diese muss im Verzeichnis das bei der Konfiguration des TFTP-Servers angegeben wurde vorliegen. Die Option `'rootpath'` gibt den Pfad des Rootdateisystems auf dem NFS Server an.

9.2.4 Kernel

Der im ELDK 3.1 enthaltene Linux-Kernel Source-Tree beinhaltet bereits mit Patches für Embedded Systems. Zur Grundkonfiguration hat Denx das Maketarget `'TQM850L_config'`

erstellt. Werden eigene Kernelquellen eingesetzt sollte die Umgebungsvariable `'ARCH=ppc'` gesetzt sein da sonst für die Architektur des Hostrechners konfiguriert wird. Danach wird der Kernel wie gewohnt mit `'make menuconfig'` konfiguriert. Per Default werden sehr viele Treiber und Eigenschaften eingebaut die, nicht unbedingt nötig sind. Die generische Konfiguration sollte an die individuellen Bedürfnisse angepasst werden.

Zuletzt wird über das Maketarget `'uImage'` ein bootfähiges Kernel-Image erstellt.

```
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > make TQM850L_config
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > export ARCH=ppc
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > make menuconfig
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > make dep
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > make uImage
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > make modules
user# /opt/eldk3.1/ppc_8xx/usr/src/linux > sudo make modules_install
\INSTALL_MOD_PATH=/opt/eldk3.1/ppc_8xx/
```

Abbildung 20: ELDK kernel

Das Image `'/opt/eldk3.1/ppc_8xx/arch/ppc/boot/images/uImage'` muss nur noch in das Verzeichnis des TFTP-Servers (`'/opt/eldk3.1/ppc_8xx/images'`) kopiert werden.

9.3 Installation und Konfiguration des Minimoduls

Das Minimodul wird werkseitig mit eigener Firmware dem „TQMonitor“ ausgeliefert. Diese Firmware ist in ihrer Funktionalität allerdings sehr beschränkt. Der TQMonitor dient dazu Software als Intel-Hexformat oder Motorola-Srecord zu laden, in den Flash-Speicher zu schreiben und auszuführen, das Programm bietet allerdings keinerlei Funktionalität für Booten über Netzwerk oder Skripten. Daher wird dringend empfohlen den TQMonitor durch das Open-Source Projekt „U-Boot“ [U-Boot] dem universal Bootloader zu ersetzen.

Die Installationsanleitung wurde mit Hilfe des Denx U-Boot and Linux Guides von Wolfgang Denk [DULG] erstellt. Dort sind weitergehende Informationen zu diesem Thema zu finden.

ACHTUNG: Bei Fehlerhafte Installation kann der Firmware kann das Minimodul unbrauchbar gemacht werden.

9.3.1 Universaler Bootloader U-Boot

Im Folgenden wird beschrieben, wie U-Boot auf dem Hostrechner kompiliert wird und in das Flash-ROM des Minimoduls installiert wird.

9.3.1.1 Kompilieren von U-Boot

Beim Kompilieren von U-Boot Version 1.1.4 traten Probleme auf, die Makefile für Beispieldateien enthält eine nicht erfüllte Abhängigkeit. Das Problem lässt sich durch das Erstellen leerer Dateien mit dem Programm `'touch'` lösen.

```
user# /opt/eldk3.1/ppc_8xx/usr/src/u-boot> touch  hello_world.srec \
timer.srec sched.srec \
      hello_world.bin timer.bin sched.bin
```

Abbildung 21: Makefile workaround

Um die Quellencode Dateien für das Board zu konfigurieren existieren spezielle Maketargets. In diesem Beispielfall ist dies `'TQM850L_config'`.

```
user# /opt/eldk3.1/ppc_8xx/usr/src/u-boot/> make TQM850L_config
```

Abbildung 22: U-Boot-Konfiguration

Danach kann das Paket kompiliert werden. Dabei ist darauf zu achten dass sich die PPC-Crosscompiler im Pfad befinden siehe Installation des ELDK.

```
user# /opt/eldk3.1/ppc_8xx/usr/src/u-boot/> make all
```

Abbildung 23: U-Boot kompilieren

9.3.2 Installation des Bootloaders im ROM

9.3.2.1 Auslesen der Hardwareinformationen

Für das erfolgreiche Einrichten der Software sind Seriennummer und MAC-Adresse des Ethernetcontrollers erforderlich, ohne diese Informationen erlischt eventuell die Garantie des Herstellers für das Minimodul.

Die Informationen sind im ROM des Minimoduls gespeichert und müssen gesichert werden bevor das U-Boot installiert wird. Da die U-Boot-Software in denselben Speicherraum wie der Firmwaremonitor geschrieben wird und die Informationen überschrieben.

Mit dem Kommando `'read <Adresse> <Länge>'` des TQMonitor können Speicherinhalte angezeigt werden.

```
MON:> read 4001FFB0
4001FFB0:  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
4001FFC0:  54 51 4D 38 32 33 4C 44 42 42 41 33 2D 45 35 30  TQM823LDBBA3-E50
4001FFD0:  2E 33 31 33 20 31 31 33 31 35 34 30 33 20 30 30  .313 11315403 00
4001FFE0:  44 30 39 33 30 37 44 36 32 37 20 34 00 00 00 00  D09307D627 4....
4001FFF0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Abbildung 24: TQMonitor – Hardwareinformation

Aus diesem String lassen sich die 4 Informationen des Minimoduls ableiten, jede durch ein Leerzeichen getrennt:

- Modul Typ & Revision: TQM823LDBBA3-E50.313
- Seriennummer: 11315403

- MAC Adresse: 00:D0:93:07:D6:27
- Anzahl zusätzlich reservierter MAC Adressen: 4

9.3.2.2 Testdownload:

Zunächst muss ein unbenutzter Speicherbereich des Flashs gelöscht werden, um dort das U-Boot-Image abzulegen. Das Flash-ROM ist über Memorymapping in den Adressraum des Minimoduls eingebunden und beginnt bei `0x4000000`.

```
MON:> erase 40100000 4013FFFF
* Erasing FLASH from 40100000h to 4013FFFFh
* Please wait
```

Abbildung 25: TQMonitor – Flash löschen

Jetzt kann die Software mit Offset `0x100000` das Flash ROM geladen werden. Die Srecord Datei kann als ASCII-Format mit der Terminalemulationssoftware geladen werden.

Platine für Dateiempfang vorbereiten

```
MON:>load 100000 flash
ready for s-record downloading into FLASH
```

Abbildung 26: TQMonitor – Dateiempfang

Datei senden, diese Anweisung ist für Kermit auf dem Hostrechner.

```
[CTRL \c]
> send U-Boot.srec
> connect
```

Abbildung 27: U-Boot – Upload

Download überprüfen

```
MON:> read 40100000
40100000:  27 05 19 56 50 50 43 42 6F 6F 74 20 31 2E 30 2E  '..VPPCBoot 1.0.
40100010:  30 2D 70 72 65 32 20 28 4A 75 6E 20 20 33 20 32  0-pre2 (Jun  3 2
40100020:  30 30 31 20 2D 20 32 33 3A 35 38 3A 34 30 29 00  001 - 23:58:40).
40100030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
\...
```

Abbildung 28: TQMonitor – Download prüfen

9.3.2.3 U-Boot installieren

Jetzt kann das U-Boot an die Startadresse des Flash-Speichers kopiert werden. Dazu muss der Schreibschutz für den Firmwaremonitor aufgehoben werden.

```
MON:>protect 1234
* Protection for sectors containing MON8xx disabled
```

Abbildung 29: TQMonitor – Schreibschutz deaktivieren

```
MON:>erase 40000000 4003ffff
* Erasing FLASH from 40000000h to 4003FFFFh
* Please wait
```

Abbildung 30: TQMonitor – Flash löschen

```
MON:>load 0 flash
* Ready for srecord download to FLASH ...
[CTRL-\ c]
>send U-Boot.srec
1 2 3 4 5 6 7 8 9 10 11 12 ...
\...
\... 6619 6620 6621 6622 6623
[file transfer complete]
>connect

* Start address 40000000
MON:>
```

Abbildung 31: TQMonitor – U-Boot laden

Wieder sollte der Ladevorgang verifiziert werden:

```
MON:>read 40000000
40000000:  27 05 19 56 50 50 43 42 6F 6F 74 20 31 2E 30 2E '..VPPCBoot 1.0.
40000010:  30 2D 70 72 65 32 20 28 4A 75 6E 20 20 33 20 32 0-pre2 (Jun 3 2
40000020:  30 30 31 20 2D 20 32 33 3A 35 38 3A 34 30 29 00 001 - 23:58:40).
40000030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
\...
MON:>
```

Abbildung 32: TQMonitor – Installation verifizieren

Die Firmware ist nun installiert und startet automatisch nach dem Reset

ACHTUNG: FALLS IRGENDETWAS SCHIEF GING, KEIN BOARDRESET durchführen sondern den alten Monitor wiederherstellen.

9.3.2.4 Wiederherstellen des Firmwaremonitors

Zunächst den Flashspeicher löschen

```
MON:>erase 40000000 4003ffff
* Erasing FLASH from 40000000h to 4003FFFFh
* Please wait
```

Abbildung 33: TQMonitor – Flash löschen

Die Firmware, die im RAM noch läuft wieder an die richtige Stelle im Flash kopieren.

```
MON:>copy monitor
Copy monitor
```

Abbildung 34: TQMonitor – Monitor wiederherstellen

Nun müssen noch die Hardwareinformationen wiederhergestellt werden.

```
MON:>sethwi TQM860LCB0A3SR50.202 10134873 00D093001234 4
* Hardware information written to 4001FFC0
MON:>
```

Abbildung 35: TQMonitor – Hardwareinformationen wiederherstellen

9.3.3 Umgang mit U-Boot

9.3.3.1 Die Kommandozeile

U-Boot bietet einige elementare Befehle zur Administration des Embedded Systems. Zu jedem Kommando gibt es eine kurze Beschreibung, die mit `'help <Kommando>'` eingesehen werden kann. `'help'` ohne weiteres Argument gibt eine Liste aller verfügbarer Befehle aus. Das Verhalten wird über Umgebungsvariablen gesteuert. Wichtig für das Laden von Images in das Flash Rom sind folgende Befehle:

Laden einer Datei ins RAM. Als Zieladresse bieten sich Adressen über `0x100000` an, die untere Adressbereich wird von U-Boot belegt.

```
=> loadb <Zieladresse>
```

Abbildung 36: U-Boot – loadb

Löschen des Flashspeichers

```
=> erase <Startadresse> <Endadresse>
```

Abbildung 37: U-Boot – erase

Kopieren vom RAM ins Flash

```
=> cp.b <Quelladresse> <Zeiladresse> <Anzahl von Bytes>
```

Abbildung 38: U-Boot – cp

Achtung: hier unterscheidet sich U-Boot von der, im [DULG] beschriebenen Vorgehen. Es muss das Suffix `'.b'` angegeben werden, da sonst Worte und nicht Bytes kopiert werden, was zu Fehler wie "copy outside of available ROM" oder "Flash not erased" führt.

9.3.3.2 Umgebungsvariablen

Alle gesetzten Variablen können mit `'printenv'` eingesehen werden. Neue Variablen werden mit `'setenv <Variable> <Wert>'` gesetzt. Soll eine Variable wieder gelöscht werden kann dies mit `'setenv <Variable>'` erreicht werden. Um die Variablen in das ROM zu schreiben, bietet U-Boot den Befehl `'saveenv'`. Dieser Befehl hebt den Schreibschutz für geschützte ROM-Sektoren auf, schreibt die Variablen und setzt den Schutz wieder.

Über Umgebungsvariablen lassen sich auch einfache Batchskripte erstellen. Diese Batchskripte können danach mit `'run <Variable>'` ausgeführt werden. Ein Beispiel dazu ist das in Abbildung 39 aufgeführte `'net_nfs'.` Bei `'run net_nfs'` werden folgende Batchskripte der Reihe nach ausgeführt:

- Über tftp wird vom Server die Datei mit dem Namen `${bootfile}` ins RAM an die Stelle `0x200000` geladen. `${bootfile}` wird durch den Inhalt der Umgebungsvariable `'bootfile'`ersetzt.
- das Skript `'nfsargs'` wird ausgeführt, das die Variable `'bootargs'`, die Bootparameter für Linux setzt.
- Das Batchskript `'addip'` setzt zusätzliche Bootparameter für den Kernel zur statischen Netzwerkkonfiguration.
- Zuletzt wird der Befehl `'bootm'` ausgeführt, der dann das ins RAM kopierte Kernel-Image startet.

Beim Einschalten des Systems wird automatisch zuerst das Skript `'preboot'` ausgeführt. Danach wird `${bootdelay}` gewartet, und dann das Skript `'bootcmd'` ausgeführt. Das bedeutet, dass die Variable `'bootcmd'` so gesetzt werden muss, dass das System in der Lage ist zu starten.

```
=> printenv
addip=setenv bootargs ${bootargs}
ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off panic=1
baudrate=115200
bootargs=root=/dev/nfs rw nfsroot=: ip=::::::off panic=1
bootcmd=run static_ip; run net_nfs
bootcount=10
bootdelay=5
bootfile=uImage
ethact=SCC ETHERNET
ethaddr=00:D0:93:07:D6:27
flash_nfs=run nfsargs addip;bootm ${kernel_addr}
flash_self=run ramargs addip;bootm ${kernel_addr} ${ramdisk_addr}
gatewayip=192.168.2.55
hostname=Minilinux
ipaddr=192.168.2.127
kernel_addr=40400000
loads_echo=1
netdev=eth
netmask=255.255.255.0
net_nfs=tftp 200000 ${bootfile};run nfsargs addip;bootm
nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}
preboot=echo; echo Welcome to the embedded world; echo booting static
```

```
configuration; echo
ramargs=setenv bootargs root=/dev/ram rw
ramdisk_addr=40100000
rootpath=/opt/eldk3.1/ppc_8xx/
serial#=TQM823LDBBA3-E50.313 11315403
serverip=192.168.2.55
static_ip=setenv ipaddr 192.168.2.127;setenv netmask 255.255.255.0;setenv
gatewayip 192.168.2.55;setenv hostname Minilinux;
static_nfs=setenv bootfile uImage; setenv serverip 192.168.2.55;setenv
rootpath /opt/eldk3.1/ppc_8xx/
stderr=serial
stdin=serial
stdout=serial
=>
```

Abbildung 39: U-Boot – printenv

9.3.4 Booten über das Netzwerk

Sind die Dienste auf dem Hostrechner gestartet reicht es im U-Boot den DHCP Client, `'dhcp'` zu starten und das Skript `'net_nfs'` auszuführen.

Leider hat `'root'` auf dem ELDK kein Passwort. Dazu muss man noch die Datei `'/opt/eldk3.1/ppc_8xx/etc/passwd'` ändern. Das einfachste vorgehen ist auf dem Hostrechner ein Passwort ändern und den Passwort-Hashwert in die Datei einzutragen:

Zum Beispiel folgender Eintrag entspricht dem Passwort „secret“ mit SHA-1 gehasht:

```
root:$1$hPvjGcL4$ZzIMk2ezSiHD2nfNVjUXI/:0:0:root:/root:/bin/bash
```

Abbildung 40: passwd (root)

9.4 Installation von Linux im Flash

Im Embedded-Bereich mach es wenig Sinn einen kompletten PC als Bootserver für die Feldgeräte zu verwenden. Meist wird ein relativ kleines Flash-ROM als Speicher für das Betriebsystem eingesetzt. Das TQ-Minimodul TQM823L bietet für diesen Zweck zwei Flash-Speicherbänke mit je 4 MB Speicher. Dieser Speicherplatz ist ausreichend, um ein Betriebsystem wie Linux mit dazugehörigen Werkzeugen zu installieren.

Im Folgenden wird nun gezeigt wie ein Kernel und ein Rootdateisystem vorbereitet, zusammengestellt und im Flash-ROM installiert wird.

9.4.1 Die Ramdisk

Im ELDK sind fertige Ramdisk-Images für PPC zu finden. Diese sind aber wie der Kernel sehr generisch und die Toolversionen veraltet. Es empfiehlt sich die Images für den eigenen Bedarf anzupassen.

Die Programme in der Ramdisk beruhen alle auf „BusyBox“ [Busybox]. BusyBox ist eine so genannte „multi-call binary“, die Funktionalität vieler kleiner Systemprogramme vereint. Das Programm wird über symbolische links aufgerufen. So sind zum Beispiel die Befehle `'ls'` oder `'ps'` nur ein link auf `/bin/busybox`. Wird `'ps'` aufgerufen wird die Prozesstabelle ausgegeben, wird der link `'ls'` aufgerufen so wird der Inhalt des Ordners angezeigt. Diese Technik ermöglicht es alle notwendigen GNU/Linux-Standardwerkzeuge in einer Datei mit einer Größe von nur 600kB bereitzustellen.

9.4.1.1 Anpassen einer Ramdisk:

Die von Denx im ELDK bereitgestellte Image Datei `'ramdisk_image.gz'` ist mit `'gzip'` gepackt und muss zunächst entpackt werden. Danach kann die Datei als Loopback-Device gemountet und wie ein reguläres Laufwerk benutzt werden. zum Mounten eines Loopback-Devices sind in der Regel Superuserrechte nötig.

```
user# /opt/eldk3.1/ppc_8xx/images > gzip -d ramdisk_image.gz
user# /opt/eldk3.1/ppc_8xx/images > sudo mount -o loop  ramdisk_image /mnt
```

Abbildung 41: ELDK – Ramdisk mount

In `/mnt` liegt nun ein standard-Linux-Root-Dateisystem. Zunächst sollte dringend die `/mnt/etc/passwd` Datei editiert werden, da DENX zwar Passwörter vergeben hat, diese aber nicht angibt. Siehe auch Booten über das Netzwerk.

Die Beispielapplikation von DENX `/usr/bin/application` wird von `'init'` beim Booten des Systems gestartet und von `init` überwacht. Dieses Vorgehen ist im Embedded Bereich Sinnvoll da so, wenn die Software abstürzt sie sofort wieder neu gestartet wird. In System V ähnlichen Systemen wie Linux wird dies über die Datei `/etc/inittab` erreicht. dort findet sich der Eintrag:

```
# Start user application
# ::respawn:/bin/application
```

Abbildung 42: ELDK – Ramdisk /etc/inittab

Die Beispielapplikation kann gelöscht werden und der Eintrag auskommentiert werden. Sinnvoll wäre zum Beispiel die Installation eines kleinen Webservers, auch das beinhaltet BusyBox, der über `'init'` gestartet wird.

Die Konfiguration einer neuen BusyBox ist relativ einfach es steht eine „ncurses“ basierte Menuoberfläche, ähnlich der des Linux Kernels, zur Verfügung. Die dort wichtigste Option ist die Auswahl des Crosscompilers unter, in diesem Beispielfall:

```
BusyBox Settings -->
        Build Options -->
             (/opt/eldk3.1/usr/bin/ppc_8xx-) Cross Compiler prefix
```

Abbildung 43: BusyBox – Crosscompiler

Es kann auch auftreten, dass wenn zu viele Werkzeuge mitkompiliert werden, dass das Dateisystem meldet, es sei kein Speicherplatz mehr vorhanden. Dieser Fehler ist unter Umständen irreführend und nicht richtig, es ist Speicherplatz vorhanden Aber es sind keine „inodes“ mehr frei. In diesem Fall sollte zunächst versucht werden bei der Konfiguration die symbolische Links, die je einen inode belegen, durch Hardlinks zu ersetzen.

```
BusyBox Settings -->
        installation options -->
                     Applets links (as hard-links)
```

Abbildung 44: BusyBox – Hardlinks

Danach muss BusyBox kompiliert und in die Ramdisk installiert werden.

```
user# /opt/eldk3.1/ppc_8xx/usr/src/busybox> make && sudo make install  \
PREFIX=/mnt
```

Abbildung 45: BusyBox kompilieren

Sind alle gewünschten Änderungen am Dateisystem vorgenommen, wird es mit `'umount'` entfernt und mit `'gzip'` gepackt. Danach muss aus der Datei wieder eine Ramdisk erstellt werden, da heißt es wird ein Dateiheader mit Name, Prüfsummen und anderen Metainformationen dazu erstellt. ELDK bietet dazu das Werkzeug `'mkimage'`.

```
user# /opt/eldk3.1/ppc_8xx/images >sudo umount /mnt
user# /opt/eldk3.1/ppc_8xx/images >gzip -v9 ramdisk_image
user# /opt/eldk3.1/ppc_8xx/images >mkimage -T ramdisk -C gzip –n \
 <Neuer Name> -d ramdisk_image.gz uRamdisk
```

Abbildung 46: ELDK – Ramdisk erstellen

Dieses Image kann über die serielle Verbindung an einen freien Speicherplatz im Flash geladen werden. Zunächst muss im U-Boot das Flash-ROM gelöscht werden, danach wird das Image ins RAM des Embedded Systems geladen und von dort in das Flash-ROM geschrieben.

```
=> erase 40100000 403FFFFF
.............. done
Erased 24 sectors
=> loadb 100000
## Ready for binary (kermit) download to 0x00100000 at 115200 bps...
[CTRL-\ c]
```

```
(Back at localhost)
------------------------------------------------
(/home/user) C-Kermit>send /opt/eldk3.1/ppc_8xx/images/uRamdisk
(/home/user) C-Kermit>connect

## Total Size      = 0x001a5c72 = 1727602 Bytes
## Start Addr      = 0x00100000

=> cp.b 100000 40100000 1A5C72     # der letzte Parameter is die
                                   # Total Size
Copy to Flash... done

=> imls
Image at 40100000:
   Image Name:   Universal
   Created:      2006-11-22   8:58:26 UTC
   Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
   Data Size:    1727538 Bytes =  1.6 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
Image at 40400000:
   Image Name:   Linux-2.4.25-min
   Created:      2006-09-23  13:37:26 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    642526 Bytes = 627.5 kB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
=>
```

Abbildung 47: U-Boot – Images laden

Auf gleiche Weise kann ein Kernel-Image an eine andere Stelle im Flash geschrieben werden. Hier in der Beispielkonfiguration wurde der Kernel an die Adresse `0x40400000` geladen.

9.4.1.2 Der CAN Bus Treiber

Als CAN Bustreiber wurde Allesandro Rubinis ocan-Treiber Version 0.92 verwendet der als Quellcode vorliegt [ocan]. Zum Kompilieren für das Embedded System sind einige Vorkehrungen nötig. Zunächst kann der Treiber auf dem Entwicklungsrechner mit dem Crosscompiler übersetzt werden. Dazu ist lediglich erforderlich in der Makefile die Variable `'KERNELDIR'` auf den Pfad der verwendeten Kernelquellen zu setzen. Weiterhin sollte überprüft werden, ob die Variable `'DEBUG'` auskommentiert ist, wird mit Debuginformationen übersetzt ist Größe des Treibers 1,3 MB was, für eine Ramdisk zu groß ist.

```
#DEBUG=y
#ifndef KERNELDIR
KERNELDIR= /opt/eldk3.1/ppc_8xx/usr/src/linux
#endif
```

Abbildung 48: Ocan – Makefile

Mit dem Befehl `'make clean && make ARCH=ppc'` wird der Treiber übersetzt.

Die Installation, insbesondere auf einer Ramdisk ist problematischer, da direkt im Ramdisk-Image einige Werkzeuge fehlen. Außerdem ist es nicht möglich, das Root-Dateisystem der Ramdisk über die gebootete Ramdisk selbst dauerhaft zu ändern.

Der einfachste Weg ist es, das Embedded System über ein Netzwerk mit demselben Kernel zu booten, der auch für die Stand-Alone-Installation mit Ramdisk verwendet wird. Danach kann per Telnet oder mit der seriellen Konsole `'make install'` auf dem Embedded System ausgeführt werden. `'make install'` legt das Kernelmodule in `'$NFS_ROOT/lib/modules/$KERNELVERSION/misc'`, und die Userspace-Tools in `'/usr/local/bin'.`

Mit dem Befehl `'ocan_load'` wird über `'insmod'` das Modul `'ocan'` geladen und dynamisch Device-Nodes in `/dev/` erstellt.

```
root@MiniLinux #/usr/src/ocan/ > make install
root@MiniLinux #~/ > ocan_load type=8,8 irq=8,8 base=1,2
```

Abbildung 49: Ocan – installieren und laden

Die installierten Dateien müssen nun auf ein Ramdisk-Image gebracht werden. Das Ramdisk-Image wird wie in Abbildung 41 gezeigt nach `'/mnt'` gemountet. Danach muss auf dem Image zuerst ein Kernelmodulverzeichnis mit den benötigten `'depmod'` Informationen erstellt werden. Der einfachste Weg ist es die Module dort mit Hilfe der Kernel-Makefile zu installieren.

```
user #/opt/eldk3.1/ppc_8xx/usr/src/linux/ > make modules_install \
INSTALL_MOD_PATH=/mnt/
```

Abbildung 50: Ocan – Kernelmodulverzeichnis

Danach sind nur noch die Device-Nodes und die Userspace-Tools auf das Ramdisk-Image zu kopieren. Die Devicenodes wurden von `'ocan_load'` auf dem Embedded System erstellt und sind somit auf dem Hostrechner in `'/opt/eldk3.1/ppc_8xx/dev/ocan/'` angelegt. Davon sind nur die Device-Nodes `'a*'` benutzt. Es reicht somit diese auf das Ramdisk-Image zu kopieren.

```
user# ~/> sudo mkdir –p /mnt/dev/ocan
user# ~/> sudo cp /opt/eldk3.1/ppc_8xx/dev/ocan/a* /mnt/dev/ocan/
user# ~>  sudo cp / opt/eldk3.1/ppc_8xx/usr/local/bin/ocan* /mnt/usr/bin/
```

Abbildung 51: Ocan – Device-Nodes und Userspace-Tools

Das skript `'ocan_load'` muss nun noch angepasst werden. Da keinerlei Device-Nodes mehr erstellt werden müssen. Es reduziert sich nun, da das Modul installiert und die Device-Nodes angelegt sind auf:

```
#!/bin/sh
insmod ocan type=8,8 irq=8,8 base=1,2
```

Abbildung 52: ocan_load auf der Ramdisk

Nun kann die die Ramdisk, wie in Abbildung 46 gezeigt, gepackt und installiert werden.

9.4.2 Der erste Start aus dem Flash

Sind die Images des Kernels und der Ramdisk installiert müssen gegebenenfalls die Umgebungsvariablen `'kernel_addr'` und `'ramdisk_addr'` in U-Boot angepasst werden. Soll Das Minimodul beim Einschalten vom Flash booten, so ist noch die Umgebungsvariable `'bootcmd'` anzupassen. Für einen ersten Test bietet sich allerdings an, die Variablen von Hand zu setzten. Dem Kernel wird die Netzwerkkonfiguration über Bootparameter statisch übergeben. Dies geschieht im vorliegenden Beispielfall über die Batchskripte `'addip'` und `'ramargs'`, die beide vom Batchskript `'flash_self'` aufgerufen werden. Diehe dazu auch Abbildung 39.

```
=> run flash_self
## Booting image at 40400000 ...
   Image Name:   Linux-2.4.25-min
   Created:      2006-09-23  13:37:26 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    642526 Bytes = 627.5 kB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
## Loading RAMDisk Image at 40100000 ...
   Image Name:   Universal
   Created:      2006-11-22   8:58:26 UTC
   Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
   Data Size:    1727538 Bytes =  1.6 MB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Loading Ramdisk to 00e09000, end 00faec32 ... OK
Linux Version 2.4.25-min (ruschi@Thinkpad) (gcc version 3.3.3 (DENX ELDK
3.1.1 3.3.3-10)) #10 Sat Sep 23 09:35:27 AMT 2006
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/ram rw
ip=192.168.2.127:192.168.2.55:192.168.2.55
:255.255.255.0:Minilinux::off panic=1
Decrementer Frequency = 187500000/60
Calibrating delay loop... 49.86 BogoMIPS
Use 'Preset loops_per_jiffy'=249344 for preset lpj.
Memory: 12952k available (1116k kernel code, 360k data, 60k init, 0k
highmem)
Dentry cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode cache hash table entries: 1024 (order: 1, 8192 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
```

```
Starting kswapd
JFFS2 version 2.2. (C) 2001-2003 Red Hat, Inc.
CPM UART driver version 0.04
ttyS0 at 0x0280 is on SMC1 using BRG1
ttyS1 at 0x0380 is on SMC2 using BRG2
pty: 256 Unix98 ptys configured
Found 2x16bit 4MByte CFI flash device of type AMD/Fujitsu standard at
40000000
Registered flash device /dev/flasha (minor 0, 4 partitions)
Found 2x16bit 4MByte CFI flash device of type AMD/Fujitsu standard at
40400000
Registered flash device /dev/flashb (minor 8, 2 partitions)
Status LED driver $Revision: 1.0 $ initialized
eth0: CPM ENET Version 0.2 on SCC2, 00:d0:93:07:d6:27
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
TQM8xxL0: Found 2 x16 devices at 0x0 in 32-bit mode
TQM8xxL0: Found 2 x16 devices at 0x400000 in 32-bit mode
 Amd/Fujitsu Extended Query Table at 0x0040
number of CFI chips: 2
cfi_cmdset_0002: Disabling erase-suspend-program due to code brokenness.
TQM flash bank 0: Using static image partition definition
Creating 7 MTD partitions on "TQM8xxL0":
0x00000000-0x00040000 : "u-boot"
0x00040000-0x00100000 : "kernel"
0x00100000-0x00200000 : "user"
0x00200000-0x00400000 : "initrd"
0x00400000-0x00600000 : "cramfs"
0x00600000-0x00800000 : "jffs"
0x00400000-0x00800000 : "big_fs"
CPM load tracking driver $Revision: 1.0 $
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 2048)
IP-Config: Complete:
      device=eth0, addr=192.168.2.127, mask=255.255.255.0,
gw=192.168.2.55,
     host=Minilinux, domain=, nis-domain=(none),
     bootserver=192.168.2.55, rootserver=192.168.2.55, rootpath=
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 1687k freed
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 60k init
init started:  BusyBox v1.2.2 (2006.11.22-08:31+0000) multi-call

BusyBox v1.2.2 (2006.11.22-08:31+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ $
```

Abbildung 53: Erster Boot aus dem Flash

10 Erfahrungen und Probleme

10.1 Erfahrungen

10.1.1 Vorgehensmodell

Während der Arbeit habe ich zunächst gelernt, in wissenschaftlichen Arbeiten zu recherchieren und ohne fremde Hilfe Informationen zu organisieren. Dadurch, dass ich meine Arbeit in Brasilien durchgeführt habe und keinen direkten Kontakt mit meinen Betreuern in Deutschland hatte, habe ich meine Fähigkeiten selbstständig und eigenverantwortlich zu arbeiten, weiter ausgebaut.

Das IAS-Vorgehensmodell für konzeptionelle Arbeiten hat mir zuerst persönlich wenig geholfen. Das Phasenmodell war nur schwierig auf meine Arbeit anwendbar. Die Trennung von Grundlagen und Konzeption war sehr schwierig, da in weiten Bereichen meiner Arbeit die Konzeption nur eine Vertiefung der Grundlagen war. Ich war am Anfang der Ansicht, dass viel unnötige Arbeit verrichtet werden muss. Erst später erkannte ich die Vorteile des IAS-Vorgehensmodells und der Einteilung in Phasen, die es im Wesentlichen erleichtern, strukturiert und diszipliniert zu arbeiten. Ich bin der Ansicht, dass das Vorgehensmodell sehr gut geeignet ist, um Studenten strukturiertes Arbeiten zu lehren.

Am Anfang war es mir nicht klar, wozu eine einzelne Person einen Projektplan mit MS-Project benötigt, der Aufwand einen solchen Projektplan zu erstellen, erschien mir unverhältnismäßig zum Nutzen. Ich war der Ansicht, dass die Aufstellung eines solchen Plans mit Ressourcenverwaltung und Arbeitspaketabhängkeiten nur in Teams und bei Arbeiten Sinn macht, die in einer ähnlichen Form zuvor schon erledigt wurden. Allerdings musste ich auch hier erkennen, dass die Erstellung eines komplexeren Projektplanes auch für Kleinprojekte nützlich ist. Sie verhindert, dass einzelne Teile der Arbeit zu lange dauern und der vorgesehene Endtermin der Arbeit nicht eingehalten werden kann. Außerdem wird durch einen Projektplan das disziplinierte Arbeiten verstärkt. Für mich war es das erste Mal mit einem komplexen Projektplan zu arbeiten, ich denke ich habe dadurch einige Modelle erlernt, Arbeiten besser zu organisieren.

Was mir persönlich am Vorgehensmodell gefällt, ist das die Anforderungen zu Beginn genau festgelegt werden und sich während der Arbeit nicht ändern.

10.1.2 Inhaltlich

Während der Studienarbeit habe ich meine Kenntnisse im über den Vorgang des Erstellens und Ausführens von Maschinencode aus Quelltext stark vertieft. Ich habe mich intensiv mit

Möglichkeiten, Grenzen und Funktionalität von Entwicklungs- und Debugwerkzeugen, wie dem GNU-C-Compiler aus der „gcc“, dem Debugger „gdb“ und verschiedenen Disassemblern und Werkzeugen zur Bearbeitung und Analyse von Programmen beschäftigt.

Auch meine bereits fundierten GNU/Linux-Kenntnisse konnte ich sehr gut einsetzen und vertiefen. In dieser Arbeit bin ich das erste Mal in Kontakt mit Linux auf Embedded Systems gekommen. Erstmals habe ich Systeme installiert und konfiguriert, die aus Ramdisks oder dem Netzwerk starten.

Dieses Wissen wird mir in vielen zukünftigen Arbeiten sehr nützlich sein.

10.2 Probleme

Die Hauptprobleme bei meiner Studienarbeit waren Probleme mit der Infrastruktur in Brasilien. Es war manchmal unmöglich zu arbeiten, weil einen halben Tag lang keine Internetverbindung des Campus bestand oder der Storm ausgefallen war. Weiterhin war es unmöglich, in Manaus ein Nullmodemkabel zur seriellen Verbindung zu kaufen, nachdem ich in vier verschieden Läden danach gesucht habe und mir niemand weiterhelfen konnte, musste ich ein solches Kabel selbst löten. Mit den Microsoft-Word-Makros zur Dokumentverwaltung gab es Anfangs große Probleme, die sich aber durch Updates der Software beseitigen ließen.

Literaturverzeichnis

[LaGö1999] Lauber, Rudolf; Göhner, Peter: Prozessautomatisierung 1. 3. Aufl., Berlin, Heidelberg, New York: Springer-Verlag, 1999.

[FeGu2006] Felix Gutbrodt: IT-Sicherheit auf der Feldebene von Automatisierungssystemen

[GöPA1] Göhner, Peter: Umdruck zur Vorlesung Prozessautomatisierung 1

http://www.ias.uni-stuttgart.de/pa1/

[DaKa2002] Kalinsky, David: Introduction to Serial Peripheral Interface – Embedded Systems Design

http://www.Embedded.com

[AtSPI2000] Atmel Corporation: In System Programming 11/2000 – Revision 0943C-11/00 http://www.atmel.com

[TIS2000] Tool Interface Standards – Portable Formats Specification Version 1.1 : ELF Executable and Linkable Format

http://www.x86.org/intel.doc/tools.html

[DoLe2000] Doug Lea: A Memory Allocator

Beschreibung : http://gee.cs.oswego.edu/dl/html/malloc.html

Quellcode mit Dokumentation: ftp://g.oswego.edu/pub/misc/malloc.c

[IEEE1149.1] IEEE 1149.1 http://www.ieee.org

[MSNT4] Microsoft Corporation: Windows NT4 Eula

download.microsoft.com/download/platformsdk/Comctl32/5.80.2614.3600/W9XNT4/EN-US/license.htm

[StZe1995] Zeigler, Stephen F. Ph.D.: Comparing Development Costs of C and Ada

http://www.adaic.com/whyada/ada-vs-c/cada_art.html

[DULG] Denx GmbH - Wolfgang Denk : Denx U-Boot and Linux User Guide

http://www.denx.de/wiki/DULG/Manual

[BusyBox] http://www.busybox.net

[TQSTK03] TQ Components GmbH : TQSTK Benutzerhandbuch

http://www.tq-components.de/

[TQM850L] TQ Compontents GmbH : TQM850L Benutzerhandbuch

http://www.tq-components.de/

[ubuntu] http://www.ubuntulinux.org

[RFC 2131] Internet Engineering Task Force – R. Dorms:

Dynamic Host Configuration Protocol 03/1997

http://www.ietf.org/rfc/rfc2131.txt

[RFC1350] Internet Engineering Task Force – K. Sollins:
The TFTP Version 2 07/1992
http://www.ietf.org/rfc/rfc1350.txt

[RFC2347] Internet Engineering Task Force – G. Malkin, A. Harkin:
TFTP Options 05/1998
http://www.ietf.org/rfc/rfc2347.txt

[RFC1813] Internet Engineering Task Force - G. Calghan, B. Pawlowski. G. Staubach:
NFS Version 3 Protocol Specification 06/1995
http://www.ietf.org/rfc/rfc1813.txt

[U-Boot] http://sourceforge.net/projects/u-boot

[Teraterm] http://hp.vector.co.jp/authors/VA002416/teraterm.html

[ocan] Alessandro Rubini Ocan driver
http://www.linux.it/~rubini/software/

[Abr91] Abraham, D. G. et al.: Transaction Security System. IBM Systems Journal, Vol. 30, No. 2, 1991

[SrArSc2004] Srivaths Ravi, Anand Raghunathan and Srimat Chakradhar: Tamper Resistance Mechanisms for Secure Embedded Systems NEC Laboratories America, Princeton, NJ 08540

[MaKu1998] Markus G. Kuhn: Cipher Instruction Attack on the Secure Microcontroller DS5002FP – IEEE TRANSACTIONS ON COMPUTERS, Vol. 47, No. 10, Oktober 1998

[DS2003] Dallas Semiconductors Application Note 2033 SRAM Based Microcontroller optimizes Security - 2004

[DS5240] Dallas Semiconductors Datasheet 5240 High Speed Secure Microcontroller 2003

[InMo2003] Ingo Molnar: "Exec Shield" new Linux security feature
Linux Kernel Mailing list 05/2003

[AlOn1996] Aleph One: Smashing the Stack for Fun an for Profit – Phrack Magazin 7(49) 11/1996

[JP1986] JP: Advanced Doug lea's malloc exploits – Phrack Magazin 7(49) 11/1996

[GeRi2002] Gerado Richarte: Four different tricks to bypass StackGuard and Stackshield Protection Core Security Technologies - 06/2002

[PaX] PaX Project http://pax.grsecurity.net/

[wikiNX] Wikipedia.org: NX-Bit

[wikiPaX] Wikipedia.org : PaX

[FIPS] Federal Information Processing Standards Publication 140-1
Security Requirements for Cryptographic Modules Januar 1994

[SeSk2005] Sergei P. Skorobogatov: Semi-invasive attacks – A new approach to hardware security analysis. University of Cambridge 2005 UCAM-CL-TR-630 ISSN 1476-2986

[Flaw] Flawfinder http://www.dwheeler.com/flawfinder

[Splint] Splint http://www.splint.org/

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, den 15.12.2006