Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Elektrotechnik

Diplomarbeit

Mechatronik

Matthias Hagl
Embedded Lua als Programmierumgebung für einen Atmel AT91SAM7X256 Mikrocontroller

Erstprüfer: Prof. Dr. Hubert Högl

Zweitprüfer: Prof. Dr.-Ing. Norbert Reifschneider

Thema erhalten am: 07.04.09

Verfasser der Diplomarbeit
Matthias Hagl
Ringstraße 26
82285 Haspelmoor
Telefon +49 8202 8475
Email haglm@rz.fh-augsburg.de

Hochschule für
angewandte Wissenschaften –
Fachhochschule Augsburg
University of Applied Sciences

An der Fachhochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Rechtsverbindliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit

„Embedded Lua als Programmierumgebung für einen Atmel AT91SAM7X256 Mikrocontroller“

selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum	Matthias Hagl

Danksagung

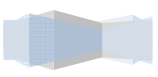
Ich möchte mich zu Beginn dieser Arbeit zuerst bei meinem Erstkorrektor Herrn Prof. Dr. Hubert Högl bedanken, ohne den dieses Thema überhaupt nicht zu Stande gekommen wäre. Zudem möchte ich mich bei Herrn Prof. Dr.-Ing. Norbert Reifschneider bedanken, der sich bereit erklärt hat, die Zweitkorrektur dieser Arbeit zu übernehmen. Abschließend möchte ich mich noch recht herzlich bei meinen Eltern bedanken, ohne deren Unterstützung mein Studium wahrscheinlich nicht so leicht machbar gewesen wäre.

Zusammenfassung

Mikrocontroller spielen in unseren täglichen Lebensablauf unbewusst eine große Rolle. Dies ist Grund genug, um sich einmal genauer mit den dafür bereitgestellten Programmierumgebungen auseinanderzusetzen.

Diese Diplomarbeit gibt dem Leser einen Überblick über das eLua-Projekt. Bei diesem Projekt handelt es sich um ein Open Source Projekt, das sich zur Aufgabe gemacht hat, die interpretierte Programmiersprache Lua auf verschiedenen Mikrocontrollern als Programmierumgebung zu integrieren.

Die Arbeit gibt dabei eine gute Zusammenfassung dessen, was die Entwickler des Projektes bereits umgesetzt haben und somit eine leicht verständliche Einführung in die Funktionalitäten der einzelnen Module. Zudem wird die Entwicklung neuer Schnittstellenmodule beschrieben und an einigen Beispielen erläutert. Das Hauptaugenmerk liegt dabei auf dem Mikrocontroller AT91SAM7X256 der Firma Atmel Corporation. Dabei werden auch der Ressourcenverbrauch und die Geschwindigkeit der Sprache untersucht. Zudem zeigt die Arbeit durch ein einfaches Anwendungsbeispiel auf, wie leicht die Programmierung von Mikrocontrollern mit Hilfe von eLua sein kann.

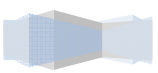


Aufgabenstellung

Die Aufgabenstellung der Diplomarbeit umfasst mehrere verschiedene Aspekte, die im Zusammenhang mit eLua[1] stehen.

Als erstes sollte einem Leser, der noch nie etwas von eLua gehört hat, ein grober Überblick gegeben werden, was eLua ist und für was man es benötigt. Dafür soll diese Arbeit eine Art Dokumentation darstellen, mit deren Hilfe sich der Leser möglichst schnell und einfach in die grundlegenden Funktionalitäten von Embedded Lua einarbeiten kann. Auf der anderen Seite soll dem Leser jedoch auch vermittelt werden, welche Vorgehensweise sinnvoll ist, um in eLua neue Schnittstellen hinzuzufügen und diese daraufhin auch zu nutzen. Zu diesem Zweck soll die allgemeine Vorgehensweise zur Schnittstellenimplementierung beschrieben werden. Desweiteren sollen verschiedene Schnittstellen implementiert werden, die diese allgemeine Vorgehensweise als praktische Beispiele veranschaulichen. Eine Untersuchung des Ressourcenverbrauchs sowie der Geschwindigkeit des eLua-Interpreters wird ebenfalls gefordert. Abschließend soll die Einfachheit der Programmierung mit eLua anhand eines Anwendungsbeispiels beschrieben werden.

[1] ist in diesem Dokument von eLua die Rede, so ist damit die Vorversion 0.5 gemeint



Inhaltsverzeichnis

1. Einführung

1.1.Motivation

Durch die vielfache Einsatzmöglichkeit von Mikrocontrollern, sind diese bereits Bestandteil unseres täglichen Lebens. Auch wenn sich die meisten Menschen dessen nicht bewusst sind, kommen im täglichen Gebrauch Mikrocontroller vielfach zum Einsatz. Beispiele hierfür sind Steuerung von Waschmaschinen, Fernsehern, oder Spielzeugrobotern. Auch im Bereich der Fahrzeugtechnologie werden Mikrocontroller in erheblichem Umfang genutzt. Somit ist es naheliegend, sich mit den Programmiertechniken dieser Steuergeräte zu befassen. Für gewöhnlich werden diese mit Compilersprachen wie C oder C++ programmiert. Die Nutzung dieser Sprachen hat jedoch einige Nachteile. Als erstes kann man hier schon einmal die Komplexität der Sprache nennen. Ein Programmieranfänger, der einen Mikrocontroller mit ANSI-C programmieren will, wird es sehr schwer haben, da man gerade beim Gebrauch von Zeigern eine gewisse Programmiererfahrung mitbringen muss. Zudem ist es in der Regel erforderlich, eine hardwarenahe Programmierung vorzunehmen. Desweiteren ist anzumerken, dass jedes einzelne Programm erst kompiliert und dann auf den Speicher des Mikrocontrollers gebrannt werden muss. Dies ist gerade während der Implementierungsphase neuer Anwendungen sehr aufwändig. Hier wird der große Vorteil der Interpretersprachen ersichtlich. Ist der Interpreter erst einmal übersetzt und auf den Mikrocontroller geladen, kann durch die Nutzung der seriellen Schnittstelle ein schnelles Überspielen von Programmen, oder Programmstücken gewährleistet werden. Der Interpreter führt diese aus und kann dann sofort wieder neue Programme empfangen. Somit stellt die Nutzung einer Interpretersprache auf Mikrocontrollern eine Art von Interaktivität und somit eine höhere Flexibilität zur Verfügung. Man kann daher seine Programme zügig testen und debuggen, ohne dass das komplette Programm fertiggestellt sein muss. Auch der geringe Speicherbedarf des Interpreters spielt bei den beschränkten Ressourcen von Mikrocontrollern eine große Rolle. Somit war Mitte der Siebziger Jahre BASIC bereits als Programmiersprache für die Vorläufer der heutigen Mikrocontroller vorhanden. Auch FORTH war zur Programmierung von Mikrocontrollern durch seinen kompakten und schnellen Code weit verbreitet, verliert allerdings zunehmend durch die unübersichtliche Syntax an Bedeu-

tung (1). Auch heute gibt es noch einige Interpretersprachen für Mikrocontroller, wie zum Beispiel PyMite, einen Python-Interpreter für Mikrocontroller. Um Interpretersprachen für Mikrocontroller attraktiver zu machen, ist unter anderem auch das eLua-Projekt[2] entstanden. Durch seine Einfachheit, Effizienz, Erweiterbarkeit, Portabilität und vor allem durch seine Nähe zu C eignet sich Lua hervorragend als Sprache zur Programmierung von Mikrocontrollern. Diese Arbeit gibt einen Einblick in dieses Projekt.

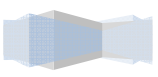
1.2.Aufbau

Die Arbeit beginnt mit einem Grundlagenkapitel, in dem das, für ein besseres Verständnis notwendige Basiswissen vermittelt wird. Es gliedert sich in fünf Abschnitte. Als erstes ein Abschnitt, der den Leser in die Grundlagen der Interpretersprache Lua einführt. Anschließend wird ein kurzer Überblick über embedded Lua (eLua) gegeben. Die nächsten beiden Unterkapitel erläutern den verwendeten Mikrocontroller und dessen Ansteuerung mit eLua. Im fünften Abschnitt wird veranschaulicht, wie der derzeitige Umgang mit eLua abläuft.

Das anschließende Kapitel erläutert, welche Funktionen die Entwickler des eLua Projektes bereits implementiert haben und inwiefern eLua bereits auf Mikrocontrollern unterstützt wird. Zudem werden die bereits implementierten Module für den Atmel AT91SAM7X256 kurz beschrieben und somit deren Funktionalitäten nahegebracht.

Aufbauend auf die vorherigen Betrachtungen beginnt der Teil der Diplomarbeit, worin dem Leser erklärt wird, wie man neue Schnittstellen zur eLua-Distribution hinzufügt. Dieser Teil der Arbeit untergliedert sich noch einmal in mehrere Teile. Der erste Teil behandelt die Schnittstellenimplementierung mit Hilfe von C – Code im Allgemeinen. Das heißt, dass die Vorgehensweise zur Schnittstellenimplementierung beschrieben wird. Im Folgenden wird dann genauer auf die Schnittstellenimplementierung eingegangen. Hierzu werden einige Schnittstellen für den AT91SAM7X256 hinzugefügt. Abschließend werden in diesem Abschnitt noch verschiedene Betrachtungen zu Ressourcenverbrauch und Geschwindigkeit von eLua getroffen.

[2] Das eLua Projekt befindet sich noch in der Entwicklungsphase, jedoch sind die bisher implementierten Teile bereits lauffähig



Zur Abrundung der Arbeit wird, mit Hilfe der in den vorhergehenden Kapiteln erstellten Module, eine kleine Beispielanwendung entwickelt. In diesem Kapitel wird somit auf die tatsächliche Nutzung von eLua eingegangen.

2. Grundlagen

In diesem Kapitel werden die Grundlagen für die weitere Arbeit geschaffen. Dem Leser wird zum einen eine Einführung in die behandelte Thematik gegeben und zum anderen werden gewisse Grundbegriffe geklärt, die helfen sollen die Arbeit besser zu verstehen.

2.1.Die Programmiersprache Lua

Die Sprache Lua wurde 1993 von den drei Softwareentwicklern Waldemar Celes, Luiz Henrique De Figueiredo und Roberto Ierusalimschy eigentlich nur für zwei Firmeninterne Projekte entwickelt. Doch durch die Eigenschaften einer einfachen, erweiterbaren, portablen und effizienten Skriptsprache verbreitete sich Lua weltweit, sodass sie heute in allen möglichen Bereichen Einsatz findet. Dazu gehören vor allem eingebettete Systeme, Webserver und Spiele. (2)

2.1.1. Definition einer Interpretersprache

Um den Begriff Interpretersprache erklären zu können, ist vorab die Funktionsweise von Compilersprachen zu betrachten.

Bei Compilersprachen wandelt der Compiler z.B. einen in ANSI-C geschriebenen Quellcode in Assemblersprache, Bytecode oder Maschinencode um. Das daraus entstandene Programm kann daraufhin vom jeweiligen Betriebssystem, im Falle eines Mikrocontrollers direkt vom Mikrocode der Hardware ausgeführt werden.

Bei einer Interpretersprache wird kein Compiler benötigt, der den Quellcode in eine ausführbare Datei übersetzt, die dann auf die jeweilige Plattform geladen wird. Interpretersprachen werden ausschließlich vom sogenannten Interpreter eingelesen. Ein Interpreter ist selbst ein ausführbares Programm. Der Interpreter liest den Code ein, analysiert ihn und führt ihn aus. Es geschieht also alles während der Laufzeit des Programmes. Dies macht sich in der Geschwindigkeit der Sprache bemerkbar. Interpretersprachen sind deutlich langsamer als Compilersprachen. Der große Vorteil von interpretier-

ten Sprachen ist jedoch, dass sie auf jeder Rechnerarchitektur ausführbar sind, auf der ihr Interpreter lauffähig ist und dass die Kompilierung von Programmen entfällt. (2)

Bekannte Interpretersprachen sind zum Beispiel BASIC, Perl, Python, FORTH und PHP, um nur einige zu nennen. Auch Skriptsprachen kann man zu Interpretersprachen zählen.

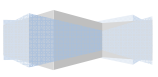
Ierusalimschy beschreibt Lua mit den nachfolgenden Worten:

> *„Lua ist eine eingebettete Sprache, also kein abgeschlossenes Paket, sondern eine Bibliothek die man mit anderen Anwendungen linken kann, um diese mit den Möglichkeiten von Lua zu erweitern“* (2).

Wenn man dieses Zitat betrachtet, könnte man meinen, dass Lua im Grunde keine eigenständige Sprache ist, sondern nur eine Bibliothek. Hier kommt allerdings der oben beschriebene Interpreter ins Spiel. Durch das aus weniger als vierhundert Zeilen Code bestehende Programm wird die Lua Bibliothek eingebunden und die Schnittstelle zum Benutzer hergestellt. Sämtliche Lua Befehle werden von dem Interpreter ausgeführt und macht Lua somit zu einer Interpretersprache. (2)

2.1.2. Grundlegende Eigenschaften der Sprache Lua

Die Sprache ist so entworfen, dass sie leicht in Software integriert werden kann, die in anderen konventionellen Sprachen geschrieben ist – vor allem in C. Der Erfolg von Lua besteht darin, nicht besser sein zu wollen als C, sondern genau dort anzusetzen, wo C gewisse Schwächen aufweist. Daher arbeitet Lua mit einem gewissen Abstand zur Hardware und somit mit dynamischen Datenstrukturen und ohne Redundanz. Außerdem sind bei Lua einfache Möglichkeiten des Testens und der Fehleranalyse vorhanden. (2)



Nachfolgend werden kurz die wichtigsten Eigenschaften der Sprache beschrieben:

- **Erweiterbarkeit**

  Lua wurde so entworfen, dass die Sprache sowohl durch Lua-Code, als auch durch C-Code erweitert werden kann. Sogar die Grundfunktionalität der Sprache wurde mit Hilfe von externen Bibliotheken implementiert. Somit ist es leicht eine Schnittstelle zwischen Lua und C/C++ zu benutzen. Zudem wurde Lua bereits mit anderen Sprachen wie z.B. Fortran, Java und auch Skriptsprachen wie Perl und Ruby benutzt. (2)

- **Einfachheit**

  Lua ist eine kleine und einfache Sprache. Es hat zwar nur wenige aber dafür mächtige Konzepte. Diese Einfachheit hat zwei nicht unwesentliche Vorteile. Zum einen ist die Sprache dadurch leicht erlernbar und zum anderen benötigt sie dadurch kaum Speicherplatz. Die gesamte Distribution kann man auf eine Floppydisk speichern. (2)

- **Effizienz**

  „Lua hat eine sehr effiziente Implementierung. Unabhängige Benchmarks haben gezeigt, dass Lua eine der schnellsten Sprachen in der Kategorie der interpretierten Skriptsprachen ist“. (2)

- **Portabilität**

  Lua läuft auf allen vorstellbaren Plattformen. Dabei ist nicht nur von Windows oder Unix die Rede. Um nur einige zu nennen: ARM Prozessoren, Mac OS-9, IBM Mainframes, PlayStation und Palm OS. Der Quellcode ist für alle Plattformen annähernd identisch. Dies liegt daran, dass sich Lua an Standard-ANSI (ISO) C hält. Man muss den Lua-Interpreter lediglich mit einem ANSI-C Compiler kompilieren und kann diesen anschließend verwenden. (2)

2.1.3. Die C-API

Im späteren Verlauf dieser Arbeit wird ersichtlich, dass es wichtig ist Erweiterungen für Lua schreiben zu können. Diese Erweiterungen werden jedoch nicht in Lua Code implementiert, sondern in C-Code.

Die C-API spielt hierbei eine wichtige Rolle. Die API ermöglicht es, dass Lua und C miteinander kommunizieren können. Ierusalimschy definiert die C-API wie folgt:

> *„Die C-API ist die Menge jener Funktionen, die es einem C-Code ermöglicht, mit Lua zu interagieren. Dazu gehören Funktionen zum Lesen und Schreiben von globalen Variablen, zum Aufrufen von Lua – Funktionen, zum Ausführen von Lua – Codestücken und zum Registrieren von Lua – Funktionen, damit diese später von Lua – Code aufgerufen werden können.“* (2)

Aus den nachfolgenden Kapiteln dieser Arbeit wird ersichtlich, dass gerade die C-API für eLua eine wichtige Rolle spielt. Da Lua – wie oben erwähnt – einen gewissen Abstand zur Hardware einhält, ist es sinnvoll die Schnittstellenimplementierung mit Hilfe der C-API umzusetzen. Hierzu jedoch in den nachfolgenden Kapiteln mehr.

2.2.Embedded Lua

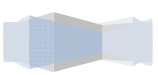
Embedded Lua ist ein Open Source Projekt, dessen Ziel darin besteht, die Programmiersprache Lua in die Embedded Softwareentwicklungswelt einzuführen. In die Welt gerufen wurde das Projekt von Bogdan Marinescu, einem Softwareentwickler aus Rumänien, und Dado Sutter, dem Leiter des Led Lab’s in der PUC Universität in Rio de Janeiro.

Die oben angesprochenen geringen Anforderungen an Hardwareressourcen der Sprache Lua sind ausschlaggebend dafür, dass sich die Sprache für eine Reihe von Mikrocontrollerfamilien eignet.

Das Ziel des Open Source Projektes besteht darin, direkt auf einem Mikrocontroller eine volle Entwicklungsumgebung bereit zu stellen, ohne auch nur ein einziges Programmierwerkzeug auf der PC Seite installieren zu müssen.

Aktuell benötigt man den PC lediglich, um Lua Programme in einem Texteditor zu schreiben bzw. zu editieren und dann auf den Mikrocontroller zu laden. Sobald das Projekt jedoch fortgeschritten ist, sollen aber auch diese Anforderungen an den PC entfallen. Ziel ist es einen einfachen Editor in den Mikrocontroller integrieren, um völlig autonom mit dem Mikrocontroller arbeiten zu können. Dazu sollen natürlich auch eine Vielzahl von Input- und Output Devices zur Verfügung stehen.

Für die anfänglichen Implementierungen haben sich die Projektverantwortlichen für die ARM Mikroprozessorfamilie entschieden. Gründe hierfür sind deren Popularität, ihre Verfügbarkeit und vor allem ihre geringen Kosten. Allerdings soll das Projekt nicht auf diese Prozessoren beschränkt bleiben. (3)



2.3.Atmel AT91SAM7X256

Der Atmel AT91SAMX256 ist Teil einer Reihe von hochintegrierten Flash–Mikrokontrollern, die auf 32 Bit ARM RISC Prozessoren basieren. Er beinhaltet einen 256 kByte Hochleistungsflashspeicher und 64 kByte SRAM. Zudem ist eine große Auswahl an Peripheriegeräten integriert, einschließlich einem Ethernet MAC 802.3 und einem CAN Controller. (4)

2.3.1. Prozessor

Beim hier verwendeten Prozessor handelt es sich um einen ARM7TDMI RISC Prozessor, der auf der ARMv4T von Neumann Architektur basiert. Die Taktfrequenz der CPU beträgt bis zu 55 MHz[3]. Es werden zwei Befehlssätze genutzt, zum einen der ARM 32 Bit Hochleistungsbefehlssatz und zum anderen der 16 Bit Thumb high code density Befehlssatz. (4)
Der Prozessor arbeitet mit einer dreistufigen Leitungsarchitektur:

- Befehlsabruf
- Befehlsinterpretation
- Ausführung des Befehls

2.3.2. Speicher

Wie oben bereits erwähnt, beinhaltet der Atmel AT19SAM7X256 einen 256 kByte großen Flashspeicher und eine 64 kByte großen SRAM. Der eingebettete dual plane Flashspeicher besteht aus einer Speicherbank mit 256 kByte und kann über die JTAG-ICE- oder die USB-Schnittstelle programmiert werden. Es sind schnelle Zugriffszeiten auf den Speicher möglich. Die Zugriffszeiten betragen im worst case Fall bei Einzelzugriffen 33,3 ns, woraus eine Taktung von 30 MHz folgt. Die eingebauten Lock Bits und das Sicherheitsbit schützen die Firmware vor versehentlichem Überschreiben und gewähren somit deren Zuverlässigkeit. (4)

[3] Im Fall von eLua wird mit einer Taktfrequenz von 48 MHz gearbeitet

2.3.3. Peripheriegeräte

Der AT91SAM7X256 stellt zudem eine Reihe von Controllern zur Ansteuerung von Peripheriegeräten zur Verfügung. Außerdem enthält er einige Schnittstellen für Bussysteme. Die nachfolgende Tabelle gibt einen kurzen Überblick.

Peripheriegerät	Abkürzung
Advanced Interrupt Controller	AIC
Parallel Input/Output Controller	PIO
Serial Peripheral Interface	SPI
USART	US
Synchronous Serial Controller	SSC
Two-wire Interface	TWI
Puls Width Modulation Controller	PWMC
USB Device Port	UDP
Timer/Counter	TC
CAN Controller	CAN
Ethernet MAC	EMAC
Analog-Digital-Converter	ADC

Tabelle 1 - Peripherie AT91SAM7X256

2.4.Kompilierung von eLua und Programmierung des Mikrocontrollers

Um eLua auf einem Mikrocontroller zu integrieren werden einige Soft- und Hardwarekomponenten benötigt. Dieser Abschnitt bringt dem Leser kurz näher, was für die Kompilierung des Interpreters und für die Programmierung des Mikrocontrollers benötigt wird.

2.4.1. Kompilierung von eLua

Dieser Absatz ist nur von Bedeutung, wenn man für eLua Erweiterungen entwickeln möchte. Falls dies nicht der Fall sein sollte, kann die Binärdatei für die jeweilige Platt-

form von der Website http://www.eluaproject.net/en/Binaries heruntergeladen werden. Somit ist die Kompilierung nicht relevant.

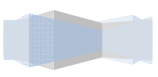
Die Kompilierung sollte man auf einem Linux Betriebssystem durchführen. Laut den Entwicklern von eLua sollte es zwar möglich sein, eLua auf einem Windows Betriebssystem zu kompilieren, dies ist allerdings nicht getestet und freigegeben. Somit ist nicht gewährleistet, dass eLua fehlerfrei funktioniert. Um eLua kompilieren zu können, muss auf dem System ein GCC/Newlib Toolchain installiert sein. Eine sehr gute Anleitung für die Installation der Toolchain für Ubuntu ist auf der Website http://wiki.ubuntuusers.de/GNU_ARM-Toolchain zu finden. Zudem muss auf dem System ein Python-Interpreter installiert sein. Dies sollte bei Linux Betriebssystemen jedoch bereits vorinstalliert sein.
Die Kompilierung erfolgt jedoch bei eLua nicht über make und makefiles wie auf Linux-Systemen üblich, sondern mit Hilfe von SCons. Somit muss auf dem jeweiligen System auch SCons installiert sein.

Sind diese Voraussetzungen erfüllt, so läuft die Kompilierung sehr einfach ab. In dem Verzeichnis, in dem eLua gespeichert ist wird SCons über die Kommandozeile mit einigen Parametern aufgerufen:

```
01 scons target=lua cpu=at91sam7x256 board=sam7-ek256
02 cpumode=thumb allocator=newlib prog
```

Listing 1 - Aufruf zur Kompilierung von eLua

Die einzelnen Parameter sind von der Plattform abhängig, auf der eLua integriert werden soll. Im Nachfolgenden werden kurz die einzelnen Parameter erörtert. Der erste Parameter `target` definiert, welche Art von Lua integriert werden soll. Hierbei gibt es die Einstellungen `lua` und `lualong`. Der Unterschied zwischen den beiden Parametern, dass bei `lualong` nur Integerzahlen und bei `lua` auch Fließkommazahlen unterstützt werden können. Definiert man hier keinen Parameter, so ist der Default-Wert `lua`. Der Parameter `cpu` gibt an, welcher Mikrocontroller verwendet werden soll. Unter `board` trägt man den Namen des jeweiligen Entwicklungsboards auf dem sich der Mikrocontroller befindet – im Falle dieser Arbeit handelt es sich hier um ein Olimexboard. Bei

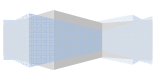


`cpumode` gibt es zwei Auswahlmöglichkeiten `thumb` für AT91SAM7X und `arm` für STR9 und LPC2888. Dem `allocator` Parameter kann man `newlib` oder `multiple` übergeben, wobei `multiple` nur dann Sinn macht, wenn man eine Unterstützung von mehreren Speichern benötigt. Dies wäre zum Beispiel der Fall, wenn man einen AT91SAM7X512 nutzt. Am Ende sollte man noch den Befehl `prog` angeben, da SCons nur eine ELF-Datei erstellt, wohingegen mit dem `prog` Befehl zusätzlich eine BIN-Datei erstellt wird.

2.4.2. Programmierung des Mikrocontrollers mit eLua

Der nächste Schritt nach der Kompilierung ist die Datenübertragung der Binärdatei vom PC auf das Testboard, auf dem der Mikrocontroller integriert ist. Für diese Übertragung gibt es zwei Möglichkeiten. Die erste Möglichkeit besteht darin, dass man Open OCD verwendet. In diesem Fall ist es sinnvoll embedded Lua über die JTAG-Schnittstelle auf das Olimex-Entwicklungsboard zu übertragen. Der große Vorteil bei dieser Variante liegt darin, dass man den Code nach der Übertragung mit Hilfe des GNU Project Debuggers debuggen kann. Dem gegenüber steht, dass Open OCD auf der Kommandozeile arbeitet und man viele verschiedene Parameter in der Konfigurationsdatei beachten muss. Die zweite Möglichkeit zur Datenübertragung im Fall des AT91SAM7X256 ist ein Tool von Atmel, mit dem man sehr schnell und einfach die Daten auf den Flash oder RAM Speicher brennen kann. Dieses Tool heißt AT91 ISP und kann unter der Webadresse http://www.atmel.com/dyn/resources/prod_documents/Install%20AT91-ISP%20v1.13.exe heruntergeladen werden. Mit dem darin enthaltenen Programm SAM-BA v2.8 kann man den Speicher des Mikrocontrollers über eine grafische Oberfläche bespielen. Für jemanden, der noch nie mit OpenOCD gearbeitet hat, ist dies mit Sicherheit die leichtere Variante. Weiter kommt in diesem Fall positiv hinzu, dass ein USB Kabel ausreicht, um die Daten übertragen zu können. Der Nachteil dieser Variante liegt darin, dass man keine Debugmöglichkeiten wie bei Open OCD hat.

Die Entscheidung, welche der beiden Varianten verwendet wird, bleibt dem jeweiligen Nutzer überlassen. Beide Varianten haben ihre Vor- und Nachteile. Bei beiden Varianten wird eLua auf den Flash geschrieben und danach davon gebootet.

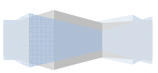


2.5.Nutzung von eLua

Sind die oben genannten Hürden genommen, steht der Nutzung von eLua nichts mehr im Wege. Man benötigt abschließend nur noch einen Terminalemulator, über den man mit dem Mikrocontroller kommunizieren kann. Geeignete Terminalemulatoren sind z.B. TeraTerm unter Windows oder minicom unter Linux. Da eLua zum derzeitigen Stand ausschließlich über das XMODEM-Protokoll kommuniziert, ist dies bei der Auswahl des Terminaleditors unbedingt zu berücksichtigen. Vor dem Start des Terminalemulators sind die folgenden Einstellungen vorzunehmen:

- Port Setup: 115200 baud
- Flow control: none
- Newline handling: CR on receive, CR+LF on send

Im Anschluss ist der Mikrocontroller neu zu starten. Am Terminal muss nun der Schriftzug eLua erscheinen. Ist dies nicht der Fall, so sind die zuvor durchgeführten Einstellungen zu prüfen und gegebenenfalls zu korrigieren. Mit recv kann man nun Lua-Dateien via XMODEM übertragen. (3)



3. Bestehende Komponenten in Embedded Lua

Dieses Kapitel beschreibt, welche Komponenten im eLua-Projekt bereits umgesetzt wurden. Dabei wird zwischen Plattform abhängigen und unabhängigen Komponenten differenziert.

3.1.Plattformunabhängige Komponenten

eLua ist bereits so weit implementiert, dass man die Skriptsprache Lua vollkommen eigenständig auf einem Mikrocontroller ablaufen zu lassen. eLua ist so strukturiert, dass man es relativ leicht an eine andere Plattform anpassen kann. Allerdings ist diese Portabilität auf Plattformen begrenzt, bei denen die Kombination von GCC und der newlib Bibliothek möglich sind. Der embedded Lua Interpreter wurde bereits auf einige Mikrocontroller abgestimmt[4].
Plattformunabhängig stehen bereits die in folgender Tabelle aufgelisteten Features zur Verfügung:

Feature	Status	Feature	Status
Beispielprogramme für eLua	teilweise implementiert und getestet	Lua PIO module	lauffähig
eLua Shell	teilweise implementiert und getestet	Lua platform data module	lauffähig
eLua terminal module	lauffähig	Lua PWM module	lauffähig
Embedded R/W file system	teilweise implementiert und getestet	Lua SPI module	implementiert und nicht getestet
Embedded ROM file system	lauffähig	Lua timer module	lauffähig
Lua binary pack/unpack module	lauffähig	Lua UART module	lauffähig
Lua bit operations module	lauffähig	Networking	lauffähig, allerdings bisher nur experimentell
Lua CPU module	lauffähig	XMODEM receive	lauffähig

Tabelle 2 - Implementierte Features in eLua (3)

[4] Mikrocontroller auf denen eLua bereits lauffähig ist: LM3S8962, LM3S6965, i386, AT91SAM7X256, AT91SAM7X512, STR912FW44, LPC2888, STR711FR2

Wie aus Tabelle 2 ersichtlich wird, stehen bereits einige ergänzenden Module zur Verfügung, die es ermöglichen mit einer Lua-Anwendung Peripheriegeräte anzusteuern. Abhängig vom Mikrocontroller kann man bereits die PIO-, SPI-, UART-, TMR, PWM- und den Netzwerkport mit Lua ansteuern.

Zudem ist es möglich eLua in zwei verschiedenen Varianten auf dem Mikrocontroller ablaufen zu lassen. Die erste Variante ist das reguläre eLua. Diese Variante arbeitet mit Fließkommanotation und ist langsamer, da Fließkommaoperationen in der Software auf dem Mikrocontroller emuliert werden müssen. Die zweite und wesentlich performantere Variante ist das Integer eLua, bei der Zahlen als Ganzzahlen dargestellt werden. Benötigt man bei dieser Variante Fließkommaoperationen so sind die entsprechenden Module hinzuzufügen.

3.2.Beschreibung der einzelnen Komponenten

In diesem Abschnitt werden kurz die einzelnen Komponenten beschrieben und ihre jeweilige Funktionsweise in eLua dargelegt.

eLua Shell

Die eLua Shell ist das grundlegende Modul für die Nutzung von eLua. Sie stellt die Schnittstelle zwischen dem Terminalemulationsprogramm und dem Mikrocontroller her. Die Shell stellt vier Funktionalitäten zur Verfügung:

- Die Interpretation von Lua-Code direkt über die Eingabe auf der Shell (Befehl: lua)
- Den Empfang von Lua-Dateien via XMODEM und deren Ausführung (Befehl: recv)
- Eine Hilfefunktion für den Umgang mit der Shell (Befehel: help)
- Die Ausgabe der Versionsnummer von eLua (Befehl: ver)

eLua terminal module

Das Terminal Modul stellt einige Funktionen zur Verfügung, mit denen sich gewisse Operationen auf dem Terminal ausführen lassen. Um nur wenige zu nennen, z.B. das Löschen des Bildschirms, Ausgabe von Buchstaben oder Strings auf dem Bildschirm. Die

genaue Auflistung der Funktionen können aus der Tabelle 21 im Anhang entnommen werden.

Embedded ROM file system

eLua besitzt ein integriertes read-only Dateisystem. Mit Hilfe dieses Systems können Programmcodes bzw. statische Dateien auf dem Mikrocontroller geöffnet und gelesen werden. Diese Dateien müssen vor der Kompilierung (SCons-Skript) im romfs-Ordner der eLua-Distribution abgelegt werden, da diese Teil des eLua Binär Images sind. Anwendungsfälle für read-only Dateien könnten z.B. statische Koordinaten zur Roboter-steuerung beinhalten. Lua-Programme können mit folgendem Aufruf von der Shell aus gestartet werden:

```
eLua# lua /rom/Beispiel.lua
```

Es ist zu beachten, dass Dateinamen inklusive Endungen die Länge von 14 Zeichen nicht überschreiten.

Lua binary pack/unpack module

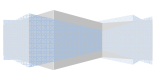
Durch dieses Modul wird es dem eLua-Nutzer ermöglicht, eine Variable eines bestimmten Datentyps in eine Datei zu speichern und diese später wieder in eine Lua-Variable zurückzuladen. Hierbei stehen zwei Methoden zur Verfügung: Pack und Unpack. Dieses Modul wurde von Luis Henrique de Figueiredo für Lua entwickelt und in eLua eingebunden. Genauere Informationen zum Gebrauch dieses Moduls können aus Quelle (5) unter lpack entnommen werden.

Lua bit operations module

Da Lua von sich aus keine Bitoperationen bereitstellt, wurde in eLua ein Bit Modul umgesetzt. Dieses Modul basiert auf einer Bibliothek für Lua von Reuben Thomas und wurde von den eLua Entwicklern noch ein wenig erweitert. Siehe Tabelle 19 und Tabelle 20 im Anhang.

Lua CPU module

Das eLua CPU Modul stellt einige Funktionen für den Zugriff auf die CPU zur Verfügung. Beispiele hierfür sind das Auslesen und Schreiben von Daten auf bestimmte Adressen oder die Ausgabe der Taktfrequenz der CPU.



Lua PIO module

Dieses Modul stellt die Funktionalitäten des programmierbaren Ein-und Ausgangscontroller zur Verfügung. Die genaue Funktionsbeschreibung wird in Kapitel 3.3 dargestellt.

Lua platform data module

Dieses Modul stellt drei Methoden zur Verfügung, mit deren Hilfe Lua feststellen kann, auf welcher Plattform es gerade läuft, welche CPU und welches Board benutzt wird.

Lua PWM module

Mit Hilfe dieses Moduls kann der Nutzer die Pulsweitenmodulation des jeweiligen Mikrocontrollers nutzen. Die genaue Funktionsbeschreibung wird in Kapitel 3.3 dargestellt.

Lua SPI module

Durch dieses Modul werden für die Mikrocontroller, die über eine SPI-Schnittstelle verfügen gewisse Funktionen bereitgestellt. Da dieses Modul noch nicht für den AT91SAM7X256 implementiert wurde, wird im Kapitel 4.4 genauer darauf eingegangen.

Lua Timer module

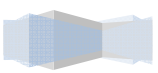
Durch das Timer-Modul wird es dem eLua-Nutzer ermöglicht gewisse Timer-spezifische Funktionen zu nutzen. Auch hierzu mehr in Kapitel 3.3.

Lua UART module

Dieses Modul ermöglicht es dem Nutzer die Funktionen der UART-Schnittstelle zu nutzen. Die genaue Funktionsbeschreibung des Moduls wird im nächsten Kapitel besprochen.

XMODEM receive

Die Möglichkeit der Nutzung des XMODEM-Protokolls ist wohl eine der wichtigsten Funktionen die eLua mit sich bringt. Der gesamte Datentransfer vom PC auf den Mikrocontroller erfolgt derzeit über das XMODEM-Protokoll. Will der Nutzer ein Lua-Programm auf dem Mikrocontroller ausführen, das nicht in das ROM Dateisystem ge-



brannt wurde, so erfolgt die Übertragung der Datei mit Hilfe von XMODEM. Es besteht jedoch auch die Möglichkeit XMODEM vor der Kompilierung abzuschalten, was jedoch zur Folge hat, dass nachträglich keine Lua-Programme übertragen werden können.

Networking

Im eLua Net-Modul werden einige Funktionen zum Gebrauch von TCP/IP bereitgestellt. Allerdings ist dieses Modul noch in einer experimentellen Phase.

3.3.Bereits implementierte Module für den Atmel AT91SAM7X256

In diesem Kapitel wird speziell auf die Softwareunterstützung für den Atmel AT91SAM7X256 eingegangen. Wichtig ist hier anzumerken, dass die Komponenten die auf dem AT91SAM7X256 lauffähig sind, auch auf dem „großen Bruder“ AT91SAM7X512 verwendbar sein dürften, da sich die Mikrocontroller nur in der Speichergröße unterscheiden.

In der nachfolgenden Tabelle 3 wird, wie auch im Kapitel vorher, ein kurzer Überblick über die Softwareunterstützung der Schnittstellen gegeben.

AT91SAM7X256	implementiert	getestet	n/a
Ethernet MAC			x
SPI			x
TWI			x
USART	x	x	
SSC			x
TIC	x	x	
PWM	x	x	
UDP			x
CAN			x
ADC			x
AIC			x
DBGU			x
PIO	x	x	

Tabelle 3 - Übersicht der Softwareunterstützung für den AT91SAM7X256

Abgesehen von den in der Tabelle aufgeführten Schnittstellen unterstützt der AT91SAM7X256 die eLua Shell, die Terminalfunktionen, das ROM Dateisystem, das Pack/Unpack Modul, das Plattform Data Modul, Bitoperatoren und selbstverständlich XMODEM. Nachfolgend wird dem Leser ein Überblick über die Funktionalitäten der bereits implementierten Schnittstellen gegeben werden.

3.3.1. Die U(S)ART-Schnittstelle

Der Universal Synchronous Asynchronous Receiver Transceiver des AT91SAM7X256 stellt eine serielle Vollduplexverbindung zur Verfügung. Um diese Verbindung bewerkstelligen zu können, müssen softwaretechnisch einige Funktionen umgesetzt werden. Dafür haben die eLua-Entwickler das UART-Modul bereitgestellt. Dieses Modul enthält vier verschiedene Funktionen, die für den Gebrauch der Schnittstelle ausreichen.

Zum einen wäre hier die Initialisierungsfunktion uart.setup(). Diese Funktion hat fünf Übergabewerte, mit deren Hilfe die Initialisierung erfolgt. Der erste Übergabeparameter ist hierbei, wie bei jeder eLua-Initialisierungsfunktion ein Integerwert *id*, mit dessen Hilfe überprüft wird, ob die angesprochene USART-Schnittstelle überhaupt vorhanden ist. Beim AT91SAM7X256 kann dies ein Wert zwischen 0 und 1 sein, da dieser Mikrocontroller über zwei dieser Schnittstellen verfügt. Der nächste Übergabewert ist *baud*, also die gewünschte Baudrate, die der Nutzer einstellen möchte. Hierbei handelt es sich um eine vorzeichenunbehaftete 32-Bit Zahl. Allerdings ist derzeit nur der asynchrone Betriebsmodus ohne oversampling verfügbar und somit ergibt sich die maximal einstellbare Baudrate nach folgender Formel:

$$Baudrate = \frac{Mikrocontrollertakt}{16} = \frac{48\,MHz}{16} = 3\frac{MBit}{s}$$

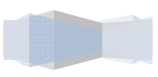
Der dritte Übergabeparameter heißt *databits*. Mit dieser Integerzahl wird die Charakter length festgelegt. Hierbei sind Werte zwischen 5 und 8 zulässig, wobei die jeweilige Zahl für die zu verwendenden Bits steht. Als nächstes kommt der Parameter *parity* für den bereits symbolische Variablen in eLua bereitstehen. Der Nutzer hat die Wahl zwischen uart.PAR_ODD für eine ungerade Parität, uart.PAR_EVEN für eine gerade Parität und uart.PAR_NONE für keine Paritätsprüfung. Durch den letzten Übergabewert werden die Stopbits eigestellt. Dafür gibt es drei einstellbare Werte uart.STOP_1 für ein Stopbit,

uart.STOP_1_5 für eineinhalb Stopbits und uart.STOP_2 für zwei Stopbits. Als Rückgabewert gibt die Funktion die aktuelle Baudrate zurück.

Die nächste Funktion die das Modul bereitstellt, ist eine Sendefunktion. Die Funktion uart.send() bekommt als erstes den Wert *id* übergeben, um feststellen zu können, welcher der beiden UART-Schnittstellen senden soll. Die nachfolgenden Übergabeparameter sind jeweils die zu übermittelnden Daten. Hierbei können beliebig viele Informationen übergeben werden, da die Funktion dies in einer Schleife abarbeitet. Der Datentyp dieser Parameter muss jedoch unsigned char sein. Es existiert noch eine zweite Sendefunktion, der Zeichenketten übergeben werden können. Die Funktion uart.sendstr()arbeitet analog zu der vorher beschriebenen Sendefunktion, eben nur mit der einen Ausnahme, dass Strings übergeben werden können.

Als Gegenstück zur Sendefunktion steht zudem die Funktion zum Empfangen von Daten bereit. Diese wird durch den Befehl uart.recv() aufgerufen. Dieser erhält drei Übergabeparameter. Um eine der beiden Schnittstellen zum Empfangen auszuwählen, wird wiederum die *id* übergeben. Der zweite Übergabeparameter definiert die *timer_id,* mit dem der Funktion die Nummer des Timers übergeben wird. Der Wert für *timer_id* kann beim AT91SAM7X256 zwischen 0 und 2 liegen. Mit dem dritten Parameter wird ein *timeout* definiert. Dafür gibt es drei verschiedene Möglichkeiten.

- uart.NO_TIMEOUT
  Der Timer wird auf 0 Sekunden eingestellt. Das heißt das Datenpaket muss sofort anliegen, ansonsten wird der Rückgabewert -1 zurückgegeben
- uart.INF_TIMEOUT
  Die Funktion wartet bis das jeweilige Datenpaket angekommen ist
- spezifischer Wert in µs
  Die Funktion wartet bis das Datenpaket anliegt oder bis die angegebene Zeit verstrichen ist. Liegen nach der angegebenen Zeit keine Daten an, wird der Rückgabewert -1 zurückgegeben



Die nachfolgende Tabelle soll abschließend einen Überblick über die Funktionen des UART-Moduls geben.

ret = uart.setup(id, baud, databits, parity, stopbits) – nimmt grundlegende Einstellungen vor	
Argumente	• id – UART Controller ID • baud – gewünschte Baudrate des UART-Controllers in Hz. Die maximale Baudrate ist dabei MCK/16 • databits – gewünschte Anzahl von Datenbits. Dabei sind Werte von 5 bis 8 Bit möglich • parity – gewünschte Art der Paritätsabfrage. Mögliche Parameter sind uart.PAR_EVEN, uart.PAR_ODD und uart.PAR_NONE • stopbits – gewünschte Länge des Stopbits. Mögliche Parameter sind uart.STOP_1, uart.STOP_1_5 und uart.STOP_2
Rückgabewerte	• ret – eingestellte UART-Baudrate
uart.send(id, in1, in2,..., inx) – sendet beliebig viele Daten	
Argumente	• id – UART-Controller ID • in1 – erstes Datenpaket • in2 – zweites Datenpaket • inx – letztes Datenpaket
Rückgabewerte	• -
uart.sendstr(id, in1, in2,..., inx) – sendet beliebig viele Strings	
Argumente	• id – UART-Controller ID • in1 – erster String • in2 – zweiter String • inx – letzter String
Rückgabewerte	• -
ret = uart.recv(id, timer_id, timeout) – empfängt ein Datenpaket	
Argumente	• id – UART-Controller ID • timer_id – Timer ID • timeout – gewünschte Timeouteinstellung. Mögliche Einstellungen sind uart.NO_TIMEOUT, uart.INF_TIMEOUT oder ein spezifischer Wert in µs
Rückgabewerte	• ret – Empfangenes Datenpaket

Tabelle 4 - Funktionsübersicht UART-Modul

3.3.2. Die Timer-Counter-Schnittstelle

Die Timer-Counter-Schnittstelle des AT91SAM7X256 Mikrocontrollers verfügt über 3 identische 16 Bit Kanäle. Die einzelnen Kanäle können unabhängig voneinander programmiert werden, um eine große Spanne von Funktionen umzusetzen. Dazu gehören Frequenzmessungen, Event Zählungen, Intervallmessungen, Pulsgeneration, das Einstellen von Verzögerungszeiten und Pulsweitenmodulation. Jeder der drei Kanäle verfügt

über drei externe, fünf interne Takteingange und zwei mehrfachverwendbare Ein- und Ausgangssignale die durch den Nutzer konfiguriert werden können. (4)

Für die Nutzung der Timerfunktionen steht in eLua das Timer-Modul zur Verfügung. In diesem Kapitel wird darauf eingegangen, welche Möglichkeiten dieses Modul bietet.

Die erste Funktion, die hier beschrieben wird, ist die Startfunktion tmr.start(), die den Timer mit der jeweiligen *id* anschaltet. Als Rückgabewert wird eine Null auf den Stack gelegt[5].

Um dem Nutzer die Möglichkeit zu geben mit Verzögerungszeiten zu arbeiten, wie zum Beispiel für die zeitliche Ansteuerung eines Gleichstrommotors, stellt das Modul die Funktion tmr.delay() bereit. Die Verzögerungsfunktion bekommt zwei Übergabewerte, die *id* des Zählers, der genutzt werden soll und die gewünschte Verzögerungszeit in µs. Die Zähler-ID muss beim AT91SAM7X256 jedoch immer zwischen 0 und 2 liegen, da er ja nur über 3 Zähler verfügt. Die Methode ruft eine plattformspezifische Funktion auf, in der die übergebene Verzögerungszeit mit Hilfe des Zählertaktes in eine Vorzeichen unbehaftete 64 Bit Zahl umgewandelt wird. Diese Zahl wird jedoch auf 65535 (= 2^{16}-1), also auf 16 Bit, begrenzt. Dies ist der Fall, da der Timer nur über 16 Bit Kanäle verfügt. Danach wird der Zähler gestartet, der dann solange zählt, bis der gewünschte Wert erreicht ist. Um Fehler bei der Nutzung der Verzögerungsfunktion zu vermeiden, hilft die nachfolgende Formel.

$$Zählwert = \frac{gewünschte\ Verzögerungszeit[\mu s] * Taktfrequenz\ des\ TC\_Kanals}{1.000.000}$$

Der daraus resultierende Zählwert darf nicht größer sein als 65535.

Damit der eben beschriebene Fehler, also die Einstellung einer nichtdurchführbaren Verzögerungszeit, nicht auftreten kann, stellt eLua zwei Hilfsfunktionen bereit: tmr.mindelay() und tmr.maxdelay(). Beide Funktionen erhalten als Übergabeparameter

[5] Der Rückgabewert kann hier je nach Plattform abweichen

den Wert *id* für die Channel-Auswahl. tmr.maxdelay() gibt dem Nutzer als Rückgabewert die maximal einstellbare Verzögerungszeit in µs zurück. Die Funktion tmr.mindelay() funktioniert analog zu tmr.maxdelay() und gibt somit die minimal einstellbare Verzögerungszeit in µs zurück. Sollte die minimal einstellbare Verzögerungszeit des Mikrocontrollers unter dem Mikrosekundenbereich liegen erhält man einen Rückgabewert von 0.

Um dem eLua-Nutzer die Möglichkeit zu geben selbst mit Zählwerten zu arbeiten, bietet das Modul zudem die Möglichkeit über die Funktion tmr.read() den aktuellen Wert des Zählers abzufragen. Als Übergabeparameter benötigt die Funktion nur den Wert id um auf den richtigen der drei Zähler zugreifen zu können. Als Rückgabewert erhält man eine vorzeichenunbehaftete 32-Bit Zahl, wobei der Wert jedoch, wie oben bereits erwähnt, nur zwischen 0 und 65535 liegen kann[5].

Eine weitere Hilfsfunktion die dem Nutzer zur Verfügung steht ist die Funktion tmr.diff(). Diese Funktion berechnet die Zeit in µs die während der Zählung von Übergabewert 1 bis Übergabewert 2 verstreicht. Als Übergabewerte bekommt die Funktion somit die Zähler-ID, den Anfangs- und Endwert der Zählung.

Als letztes stellt das Modul noch die Funktionen tmr.setclock() und tmr.getclock() zur Verfügung. Durch getclock() kann man sich die aktuelle Taktfrequenz des jeweiligen Kanals ausgeben lassen. Die Funktion setclock() bekommt abgesehen von der ID des jeweiligen Kanals auch noch eine Taktfrequenz[6] übergeben. Wie der Funktionsname bereits sagt, stellt setclock() dann die gewünschte Taktfrequenz ein und gibt diese dann auch als Rückgabewert zurück.

Die abschließende Übersicht in Tabelle 5 stellt noch einmal die Funktionalitäten des Moduls im Überblick dar.

[6] Die einstellbaren Taktfrequenzen können aus Tabelle 32-1 der Quelle (4) entnommen werden

ret = timer.setclock(id, clock) – stellt die Taktfrequenz des jeweiligen Timers ein	
Argumente	• id – Timer ID • clock – gewünschte Taktfrequenz des Timers in Hz
Rückgabewerte	• ret – eingestellte Taktfrequenz in Hz
ret = timer.getclock(id) – gibt die eingestellte Taktfrequenz zurück	
Argumente	• id – Timer ID
Rückgabewerte	• ret – eingestellte Taktfrequenz in Hz
ret = timer.start(id) – startet den Timer	
Argumente	• id – Timer ID
Rückgabewerte	• ret = aktueller Zählwert
timer.delay(id, delay) – wartet eine bestimmte Zeit	
Argumente	• id – Timer ID • delay – gewünschte Wartezeit in µs
Rückgabewerte	• -
ret = timer.mindelay(id) – gibt minimal einstellbare Verzögerungszeit zurück	
Argumente	• id – Timer ID
Rückgabewerte	• ret – minimal einstellbare Verzögerungszeit in µs
ret = timer.maxdelay(id) – gibt maximal einstellbare Verzögerungszeit zurück	
Argumente	• id – Timer ID
Rückgabewerte	• ret – maximal einstellbare Verzögerungszeit in µs
ret = timer.read(id) – gibt den Zählerstand zurück	
Argumente	• id – Timer ID
Rückgabewerte	• ret – aktueller Zählerstand
ret = timer.diff(id, end, start) – gibt die verstrichene Zet zwischen 2 Zählerständen zurück	
Argumente	• id – Timer ID timer_id – Timer ID • end – Endwert • start – Startwert
Rückgabewerte	• ret – verstrichene Zeit in µs

Tabelle 5 - Funktionsübersicht Timer-Modul

3.3.3. Die Pulsweitenmodulations-Schnittstelle

Der AT91SAM7X256 Mikrocontroller verfügt über einen eigenen Pulsweiten Modulations Controller. Mit Hilfe diesen Controllers können vier verschiedene Kanäle zur Erzeugung von Rechtecksignalen unabhängig voneinander angesteuert werden. Der Nutzer kann die charakteristischen Eigenschaften des Signals, wie die Periodendauer, das Puls-Pause-Verhältnis und die Polarität über die Programmierschnittstelle verändern (4). Nachfolgend wird ein kurzer Überblick über die Ansteuerung der Schnittstelle durch eLua gegeben. Zu diesem Zweck werden die umgesetzten Funktionen in eLua beschrieben.

Mit der Initialisierungsfunktion pwm.setup() kann man die Periodendauer und das Puls-Pause-Verhältnis jedes einzelnen PWM-Kanals einstellen. Die Funktion erhält drei Übergabewerte. Der erste Wert ist die *id* des einzustellenden Kanals. Da der AT91SAM7X256 über vier PWM-Kanäle verfügt, können diese Werte zwischen 0 und 3 liegen. Der zweite Übergabeparameter ist die Frequenz, also der Kehrwert der Periodendauer. Um die Taktfrequenz des Pulsweiten Modulationscontrollers zu berücksichtigen, wird die Umrechnung intern mit folgender Formel durchgeführt:

$$Periodendauer = \frac{PWM\ Taktfrequnez}{Gewünschte\ Signaltaktfrequenz}$$

Die Periodendauer wird dann durch die Funktion in das Period Register des angesprochenen Kanals geschrieben. Als letzten Übergabeparameter bekommt die Funktion, wie bereits oben angesprochen das gewünschte Puls-Pause-Verhältnis in Prozent. Dabei können Werte zwischen 0 und 100 vergeben werden. Somit ergeben sich folgende Formeln für die An- und Ausschaltzeit:

$$Anschaltzeit = \frac{Periodendauer * Puls\_Pause\_Verhältnis}{100}$$

$$Ausschaltzeit = \frac{Periodendauer * (100 - Puls\_Pause\_Verhältnis)}{100}$$

Die Funktion berechnet intern die Anschaltzeit und schreibt diese dann in das Duty Cycle Register des Kanals. Die Initialisierungsfunktion gibt dem User die eingestellte Signalfrequenz zurück.

Um auf die Taktfrequenz des PWM Controllers zugreifen zu können, stellt eLua zwei Funktionen zur Verfügung. Mit Hilfe der Funktion pwm.setclock() lässt sich die Frequenz einstellen. Der PWM Controller des AT91SAM7X256 ist hierbei in zwei verschiedene Blöcke gegliedert. Die Kanäle 0 und 1 sind in Block A und die Kanäle 2 und 3 sind in Block B zusammengefasst. Das Positive an der Implementierung der Funktionen in eLua ist, dass sich der Anwender nur bedingt überlegen muss, welche Kanäle, wie zusammengefasst sind, da er einfach nur die Channel-ID übergeben muss und die Funktion

setclock() übernimmt die Eintragung der Werte in den richtigen Block. Somit ergeben sich als Übergabeparameter die Channel-ID und die gewünschte Taktfrequenz. Die Funktion errechnet intern, die für die gewählte Taktfrequenz benötigten Werte für den Prescaler und den Divisor, die in das Mode Register des PWM-Controllers eingetragen werden[7]. Die Funktion pwm.getclock() bekommt ebenfalls die Channel-ID übegeben und gibt wie auch setclock() die aktuelle Taktfrequenz des Blockes zurück.

Die beiden verbleibenden Funktionen des Moduls sind die Start- und Stopfunktionen der Pulsweitenmodulation. Beide Funktionen erhalten als Übergabeparameter die Channel-ID. pwm.start() konfiguriert die benötigten Pins und startet die Signalerzeugung. Die inverse Funktion hierzu ist pwm.stop(). Diese stoppt die Signalerzeugung und gibt die Pins wieder frei. Beide Funktionen haben keine Rückgabewerte.

Abschließend soll die folgende Übersicht, wie in den vorherigen Kapiteln, die Funktionen des Moduls kurz zusammenfassen:

ret = pwm.setup(id, clock, duty) – nimmt die grundlegenden Einstellungen vor	
Argumente	• id – PWM ID • clock – gewünschte Signalfrequenz in Hz • duty – gewünschtes Puls-Pause-Verhältnis. Hierbei sind Werte zwischen 0 und 100 zulässig
Rückgabewerte	• ret – eingestellte Signalfrequenz in Hz
ret = pwm.setclock(id, clock) – stellt die gewünschte Taktfrequenz des PWM-Blockes ein	
Argumente	• id – PWM ID • clock – gewünschte Taktfrequenz in Hz
Rückgabewerte	• ret – eingestellte Taktfrequenz in Hz
ret = pwm.getclock(id) – gibt die Taktfrequenz des PWM-Blockes zurück	
Argumente	• id – PWM ID
Rückgabewerte	• ret – eingestellte Taktfrequenz in Hz
pwm.start(id) – startet die Pulsweitenmodulation	
Argumente	• id – PWM ID
Rückgabewerte	• -
pwm.stop(id) – stoppt die Pulsweitenmodulation	
Argumente	• id – PWM ID
Rückgabewerte	• -

Tabelle 6 - Funktionsübersicht PWM-Modul

[7] Die möglichen Einstellungen für Prescaler und Divisor können auf Seite 434 aus (4) entnommen werden

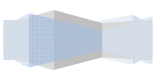
3.3.4. Die programmierbare Input/Output-Schnittstelle

Eine der wahrscheinlich wichtigsten Schnittstellen ist die programmierbare parallele Ein- und Ausgangsschnittstelle des AT91SAM7X256. Sämtliche Ein- und Ausgänge sind über den PIO-Controller mit den anderen Peripheriegeräten verbunden. Die einzige Ausnahme hierbei sind die analogen Eingänge AD4 bis AD7[8]. Der PIO-Controller verwaltet 32 voll programmierbare Ein- bzw. Ausgänge. Jeder Ein- bzw. Ausgang kann entweder als Anschluss für die integrierten Peripheriegeräte dienen, oder für den allgemeinen Verbrauch als Ein- bzw. Ausgangspin verwendet werden. Jeder Pin hat einen festen Wert in den 32-Bit PIO-Registern. Außerdem ermöglicht es der PIO-Controller, bis zu zwei verschiedene interne Peripheriegeräte auf einem Pin zu multiplexen. Dies geschieht durch den Gebrauch von zwei Auswahlregistern – PIO_ASR und PIO_BSR. Welches der beiden Peripheriegeräte gerade den Pin nutzt, kann man im AB Select Status Register ablesen. Enthält das Register den Wert 0 so wird PIO_ASR genutzt, der Wert 1 zeigt die Nutzung von PIO_BSR an. Diese Mehrfachnutzung kann allerdings nur bei Ausgängen erfolgen. Jeder der Ein- bzw. Ausgangspins verfügt über einen integrierten Pull-up Widerstand, der durch das Pull-up Enable Register und das Pull-up Disable Register an- bzw. ausgeschaltet werden kann (4). Die nachfolgenden Ausführungen sollen dem Leser die Verwendung des PIO-Controllers mit eLua nahebringen.

Die Ansteuerung des PIO-Controllers erfolgt, wie bei jedem Peripheriegerät in eLua über ein dafür geschriebenes Softwaremodul. In dem Modul werden symbolische Variablennamen für die einzelnen Pins und für ganze Ports verwendet. In eLua kann man auf Pins mit pio.PA_0 bis pio.PB_30 und auf Ports mit pio.PA bis pio.PF zugreifen. Zudem stellt das PIO-Modul sowohl Funktionen zur Ansteuerung von Pins, als auch von Ports zur Verfügung. Als erstes werden hier die „Pin-Funktionen“, im Anschluss die „Port-Funktionen“ beschrieben.

Um einen oder mehrere Pins als Ein- oder Ausgang zu definieren, stellt eLua die Funktionen pio.input() und pio.output()zur Verfügung. Beide Funktionen arbeiten vom Prinzip her exakt gleich. Als Übergabeparameter fungieren ein oder mehrere symbolische Na-

[8] Vergleiche hierzu das Blockdiagramm auf Seite 4 von Quelle (4)



men der Pins. Die Funktion überprüft zunächst einmal, ob der oder die besagten Pins vorhanden sind. Falls die Pins nicht vorhanden sind, geben die beiden Funktionen eine Fehlermeldung auf der Shell aus. Kommt es zu keinem Fehler, so wird die gewünschte Operation mit Hilfe von unterstützenden Funktionen ausgeführt. Die Funktion pio.input() definiert den gewählten Pin daraufhin als Eingang und pio.output() definiert diesen als Ausgang.

Sind Pins als Ausgänge definiert, bietet eLua einige Funktionen um mit diesen Ausgängen zu arbeiten. Die Funktion pio.clear() erhält analog den vorhergehenden Funktionen wiederum einen oder mehrere Pins. Somit werden die Werte aller angegebenen Pins auf 0 gesetzt. Umgekehrt gilt für die Funktion pio.set(), dass alle angegebenen Pins auf den Wert 1 gesetzt werden. Für die direkte Werteingabe steht die Funktion pio.setpin() zur Verfügung. Zusätzlich zu den Pins wird im ersten Übergabeparameter definiert, welcher Wert, 0 oder 1, für alle angegebenen Pins gesetzt werden soll.

Ist ein Pin als Eingang registriert, so ist es in eLua möglich mit der Funktion pio.get() den an den übergebenen Pins anliegenden Wert abzufragen. Die Funktion legt dann nacheinander, d.h. in der Reihenfolge der übergebenen Pins die Werte auf dem Stack ab[9].

Die nächsten drei Funktionen befassen sich mit den integrierten Pull-up und Pull-down Widerständen. pio.pullup() und pio.pulldown()[10] aktivieren bei ihrem Aufruf die integrierten Pull-up bzw. Pull-down Widerstände der übergebenen Pins, falls diese über einen solchen Widerstand verfügen. Mit pio.nopull() kann man die integrierten Widerstände der jeweiligen Pins wieder abschalten.

Die letzte Pin-spezifische Funktion des Moduls ist pio.pin(). Diese Funktion bekommt einen symbolischen Namen eines Pins übergeben und gibt die physikalische Pin-Nummer zurück.

[9] Die Werte können hier entweder 0 oder 1 sein
[10] pio.pulldown() wird her nur vollständiger weise erwähnt, da der AT91SAM7X256 nur integrierte Pull-up Widerstände besitzt

Die Port-spezifischen Funktionen können analog zu den Pin-spezifischen Funktionen verwendet werden. Anstelle der symbolischen Pin-Variablen werden nun die Variablen der Ports übergeben. Die genaue Auflistung der Funktionen kann der abschließenden Übersichttabelle des Moduls entnommen werden. Aufgrund der Größe der Tabelle 23 befindet sich diese im Anhang.

4. Implementierung von Schnittstellen

In diesem Teil der Arbeit wird speziell darauf eingegangen, wie neue Schnittstellen in eLua implementiert und somit integriert werden können. Als erstes wird auf die allgemeine Vorgehensweise eingegangen, im Anschluss wird die Implementierung am Beispiel verschiedener Schnittstellen beschrieben.

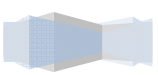
4.1.Allgemein Vorgehensweise

Um die Portabilität von eLua möglichst optimal gewährleisten zu können, wird darauf geachtet, den Quellcode möglichst allgemein zu halten. Da eLua auf den unterschiedlichsten Mikrocontrollern verfügbar sein soll, haben sich die Entwickler entschieden, das Projekt in plattformunabhängige und plattformspezifische Module zu unterteilen. Die plattformunabhängigen Module stellen die Schnittstelle zu Lua bereit. Die plattformspezifischen Module ermöglichen die Implementierung der unterschiedlichsten Hardware und deren Anforderungen.

4.1.1. Plattformunabhängige Module

Die allgemeinen Module unterstützen, wie oben bereits erwähnt, die Portabilität der einzelnen Schnittstellenmodule. Diese C-Dateien werden beim eLua-Projekt im Verzeichnis „elua-5.0\src\modules“ abgespeichert.

Die wichtigste Datei in diesem Verzeichnis ist der Header mit dem Namen „auxmods.h“. In dieser Headerdatei werden die Bibliotheksnamen aller Behelfsmodule, die später in eLua registriert werden sollen, in Form von symbolischen Kostanten definiert. Außerdem werden sämtliche Prototypen der Lua Registrierungsfunktionen der einzelnen Module aufgelistet. Die wichtigste Funktion, besser gesagt das wichtigste Makro in diesem Header ist jedoch *MOD_CHECK_ID(mod, id)*. Dieses Makro spielt später bei jedem Lua-Funktionsaufruf eine wichtige Rolle. Sie überprüft, ob die jeweilige Plattform ein spezielles Schnittstellenmodul besitzt und gibt eine Fehlermeldung zurück, falls dies nicht der Fall sein sollte.

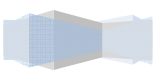


Die restlichen Dateien in diesem Verzeichnis sind C-Sourcecode-Dateien, also die jeweiligen plattformunabhängigen Module für bestimmte Funktionalitäten, wie zum Beispiel pio.c für die parallele Ein- und Ausgabeschnittstelle oder pwm.c für die Pulsweitenmodulation. In den C-Source-Dateien werden die Bibliotheksfunktionen definiert, die später von Lua auf den Mikrocontrollern aufgerufen werden können. Je nach Art der Schnittstelle werden die benötigten Funktionen definiert. Bei UART gibt es z. B. eine Sendefunktion, eine Empfangsfunktion und eine String-Sendefunktion. Sämtliche hier definierten Funktionen stellen die Schnittstelle zwischen Lua und den plattformabhängigen Funktionen dar. Die weiteren Funktionsaufrufe innerhalb der Bibliotheksfunktionen beziehen sich dann direkt auf die plattformspezifischen Module, die nicht mehr allgemein anwendbar sind, mehr dazu im Kapitel 4.1.2.

Zudem wird sozusagen global eine Headerdatei definiert, in der die Prototypen der plattformspezifischen Funktionen definiert werden. Die Datei trägt den Namen „platform.h“ und ist unter dem Verzeichnis „elua-5.0\inc“ gespeichert. In ihr werden die Prototypen und symbolischen Konstanten für sämtliche Mikrocontroller definiert. Benötigt eine Plattform diese Prototypen und Konstanten nicht, werden sie in der Datei „platform.c“ einfach nicht ausprogrammiert bzw. verwendet. Somit erhält man einen plattformunabhängigen Header, in dem man die benötigten Prototypen aller neuen Schnittstellen definiert.

Abschließend stellt sich die Frage, wie Bibliotheksfunktionen als solche innerhalb von Lua registriert werden. Die Antwort auf diese Frage steckt in der Funktion *luaopen_MODULENAME()*, wobei MODULNAME für den Namen der jeweiligen Schnittstelle steht. In dieser Funktion wird die Methode *luaL_register()* aufgerufen. *luaL_register()* hat drei Übergabewerte: *lua_State* L*[11], *const char* libname und luaL_Reg * l*. Die Variable *l* ist ein Feld, in dem eine „Karte“ der Modulfunktionen abgelegt ist. Das folgende Listing 2 soll den Aufbau eines solchen Feldes verdeutlichen. Diese Registrierung erfolgt in den Quellcodedateien der plattformunabhängigen Module.

[11] In der dynamischen Struktur lua_State wird der gesamte Zustand der Lua Bibliothek verwaltet. Jeder Funktion in Lua wird ein Zeiger auf diese Zustandsvariable übergeben(lua_State*).



```
01 static const luaL_reg uart_map[] =
02 {
03 { "setup",  uart_setup },
04 { "send", uart_send },
05 { "recv", uart_recv },
06 { "sendstr", uart_sendstr },
07 { NULL, NULL }
08 };
```

Listing 2 - Bespiel Functionmap (3)

Die Feldeinträge bestehen aus zwei Teilen, dem Namen der Funktion (z.B. send) und einem Zeiger auf die Funktion (z.B. uart_send). Das Feld muss immer mit dem Eintrag {NULL, NULL} abgeschlossen werden.
*luaL_register()*erstellt eine Tabelle auf dem Stack, die den Namen der Variable *libname* trägt und registriert diese als geladenes Modul mit sämtlichen Funktionen die in der Variablen *l* gespeichert sind.

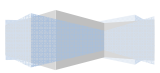
Für die Kompilierung mit *SCons* muss man in der Datei „SConstruct“ die Module angeben die übersetzt werden sollen. Dazu trägt man bei der Variablen *module_names* den Dateinamen des jeweiligen Moduls ein, z.B. „uart.c“.

Wichtig ist, abschließend noch einmal zu erwähnen, dass diese plattformunabhängigen Module nie wieder verändert werden müssen, sobald sie einmal geschrieben und getestet sind. Auch nicht, wenn embedded Lua auf einem anderen Mikrocontroller laufen soll. Es müssen beim Hinzufügen eines neuen Controllers ausschließlich die plattformspezifischen Module für den jeweiligen Mikrocontroller neu- bzw. umgeschrieben werden.

4.1.2. Plattformspezifische Module

Obwohl eLua portabel für jeden Mikrocontroller gestaltet werden soll, kommt man nicht um einige plattformspezifischen Erweiterungen herum, die exakt auf die jeweilige Hardware abgestimmt sind. Dieses Kapitel stellt möglichst allgemein dar, wie ein solches plattformspezifisches Modul implementiert und eingebunden wird.

Zunächst muss man auf die Headerdatei „platform_conf.h“ eingehen. Diese Datei muss in jedem „Plattformverzeichnis“ angelegt sein. In ihr wird festgelegt, welche Komponenten von eLua für den jeweiligen Mikrocontroller eingebunden werden sollen. Darüber hin-



aus werden einige Konfigurationsdaten mit Hilfe von symbolischen Konstanten definiert. Der wichtigste Abschnitt der Datei für die Betrachtung aus der Sicht der Schnittstellenimplementierung, ist jedoch der, in dem definiert wird, welche Behelfsbibliotheken für den entsprechenden Mikrocontroller in eLua eingebunden werden sollen.

Listing 3 zeigt einen Ausschnitt aus „platform_conf.h“ in dem die Behelfsbibliotheken für den AT91SAM7X256 definiert werden. Die Bibliotheken werden in der symbolischen Konstanten LUA_PLATFROM_LIBS abgelegt.

```
01 #define LUA_PLATFORM_LIBS\
02 { AUXLIB_PIO, luaopen_pio },\
03 { AUXLIB_CAN, luaopen_can },\
04 { AUXLIB_AIC, luaopen_aic },\
05 { AUXLIB_TMR, luaopen_tmr },\
06 { AUXLIB_PD, luaopen_pd },\
07 { AUXLIB_UART, luaopen_uart },\
08 { AUXLIB_TERM, luaopen_term },\
09 { AUXLIB_PWM, luaopen_pwm },\
10 { AUXLIB_PACK, luaopen_pack },\
11 { AUXLIB_BIT, luaopen_bit },\
12 { LUA_MATHLIBNAME, luaopen_math }
```

Listing 3 - Definition der Behelfsbibliotheken für AT91SAM7X512 (3)

Der Aufbau dieser Konstanten wird durch das Listing 3 deutlich. Der erste Parameter ist immer der Bibliotheksname, der zweite Parameter ist der Funktionsname der Registrierungsfunktion aus dem plattformunabhängigen Teil der Implementierung (siehe Seite 31 *luaopen*).

Anmerkung: Die tatsächliche Registrierung der Behelfsbibliotheken geschieht beim Kompilieren von eLua zusammen mit der Registrierung der Lua-Standardbibliotheken (siehe Seite VII **elua-0.5\src\lua\linit.c**).

In der Datei „platform.c“ werden nun die in der Headerdatei „platform.h“ definierten Prototypen ausprogrammiert. Dabei sollte man sich immer vor Augen halten, dass nicht alle Funktionen ausprogrammiert werden müssen, die in dem Header definiert sind. Die Funktion platform_init() muss jedoch auf jeden Fall entwickelt werden. Mit Hilfe dieser Funktion werden grundlegende Einstellungen im Bezug auf den Mikrocontroller vorgenommen. Beim AT91SAM7X256 werden zum Beispiel sämtliche Peripheriegeräte, die

vom Mikrocontroller verwendet werden im Power Management Controller (PMC) aktiviert. Die Initialisierungsfunktion sollte PLATFORM_OK als Rückgabeparameter haben. Falls bei der Initialisierung etwas schief geht und der Rückgabewert somit „False“ ist, hat dies fatale Folgen, denn dann läuft das Main-Programm in eine Endlosschleife und eLua kann nicht auf dem Mikrocontroller integriert werden. Die restlichen Funktionen sind, wie oben bereits erwähnt nicht unbedingt notwendig. Bei ihnen handelt es sich um Funktionen, die von den in den plattformunabhängigen Modulen definierten Funktionen aufgerufen werden, um die jeweilige Schnittstelle anzusteuern. Damit sind die in „platform.c“ definierten Funktionen das Bindeglied zwischen den eLua-Funktionen und der Hardware. Abschließend sollte man noch die Funktion platform_MODULENAME_exists()[12] erwähnen. Mit Hilfe dieser Funktion werden nochmals Fehler abgefangen. Sie wird bei jedem eLua-Funktionsaufruf in den plattformunabhängigen Modulen aufgerufen, um sicherzustellen, ob die jeweilige Schnittstelle bei der aktuellen Plattform überhaupt vorhanden ist. Wie aus Listing 4 ersichtlich wird, handelt es sich dabei nur um eine Funktion die einen Wert *id* als Übergabeparameter bekommt und diese mit einer hardwarespezifischen Vorgabe vergleicht.

```
01 int platform_uart_exists( unsigned id )
02 {
03   return id <= 1;
04 }
```

Listing 4 - Beispiel platform_MODULNAME_exists()

Der Wert von ID ist dabei abhängig von der Anzahl der Ports der jeweiligen Schnittstelle.

Um die Datei „platform.c“ nicht unnötig aufzublähen, werden die von den Herstellern der Mikrocontroller vordefinierten Funktionen in eigenständigen Source-Code-Dateien (z.B „can.c“) und den zugehörigen Headerdateien abgelegt. Diese Strukturen befinden sich in dem Verzeichnis des jeweiligen Controllers.

Für die Kompilierung gibt es, wie bei den plattformunabhängigen Modulen, eine Datei, in der festgelegt wird, welche Dateien übersetzt werden sollen. Diese Datei ist diesmal ein Python-File mit dem Namen „conf.py“. Hier werden in der Variablen *specific_files* die Namen der zu verwendenden plattformspezifischen Dateien konfiguriert (z.B. „uart.c“).

[12] MODULNAME ist der Name eines Schnittstellenmoduls, wie z.B. can, pwm oder uart

Abbildung 12 - Vorgehensweise zur Schnittstellenimplementierung im Anhang stellt die Vorgehensweise der Schnittstellenimplementierung noch einmal übersichtlich dar.

4.2.Implementierung der Analogen Eingänge

Die erste Schnittstelle die implementiert wird, ist zugleich die denkbar Einfachste – die analogen Eingänge. Im Folgenden wird kurz darauf eingegangen, welche Funktionen der Mikrocontroller im Bezug auf die analogen Eingänge mit sich bringt. Anschließend wird die Umsetzung dieser Funktionen in eLua beschrieben.

4.2.1. Allgemeines zu den analogen Eingängen

Die AT91SAM7X-Mikrocontrollerfamilie besitzt neun analoge Eingänge. Acht dieser Eingänge sind über einen Multiplexer mit einem Analog-Digital-Wandler (ADC) verbunden. Durch den Multiplexer wird bewerkstelligt, dass der AD-Wandler die einzelnen Pins abarbeiten kann. Man kann die analogen Eingänge zudem in zwei Arten aufteilen. Die Eingange AD4 bis AD7 sind direkt mit dem Multiplexer verbunden. Bei den restlichen vier Eingängen ist der Programmierbare Input/Output Controller (PIO) dazwischen geschalten. Dies spielt insofern eine entscheidende Rolle, da der Entwickler die Eingänge AD0 bis AD3 somit später nicht nur mit Hilfe des ADC Moduls einbinden, sondern die Pins auch mit Hilfe des PIO Moduls registrieren muss. Der Neunte Eingang ist direkt mit dem AD-Wandler verbunden. Bei diesem Eingang handelt es sich um ADVREF. Der Eingang gibt eine Referenzspannung für die Eingänge an, mit deren Hilfe der AD-Wandler die Werte der einzelnen Eingänge errechnet. Abbildung 1 soll dies schematisch darstellen:

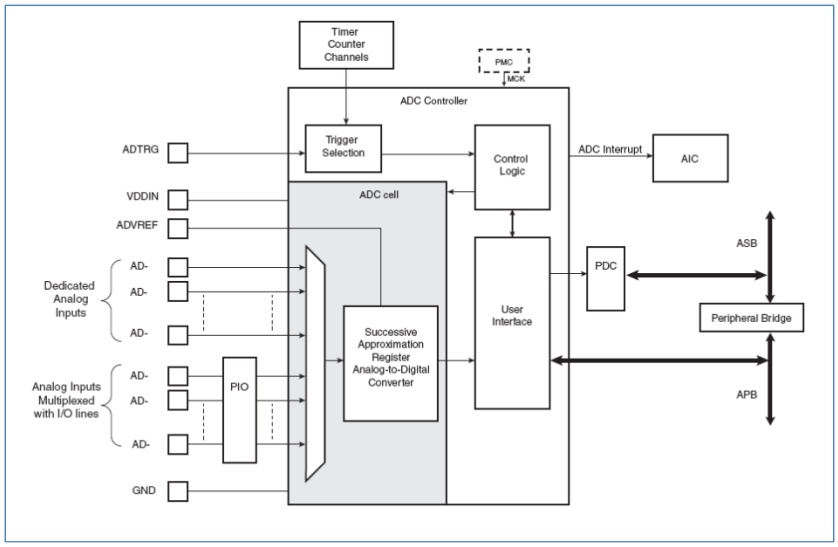


Abbildung 1 - Schematische Darstellung ADC Controller (4)

Wie aus der Grafik ersichtlich wird, ist der ADC Controller auch mit dem Interruptcontroller (AIC) verbunden, wodurch die Auswertung der Daten über Interrupts bewerkstelligt werden kann.

Abschließend wird in diesem Abschnitt die Konvertierung der Daten beschrieben. Für diese Umwandlung kann man zwischen einer 8 und 10 Bit-Auflösung auswählen. Wie oben bereits erwähnt wurde, dient hierbei der Wert ADVREF als Referenzwert für die Umwandlung, was wiederum zur Folge hat, dass nur Werte zwischen 0 Volt und dem Referenzwert anliegen dürfen.

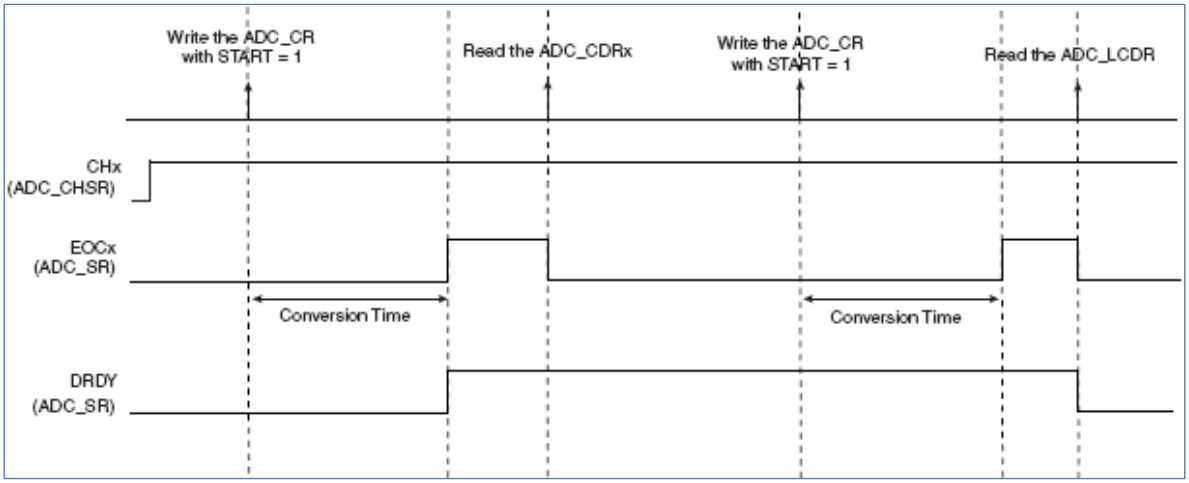


Abbildung 2 - Bereitstellung der umgewandelten Daten (4)

Abbildung 2 stellt den zeitlichen Verlauf der Bereitstellung der Daten dar. Aus dieser Darstellung erkennt man, dass die Bereitstellung der Daten eine gewisse Zeit benötigt, die sogenannte „Conversion Time“. Diese Zeit ist von der Taktfrequenz abhängig. Ist die Umwandlung beendet, stehen die Daten im Channel Data Register des jeweiligen Kanals zur Verfügung. Zudem wird das End of Conversion Bit des Kanals und das Data Ready Register gesetzt. Sobald die Daten aus dem Channel Data Register ausgelesen werden, wird das End of Conversion Bit wieder rückgesetzt. Das Data ready Register wird erst wieder rückgesetzt, wenn ein Zugriff auf Last Converted Data Register stattgefunden hat. Hierbei ist es wichtig anzumerken, dass der jeweilige Channel aktiviert sein muss, damit korrekte Daten in den Registern stehen. Sind mehrere Kanäle aktiviert, so werden diese nacheinander abgearbeitet.

4.2.2. Umsetzung der Implementierung

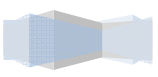
Um die problemlose Nutzung der AD-Wandlung in eLua gewährleisten zu können, sind einige Funktionen nötig. In diesem Kapitel wird Schritt für Schritt an diese Funktionen herangeführt. Zusätzlich wird beschrieben, wie die Funktionen in eLua anzuwenden sind.

Anmerkung: Die hardwareunabhängigen eLua-Funktionen sind in der Datei „elua-0.5\src\modules\adc.c“ gespeichert. Die hardwarespezifischen Funktionen befinden sich in den Dateien „platform.c“ und „adc.c“ im Verzeichnis „elua-0.5\src\platform\at91sam7x“.

ADC Initialisierung

Wie in Kapitel 4.2.1 erwähnt wurde, gibt es für die Aufbereitung der Signale, die an den analogen Eingängen anliegen bestimmte Voraussetzungen. Mit der Initialisierungsfunktion sollen diese Grundeinstellungen vorgenommen werden. Die erforderlichen Einstellungen sind in diesem Fall die Taktfrequenz des ADC Controllers, die Startup Zeit, die Sample & Hold Zeit und die Auflösung der Datenumwandlung.

Als Schnittstelle zwischen eLua und der Hardware dient die Funktion adc_setup(). Aufgerufen wird sie im Lua-Code mit dem Ausdruck adc.setup(). Die Funktion erhält als Übergabeparameter die oben angesprochenen Einstellungswerte. Diese werden von der Funktion aus dem Stack ausgelesen und an eine plattformspezifische Funktion weitergegeben. Das nachfolgende Listing zeigt einen Ausschnitt dieser Funktion:



```
01 static int adc_setup( lua_State* L )
02 {
03 unsigned id,adc_freq,resolution,sample_hold_time,startuptime,tmp;
04 id = luaL_checkinteger(L,1);
05 MOD_CHECK_ID(adc,id);
06 adc_freq = luaL_checkinteger(L,2);
07 resolution = luaL_checkinteger(L,3);
08 sample_hold_time = luaL_checkinteger(L,4);
09 startuptime = luaL_checkinteger(L,5);
10 tmp = platform_adc_setup(id,adc_freq,sample_hold_time,startuptime);
11         .
12         .
13         .
14 lua_pushinteger( L, tmp );
15 return 1;}
```

Listing 5 - ADC Initialisierungsfunktion

Die Funktion platform_adc_setup() greift auf eine von Atmel bereitgestellte Funktion zu, welche für die übergebenen Parameter die jeweiligen Werte berechnet und diese in die dafür vorgesehenen Register einträgt.

Man benötigt hierbei drei Parameter:

- PRESCALER zur Einstellung der ADC Taktfrequenz
- SHITIM für die Sample & Hold Zeit
- STARTUP für die Einstellung der Startup Zeit

Die Werte berechnen sich nach folgenden Formeln:

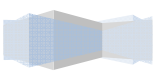
$$PRESCALER = \frac{Mikrocontrollertakt}{ADC\_Takt * 2} - 1$$

$$SHITIM = Sample\ \&\ Hold\ Zeit * ADC\_Takt$$

$$STARTUP = \frac{Startuptime * ADC\_Takt}{8} - 1$$

Falls die angegebenen Werte fehlerhaft sein sollten, (z.B. die gewünschte Taktfrequenz ist zu groß)wird in der eLua-Shell eine Fehlermeldung ausgegeben. Durch den Wert in der Variable *resolution* stellt die Funktion die gewünschte Auflösung ein. Die Auflösung kann, wie oben bereits erwähnt, auf 8 Bit oder 10 Bit eingestellt werden. Somit ist der Übergabeparameter entweder 8 oder 10.

Nach fehlerfreier Initialisierung können die einzelnen Eingänge aktiviert und benutzt werden.



Kanal Aktivierung und Deaktivierung

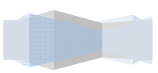
Wurde der ADC Controller, wie oben beschrieben, initialisiert, kann man damit beginnen, die benötigten Eingänge zu aktivieren. Dazu wird die Aktivierungsfunktion von eLua benötigt. Die Funktion wird durch den Befehl adc.enable() aufgerufen. Als Übergabeparameter bekommt die Funktion die Pin-Nummer des Eingangs – beim Eingang AD6 ist der Übergabeparameter also die Zahl 6. Die Anschaltfunktion ruft ihrerseits wieder eine plattformunabhängige Funktion auf, die eine 1 an die richtige Stelle des Channel Enable Registers des ADC Controllers schreibt. Die Deaktivierungsfunktion erfolgt durch den Aufruf adc.disable() analog zur Aktivierungsfunktion, mit dem Unterschied, dass diese den Wert 1 im Channel Disable Register des ADC Controllers setzt.

Start der AD-Wandlung

Mit dem Befehl adc.startconv() ist es möglich die an den Eingängen anliegenden Spannungssignale in einen digitalen Wert umzuwandeln. Die Konvertierung erfolgt durch den AD-Wandler. Hierfür ruft die Funktion eine plattformspezifische Routine auf, die im ADC Control Register das Startbit für den Beginn der AD-Wandlung setzt.

Der Datenabruf

Um die konvertierten Daten abrufen zu können, steht die Anweisung adc.getData() zur Verfügung. Diese greift mit Hilfe einer plattformspezifischen Implementierung auf ein Register des Mikrocontrollers zu. Es liest den Wert aus dem ADC Channel Data Register des angegebenen Kanals und gibt je nach Auflösung einen entsprechenden Wert zwischen 0 und 1023 (bei einer Auflösung von 10 Bit) bzw. zwischen 0 und 255 (bei einer Auflösung von 8 Bit) zurück. Der Wert, wird wie bei eLua üblich, auf dem Stack abgelegt. Die Auswertung des Ergebnisses obliegt dem Benutzer.



Interrupt Aktivierung und Deaktivierung

Um gewährleisten zu können, dass die analogen Eingänge auch mit Hilfe von Interrupts ausgewertet werden können, stellt das ADC-Modul die Funktionen adc.enint() und adc.disint() zur Verfügung. Mit Hilfe dieser beiden Funktionen kann der eLua-Nutzer die Interrupt-Unterstützung des jeweiligen Kanals aktivieren bzw. deaktivieren. Somit haben beide Funktionen als Übergabeparameter eine Zahl von 0 bis 7, also die Kanalnummer. Zurückgegeben wird ein String mit dem Text „interrupt enabled“ oder „interrupt disabled“. Wichtig ist hier jedoch noch anzumerken, dass zur Nutzung der Interrupt-Funktionen zuerst Einstellungen im Advanced Interrupt Controller vorgenommen werden müssen – hierzu mehr im Kapitel 4.3.

Abschließend wird in diesem Kapitel noch ein kurzer Überblick der Funktionen und ihrer Parameter in tabellarischer Form gegeben.

ret = adc.setup(id, clock, solution, shtime, starttime) – nimmt grundlegende Einstellungen vor	
Argumente	• id – ADC Controller ID • clock – gewünschte Taktfrequenz des ADC-Controllers in Hz. Dabei sind Werte zwischen 24MHz und 375kHz möglich • solution – gewünschte Auflösung 8 oder 10 Bit • shtime – gewünschte Sample&Hold-Zeit in s. Dabei sind Werte zwischen 0 und 16/clock möglich • starttime – gewünschte Startup-Zeit. Werte zwischen 8/clock und 128/clock möglich
Rückgabewerte	• ret – eingestellte ADC-Frequenz
adc.enable(channel) – aktiviert den jeweiligen ADC Pin	
Argumente	• channel – ADC-ID
Rückgabewerte	• -
adc.disable(channel) – deaktiviert den jeweiligen ADC Pin	
Argumente	• channel – ADC Kanal ID
Rückgabewert	• -
ret = adc.startconv() – startet die AD-Wandlung	
Argumente	• -
Rückgabewerte	• ret – Kontrollstring „conversion started“
ret = adc.getData(channel) – gibt die Daten des jeweiligen ADC Pins zurück	
Argumente	• channel – ADC-ID
Rückgabewerte	• ret – Wert der AD-Wandlung des entsprechenden Kanals
ret = adc.enint(channel) – aktiviert einen bestimmten Interrupt des AD-Wandlers	
Argumente	• channel – Art des Interrupts (siehe Datenblatt)
Rückgabewert	• ret – Kontrollstring „Interrupt is enabled“
ret = adc.disint(channel) – deaktiviert einen bestimmten Interrupt des AD-Wandlers	
Argumente	• channel – Art des Interrupts (siehe Datenblatt)
Rückgabewert	• ret – Kontrollstring „Interrupt is disabled“

Tabelle 7 - Funktionsübersicht des ADC-Moduls

4.2.3. Anwendungsbeispiel ADC-Modul

Um dem Leser die Nutzung des ADC-Moduls noch einmal zu verdeutlichen, legt dieses Kapitel das ADC-Modul anhand eines kurzen Beispiels dar. In diesem Beispiel soll die Spannung ausgegeben werden, die am Trimmer des Olimex Entwicklungsboards anliegt.

```
01 print "Bitte am Trimmer des OlimexBoards die Voltzahl einstellen"

02 adc.setup(0,5000000,10,8,1200)
03 adc.enable(6)

04 while true do
05   adc.startconv()
06   tmp = adc.getData(6)
07   tmp = (3.3 * tmp)/1023  --VREF = 3.3V
08   if (tmp == 3.3) then
09     print "end"
10     break
11   end
12 tostring(tmp)
13 print( "Voltage in V" .. tmp)
14 end
```

Listing 6 - Beispielprogramm ADC-Modul

Das Programm initialisiert den ADC-Controller, aktiviert den Kanal AD6 und prüft in einer Schleife den Wert am Potentiometer. Da der Wert des AD-Wandlers bei einer Auflösung von 10 Bit zwischen 0 und 1023 liegen kann, erfolgt zur Angabe der Spannung eine Umrechnung. Dreht man das Potentiometer auf den rechten Endanschlag, welcher dem Maximalwert von 3,3 Volt entspricht, gibt das Programm „end“ aus und verlässt die Schleife.

4.3.Implementierung von Interrupts

Da für die meisten Anwendungen mit Mikrocontrollern in der Praxis Interruptroutinen eine wichtige Rolle spielen, soll auch eLua Interrupts unterstützen. Im Folgenden wird kurz darauf eingegangen, welche Funktionen der Mikrocontroller im Bezug auf Interrupts mit sich bringt und danach wird die Umsetzung dieser Funktionen in eLua beschrieben. Abschließend wird auf die Probleme bei der Nutzung von Interrupts in eLua eingegangen.

4.3.1. Allgemeines zum Advanced Interrupt Controller

Die AT91SAM7X-Mikrocontrollerfamilie verfügt über einen eigenen Interrupt Controller. Der sogenannte Advanced Interrupt Controller kann bis zu 32 Interrupt-Funktionen verwalten. Dieser Controller verwaltet die nFIQ[13]- und nIRQ[14]-Eingänge des ARM Prozessors. Der Controller verfügt zudem über acht verschieden Prioritätsstufen, mit denen jede einzelne Interrupt-Quelle belegt werden kann. Abbildung 3 zeigt den detaillierten Aufbau des AIC-Controllers.

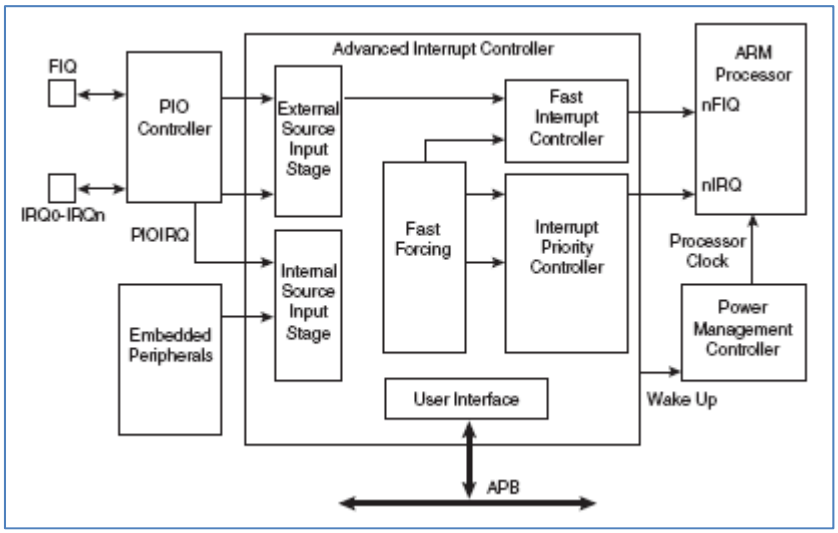


Abbildung 3 - Schematische Darstellung des Advanced Interrupt Controllers (4)

[13] FIQ = Fast Interrupt Request
[14] IRQ = Standard Interrupt Request

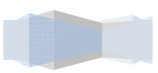
Wie aus dem Aufbau hervorgeht, kann der Advanced Interrupt Controller Interrupts verarbeiten, die sowohl von externen Quellen, als auch von internen Quellen ausgelöst werden. Somit sind auch sämtliche integrierten Peripheriegeräte mit dem Controller verbunden. Bei der Initialisierung des AIC muss man keine Einstellungen im Power Management Controller vornehmen, da der Interrupt-Controller dauerhaft aktiviert ist.

Bevor man einen Interrupt für eine bestimmte Quelle, sei es ein Peripheriegerät oder der FIQ, aktiviert, muss man für diese Interrupt-Quelle im AIC Source Mode Register und im AIC Source Vector Register einige Einstellung vornehmen. In das jeweilige AIC Source Mode Register werden Angaben zur Priorität des Interrupts gemacht. Außerdem kann man bei externen Interrupt-Quellen die Flanken auswählen, bei denen der Interrupt ausgelöst werden soll. In das AIC Source Vector Register wird die Speicheradresse der jeweiligen Interrupt-Handler-Funktion geschrieben. Dabei ist zu beachten, dass es sich bei der Speicheradresse um eine 32-Bit Zahl handelt. Da 32 verschiedene Interrupt-Quellen unterschieden werden können gibt es von den beiden eben genannten Registern jeweils 32 Stück.

4.3.2. Umsetzung der Implementierung

Um die Nutzung von Interrupts in eLua gewährleisten zu können, sind einige Funktionen nötig. In diesem Kapitel wird Schritt für Schritt an diese Funktionen herangeführt. Zusätzlich wird beschrieben, wie die Funktionen in eLua anzuwenden sind.

Anmerkung: Die hardwareunabhängigen eLua-Funktionen sind in der Datei „elua-0.5\src\modules\aic.c“ gespeichert. Die hardwarespezifischen Funktionen befinden sich in den Dateien „platform.c“ und „aic.c“ im Verzeichnis „elua-0.5\src\platform\at91sam7x“.



AIC Initialisierung

Die Initialisierung des Advanced Interrupt Controllers wird durch den Aufruf aic.setup() gestartet. Hinter diesem Aufruf verbirgt sich die Routine aus Listing 7. Die Funktion überprüft zuerst, ob der globale *lua_state* L1* bereits initialisiert wurde. Ist dies nicht der Fall werden die Einträge des globalen Zeigerfeldes *handler[]* mit *NULL* vorbelegt. Dadurch wird gewähreistet, dass beim späteren Aufruf eines Interrupts keine uninitialisierten Interrupts abgearbeitet werden können. Diese Vorbelegung wird allerdings immer nur beim ersten Aufruf von aic.setup() durchgeführt, da in Zeile 10 der globale Zustand mit dem aktuellen Lua-Zustand verknüpft wird. Ab Zeile 12 werden nun die Übergabeparameter der Funktion abgearbeitet. Bei der ersten übergebenen Variablen handelt es sich um den Wert *id,* mit deren Hilfe über das Makro MOD_CHECK_ID() abgefragt wird, ob der Advanced Interrupt Controller vorhanden ist. Somit ist dabei nur ein Wert von 0 möglich, da der AT91SAM7X nur über einen solchen Controller verfügt.

```
01 static int aic_setup( lua_State* L )
02 {
03   if (L1 == NULL){
04     int i;
05     for(i=0; i < PLATFORM_AIC_NUMBERS; i++){
06       handler[i] = NULL;
07     }
08   }
09   unsigned id, inumber, mode;
10   L1 = L;
11
12   id = luaL_checkinteger( L, 1 );
13   MOD_CHECK_ID( aic, id );
14   inumber = luaL_checkinteger(L, 2);
15   mode = luaL_checkinteger(L, 3);
16   handler[inumber] = (char*) luaL_checkstring( L, 4);
17
18   platform_aic_setup(inumber, mode, int_wrapper);
19   lua_pushstring(L, handler[inumber]);
20
21   return 1;
22 }
```

Listing 7 - AIC Initialisierungsfunktion

Mit dem nächsten Parameter wird der Wert der Schnittstelle übergeben, für die der jeweilige Interrupt initialisiert werden soll. Um hierbei Zahlen mit den jeweiligen Schnittstellen in Verbindung bringen zu können, hilft die Tabelle 9 – Übersicht Interrupt-Belegung. Anschließend wird *mode* übergeben. Mit Hilfe dieser Variablen wird die Priorität des

Interrupts eingestellt. Mögliche Werte sind dabei 0 bis 8. Bei dem letzten Parameter der Funktion handelt es sich um den Funktionsnamen der Lua-Funktion, die aufgerufen werden soll, falls der jeweilige Interrupt auftritt. Sind die Übergabewerte vom Stack ausgelesen, so startet die Funktion in Zeile 18 eine plattformspezifische Routine, die mit Hilfe einer Atmel-Methode die entsprechenden Werte in die Register einträgt. Bei diesem Aufruf wird nun jedoch in das Interrupt Vector Register nicht die Adresse der aufzurufenden Lua-Funktion eingetragen, sondern die der Routine int_wrapper().

```
01 static void int_wrapper(void)
02 {
03   int number;
04   number = platform_get_isource_nbr();
05   if (handler[number] != NULL){
06     lua_getglobal(L1, handler[number]);
07     lua_call(L1,0,0);
08     lua_pop(L1,1);
09   }
10 }
```

Listing 8 - Die Funktion int_wrapper()

Listing 8 zeigt die eben genannte „Verpackungsfunktion“. Diese Funktion wird bei jedem auftretenden Interrupt-Signal, das durch aic.setup() initialisiert wurde, aufgerufen. Die Routine überprüft in Zeile 04, welche Schnittstelle einen Interrupt ausgelöst hat. Mit dem Ausdruck in Zeile 06 wird der dieser Schnittstelle zugeordnete Name einer Lua-Funktion auf den Stack gelegt. Der folgende Aufruf führt die Lua-Methode mit dem Namen aus, der als oberster Eintrag auf dem Stack liegt. Somit wird der entsprechende Interrupt-Handler der Schnittstelle gestartet. Mit lua_pop() wird der Funktionsname abschließend vom Stack gelöscht. Somit verwaltet die Funktion int_wrapper() die Aufrufe der Interrupts. Abschließend muss man noch auf den Rückgabewert der AIC Initialisierungsfunktion eingehen. Hierbei handelt es sich um einen Kontrollstring, der den Namen der Lua-Interrupt-Handler-Routine enthält.

Interrupt Aktivierung und Deaktivierung

Um nun Interrupts für die jeweilige Schnittstelle nutzen zu können, muss man diese jedoch zusätzlich aktivieren bzw. deaktivieren. Dazu stehen die Funktionen aic.enable()

und aic.disable() zur Verfügung. Beide bekommen als Parameter den Wert der Schnittstelle, für den die Interrupt-Unterstützung aktiviert oder deaktiviert werden soll.
Vor der Aktivierung muss jedoch die Initialisierung vorgenommen werden!

Anmerkung: Bei jeder Schnittstelle sollten Funktionen zur Definition der Art des Interrupts geschrieben werden, wie z.B. adc.enint().

Abschließend stellt eine Tabelle die Funktionen des AIC-Moduls dar.

ret = aic.setup(id, number, mode, handler) – nimmt grundlegende Einstellungen vor	
Argumente	• id – AIC Controller ID • number – Nummer der Schnittstelle für den der Interrupt initialisiert werden soll • mode – gewünschte Priorität • handler – Name des Lua-Interrupt-Handlers
Rückgabewerte	• ret – eingestellter Lua-Interrupt-Handler
aic.enable(number) – aktiviert die Interrupt-Unterstützung der jeweiligen Schnittstelle	
Argumente	• number – Nummer der Schnittstelle
Rückgabewerte	• -
aic.disable(number) – deaktiviert die Interrupt-Unterstützung der jeweiligen Schnittstelle	
Argumente	• number – Nummer der Schnittstelle
Rückgabewert	• -

Tabelle 8 - Funktionsübersicht AIC-Modul

AT91SAM7X256	Zuordnung	AT91SAM7X256	Zuordnung
FIQ	0	PWM	10
SYS	1	UDP	11
PIOA	2	TC0	12
PIOB	3	TC1	13
SPI0	4	TC2	14
SPI1	5	CAN	15
USART0	6	Ethernet Mac	16
USART1	7	ADC	17
SSC	8	IRQ0	30
TWI	9	IRQ1	31

Tabelle 9 - Übersicht Interrupt-Belegung

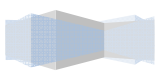
4.3.3. Problem bei der Nutzung von Interrupts

Bei der Nutzung von Interrupts mit eLua stößt man auf ein großes Problem. Offensichtlich handelt es sich hierbei um die Unterbrechung von Lua. Die Funktionen, die im vorherigen Kapitel beschrieben wurden, arbeiten im Bezug auf die Hardware einwandfrei. Definiert man einen Schnittstellen-Interrupt mit aic.setup, wie in Listing 9, für die ADC-Schnittstelle, aktiviert diesen und stellt mit Hilfe der Funktion adc.enint() einen „End of Coversion“-Interrupt ein, so wird dieser wie gewollt ausgeführt. Das Problem besteht nun darin, dass sobald der Interrupt ausgelöst wird und somit die Funktion test() aufgerufen wird, die Ausführung des aktuellen Lua-Zustandes unterbrochen wird. Die Funktion test() wird dann zwar noch ohne Probleme ausgeführt, kann jedoch am Ende nicht mehr mit der Fortsetzung des ursprünglichen Lua-Codes fortfahren.

```
01 function test()
02   print("interrupt")
03   print(adc.getData(6))
04   collectgarbage("collect")
05 end
06
07 adc.setup(0,5000000,10,8,1200)
08 adc.enable(6)
09 print(“Interrupt handler: ” .. aic.setup(0,17,0,"test"))
10 aic.enable(17)
11 adc.enint(6)
12 print(adc.startconv())
13 for i = 0,100 do
14   print(tostring(i))
15   print(adc.startconv())
16 end
```

Listing 9 - Beispiel AIC-Modul

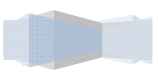
Daraus entsteht die Idee, mit der Lua-Debug-Funktion lua_sethook() zu arbeiten. Diese Funktion kann die Ausführung eines Lua-Zustandes unterbrechen. Die anfängliche Idee besteht darin, mit Hilfe von lua_sethook() regelmäßig einen Hook aufzurufen, der überprüft, ob ein Interrupt ausgelöst wurde und diesen daraufhin ausführt. Diese Lösung scheitert jedoch daran, dass sobald ein Interrupt auftritt auch die Hooks unterbrochen werden und somit wieder das zuvor beschriebene Symptom auftritt.
Da diese Variante auch nicht funktioniert, wird versucht mit Coroutinen zu arbeiten. Die Idee ist, jedes Lua-Programm von vornherein als Coroutine zu definieren. Somit muss man sein Programm in eine Funktion packen, die dann als Coroutine fungiert. Daraus



resultiert, dass das Hauptprogramm nur noch aus einem Aufruf besteht, der das eigentliche Hauptprogramm als Coroutine definiert und danach ausführt. Tritt nun ein Interrupt auf, so wird mit dem Interrupt-Wrapper die Ausführung der Coroutine gestoppt. Dazu werden in die Funktion int_wrapper() die Aufrufe lua_yield() und lua_resume() eingefügt. Mit ersterem wird die Coroutine unterbrochen und mit dem zweiten wird sie fortgesetzt. Durch diese Variante wird es ermöglicht, dass das Hauptprogramm weiter ausgeführt wird. Bei Programmende wird jedoch auch kein Ausstiegspunkt gefunden. Zudem kann dabei immer nur ein Interrupt ausgeführt werden, da die Coroutine durch die Fortsetzung in der Interrupt-Funktion zum Teil des Interrupt-Handlers wird und somit nur von Interrupts mit höherer Priorität unterbrochen werden kann.

Als möglicher Lösungsansatz bleibt somit nur noch die Erweiterung der Bedingungen für die Auslösung eines Hooks. Derzeit kann man Hooks auslösen, wenn Lua eine Funktion aufruft, aus einer Funktion zurückkehrt, eine neue Codezeile interpretiert wird, oder eine bestimmte Anzahl von Anweisungen durchgeführt wurde. Die Idee besteht nun darin, eine Bedingung hinzuzufügen, die einen Hook startet, wenn ein Interrupt ausgelöst wird. Diese Idee stammt von den beiden Entwicklern, die das eLua-Projekt ins Leben gerufen haben.

Anmerkung: Auf die Ausführung dieser Idee wurde in dieser Arbeit jedoch aus zeitlichen Gründen verzichtet.



4.4.Implementierung des Serial Peripheral Interfaces

Dieses Kapitel befasst sich mit der Implementierung der SPI-Schnittstellenerweiterung des AT91SAM7X256 für eLua. Im Folgenden wird kurz darauf eingegangen, wie die SPI-Schnittstelle des Mikrocontrollers aufgebaut ist und wie das von Motorola entwickelte Bussystem funktioniert. Danach werden die durch eLua bereitgestellten Funktionalitäten und deren Umsetzung beschrieben. Zum Abschluss des Kapitels erläutert ein Anwendungsbeispiel den Umgang mit den SPI-Funktionen von eLua.

4.4.1. Allgemeines zum Serial Peripheral Interface

Wie in der Einführung kurz erwähnt, ist die SPI-Schnittstelle ein von der Firma Motorola entwickeltes Bussystem. Dieses Bussystem stellt einen lockeren Standard für einen synchronen seriellen Datenbus dar. Die mit dem SPI umgesetzten digitalen Schaltungen werden nach dem Master/Slave Prinzip aufgebaut. Das bedeutet, dass ein Teilnehmer Master ist und beliebig viele andere Teilnehmer als Slaves fungieren. Der Master gibt im wahrsten Sinne des Wortes den Takt vor und hat uneingeschränkte Rechte auf die gemeinsamen Ressourcen, wohingegen die einzelnen Slaves auf die Aufforderung des Masters warten müssen, um die jeweilige Ressource nutzen zu können. (6)

Die folgende Abbildung 4 beschreibt den grundlegenden Aufbau einer SPI-Topographie.

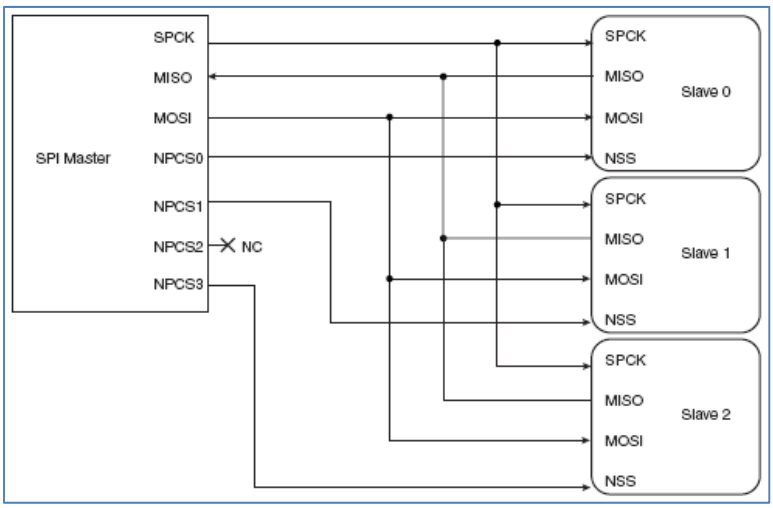


Abbildung 4 - SPI-Topologie AT91SAM7X256 (4)

Die SPI-Schnittstelle verfügt über zwei Steuerleitungen und zwei Datenleitungen, die in einer sternförmigen Busstruktur zum Master hin angeordnet sind. Wie aus obiger Abbildung ersichtlich wird, gehen die beiden Steuerleitungen SPCK und NCPS immer vom SPI-Master zu den SPI-Slaves. In diesem Fall ist die zweite Steuerleitung NCPS in vier verschiedene Leitungen aufgespalten. Durch diese sogenannten Chip-Select- oder Slave-Select-Leitungen kann der Master einen bestimmten Slavebaustein für den Datenaustausch auswählen. Die andere Steuerleitung SPCK steht für Serial Clock und dient dem Master dazu, die Datenrate zu regulieren. Hier ist noch einmal wichtig anzumerken, dass ausschließlich der Master diese Regulierung vornehmen kann. Auf die Einstellungsmöglichkeiten wird im folgenden Kapitel genauer Bezug genommen. Bei den Datenleitungen geht nun eine vom Master als Ausgang auf die Slaves und umgekehrt. Die sogenannte MOSI-Leitung geht vom Master auf die Slaves. MOSI steht hierbei für Master out/Slave in. Die MISO-Leitung ist das analoge Gegenstück dazu und verläuft von den Slaves zum Master. MISO steht daher in diesem Fall für Master in/Slave out.

Die Anzahl der SPI-Slaves, die mit einem SPI-Master verbunden werden können, hängt von der Anzahl der Chip-Select-Leitungen des Masters ab. Beim AT91SAM7X256 wären dies im Prinzip vier Chip-Select-Leitungen. Als Besonderheit bei diesem Mikrocontroller können diese Leitungen jedoch mit einem 4-to-16 Bit Decoder verbunden werden, wodurch bis zu 16 verschiedene Slaves angeschlossen werden können. Die Datenübertragung kann in verschiedenen Modi ablaufen. Diese unterschiedlichen Modi sind von der Taktpolarität CPOL und der Taktphase CPHA abhängig. Da jeder der beiden Werte entweder 0 oder 1 sein kann, ergeben sich somit vier Übertragungsmodi. Diese Übertragungsarten definieren, wann ein gesendeter Wert übernommen wird. Bei der Einstellung von CPOL = 0 ist der inaktive Zustand von SPCK bei einer logischen 0. Hingegen ist bei einem Wert von CPOL = 1 der inaktive Zustand von SPCK bei einer logischen 1. Durch die Taktphase CPHA wird festgelegt, bei welcher Taktflanke der Wert übernommen wird. Ist CPHA = 0, so wird der Wert bei der ersten Taktflanke und bei CPHA = 1 bei der zweiten Taktflanke übernommen. Das eben beschriebene verdeutlichen die folgenden Abbildungen. (6)

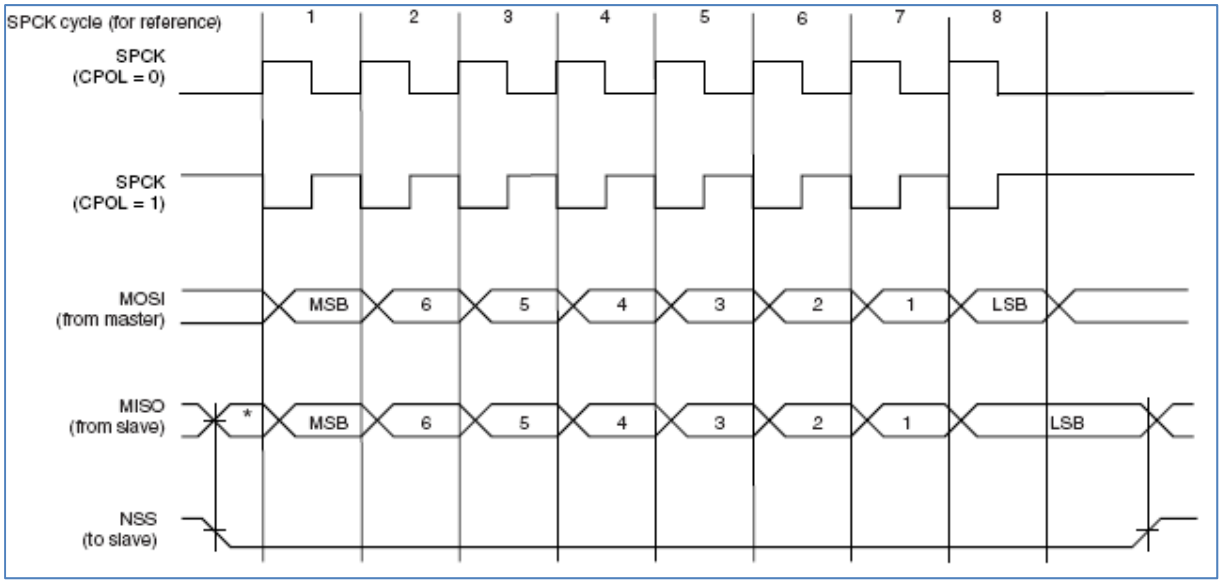


Abbildung 5 - SPI-Datenübertragung bei CPHA = 0 (4)

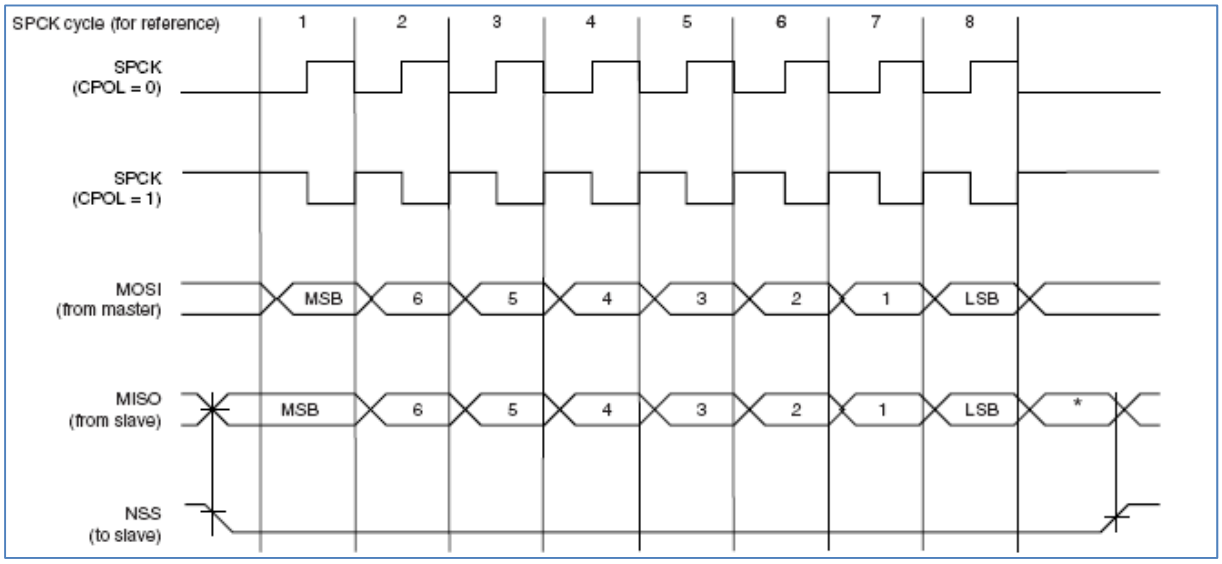


Abbildung 6 - SPI-Datenübertragung bei CPHA = 1 (4)

Bei der Datenübertragung ist es wichtig, dass sowohl Master als auch Slave im selben Modus arbeiten.

Da die beiden SPI-Schnittstellen des AT91SAM7X256 entweder als Master oder als Slave verwendet werden können, soll die folgende Tabelle 10 eine Übersicht über die situationsbedingte Pinbelegung geben.

Pin Name	Pin Description	Type	
		Master	Slave
MISO	Master In Slave Out	Input	Output
MOSI	Master Out Slave In	Output	Input
SPCK	Serial Clock	Output	Input
NPCS1-NPCS3	Peripheral Chip Selects	Output	Unused
NPCS0/NSS	Peripheral Chip Select/Slave Select	Output	Input

Tabelle 10 - Pinbelegung der SPI-Schnittstelle des AT91SAM7X256 (4)

Der Einsatz der SPI-Schnittstelle ist besonders wegen seiner Vielseitigkeit im Bezug auf die Ansteuerung von Peripheriegeräten interessant. Mit Hilfe der SPI-Schnittstelle lassen sich die folgenden Peripheriegeräte ansteuern:

- *Analog/Digital- und Digital/Analog-Wandler*
- *EEPROM- und FLASH-Speicher*
- *Real Time Clocks*
- *Temperatur- und Drucksensoren*
- *andere Controller wie z.B. CAN-, USB-, UART- oder LCD-Controller* (7)

4.4.2. Umsetzung der Implementierung

Um die problemlose Nutzung der SPI-Funktionen in eLua gewährleisten zu können, sind einige Funktionen nötig. In diesem Kapitel wird der Leser Schritt für Schritt an diese Funktionen herangeführt. Zusätzlich wird die Anwendung der Funktionen innerhalb von eLua beschrieben.

Anmerkung: Die hardwareunabhängigen eLua-Funktionen sind in der Datei „elua-0.5\src\modules\spi.c“ gespeichert. Die Hardwarespezifischen Funktionen befinden sich in den Dateien „platform.c“ und „spi.c“ im Verzeichnis „elua-0.5\src\platform\at91sam7x“.

SPI Initialisierung

Die erste Funktion die in diesem Kapitel angesprochen wird, ist die Initialisierungsfunktion des SPI-Moduls. Mit ihrer Hilfe werden die Grundeinstellungen der SPI-Schnittstelle vorgenommen. Die Funktion wird mit dem Befehl spi.setup() aufgerufen. Sie bekommt sechs verschiedene Übergabeparameter und legt nach ihrem korrekten Ablauf einen Rückgabeparameter auf den Stack. Nachfolgend wird Schritt für Schritt auf die einzelnen Parameter der Funktion eingegangen. Der erste Parameter ist wie bei allen eLua-Funktionen der Wert id. Da der AT91SAM7X256 Mikrocontroller über zwei unterschiedliche SPI-Schnittstellen verfügt, kann der Wert *id* entweder 0 oder 1 sein. Somit wird je nach Eingabe SPI0 oder SPI1 initialisiert. Der nächste Parameter bezieht sich auf den jeweiligen Betriebsmodus der Schnittstelle. Mit seiner Hilfe kann man die Schnittstelle als MASTER oder SLAVE deklarieren. Die Entwickler stellen dabei zur Übersichtlichkeit symbolische Konstanten zur Verfügung. Somit kann der Nutzer spi.MASTER übergeben, um die Schnittstelle im MASTER-Modus zu initialisieren. Um das jeweilige Interface als SLAVE zu deklarieren steht analog dazu der symbolische Name spi.SLAVE bereit. Intern sind die beiden Konstanten als Integerwerte – 1 für MASTER und 0 für SLAVE – gespeichert. Somit könnte der Nutzer auch diese beiden Zahlenwerte verwenden. Der nächste Übergabewert spielt nur für den Modus als Master eine Rolle, muss jedoch immer übergeben werden. Mit Hilfe dieses Parameters wird die Taktfrequenz der SPI-Schnittstelle eingestellt. Der Wert der Taktfrequenz kann hierbei höchstens bei der vollen Taktfrequenz der CPU, im Fall des AT91SAM7X256 bei 48 MHz, oder mindestens bei rund 188 kHz liegen. Falsche Eingaben werden durch eine interne Logik abgefangen. Um jedoch beim Programmieren nicht in irgendwelche „Fettnäpfchen“ zu treten, sollte man nur Werte eingeben, die sich nach folgender Formel berechnen lassen:

$$Taktfrequenz = \frac{Taktfrequenz\ des\ Mikrocontrollers}{F}\ , wobei\ F \in [1; 255] \cap \mathbb{Z}$$

Damit auch ohne diese Berechnung keine Fehler unterlaufen, legt die Funktion die eingestellte Taktfrequenz als Rückgabewert auf den Stack.
Den vierten und fünften Übergabeparameter kann man genaugenommen zusammenfassen. Hierbei handelt es sich um CPOL und CPHA. Wie in Kapitel 4.4.1 bereits erwähnt,

handelt es sich hierbei um die Taktpolarität und die Taktphase. Beide Werte können entweder 0 oder 1 sein. Zusammenfassen kann man beide, da sie zusammen den Modus beschreiben, mit dem die Daten übermittelt werden. Die Einstellungen für den gewünschten Modus kann nach Abbildung 5 und Abbildung 6 übernommen werden. Nun kommen wir zum letzten Übergabewert dieser Funktion. Hierbei handelt es sich um die Anzahl der Bits, die pro Übertragung weitergegeben werden können. Hierbei kann der Nutzer Werte zwischen 8 und 16 eingeben. Abschließend sollte man erwähnen, dass die Initialisierungsfunktion das Interface nach der Einstellung der Werte aktiviert.

Slave Select und Slave Unselect

Die Funktionen, die nun besprochen werden sollen, dienen dazu, bei einem Masterinterface die gewünschte Slave Select Leitung zu aktivieren bzw. zu deaktivieren. Hierbei ist anzumerken, dass die Slave Select Leitung low-aktiv ist, also der Slave fühlt sich nur dann angesprochen, wenn sein Slave Select Eingang auf Masse gezogen wird.
Die beiden Funktionen arbeiten auf eine einfache Art und Weise. Da sie prinzipiell die gleiche Aktivität durchführen, greifen auch beide auf ein und dieselbe plattformspezifische Funktion zu. Der einzige Unterschied besteht darin, dass die Slave Select Funktion intern eine 1, die Slave Unselect Funktion eine 0 übergibt. Beide eLua Funktionen bekommen als Übergabewerte die gleichen Parameter. Als ersten Wert bekommen sie die *id* eines SPI-Masters. Sollte die Schnittstelle, die mit der übergebenen *id* übereinstimmt, nicht im Master-Modus aktiviert sein, so wirft die Funktion eine Fehlermeldung aus. Ansonsten verwenden die Funktionen den zweiten Übergabeparameter dafür, die angeforderte Slave Select Leitung zu aktivieren bzw. zu deaktivieren. Hierbei können beim AT91SAM7X256 Werte zwischen 0 und 3 ausgewählt werden. Die plattformspezifische Funktion manipuliert daraufhin das SPI Mode Register im Feld PCS. Wird Slave Unselect aufgerufen, so wird die Leitung NPCS0 per Default auf Masse gezogen.

SPI Datenübertragung

Das nachfolgende Listing 10 stellt die plattformunabhängige Implementierung der der Datenübertragungsfunktion des SPI-Moduls dar.

```
01 static int spi_send( lua_State* L )
02 {
03   spi_data_type value;
04   int total = lua_gettop( L ), i, id;

05   id = luaL_checkinteger( L, 1 );
06   MOD_CHECK_ID( spi, id );
07   for( i = 2; i <= total; i ++ )
08   {
09     value = luaL_checkinteger( L, i );
10     platform_spi_send_recv( id, value );
11   }
12   return 0;
13 }
```

Listing 10 - SPI-Datenübertragungsfunktion

Wie man aus dem Quellcode erkennen kann, bekommt die Funktion spi.send() als ersten Übergabeparameter den Wert *id*. Mit Hilfe der Funktion MOD_CHECK_ID wird, wie in Kapitel 4.1.1 beschrieben überprüft, ob die SPI-Schnittstelle mit der gewählten *id* vorhanden ist. Ist diese Schnittstelle verfügbar, so werden in der for-Schleife total[15] – 1 Werte vom Stack geholt und danach an die plattformspezifische SPI-Sende/Empfangsfunktion[16] übermittelt. Der Datenübertragungsfunktion können also beliebig viele Werte übergeben werden. Diese werden nacheinander über die Datenleitung übertragen. Hardwaretechnisch gesehen erfolgt dies durch das Schreiben in das sogenannte SPI Transmit Data Register.

Wichtig ist anzumerken, dass das Schreiben in das SPI Transmit Data Register erst dann ausgeführt wird, wenn im Statusregister der SPI-Schnittstelle die Information vorhanden ist, dass das Schieberegister leer ist, d.h. dass alte Daten bereits übermittelt wurden.

[15] Wobei total die Anzahl der übergebenen Werte darstellt
[16] Mehr hierzu bei der Empfangs-Funktion der SPI-Schnittstelle

SPI Datenabruf

Die letzte Funktion dieser Schnittstelle ist die SPI-Datenabruf-Funktion, wobei der Name hier nicht ganz korrekt ist. Im Grunde handelt es sich dabei um eine Sende-Funktion, die nach dem Senden auch Daten empfängt. Der Funktionsaufbau ist der im vorherigen Abschnitt beschriebenen SPI-Datenübertragungsroutine sehr ähnlich. Der einzige Unterschied besteht darin, dass nicht nur die übergebenen Daten übermittelt werden, sondern auch der Rückgabewert der in Listing 11 beschriebenen plattformspezifischen Funktion genutzt wird.

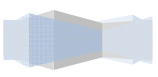
```
01 spi_data_type platform_spi_send_recv( unsigned id,spi_data_type
   data)
02 {
03   AT91S_SPI *pSpi;

04   if (id == 0)pSpi = AT91C_BASE_SPI0;
05   else pSpi=AT91C_BASE_SPI1;

06   SPI_Write(pSpi, data);
07   return SPI_Read(pSpi);
08 }
```

Listing 11 - Plattformspezifische SPI-Datenübertragungsfunktion

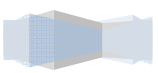
Hierzu wird mit Hilfe der Funktion SPI_Read() das SPI Receive Data Register ausgelesen. Da SPI eine maximale Datengröße von 16 Bit haben kann, werden nur die ersten 16 Bit des Registers ausgelesen. Dies geschieht mit einer bitweisen UND-Verknüpfung des 32-Bit Registers mit der hexadezimalen Zahl 0xFFFF.
Der Rückgabewert der plattformspezifischen Funktion wird durch die eLua-Methode auf den Stack gelegt. Auch für diese Funktion gilt, dass beliebig viele Werte übergeben werden können. Es können allerdings nur so viele Werte empfangen werden, wie auch übertragen werden.



Abschließend soll in diesem Kapitel eine Tabelle die Funktionen des Moduls kompakt und übersichtlich darstellen.

ret = spi.setup(id, mode, clock, cpol, cpha, databits) – nimmt grundlegende Einstellungen vor	
Argumente	• id – SPI Controller ID • mode – gewünschter Funktionsmodus. Parameter sind dabei spi.MASTER oder spi.SLAVE • clock – gewünschte Taktfrequenz des SPI-Controllers in Hz. Dabei sind Werte zwischen 48MHz und 188235Hz möglich • cpol – gewünschte Taktpolarität. Dabei sind Werte von 0 und 1 erlaubt • cpha – gewünschte Taktphase. Dabei sind Werte von 0 und 1 erlaubt • databits – gewünschte Anzahl von Datenbits pro übermitteltem Paket. Dabei sind ganzzahlige Werte von 8 bis 16 erlaubt
Rückgabewerte	• ret – eingestellte SPI-Frequenz
spi.select(id, slave) – aktiviert die jeweilige Chipselect-Leitung	
Argumente	• id – SPI-ID • slave – zu aktivierende Chipselect-Leitung
Rückgabewerte	• -
spi.unselect(id, slave) – deaktiviert die jeweilige Chipselect-Leitung	
Argumente	• id – SPI-ID • slave – zu deaktivierende Chipselect-Leitung
Rückgabewert	• -
spi.send(id, in1, in2,..., inx) – sendet beliebig viele Daten	
Argumente	• id – SPI-ID • in1 – erstes Datenpaket • in2 – zweites Datenpaket • inx – letztes Datenpaket
Rückgabewerte	• -
ret1, ret2, ..., retx = spi.send_recv(id, in1, in2,..., inx) – sendet und empfängt beliebig viele Daten, wobei die Anzahl der empfangenen gleich der Anzahl der gesendeten Daten ist	
Argumente	• id – SPI-ID • in1 – erstes Datenpaket • in2 – zweites Datenpaket • inx – letztes Datenpaket
Rückgabewerte	• ret1 – Wert des ersten empfangenen Datenpakets • ret2 - Wert des zweiten empfangenen Datenpakets • retx - Wert des x-ten empfangenen Datenpakets

Tabelle 11 - Funktionsübersicht SPI-Modul

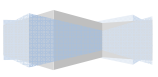


4.4.3. Anwendungsbeispiel SPI-Modul

Um dem Leser die Nutzung des SPI-Moduls noch einmal zu verdeutlichen, legt dieses Kapitel das SPI-Modul anhand eines kurzen Beispiels dar. In diesem Beispiel wird die SPI1-Schnittstelle als Master definiert. Danach werden die Zahlen 1 bis 100 an den Slave mit der Chip Select Leitung 0 übermittelt. Nach der Übermittlung von Daten wird die erhaltene Antwort ausgegeben. Listing 12 zeigt die Umsetzung dieses einfachen Beispiels:

```
01 npcs0_1 = pio.PA_21  --Chip Select Leitung Slave 0
02 spck_1  = pio.PA_22  --SPI Taktleitung
03 mosi_1  = pio.PA_23  --Master out/Slave In
04 miso_1  = pio.PA_24  --Master in/Slave out
05
06 pio.output(npcs0_1, miso_1, mosi_1, spck_1)
07 spi.setup(1,spi.MASTER,48000000,1,0,8)
08 spi.select(1,0)
09
10 for i = 1,100 do
11   print(tostring(spi.send_recv(1,i)))
12 end
```

Listing 12 - Übertragung von Daten via SPI1-Schnittstelle



4.5.Implementierung der TWI-Schnittstelle

In diesem Kapitel wird auf die letzte Schnittstelleimplementierung eingegangen, die in dieser Arbeit entwickelt wird. Hierbei handelt es sich um das TWI-Peripheriegerät des AT91SAM7X256. Wie bereits in den vorhergehenden Kapiteln zur Schnittstellenentwicklung, untergliedert sich dieser Abschnitt in einen Teil der allgemein auf die TWI-Schnittstelle des Atmel Mikrocontrollers eingeht und in einen Teil in dem die implementierten Module beschrieben werden. Zum Abschluss des Kapitels wird eine Beispielanwendung beschrieben, die den Umgang mit den TWI-Funktionen von eLua veranschaulicht.

4.5.1. Allgemeines zur TWI-Schnittstelle

In diesem Abschnitt wird die Arbeitsweise der TWI-Schnittstelle, sowie deren Eigenschaften erläutert.

TWI steht für Two Wire Interface und ist ursprünglich von Philips Semiconductors im Jahre 1980 entwickelt worden. Bei Philips heißt die Schnittstelle allerdings nicht Two Wire Interface, sondern I^2C-Bus. Der I^2C-Bus von Philips wird von anderen Firmen aus patentrechtlichen Gründen Two Wire Interface genannt. Der I^2C-Bus wurde von Philips für die Kommunikation zwischen verschiedenen IC's entwickelt[17] und ist aus diesem Grund eigentlich nur für kleinere Distanzen geeignet. Je nach Datenrate können jedoch auch mehrere Meter mit diesem Bussystem überbrückt werden. Das Two Wire Interface arbeitet, wie die SPI-Schnittstelle, nach dem Master/Slave-Prinzip. Der Unterschied ist jedoch, dass beim I^2C-Bus mehrere Master miteinander vernetzt sein können. Ist dies der Fall spricht man von einem Multi Master Bussystem. Damit bei der Verwendung mehrerer Master keine Fehler entstehen wird die Kommunikation in diesem Fall über ein Protokoll geregelt.
Die Übertragung von Daten wird durch den oder die Master gesteuert. Beim Übertragungsbeginn legt der Master die Adresse des Slaves, mit dem er kommunizieren möchte, auf den Bus. Hierbei gibt es zwei unterschiedliche Adressierungsarten. Die erste ist eine

[17] Daher auch der Name I^2C(=IIC), der für Inter IC Bus steht

7-Bit Adresscodierung, bei der 128 Slaves[18] angeschlossen werden können. Die zweite Adressierungsart ist eine 10-Bit Codierung, bei dem somit maximal 1024 Slaves vernetzt werden können. Nach der Adressübermittlung wird die Information übertragen, ob der Master Daten empfangen oder senden möchte. Ist der Lese- bzw. Schreibvorgang abgeschlossen, so übermittelt der Master ein Stop-Signal und gibt somit den Bus wieder frei. (8) (9)

Als maximale Taktrate wurde ursprünglich nur eine Frequenz von 400kHz im Fast Mode gewählt. Durch die Spezifikationen 2.0 und 3.0 wurden diese jedoch erhöht. Die folgende Tabelle gibt eine Übersicht über die verschiedenen Geschwindigkeitsmodi.

Modus	Maximale Taktfrequenz
Standard mode	100 kHz
Fast mode	400 kHz
Fast mode plus (Spezifikation 3.0)	1 MHz
High speed mode (Spezifikation 2.0)	3,4 MHz

Tabelle 12 - Übertragungsmodi I²C-Bus

Die TWI-Schnittstelle des AT91SAM7X256 kann nur im Master-Modus betrieben werden. Als Adressierungsarten stellt sie jedoch beide Varianten, also 7 Bit und 10 Bit, zur Verfügung. Der Controller unterstützt lediglich die Geschwindigkeitsmodi Standard oder Fast. Wie in Abbildung 7 dargestellt ist, ist der Power Management Controller mit der TWI-Schnittstelle verbunden. Somit muss der Two Wire Interface Controller vor seinem Gebrauch im Power Management Controller aktiviert werden. Zudem wird aus dem TWI-Controller eine Leitung auf den Advanced Interrupt Controller geführt, die es ermöglicht bei gewissen Ereignissen, wie zum Beispiel beim Ende einer Übertragung, einen Interrupt auszulösen. Aus der Abbildung 7 wird außerdem ersichtlich, woher das Two Wire Interface seinen Namen erhält. Die gesamte Kommunikation läuft bei dieser Schnittstelle über zwei Leitungen. Im Falle des AT91SAM7X256 sind dies der TWCK- und der TWD-Pin,wobei TWCK die Taktleitung und TWD die Datenleitung des Busses darstellen.

[18] Tatsächlich sind es nur 112 Slaves, da 16 Adressen für Sonderzwecke reserviert sind

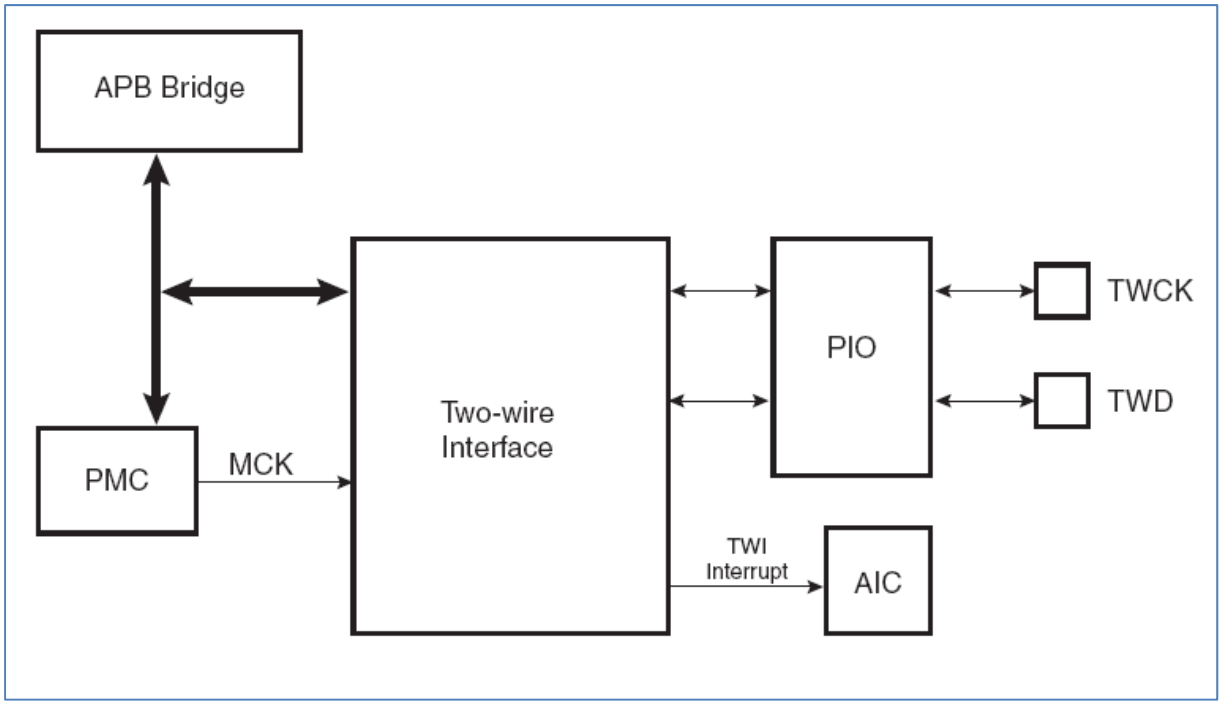


Abbildung 7 - Blockschaltbild des TWI-Interfaces (4)

4.5.2. Umsetzung der Implementierung[19]

Um die problemlose Nutzung des TWI-Moduls in eLua gewährleisten zu können sind einige Funktionen nötig. In diesem Kapitel wird Schritt für Schritt an diese Methoden herangeführt und die Anwendung innerhalb von eLua beschrieben.

Anmerkung: Die hardwareunabhängigen eLua-Funktionen sind in der Datei „elua-0.5\src\modules\twi.c“ gespeichert. Die Hardwarespezifischen Funktionen befinden sich in den Dateien „platform.c“ und „twi.c“ im Verzeichnis „elua-0.5\src\platform\at91sam7x“.

TWI Initialisierung

Mit der TWI Initialisierung twi.setup() werden die für die Nutzung des Two Wire Interfaces benötigten Einstellungen vorgenommen. Der Funktion werden zwei Parameter

[19] Das TWI-Modul funktioniert nur bei Gleitkomma-eLua, da es für die Einstellung einiger Register Gleitkommaberechnungen durchführt.

übergeben. Der erste Parameter ist wie gewohnt die *id* mit dem die Schnittstelle, die benutzt werden soll, ausgewählt wird. Für den AT91SAM7X256 darf dieser Parameter nur den Wert 0 haben, da der Mikrocontroller nur über eine einzige TWI-Schnittstelle verfügt. Die eLua-Funktion überprüft diesen Parameter, wie bei allen Schnittstellen, mit Hilfe der Funktion MOD_CHECK_ID(). Ist der erste Parameter in Ordnung, dass heißt die Schnittstelle mit der ausgewählten *id* existiert, so liest die Funktion den zweiten Übergabeparameter aus. Bei diesem Wert handelt es sich um die einzustellende Taktfrequenz. Falls der Nutzer den Standard- oder Fast-Modus verwenden will, kann er die Taktfrequenz mit Hilfe der beiden symbolischen Namen twi.STANDARDMODE und twi.FASTMODE übergeben. Die beiden Übergabeparameter *id* und *clock* werden an die plattformspezifische Funktion platform_twi_setup(), die in Listing 13 dargestellt ist, weitergegeben.

```
01 u32 platform_twi_setup( unsigned id, u32 clock)
02 {
03   u32 ret;
04   AT91S_PIO *pPio = AT91C_BASE_PIOA;
05   ret = TWI_ConfigureMaster(AT91C_BASE_TWI, clock,
   BOARD_MCK);
06   if (ret == 1)return 1;
07   pPio->PIO_PDR = 0x00000400 | 0x00000800;
08   pPio->PIO_ASR = 0x00000400 | 0x00000800;
09   pPio->PIO_MDER = 0x00000400 | 0x00000800;
10   return ret;
11 }
```

Listing 13 - Plattformspezifische TWI-Initialisierungsfunktion

Die plattformspezifische Funktion stellt in Zeile 05 mit Hilfe einer von Atmel bereitgestellten Funktion die Taktfrequenz des Two Wire Interfaces ein. Die Einstellung geschieht über die Werte ckdiv, cldiv und chdiv im TWI Clock Waveform Generator Register. Dabei ist cldiv der Teiler für die Pause-Zeit und chdiv der Teiler für die Puls-Zeit. Die Zeiten lassen sich nach folgenden Formeln berechnen.

$$T_{aus} = \left(cldiv * 2^{ckdiv} + 3\right) * Periodendauer\ des\ MCK\_Taktes \quad (4)$$

$$T_{an} = \left(chdiv * 2^{ckdiv} + 3\right) * Periodendauer\ des\ MCK\_Taktes \quad (4)$$

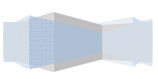
Somit kann man theoretisch ein Puls-Pause-Verhältnis mit Hilfe dieser Register einstellen. Im Fall von eLua wird hier allerdings immer ein Puls-Pause-Verhältnis von 50% eingestellt. Somit ist die Einschaltzeit T_{an} gleich der Ausschaltzeit T_{aus}. Daraus ergibt sich für die TWI-Taktfrequenz mit den beiden oben genannten Formeln der folgende Ausdruck.

$$Taktfrequenz\ TWI = \frac{1}{T_{an} + T_{aus}}$$

Ist die Einstellung der Taktfrequenz korrekt vorgenommen worden, so stellt die Initialisierungsfunktion den PIO-Controller so ein, dass die Pins TWCK und TWD unter der Kontrolle der TWI-Schnittstelle sind. Außerdem wird in Zeile 09 die Open Drain Unterstützung der beiden Pins aktiviert. Abschließend gibt die Funktion die eingestellte Taktfrequenz an die plattformunabhängige Funktion zurück, die diese anschließend auf dem Stack ablegt. Sollte bei der Einstellung der Taktfrequenz ein Fehler unterlaufen sein, so wird die Fehlermeldung „Unable to configure given clock“ ausgegeben.

TWI Datenübertragung

Als nächstes wird in diesem Kapitel die TWI Datenübertragung twi.send() beschrieben. Die Funktion bekommt als Parameter, abgesehen vom Wert *id,* die Adresse des Empfängers und ein Anzahl beliebiger Daten. Listing 14 zeigt die Umsetzung der Funktion twi.send(). Nachdem die Übergabeparameter vom Stack ausgelesen wurden und die übliche Überprüfung der Schnittstelle stattgefunden hat, wird in Zeile 11 die plattformspezifische Methode platform_twi_start_write() aufgerufen. Diese wiederum nutzt eine Atmel-Funktion, um den Startbefehl für die Übertragung von Daten auf den Bus zu schreiben. Hierzu wird im TWI Master Mode Register die Adresse des anzusprechenden Slaves abgelegt. Außerdem wird der Operationsmodus Master Transmitter Modus aktiviert. Die Atmel-Funktion schickt nach den vorgenommenen Einstellungen bereits das erste Datenbyte. Verläuft die Übertragung ohne Fehler, wird die Übermittlung der restlichen Daten mit Hilfe der for-Schleife in Zeile 13 abgearbeitet. Sollte im Laufe der Übertragung ein Fehler auftreten, so gibt die Lua Funktion eine entsprechende Fehlermeldung aus(Zeile 12 + 17).

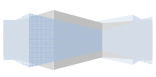


```
01 static int twi_send( lua_State* L )
02 {
03   unsigned char value;
04   int total = lua_gettop( L ), i, id;
05   unsigned char address;
06
07   id = luaL_checkinteger( L, 1 );
08   MOD_CHECK_ID( twi, id );
09   address = luaL_checkinteger( L, 2 );
10   value = luaL_checkinteger( L, 3 );
11   if(platform_twi_start_write(id, address,value) == -1)
12     return luaL_error( L, "TWI timeout" );
13   for( i = 4; i <= total; i ++ )
14   {
15     value = luaL_checkinteger( L, i );
16     if(platform_twi_send( id, value ) == -1)
17       return luaL_error( L, "TWI timeout" );
18   }
19   return 0;
20 }
```

Listing 14 - TWI-Datenübertragungsfunktion

TWI Datenabruf

Schlussendlich wird die Funktion zum Datenabruf betrachtet. Prinzipiell arbeitet twi.recv() wie die Sendefunktion. Sie bekommt allerdings statt den Daten, die gesendet werden sollen, die Anzahl der Daten, die empfangen werden sollen. Somit hat die Funktion drei Übergabeparameter: *id*, *address* und *number*. Der TWI-Master sendet nun wieder einen Startbefehl an den Slave mit der angegebenen Adresse, jedoch mit der Information, dass er nun Daten empfangen möchte. Daraufhin sendet der Slave solange Daten bis der Master wieder einen Stopbefehl sendet. Dieser Befehl wird allerdings erst gesendet, wenn der Master die Anzahl von *number* an Bytes erhalten hat. Im Falle eines korrekten Ablaufs der Routine werden die Daten auf dem Stack abgelegt. Beim Auftreten eines Fehlers während der Übertragung, d. h. wenn die Übertragung zu lange dauert, wird wie auch bei der Sendefunktion die Fehlermeldung „TWI timeout“ ausgegeben.



Die nachfolgende Tabelle gibt einen abschließenden Überblick über die Funktionen des TWI-Moduls.

ret = twi.setup(id, clock) – nimmt grundlegende Einstellungen vor	
Argumente	• id – TWI Controller ID • clock – gewünschte Taktfrequenz des TWI-Controllers in Hz. Dabei ist die Übergabe von twi.STANDARDMODE(= 100kHz) und twi.FASTMODE(= 400 kHz) möglich. Bei anderen Einstellungen sollte man im Datenblatt nachlesen, ob diese möglich sind!
Rückgabewerte	• ret – eingestellte TWI-Frequenz
twi.send(id, address, in1, in2,..., inx) – sendet beliebig viele Daten	
Argumente	• id – TWI-Controller ID • address – 7 Bit Adresse des Empfängers • in1 – erstes Datenpaket • in2 – zweites Datenpaket • inx – letztes Datenpaket
Rückgabewerte	• -
ret1, ret2, ..., retx = twi.recv(id, address, number) –empfängt number-viele Daten	
Argumente	• id – TWI-Controller ID • address – Adresse des Senders • number – Anzahl der zu empfangenen Datenpakete
Rückgabewerte	• ret1 – Wert des ersten empfangenen Datenpakets • ret2 - Wert des zweiten empfangenen Datenpakets • retx - Wert des x-ten empfangenen Datenpakets

Tabelle 13 - Funktionsübersicht TWI-Modul

4.5.3. Anwendungsbeispiel TWI-Modul

Dieses abschließende Anwendungsbeispiel verdeutlicht noch einmal die Nutzung des TWI-Moduls. Dabei wird das Two Wire Interface im Fast Mode, also mit eines Taktfrequenz von 400 kHz aktiviert. Danach sollen die Zahlen 1 bis 4 an den Slave mit der Adresse 0x70 übermittelt werden. Sobald die Datenübertragung korrekt abgelaufen ist, fordert der Master vom gleichen Slave drei Werte an. Diese Werte werden dann auf dem Terminal ausgegeben.

```
01 pio.output(pio.PA_10, pio.PA_11)
02 print(tostring(twi.setup(0,twi.FASTMODE)))
03
04 twi.send(0,0x70, 1,2,3,4)
05
06 a,b,c = twi.recv(0, 0x70, 3)
07 print(tostring(a))
08 print(tostring(b))
09 print(tostring(c))
```

Listing 15 - Anwendungsbeispiel TWI-Modul

4.6.Ressourcenverbrauch und Geschwindigkeit

Dieser Abschnitt unterteilt sich in zwei verschiedene Bereiche. Der erste Bereich bringt dem Leser den Ressourcenverbrauch von eLua nahe. Das heißt es wird einen Überblick darüber gegeben, wie viel Speicher eLua den Stack bereitstellt. Zudem wird beschrieben, wie viel Speicherplatz das bss- und data-Segment verbraucht. Daraus resultierend ergibt sich, die vom Interpreter selbst benötigte Speicherkapazität. Dieser Ressourcenverbrauch ist davon abhängig, welches eLua verwendet wird. Wie in Kapitel 2.4 angesprochen gibt es zwei verschiedene eLua-Varianten, das Ganzzahl- und Gleitkomma-eLua. Die folgende Tabelle zeigt den jeweiligen Speicherverbrauch der beiden Varianten an.

Ressourcenart	Speicherbedarf Ganzzahl-eLua	Speicherbedarf Gleitkomma-eLua
bss-Segment	1284 Byte	972 Byte
data-Segment	2236 Byte	2192 Byte
stack	288 Byte	288 Byte
Gesamter Verbrauch	177728 Byte	134608 Byte

Tabelle 14 - Ressourcenverteilung von eLua

Der zweite Teil dieses Kapitels befasst sich mit der Geschwindigkeit mit der eLua ein externes Peripheriegerät über das PIO-Modul ansteuern kann. Dies ist ein sehr wichtiger Aspekt, falls man zum Beispiel einen externen Motortreiber mit Hilfe von eLua steuern möchte. Hierzu werden mit einem Oszilloskop mehrere Messungen durchgeführt. Es werden nacheinander die Frequenzen eines Rechtecksignals gemessen. Dieses Rechtecksignal wird durch das An- und Ausschalten eines PIO-Ausgangpins erzeugt.

Anmerkung: Die Messung wird sowohl für Integer- als auch für Gleitkomma-eLua durchgeführt.

Bei der ersten Messung wird immer nur ein Pin an- bzw. abgeschaltet. Tabelle 15 zeigt die aus dieser Messung resultierenden Ergebnisse.

eLua-Variante	Frequenz	Resultierende Ansteuerungszeit
Gleitkomma	4,303 kHz	232,4 µs
Ganzzahl	4,579 kHz	218,4 µs

Tabelle 15 - Geschwindigkeitsmessung beim Pulsen an einem Pin

Man kann zwar einen Unterschied zwischen den beiden Varianten erkennen, jedoch ist dieser mit 276 Hz bzw. 14 µs relativ klein. Für Anwendungen die schnelle Reaktionen erfordern, sollte man dennoch abwägen, ob die Verwendung von Gleitkommazahlen zwingend erforderlich ist.

Ein interessanter Aspekt für die Anwendung von eLua ergibt sich aus der nächsten Messung die durchgeführt wurde. Das Oszilloskop zeichnet wieder dasselbe Rechtecksignal wie bei der vorhergehenden Messung auf. Jedoch werden bei dieser Messung mehrere Pins an- bzw. ausgeschalten. Dieses Pulsen von mehreren Pins wird einmal durch drei getrennte Funktionsaufrufe[20] und einmal durch die Übergabe von drei Pins an dieselbe Funktion[21] bewerkstelligt. In Tabelle 16 sind die Ergebnisse dieser Messung aufgeführt.

Anzahl der Pins	Frequenz bei Variante a)	Frequenz bei Variante b)
2 Pins	2,155/2,299 kHz[22]	2,525/2,708 kHz[22]
3 Pins	1,493/1,534 kHz[22]	1,792/1,923 kHz[22]

Tabelle 16 - Geschwindigkeitsmessung beim Pulsen an mehreren Pins

Durch diese Messung wird gezeigt, dass eLua durch seinen Funktionsaufbau, d.h. durch die Möglichkeit mehrere Pins „gleichzeitig“ bearbeiten zu können, einen Geschwindigkeitsvorteil von bis zu 25% erreicht. Daher sollte man bei der Programmierung darauf achten möglichst viele Operationen desselben Typs in einer Funktion zusammenzufassen.

[20] Variante a)
[21] Variante b)
[22] Gleitkomma-eLua/Ganzzahl-eLua

5. Anwendungsbeispiel mit eLua auf einem AT91SAM7X256

Nachdem die allgemeine Vorgehensweise der Implementierung von Schnittstellen und einige Beispiele der Schnittstellenimplementierung beschrieben wurden, wird in diesem Kapitel abschließend ein Anwendungsbeispiel mit eLua erstellt. Diese Anwendung verwendet zu großen Teilen die SPI- und ADC-Schnittstelle des AT91SAM7X256. Im Folgenden werden die Anforderungen an die Anwendung beschrieben. Anschließend wird auf die benutzte Hardware eingegangen und die Umsetzung der Implementierung dokumentiert.

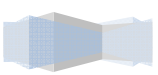
5.1. Anforderung an das Anwendungsbeispiel

Als Beispiel soll die Ansteuerung des LCD-Displays, welches auf dem Olimex Entwicklungsboard integriert ist, gewährleistet werden. Hierbei sollen mehrere Varianten durch den Programmcode abgedeckt werden.

Die erste Variante soll eine Kombination aus der Ansteuerung des LCD-Displays und der ADC-Schnittstelle des Boards bzw. des Mikrocontrollers sein. Das LCD-Display soll hierbei Pixel für Pixel farbig hinterlegt werden. Durch die Verstellung eines Potentiometers sollen verschiedene Farben auf dem LCD-Display erscheinen.

Als zweite Variation der Ansteuerung des LCD-Displays soll nun ein Bild vom ROM Dateisystem eLua's geladen werden. Da der Vorgang durch die Größe der Datei und die Interpretation des Quellcodes relativ lange dauern kann, soll dem Nutzer über den Terminalemulator der aktuelle Fortschritt in Prozent angezeigt werden. Außerdem soll jederzeit die Möglichkeit bestehen, das Laden der Bitmap abzubrechen.

Da, wie oben bereits angemerkt, die Bilddatei relativ groß ist, soll eine Bilddatei direkt von der Festplatte des Rechners, auf dem die Emulationssoftware läuft, auf das Display gespielt werden. Hierzu soll abgesehen von der SPI-Schnittstelle auch die UART-Schnittstelle verwendet werden, um mit Hilfe des XMODEM-Protokolls die Dateien von der Festplatte zu übertragen.



Anmerkung: Die Pixelinformationen liegen in Form einer Textdatei vor, in der einzelne Bytes aufgelistet sind.

5.2.Allgemeines

Dieses allgemeine Kapitel legt dem Leser erst einmal die grundlegenden Eigenschaften des LCD-Displays und dessen Verdrahtung auf dem Olimex Entwicklungsboard SAM7-EX256 dar. Dieser Abschnitt wurde in Anlehnung an die LCD-Beschreibung von James P. Lynch (10) erstellt.

Bei dem in dieser Arbeit verwendeten Display handelt es sich um das 6100 LCD Display von der Firma Nokia. Olimex liefert Boards mit zwei unterschiedlichen Displaycontrollern aus. Zum einen mit dem Philips PCF 8833 und zum anderen mit dem Epson S1D15G00. Der Nutzer kann dies durch einen Aufkleber auf dem Display unterscheiden. Der Controller von Philips ist mit G12, der von Epson mit G8 gekennzeichnet. Für diese Diplomarbeit wird ein Olimexboard mit einem Epson Displaycontroller verwendet. Das Nokia 6100 Display verfügt über eine Auflösung von 132x132 Pixel. Die nachfolgende Abbildung zeigt die Pixelbelegung des Displays im normalen und inversen Modus.

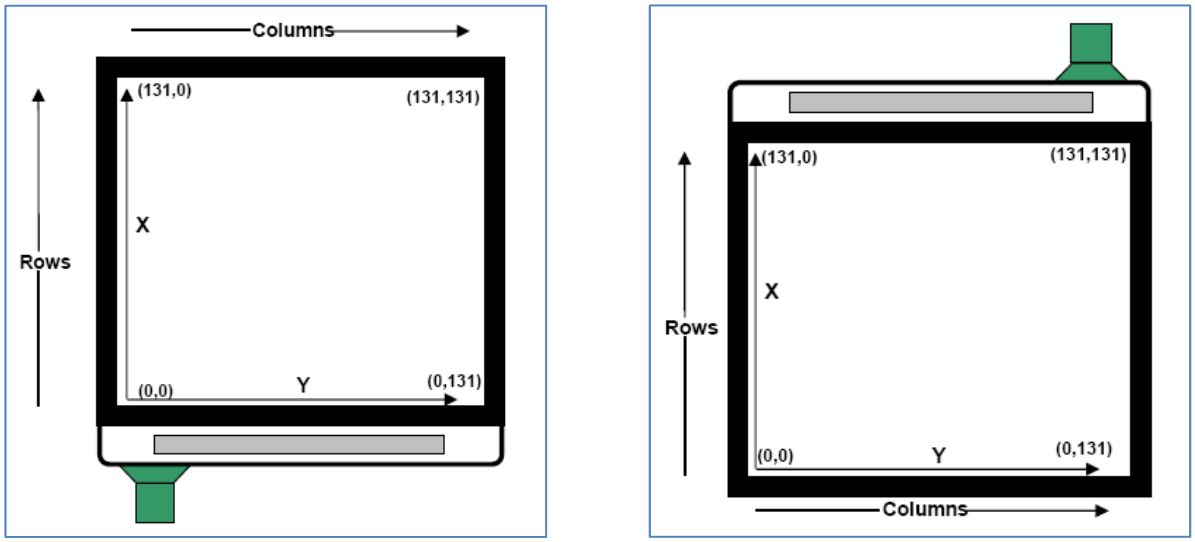


Abbildung 8 - LCD Normale und Inverse Ausrichtung (10)

Die Ansteuerung des LCD Controllers erfolgt über die SPI-Schnittstelle des Mikrocontrollers. Hierbei ist darauf zu achten, dass nur die SPI0-Schnittstelle des AT91SAM7X256 für die Ansteuerung verwendet werden kann, da nur diese, wie Abbildung 9 zeigt, mit dem Display verbunden ist.

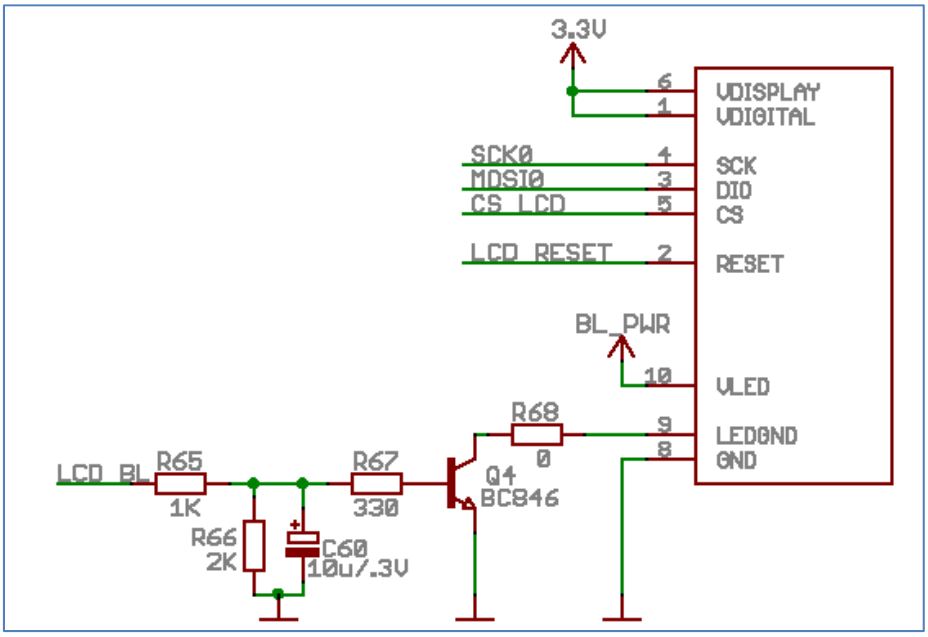


Abbildung 9 - Verdrahtung LCD Display (11)

Aus dieser Abbildung ist ebenfalls ersichtlich, dass das Display mit einer Versorgungsspannung von 3,3 Volt betrieben wird. Zudem kann man die später für die Umsetzung der Anforderungen wichtigen Pins ablesen. Abgesehen von den SPI Steuerleitungen sind dies die Reset-Leitung und der Backlight-Pin des Displaycontrollers, der über einen Transistor angesteuert wird. Tabelle 17 soll die für die Ansteuerung benötigten PIO-Pins darstellen.

Pio Pin	Funktion
PA12	Slave Select Leitung des LCD Displays
PA17	Master Out / Slave In
PA18	SPI Taktleitung
PA2	LCD Reset
PB20	LCD Backlight

Tabelle 17- Benötigte Pins zur Ansteuerung des LCD Displays

Für die Datenübertragung muss die SPI-Schnittstelle auf 9 Bit eingestellt werden, wobei das neunte Bit hierbei als Steuerbit dient. Ist das Steuerbit 0 so erkennt der Display-Controller, dass es sich bei den 8 Bits um ein Steuersignal handelt. Liegt jedoch eine logische 1 an der Position des neunten Bits an, so sind die nachfolgenden 8 Bits Datenbits.

Die Übersicht der Steuersignale kann auf Seite 33 des Epson Datenblattes (12) entnommen werden.

5.3.Umsetzung der Anwendung

In diesem Abschnitt wird dem Leser nun die Implementierung der geforderten Funktionalitäten, des unter Kapitel 5.1 beschriebenen Anwendungsbeispiels, geschildert. Als erstes wird hierbei auf die Initialisierung des LCD Displays mit eLua eingegangen. Danach werden die einzelnen Programmbausteine Schritt für Schritt erläutert.

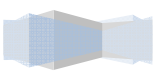
5.3.1. Initialisierung des LCD Displays

Die Initialisierung des LCD Displays ist in zwei verschiedene Funktionen aufgeteilt. Die erste Funktion heißt init_lcd() und konfiguriert die Hardware, die zur Ansteuerung des Displays benötigt wird.

```
10 function init_lcd()
11   pio.output(backlight_lcd,resetpin_lcd, miso, mosi,
   spck)
12   pio.set(backlight_lcd, resetpin_lcd)
13   spi.setup(0,spi.MASTER,125000,1,0,9)
14   spi.select(0,0)
15 end
```

Listing 16 - Hardwareinitialisierung für die LCD Ansteuerung

In Zeile 02 stellt die Funktion die in Tabelle 17 beschriebenen Pins als Ausgänge ein. Als nächstes werden sowohl der Pin für die Hintergrundbeleuchtung, als auch der Reset des LCD Controller auf den Wert 1 gesetzt. Dadurch wird die Hintergrundbeleuchtung des LCD Displays eingeschaltet. Der Reset-Eingang des Controllers ist Low-aktiv und somit ist sichergestellt, dass keine Aktivität durchgeführt wird. In Zeile 04 wird die SPI0-



Schnittstelle des Mikrocontrollers als Master definiert, die Taktfrequenz von 125kHz und eine Datenübertragung von 9 Bit eingestellt. Mit der letzten Anweisung wird der Slave Nummer 0, also das LCD Display ausgewählt. Diese Anweisung wäre eigentlich nicht nötig, da das eLua SPI-Modul als Default Chip Select Leitung NCPS0 verwendet, wird allerdings zur Übersichtlichkeit mit eingefügt.

Die zweite Funktion, die bei der Initialisierung aufgerufen wird, ist die Konfigurationsfunktion für das LCD Display. Mit ihrer Hilfe wird das LCD Display erst einmal zurückgesetzt und danach mit den Grundeinstellungen bespielt.

```
01 function configure_lcd()
02   pio.setpin(0,resetpin_lcd)
03   tmr.delay(0,1000)
04   pio.set(resetpin_lcd)
05   tmr.delay(0,1000)

06   lcd.sendcmd(0,202)
07   lcd.senddata(0,0,32,10)
08   lcd.sendcmd(0,209)
09   tmr.delay(0,10000)
10   lcd.sendcmd(0,148)
11   lcd.sendcmd(0,130)
12   lcd.senddata(0,0)
13   lcd.sendcmd(0,32)
14   lcd.senddata(0,15)
15   lcd.sendcmd(0,167,169,188)
16   lcd.senddata(0,0,3,2)
17   lcd.sendcmd(0,117)
18   lcd.senddata(0,0,131)
19   lcd.sendcmd(0,21)
20   lcd.senddata(0,0,131)
21   tmr.delay(0,1000)
22 end
```

Listing 17 - LCD Konfiguration

Listing 17 zeigt die eben genannte Konfigurationsfunktion. In den Zeilen 02 bis 05 wird das Display über den Low-aktiven Eingang zurückgesetzt. Für die Einstellungen und die Datenübertragung auf das LCD Display wird ein spezielles eLua-Modul programmiert, das im Grunde nur auf die SPI-Sende-Funktionen zugreift. Der Unterschied zum SPI-Modul ist jedoch, dass das neunte Bit – das Steuerbit für den LCD Controller – automatisch, je nach Befehlsaufruf konfiguriert wird. So ist dieses Bit beim Aufruf lcd.sendcmd() gleich dem Wert 0 und der LCD Controller erkennt, dass es sich hierbei um ein Steuer-

signal handelt. Demnach ist das neunte Bit beim Aufruf lcd.senddata() gleich dem Wert 1.

Anmerkung: Die genaue Funktionsbeschreibung des LCD Moduls kann aus Tabelle 24 im Anhang entnommen werden. Auf die Verwendung von symbolischen Konstanten für die Steuerbefehle wurde bewusst verzichtet, da diese zu viel Platz auf dem Speicher verbrauchen würden. Somit werden bei Steuerbefehlen nur die Zahlenwerte übergeben.

Der erste Steuerbefehl zur Konfiguration des LCD Displays in Zeile 06 lautet Display Control. Dem Befehl folgen 3 Übergabeparameter die in Zeile 07 via lcd.sendcmd() übertragen werden. Mit dem ersten Parameter wird der Default-Wert für die Frequenzteilung eingestellt. Mit Hilfe des nächsten Parameters wird dem Controller die Anzahl der Zeilen des Displays mitgeteilt. Der übergebene Wert ergibt sich nach folgender Formel.

$$\text{Übergabewert} = \frac{Anzahl\ der\ Zeilen}{4} - 1 \quad (10)$$

Somit ergibt sich im Falle des Olimex Entwicklungsboards ein Wert von 32. Mit dem letzten Wert werden die „Inversely highlighted lines“ auf den Wert 10 eingestellt. Als nächstes wird in Zeile 08 der interne Oszillator des Displays angeschaltet. Um dafür genug Zeit bereitzustellen wird der nächste Befehlsaufruf durch das eLua Timer-Modul um 10 ms verzögert. Mit der Übergabe von 148 als Steuerbefehl verlässt man den „Sleepmodus“. Mit dem Steuerbefehl in Zeile 11 und der darauf folgenden Datenübertragung wird der Temperaturgradient der Spannung für die Flüssigkristalle auf -0,05 $\%/_{\circ C}$ eingestellt. Mögliche andere Werte wären -0,1, -0,15 oder -0,2 $\%/_{\circ C}$.
Durch lcd.sendcmd(0, 32) in Zeile 13 und den folgenden Übergabeparameter wird die gesamte Spannungsversorgung aktiviert. Mit dem Befehl in Zeile 15 wird der Inverse Display Modus ausgewählt, die Auswahl eines bestimmten Displaybereichs deaktiviert und der Steuerbefehl zur Einstellung der internen Datenverarbeitung des Controllers übergeben. Die in der folgenden Zeile gesendeten Daten spezifizieren die interne Datenverarbeitung. Der erste übergebene Wert stellt das Display so ein, dass die Zeilen des Displays von oben links Zeile für Zeile gefüllt werden. Mit dem nächsten Parameter wird

die RGB Anordnung eingestellt. Abschließend wird die gewünschte Graustufe parametriert.

Durch die beiden letzten Steuerbefehle und deren Übergabeparameter werden sowohl für die Zeilen, als auch für die Spalten die Start- und Endwerte gesetzt, in denen die Pixel mit den Farbinformationen bespielt werden sollen. Da in diesem Fall das gesamte Display genutzt werden soll, geht der Wertebereich bei Zeilen und Spalten von 0 bis 131.

5.3.2. Das Auswahlmenü

Nachdem die Initialisierung beendet ist, startet eLua das eigentliche „Hauptprogramm“. Da das Programm unterschiedliche Funktionalitäten im Bezug auf das LCD Display darstellen soll, ist das Hauptprogramm ein Menü zur Auswahl dieser Funktionen. Hierbei wird von dem Nutzer auf dem Terminalemulator eine Eingabe gefordert. Die folgende Abbildung zeigt diese Aufforderung.

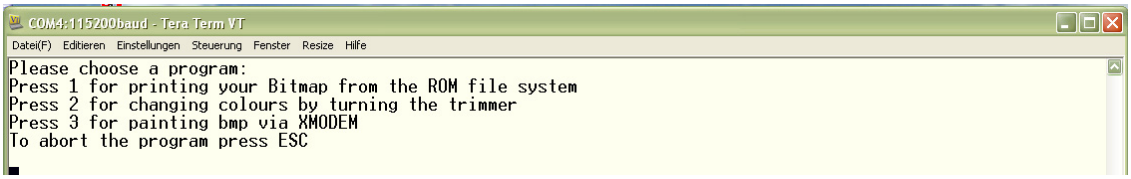


Abbildung 10 - Auswahlmenü LCD Programm

Über das eLua Terminal Modul wird solange auf eine Eingabe des Nutzers gewartet, bis dieser eine korrekte Auswahl getroffen hat. Sollte der Nutzer eine falsche Eingabe tätigen, gibt das Programm eine Fehlermeldung aus und wartet erneut auf eine Eingabe. Durch Drücken der ESCAPE-Taste kann der Benutzer das Programm beenden. Bei einer korrekten Eingabe führt das Programm die jeweilige Funktion aus und startet nach Beendigung der gewählten Funktion erneut das Auswahlmenü.

5.3.3. Ausgabe verschiedener Farben auf dem LCD Display

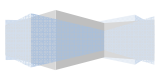
Die erste geforderte Programmfunktionalität besteht aus dem Zusammenspiel des ADC- und LCD-Moduls. Diese Anwendung mag zwar aus der Sicht des Programmieraufwandes recht einfach erscheinen, verdeutlicht jedoch noch einmal sehr schön die Funktionswei-

se der ADC-Schnittstelle unter eLua. Wie im vorherigen Kapitel bereits beschrieben, wird diese Funktion aus dem Auswahlmenü aufgerufen. Das folgende Listing 18 zeigt den Aufbau der Funktion poti().

```
01 function poti()
02   adc.setup(0,5000000,10,8,1200)
03   adc.enable(6)
04   lcd.sendcmd(0,175)
05   lcd.sendcmd(0,92)
06     while true do
07       if term.getch(term.NOWAIT) == 267 then
08         term.clrscr()
09         term.gotoxy(0,0)
10         print "Changing colours aborted!\n"
11         lcd.sendcmd(0,174)
12         return
13       end
14       adc.startconv()
15       value = adc.getData(6)
16       if value <= 204 then
17         paint(1)
18       elseif value > 204 and value <= 408 then
19         paint(2)
20       elseif value > 408 and value <= 612 then
21         paint(3)
22       elseif value > 612 and value <= 816 then
23         paint(4)
24       else
25         paint(5)
26       end
27     end
28 end
```

Listing 18 - Funktion zur Ausgabe verschiedener Farben auf dem LCD

Um den Trimmer auf dem Olimex Entwicklungsboard nutzen zu können, wird zu Beginn die ADC-Schnittstelle initialisiert. Hierbei wird eine Auflösung von 10 Bit eingestellt, um später bei der Abfrage der Daten Werte zwischen 0 und 1023 bekommen zu können. Mit dem Befehl adc.enable(6) in Zeile 03 wird der ADC-Pin aktiviert, da dieser auf dem Board mit dem Potentiometer verbunden ist. Daraufhin wird mit dem LCD Steuerbefehl lcd.sendcmd(0,175) das Display eingeschalten. Ist das Display aktiviert, so folgt ein weiterer Steuerbefehl, mit dessen Hilfe dem Controller der Befehl übermittelt wird, dass ab sofort Pixeldaten auf den RAM-Speicher des LCD-Displays geschrieben werden sollen. Durch die Zeilen 11 bis 17 wird gewährleistet, dass der Nutzer jederzeit über die ESC-Taste in das Hauptmenü zurückkehren kann. Bricht der User das Programm nicht ab, so wird die Analog/Digitalwandlung der am Trimmer anliegenden Spannung durchgeführt.

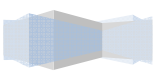


Der daraus resultierende Wert wird daraufhin mit fünf verschiedenen Wertebereichen abgeglichen. Für jeden Wertebereich ist eine Farbe hinterlegt, die dann via SPI an den Display-Controller übermittelt wird. Pro Schleifendurchlauf wird eine Pixelinformation übermittelt, danach beginnt der Vorgang von neuem. Das Programm endet erst durch die Eingabe von ESC durch den Nutzer.

5.3.4. Laden einer Grafik aus dem ROM Dateisystem von eLua

Nachdem im vorherigen Kapitel für den Anfang nur einzelne Pixel an das LCD Display übermittelt wurden, liegt es nun nahe, dass man vollständige Bilder auf dem Display anzeigen will. Wie in der Einführung bereits angemerkt, soll diese Bildübertragung in zwei unterschiedlichen Varianten geschehen. Dieser Abschnitt beschäftigt sich nun mit der Version, in der die Bildinformationen bereits mit der eLua-Quelldatei auf dem Mikrocontroller integriert sind. Um genauer zu sein, die Bildinformationen befinden sich hier auf dem ROM-Dateisystem von eLua. Um dabei die kleine Speicherkapazität des Mikrocontrollers nicht zu sehr zu belasten, werden aus der Datei[23] sämtliche Leerzeichen und alle anderen überflüssigen Zeichen entfernt, so dass pro Zeile nur noch 48 Zeichen benötigt werden.

[23] Die Textdatei ist ursprünglich eine C-Header Datei, die durch das Programm BmpToArray von der Olimex-Webpage http://www.olimex.com/dev/soft/BmpToArray.zip erstellt wurde



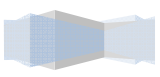
Listing 19 zeigt die Lua-Funktion print_bmp(), mit deren Hilfe die Daten vom ROM-Dateisystem ausgelesen und interpretiert werden.

```
01 function print_bmp()

02   f = io.open("/rom/bmp.txt")
       .
       .
       .
07   lcd.sendcmd(0,92)
08   while content ~= nil do
09     content = f:read("*l")
10     counter = counter + 1
11     tmp = math.floor(counter*100/1057)
12     term.gotoxy(0,12)
13     print("Progress[%]:" .. tostring(tmp))
14     collectgarbage("collect")
15     if term.getch(term.NOWAIT) == 267 then
16       term.clrscr()
17       term.gotoxy(0,0)
18       print "Loading bitmap aborted!\n"
19       lcd.sendcmd(0,174)
20       return
21     end
22     if content ~= nil then
23       i = 1
24       while i <= string.len(content) do
25         tmp = string.sub(content,i,i+1)
26         if  tmp ~= nil then
27           test = "0x" .. tmp
28           test = tonumber(test)
29           if test ~= nil then
30             lcd.senddata(0,test)
31           end
32           test = 0
33         end
34         i = i+2
35       end
36     end
37   end
38   tmr.delay(0,10000)
39   lcd.sendcmd(0,175)
40 end
```

Listing 19 - Funktion zur Darstellung eines Bildes vom ROM Dateisystem

Mit dem ersten Befehl, in Zeile 02, den die Funktion ausführt, wird die auf dem ROM liegende Textdatei geöffnet. Danach wird, wie auch beim Zeichnen von Pixeln im vorherigen Kapitel, der Befehl an den Displaycontroller übermittelt, dass die Datenübertragung beginnt. Jetzt beginnt das eigentliche Auslesen der Datei. Mit der Anweisung `content = f:read("*l")` in Zeile 09 wird eine Zeile der Textdatei in die Variable content gespeichert. Mit dieser Variablen wird dann im Folgenden weitergearbeitet.



Zuvor werden allerdings noch einige Anweisungen durchgeführt, die dem Nutzer zum einen den Fortschritt der Datenübertragung aufzeigen und zum anderen die Möglichkeit bereitstellen, das Programm jederzeit abbrechen zu können.

Um nun mit der Variable *content* weiterarbeiten zu können wird die Lua String Bibliothek verwendet. In einer Schleife wird der String aus der Variablen *content* nun in Substrings zerlegt, die immer zwei Zeichen lang sind. Diese Substrings werden daraufhin mit den Zeichen „0x“ verkettet. Dieser daraus entstandene Substring wird dann in Zeile 28 in eine Zahl umgewandelt. Diese Zahl steht für einen Teil einer Pixelinformation und wird daraufhin via SPI-Schnittstelle an den LCD-Controller übermittelt. Diese Aktionen werden nun solange durchgeführt, bis das Dateiende erreicht ist. Ist das Dateiende erreicht, so wartet die Funktion einen Moment und sendet dann den Steuerbefehl zur Aktivierung des Displays an den LCD-Controller.

5.3.5. Laden einer Grafik via XMODEM vom Hostrechner

Der letzte Abschnitt des Kapitels 5.3 befasst sich mit der zweiten Variante der Darstellung eines Bildes auf dem LCD Displays des Olimex Entwicklungsboards. In diesem Fall sollen die Bildinformationen direkt vom Hostrechner via XMODEM an den Displaycontroller übertragen werden. Diese Version der Datenübertragung bringt den Vorteil mit sich, dass zum einen keinerlei Speicherplatz des Mikrocontrollers für die Bildinformationen verwendet werden muss und zum anderen, dass der Nutzer eine beliebige Datei darstellen kann, ohne eLua jedes Mal neu zu kompilieren. Wie Listing 20 zeigt handelt es sich bei dieser Variante auch um das kürzeste Lua-Programm der drei umgesetzten Anwendungsbeispiele.

```
01 print "Lua is waiting for your bmp"
02 lcd.sendcmd(0,92)
03 lcd.sendbmp()
04 tmr.delay(0,10000)
05 lcd.sendcmd(0,175)
06 term.clrscr()
07 term.gotoxy(0,0)
```

Listing 20 - Aufruf zum Bildempfang via XMODEM

Dies liegt allerdings nur an der Funktion lcd.sendbmp(). Hinter diesem Aufruf verbirgt sich eine etwas größere Funktion des LCD-Moduls. Beim Starten dieser Funktion wird intern ein Aufruf zum Erhalt von XMODEM Daten gestartet. Dieser Aufruf ist im Prinzip gleich mit dem Shell-Befehl `recv`, mit der Ausnahme, dass anstelle von Lua-Code der übertragen und ausgeführt wird, eben die gewünschten Bildinformationen übermittelt werden. Während dieser Übermittlung werden die Bilddaten verarbeitet und durch die SPI-Schnittstelle an den LCD-Controller übergeben. Listing 21 zeigt den Ausschnitt des C-Programms, das dafür zuständig ist.

```
01 for (counter = 0;counter < pktsize;counter = counter +2)
02 {
03   int tmp=0;
04   if (xmbuf[3+counter] != '\x1A' && xmbuf[3+counter+1] != '\x1A')
05   {
06     if (xmbuf[3+counter] != 10 && xmbuf[3+counter+1] != 10){
07     tmp = 16*helper(xmbuf[3+counter]) + helper(xmbuf[3+counter+1]);
08     platform_spi_send_recv_lcd( 0,1,tmp);}
09   }
10 }
```

Listing 21 - C Funktion zur Übergabe der XMODEM Daten an den LCD Controller

Auch hier werden die Daten wieder in einer Iteration an die SPI-Schnittstelle weitergegeben. Der Unterschied ist, dass nicht mehr zeilenweise, sondern paketweise vorgegangen wird. Bei dem Feld `xmbuf[]` handelt es sich um ein solches XMODEM-Paket. Da die Daten beim XMODEM-Protokoll erst ab dem vierten Byte übermittelt werden, werden die ersten drei Byte bei der Verarbeitung vernachlässigt. In Zeile 04 wird überprüft, ob es sich bei den übermittelten Daten nicht um Füllzeichen des Protokolls handelt. Zeilenumbrüche werden durch die Abfrage in Zeile 06 abgefangen. Handelt es sich bei den Daten weder um Füllzeichen, noch um Zeilenumbrüche wird eine Funktion zur Umrechnung der char-Zeichen in int-Zahlen aufgerufen. Danach werden die Daten wie gewohnt über eine plattformspezifische Funktion an den Displaycontroller übermittelt.

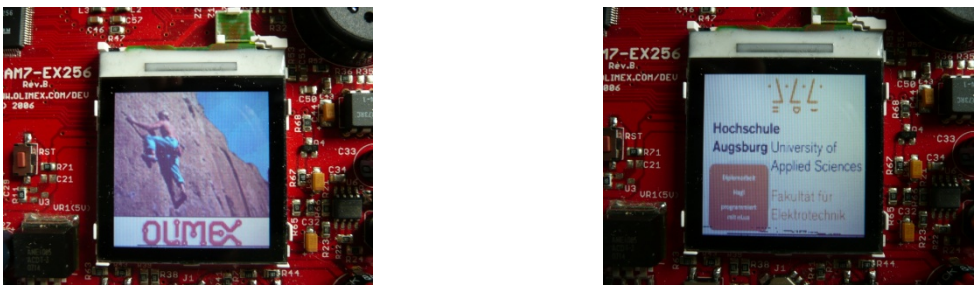


Abbildung 11 - Darstellung von Bildern auf dem LCD-Display

6. Fazit

Embedded Lua ist definitiv eine sehr gute Variante einer Interpretersprache für einen Mikrocontroller. Gerade durch die Leichtigkeit, mit der man Schnittstellen hinzufügen kann, zeichnet sich diese embedded Sprache aus. Dies liegt wohl an dem hervorragendem C-Application Programming Interface, das es ermöglicht die Schnittstelle zwischen Lua und der Hardware mit Ansi-C zu programmieren. Somit konnten in dieser Arbeit auch einige Schnittstellen zu den bestehenden hinzugefügt werden. Die folgende Tabelle soll noch einmal einen kurzen Überblick darüber geben.

AT91SAM7X256	implementiert	getestet	n/a
Ethernet MAC			x
SPI	x	x	
TWI	x	x	
USART	x	x	
SSC			x
TIC	x	x	
PWM	x	x	
UDP			x
CAN			x
ADC	x	x	
AIC	teilweise impl.		
DBGU			x
PIO	x	x	
LCD	x	x	

Tabelle 18 - Abschließender Überblick über die Schnittstellenimplementierung des AT91SAM7X

Es hat sich im Laufe der Arbeit auch gezeigt, dass man, sobald die Schnittstellen implementiert wurden, eine sehr mächtige und leicht verständliche Sprache zur Programmierung von Mikrocontrollern hat. Wenn eLua einmal den Entwicklungsstatus überwunden hat, dann wäre es gut vorstellbar, diese als „Einsteiger-Sprache“ für die Programmierung von Mikrocontrollern zu nutzen. Dafür spricht gerade der Einfache Umgang[24], dass teilweise beliebig viele Parameter übergeben und Rückgabewerte ausgelesen werden können. Der Hauptaspekt für eLua als „Einsteiger-Sprache“ ist jedoch, dass man ohne detaillierte Kenntnisse hardwarenahe Funktionalitäten entwickeln kann, da dies bereits durch

[24] z.B. dass der Entwickler nicht auf Datentypen achten oder Variablen vor Gebrauch initialisieren muss

die dafür vordefinierten Funktionen abgedeckt wird. Sollte es darüber hinaus notwendig sein, den vordefinierten Funktionsumfang zu erweitern, so kann dies jederzeit durch Hinzufügen von geeigneten C-Routinen realisiert werden.

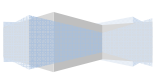
Zudem muss man die einfache Möglichkeit zur Fehlersuche erwähnen. Da für die Ausführung der Applikation keine Kompilierung notwendig ist, können Quellcodes bzw. kleine Programmteile sehr schnell aufgespielt und auf Fehler untersucht werden. Dies kann gerade bei größeren Projekten einen immensen Geschwindigkeitsvorteil mit sich bringen.

Eine weitere mächtige Eigenschaft ist die Nutzung des XMODEM-Protokolls. Es ist durchaus vorstellbar, dass man wie es im LCD-Modul der Fall war, für andere Zwecke als zur Übertragung von Quellcode nutzt. Somit ergibt sich eine Möglichkeit mit dem Hostrechner Daten auszutauschen, was gerade bei der geringen Speicherkapazität von Mikrocontrollern eine erhebliche Rolle spielen könnte.

Als letztes sehr gutes Feature muss man hier mit Sicherheit auch noch einmal das ROM-Dateisystem erwähnen. Mit dessen Hilfe kann man, je nach Notwendigkeit eine kleine statische Datenbank aufbauen, die systemseitig vor überschreiben geschützt ist.

Bei allen positiven Eigenschaften von eLua ist festzustellen, dass diese Sprache noch in den Kinderschuhen steckt. So wird gerade im Mikrocontroller-Umfeld lediglich ein relativ geringer Umfang der möglichen Schnittstellen abgedeckt. . Dies wird sich jedoch von Version zu Version verbessern. Mit der kommenden Version 0.6 werden weitere Schnittstellenmodule zur Verfügung stehen, zudem werden andere Prozessorarchitekturen unterstützt.

Da Interrupts beim Gebrauch von Mikrocontrollern eine wichtige Rolle spielen, stellt die fehlende Interrupt-Unterstützung ein großes Manko dar.
Sollte diese Funktionalität bis zum offiziellen Release zur Verfügung stehen, so dürfte eLua eine deutliche Erleichterung in der Programmierung von Mikrocontrollern bringen.



Anhang

Wichtige Header, Dateien, Funktionen und Strukturen

Header und Dateien

elua-0.5\src\lua\lua.h

In der Headerdatei „lua.h“ sind die grundlegenden in Lua enthaltenen Funktionen definiert. Darunter befinden sich Funktionen zum Erstellen einer neuen Lua- Umgebung, zum Aufrufen von Lua- Funktionen, zum Lesen und Schreiben von globalen Variablen in der Lua- Umgebung und zum Registrieren neuer Funktionen. Die in „lua.h“ definierten Funktionen haben alle das Präfix „lua_“.

elua-0.5\src\lua\lauxlib.h

„lauxlib.h“ definiert die in der Hilfsbibliothek „auxlib.h“ enthaltenen Funktionen. Auxlib hat keinen Zugriff auf Elemente, die sich intern in Lua abspielen. Ihre gesamte Aufgabe wird über die C- API erledigt. Alle Definitionen aus „lauxlib.h“ beginnen mit „luaL_“.

elua-0.5\src\lua\lualib.h

In „lualib.h“ sind Funktionen zum Öffnen von Bibliotheken definiert.

elua-0.5\src\lua\linit.c

In dieser Datei werden sämtliche Lua-Bibliotheken aufgelistet, die bei der Kompilierung des Interpreters integriert werden sollen.

elua-0.5\src\modules\auxmods.h

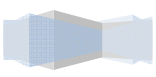
In der Headerdatei „auxmods.h“ werden sämtliche Namen der eLua Behelfsbibliotheken definiert. Außerdem enthält die Datei die Prototypen der Funktionen zur Integration der Bibliotheken in eLua.

elua-0.5\src\platform\at91sam7x\platform_conf.h

Jeder Mikrocontroller hat seine eigene „platform_conf.h“ Datei. In dieser Datei wird definiert, welche Komponenten der Controller besitzt und welche Behelfsbibliotheken in eLua integriert werden sollen.

elua-0.5\SConstruct

Allgemeine Konstruktionsdatei für SCons, mit deren Hilfe die Module ausgewählt werden, die übersetzt werden sollen. Die Datei verweist bei der weiteren Kompilierung auf das plattformspezifische „conf.py“.



elua-0.5\src\platform\at91sam7x\conf.py

Spezielle Konfigurationsdatei für die AT91SAM7X-Mikroprozessorfamilie. Darin werden sind die plattformspezifischen Dateien, die übersetzt werden sollen aufgelistet.

elua-0.5\inc\platform.h

Plattformunabhängiger Header, in dem sämtliche Prototypen für „platform.c“ definiert werden.

elua-0.5\src\platform\at91sam7x\platform.c

Diese Datei bildet die Schnittstelle zwischen den plattformunabhängigen und den hardwarespezifischen Funktionen von eLua.

elua-0.5\src\modules\adc.c

Diese Datei enthält die plattformunabhängigen Funktionen des ADC-Moduls.

elua-0.5\src\platform\at91sam7x\adc.c

Von Atmel bereitgestellte Datei mit Funktionen, die den AD-Wandler der AT91SAM7X-Mikrocontrollerfamilie betreffen.

elua-0.5\src\platform\at91sam7x\adc.h

Zu „elua-0.5\src\platform\at91sam7x\adc.c“ gehörige Headerdatei.(ebenfalls von Atmel bereitgestellt)

elua-0.5\src\modules\aic.c

Diese Datei enthält die plattformunabhängigen Funktionen des AIC-Moduls.

elua-0.5\src\platform\at91sam7x\aic.c

Von Atmel bereitgestellte Datei mit Funktionen, die den Advanced Interrupt Controller der AT91SAM7X-Mikrocontrollerfamilie betreffen.

elua-0.5\src\platform\at91sam7x\aic.h

Zu „elua-0.5\src\platform\at91sam7x\aic.c“ gehörige Headerdatei.(ebenfalls von Atmel bereitgestellt)

elua-0.5\src\modules\spi.c

Diese Datei enthält die plattformunabhängigen Funktionen des SPI-Moduls.

elua-0.5\src\platform\at91sam7x\spi.c

Von Atmel bereitgestellte Datei mit Funktionen, die die SPI-Schnittstelle der AT91SAM7X-Mikrocontrollerfamilie betreffen.

elua-0.5\src\platform\at91sam7x\spi.h

Zu „elua-0.5\src\platform\at91sam7x\spi.c“ gehörige Headerdatei.(ebenfalls von Atmel bereitgestellt)

elua-0.5\src\modules\twi.c

Diese Datei enthält die plattformunabhängigen Funktionen des TWI-Moduls.

elua-0.5\src\platform\at91sam7x\twi.c

Von Atmel bereitgestellte Datei mit Funktionen, die die TWI-Schnittstelle der AT91SAM7X-Mikrocontrollerfamilie betreffen.

elua-0.5\src\platform\at91sam7x\twi.h

Zu „elua-0.5\src\platform\at91sam7x\twi.c“ gehörige Headerdatei.(ebenfalls von Atmel bereitgestellt)

elua-0.5\src\modules\lcd.c

Diese Datei enthält die plattformunabhängigen Funktionen des LCD-Moduls.

Lua_Quellcode\beispiel_adc.lua

Diese Datei enthält das Beispielprogramm für das ADC-Modul.

Lua_Quellcode\beispiel_aic.lua

Diese Datei enthält das Beispielprogramm für das AIC-Modul.

Lua_Quellcode\beispiel_spi.lua

Diese Datei enthält das Beispielprogramm für das SPI-Modul.

Lua_Quellcode\beispiel_twi.lua

Diese Datei enthält das Beispielprogramm für das TWI-Modul.

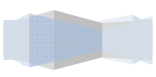
CD\Lua_Quellcode\projektlcd\beispielanwendung.lua

Diese Datei enthält das Anwendungsbeispiel aus Kapitel 5.

Funktionen

lua_State *lua_newstate(lua_Alloc f, void *ud)

Öffnet eine neue Lua-Umgebung. Diese Umgebung enthält noch keine einzige vordefinierte Funktion. Die neue Lua-Umgebung wird in der Zustandsvariablen lua_State gespeichert.



lua_State *lua_newthread(lua_State *L)

Erstellt einen neuen Thread und legt diesen auf den Stack. Die Funktion gibt einen lua_State Zeiger der auf diesen Thread zeigt zurück.

void luaL_openlibs(lua_State *L)

Diese Funktion öffnet alle Standardbibliotheken.

int luaL_loadbuffer(lua_State *L, const char *buff, size_t sz, const char *name)

Mit dieser Funktion wird jede Benutzereingabe eingelesen. Falls dies ohne Fehler geschieht, wird 0 zurückgegeben und der resultierende Chunk wird auf dem Stack abgelegt.

int lua_pcall(lua_State *L, int nargs, int nresults, int errfunc)

Holt Chunk vom Stack und ruft diesen im geschützten Modus auf.

void lua_close(lua_State *L)

Schließt die Lua-Umgebung.

int lua_checkstack(lua_State *L, int extra)

Überprüft, ob auf dem Stack genügend Platz für die gewünschte Operation vorhanden ist.

void lua_getglobal(lua_State *L, const char *name)

Die Funktion legt den Wert des Übergabeparameters name auf den Stack.

void lua_pop(lua_State *L, int n)

Die Funktion löscht die obersten n Elemente vom Stack.

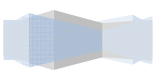
Strukturen

lua_State

In der Lua-Bibliothek werden keinerlei globale Variablen definiert. Der gesamte Zustand wird in der dynamischen Struktur lua_State verwaltet. Jede Lua-Funktion bekommt als Parameter einen Pointer auf diese Struktur. (2)

lua_Hook

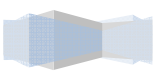
Typ für Funktionen die zum debuggen gedacht sind.



Ausgelagerte Tabellen

ret = bit.bnot(value) – gibt das inverse des übergebenen Wertes zurück	
Argumente	• value – Zahl die negiert werden soll
Rückgabewerte	• ret – inverse Zahl zu value
ret = bit.band(v1, …, vx) – Logische UND-Verknüpfung	
Argumente	• v1 – erster Wert • vx – letzter Wert
Rückgabewerte	• ret – Ergebnis der UND-Verknüpfung der Zahlen von v1 bis vx
ret = bit.bor(v1, …, vx) – Logische ODER-Verknüpfung	
Argumente	• v1 – erster Wert • vx – letzter Wert
Rückgabewerte	• ret – Ergebnis der ODER-Verknüpfung der Zahlen von v1 bis vx
ret = bit.bxor(v1, …, vx) – Logische EXKLUSIV-ODER-Verknüpfung	
Argumente	• v1 – erster Wert • vx – letzter Wert
Rückgabewerte	• ret – Ergebnis der EXKLUSIV-ODER-Verknüpfung der Zahlen von v1 bis vx
ret = bit.lshift(value, pos) – verschiebt value um pos-Stellen nach links	
Argumente	• value – zu bearbeitender Wert • pos – Anzahl der Stellen, um die die Zahl verschoben werden soll
Rückgabewerte	• ret – Ergebnis der Schiebeoperation
ret = bit.rshift(value, pos) – verschiebt value um pos-Stellen nach rechts. Das Vorzeichen wird dabei nicht beachtet	
Argumente	• value – zu bearbeitender Wert • pos – Anzahl der Stellen, um die die Zahl verschoben werden soll
Rückgabewerte	• ret – Ergebnis der Schiebeoperation
ret = bit.arshift(value, pos) – verschiebt value um pos-Stellen nach rechts. Das Vorzeichen wird dabei beachtet	
Argumente	• value – zu bearbeitender Wert • pos – Anzahl der Stellen, um die die Zahl verschoben werden soll
Rückgabewerte	• ret – Ergebnis der Schiebeoperation
ret = bit.bit(bitno) – verschiebt 1 um bitno-Stellen nach links	
Argumente	• bitno – Anzahl der Stellen, um die die 1 verschoben werden soll
Rückgabewerte	• ret – Ergebnis der Schiebeoperation
ret1, …, retx = bit.set(bitno, v1, …, vx) – belegt das Bit an der Position bitno mit 1	
Argumente	• bitno – Position an der das Bit gesetzt werden soll • v1 – erster Wert • vx – letzter Wert
Rückgabewerte	• ret1 – Ergebnis vom Setzen beim ersten Wert • retx – Ergebnis vom Setzen beim letzten Wert
ret1, …, retx = bit.clear(bitno, v1, …, vx) – belegt das Bit an der Position bitno mit 0	
Argumente	• bitno – Position an der das Bit gesetzt werden soll • v1 – erster Wert • vx – letzter Wert
Rückgabewerte	• ret1 – Ergebnis vom Setzen beim ersten Wert • retx – Ergebnis vom Setzen beim letzten Wert

Tabelle 19 - Funktionsübersicht Bit-Modul Teil 1



ret = bit.isset(bitno, value) – überprüft ob das Bit an der Position bitno 1 ist	
Argumente	• bitno – Position die überprüft werden soll • value – Variable
Rückgabewerte	• ret – true bei 1, false bei 0
ret = bit.isclear(bitno, value) – überprüft ob das Bit an der Position bitno 0 ist	
Argumente	• bitno – Position die überprüft werden soll • value – Variable
Rückgabewerte	• ret – true bei 0, false bei 1

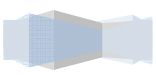
Tabelle 20 - Funktionsübersicht Bit-Modul Teil 2

term.clrscr() – löscht alle Zeichen vom Bildschirm	
Argumente	• -
Rückgabewerte	• -
term.clreol() – löscht alle Zeichen von der aktuellen Cursorposition bis zum Ende der Zeile	
Argumente	• -
Rückgabewerte	• -
term.gotoxy(x,y) – verschiebt den Cursor auf die übergebenen Koordinaten	
Argumente	• x – x-Koordinate • y –y-Koordinate
Rückgabewerte	• -
term.up(delta) – verschiebt den Cursor um delta Zeilen nach oben	
Argumente	• delta – Anzahl der Zeilen
Rückgabewerte	• -
term.down(delta) – verschiebt den Cursor um delta Zeilen nach unten	
Argumente	• delta – Anzahl der Zeilen
Rückgabewerte	• -
term.right(delta) – verschiebt den Cursor um delta Zeichen nach rechts	
Argumente	• delta – Anzahl der Zeichen
Rückgabewerte	• -
term.left(delta) – verschiebt den Cursor um delta Zeichen nach links	
Argumente	• delta – Anzahl der Zeichen
Rückgabewerte	• -
ret = term.lines() – gibt die Anzahl von Zeilen zurück	
Argumente	• -
Rückgabewerte	• ret – Anzahl der Zeilen
ret = term.cols() – gibt die Anzahl von Spalten zurück	
Argumente	• -
Rückgabewerte	• ret – Anzahl der Spalten
term.put(c1, …, cx) – schreibt x übergebene Buchstaben auf die Shell	
Argumente	• c1 – erstes Zeichen • cx – letztes Zeichen
Rückgabewerte	• -
term.putstr(s1, …, sx) – schreibt x übergebene Strings auf die Shell	
Argumente	• c1 – erstes Zeichen • cx – letztes Zeichen
Rückgabewerte	• -
ret = term.cursorx() – gibt die X-Koordinate des Cursors zurück	
Argumente	• -
Rückgabewerte	• ret – X-Koordinate
ret = term.cursory() – gibt die Y-Koordinate des Cursors zurück	
Argumente	• -
Rückgabewerte	• ret – Y-Koordinate
ret = term.getch(wait) – wartet auf eine Zeicheneingabe und gibt diese zurück	
Argumente	• wait – Dauer der Wartezeit. Mögliche Werte sind term.WAIT und term.NOWAIT
Rückgabewerte	• ret – eingegebenes Zeichen

Tabelle 21 - Funktionsübersicht Terminal Modul

pio.input(pin1, ..., pinx) – definiert Pins als Eingänge	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
pio.ouput(pin1, ..., pinx) – definiert Pins als Ausgänge	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
pio.set(pin1, ..., pinx) – legt eine logische 1 an die jeweiligen Pins	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
pio.clear(pin1, ..., pinx) – legt eine logische 0 an die jeweiligen Pins	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
ret1, retx = pio.get(value, pin1, ..., pinx) – gibt den anliegenden Wert der Pins zurück	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• ret1 – Wert des ersten Pins • retx – Wert des x-ten Pins
pio.setpin(value, pin1, ..., pinx) – legt den Wert valua an die jeweiligen Pins	
Argumente	• value • pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
pio.pullup(pin1, ..., pinx) – aktiviert den Pullup-Widerstand der jeweiligen Pins	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• .
pio.pulldown(pin1, ..., pinx) – aktiviert den Pulldown-Widerstand der jeweiligen Pins	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• ret – verstrichene Zeit in µs
pio.nopull(pin1, ..., pinx) – deaktiviert den Pull-Widerstand der jeweiligen Pins	
Argumente	• pin1 – erster Pin • pinx – letzter Pin
Rückgabewerte	• -
ret = pio.pin(pinname) – gibt physikalische Pin-Nummer zurück	
Argumente	• pinname – symbolischer Pin-Name
Rückgabewerte	• ret - physikalische Pin-Nummer
pio.port_input(port1, ..., portx) – definiert Ports als Eingänge	
Argumente	• port1 – erster Port • portx – letzter Port
Rückgabewerte	• -

Tabelle 22 - Funktionsübersicht PIO Modul Teil 1



pio.port_ouput(port1, ..., portx) – definiert Ports als Ausgänge	
Argumente	• port1 – erster Port • portx – letzter Port
Rückgabewerte	• -
pio.setport(value,port1, ..., portx) – legt einen Pegel an den jeweiligen Ports an	
Argumente	• value – gewünschter Wert. Möglich sind dabei 0 oder 1 • port1 – erster Port • portx – letzter Port
Rückgabewerte	• -
ret1, ..., retx = pio.getport(port1, ..., portx) – gibt den anliegenden Wert der Ports zurück	
Argumente	• port1 – erster Port • portx – letzter Port
Rückgabewerte	• ret1 – Wert des ersten Ports • retx – Wert des x-ten Ports
ret = pio.port(portname) – gibt physikalische Port-Nummer zurück	
Argumente	• portname – symbolischer Name des Ports
Rückgabewerte	• ret - physikalische Pin-Nummer

Tabelle 23 - Funktionsübersicht PIO Modul Teil 2

lcd.sendcmd(id, cmd1, ..., cmdx) – sendet Steuerbefehle an den LCD-Controller	
Argumente	• cmd1 – erster Steuerbefehl • cmdx – letzter Steuerbefehl
Rückgabewerte	• -
lcd.senddata(id, data1, ..., datax) – sendet Daten an den LCD-Controller	
Argumente	• data1 – erstes Datenbyte • datax – letztes Datenbyte
Rückgabewerte	• -
lcd.sendbmp() – wartet auf die Übertragung von Bilddaten via XMODEM, um diese an den LCD-Controller zu senden	
Argumente	• -
Rückgabewerte	• -

Tabelle 24 - Funktionsübersicht LCD Modul

Ausgelagerte Abbildungen

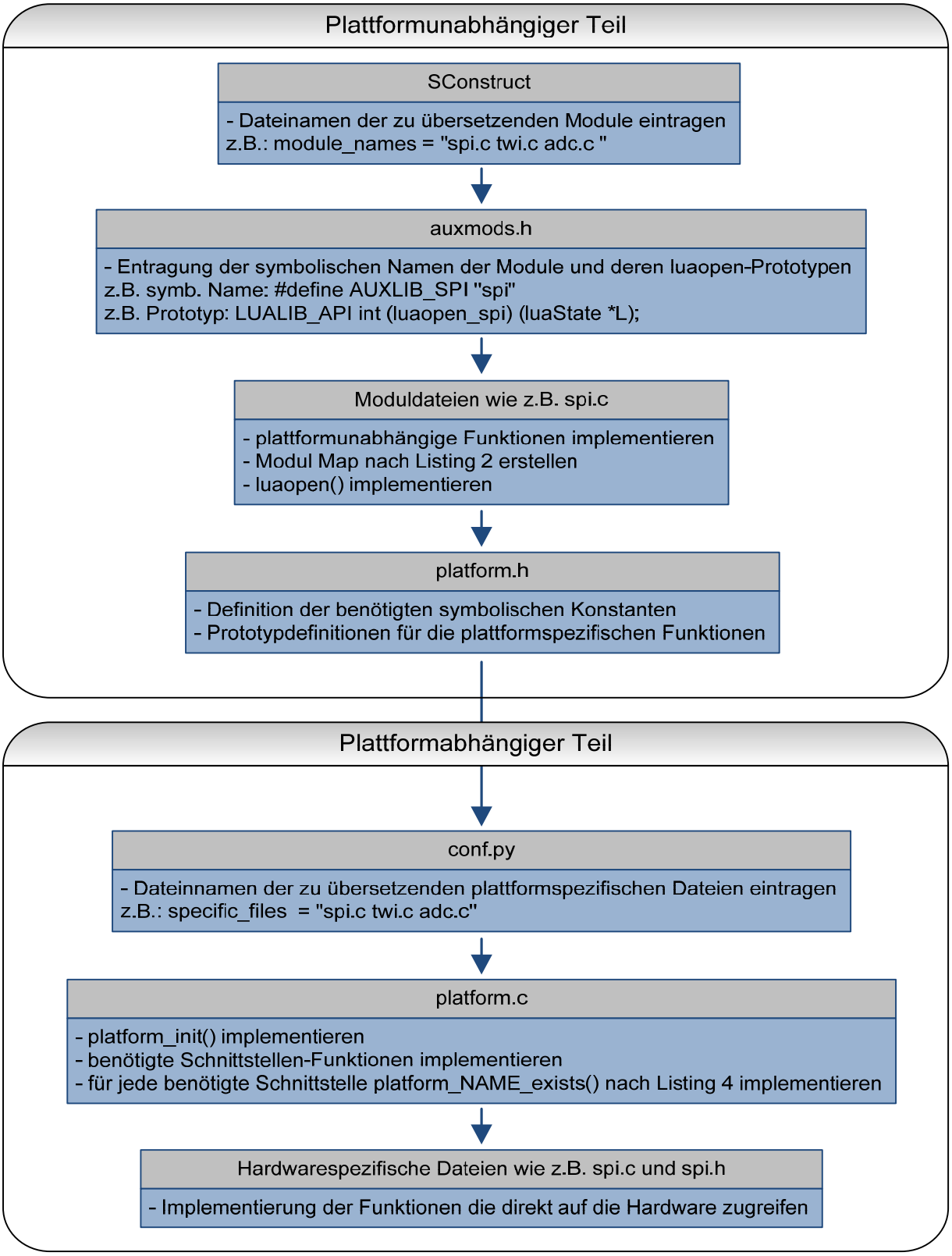


Abbildung 12 - Vorgehensweise zur Schnittstellenimplementierung

Literaturverzeichnis

1. Wikipedia Mikrocontroller. [Online] [Zitat vom: 03. 06 2009.] http://de.wikipedia.org/wiki/Mikrocontroller.

2. **Ierusalimschy, Roberto.** *Programmieren in Lua.* Rio de Janeiro : Open Source Press, 2006. ISBN 3-937514-22-8.

3. **Marinescu, Bogdan und Sutter, Dado.** eluaproject. [Online] http://www.eluaproject.net/.

4. **Atmel Corporation.** Atmel AT91SAM7X512/256/128 Datenblatt. 2009.

5. **Figueiredo, Luis Henrique de.** Luiz Henrique de Figueiredo: Libraries and tools for Lua. [Online] [Zitat vom: 14. 07 2009.] http://www.tecgraf.puc-rio.br/~lhf/ftp/lua/.

6. Wikipedia SPI. [Online] [Zitat vom: 25. 08 2009.] http://de.wikipedia.org/wiki/Serial_Peripheral_Interface.

7. SPI - RN - Wissen. [Online] [Zitat vom: 03. 08 2009.] http://www.rn-wissen.de/index.php/SPI.

8. Wikipedia TWI. [Online] [Zitat vom: 01. 09 2009.] http://de.wikipedia.org/wiki/I%C2%B2C.

9. mikrocontroller.net TWI. [Online] [Zitat vom: 01. 09 2009.] http://www.mikrocontroller.net/articles/I2C.

10. **Lynch, James P.** *Nokia 6100 LCD Display Driver.* [PDF-Datei]

11. Schaltplan SAM7-EX256.

12. *Epson Datasheet S1D15G10D08B000.* [PDF-Datei] Wilsonville : Seiko Epson Coperation, 2002.

13. **Schrüfer, Elmar.** *Elektrische Messtechnik.* München : Carl Hanser Verlag München, 2007. ISBN 978-3-446-40904-0.

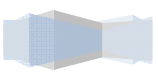
14. Wikipedia - Die freie Enzyklopedie. [Online] http://www.wikipedia.org/.

15. Wikipedia CAN. [Online] [Zitat vom: 16. 09 2009.] http://de.wikipedia.org/wiki/Controller_Area_Network.

16. **Tanenbaum, Andrew S.** *Computernetzwerke.* München : Pearson Studium, 2003. ISBN 978-3-8273-7046-4.

17. Wikipedia Ethernet. [Online] [Zitat vom: 16. 09 2009.] http://de.wikipedia.org/wiki/Ethernet.

18. Wikipedia Flash-Speicher. [Online] [Zitat vom: 16. 04 2009.] http://de.wikipedia.org/wiki/Flash-Speicher.



19. mikrocontroller.net PWM. [Online] [Zitat vom: 29. 04 2009.] http://www.mikrocontroller.net/articles/Pulsweitenmodulation.

20. SCons: A software construction tool. [Online] [Zitat vom: 29. 04 2009.] http://www.scons.org/.

21. mikrocontroller.net. [Online] www.mikrocontroller.net.

22. **R. Ierusalimschy, L. H. de Figueiredo, W. Celes.** Lua 5.1 Reference Manual. [Online] August 2006. http://www.lua.org/manual/5.1/.

23. **Kaiser, Ulrich und Kecher, Christoph.** *C/C++ Das umfassende Lehrbuch.* Bonn : Galileo Press, 2008. ISBN 978-3-89842-839-2.

24. **R. Ierusalimschy, L. H. de Figueiredo, W. Celes.** *Lua 5.1 Reference Manuel.* August 2006. ISBN 85-903798-3-3 .

25. Wikipedia - Interpretersprachen. [Online] [Zitat vom: 14. 07 2009.] http://de.wikipedia.org/wiki/Interpreter.

26. Wikipedia Pulsweitenmodulation Englisch. [Online] [Zitat vom: 03. 06 2009.] http://en.wikipedia.org/wiki/Pulse-width_modulation.

27. Wikipedia Open Circuit. [Online] [Zitat vom: 14. 07 2009.] http://de.wikipedia.org/wiki/Open_circuit#Pull-down.

28. Wikipedia Paritätsbit. [Online] [Zitat vom: 16. 05 2009.] http://de.wikipedia.org/wiki/Parit%C3%A4tsbit.

29. **Figueiredo, Luiz Henrique de.** Lua Libraries and Tools. [Online] [Zitat vom: 18. 07 2009.] http://www.tecgraf.puc-rio.br/~lhf/ftp/lua/.

30. Olimex Entwicklungsboards. [Online] [Zitat vom: 03. 05 2009.] http://www.olimex.com/dev/.

31. Informatik Uni Basel. [Online] [Zitat vom: 11. 04 2009.] http://informatik.unibas.ch/lehre/ss07/cs506/_Downloads/lua.pdf.

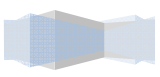
32. **Ierusalimschy, Roberto, de Figueiredo, Luiz Henrique und Celes, Waldemar.** Lua 5.1 Reference Manual. [Online] [Zitat vom: 24. 04 2009.] http://www.lua.org/manual/5.1/manual.html#4.

33. —. *Lua 5.1 Reference Manual.* s.l. : Lua.org, 2009. 85-903798-3-3.

34. ELUA Documentation. [Online] [Zitat vom: 18. 06 2009.] http://elua.berlios.de/beta/.

35. eLua Development forum & mailing list archive. [Online] [Zitat vom: 15. 06 2009.] http://n2.nabble.com/eLua-Development-f2368040.html.

36. Wikipedia BASIC. [Online] [Zitat vom: 03. 06 2009.] http://de.wikipedia.org/wiki/BASIC.



37. Microcontroller Programming Languages. [Online] 03. 06 2009. http://www.esacademy.com/automation/faq/primer/7.htm.

38. PyMite-PythonInfo Wiki. [Online] [Zitat vom: 03. 06 2009.] http://wiki.python.org/moin/PyMite.

39. mikrocontroller.net FORTH. [Online] [Zitat vom: 03. 06 2009.] http://www.mikrocontroller.net/articles/Forth.

Glossar

ADC – Analog Digital Converter

Ein Analog/Digital-Wandler wandelt analoge Daten in digitale Werte um. Mit der Hilfe eines AD-Wandlers können somit elektrische Messgrößen umgewandelt und mit dem Rechner oder Mikrocontroller ausgewertet werden. Dabei wird oft ein Sample&Hold-Glied in Verbindung mit einem Multiplexer vorgeschalten, um mehrere Messgrößen nacheinander abarbeiten zu können. (13)

AIC – Advanced Interrupt Controller

Der Advanced Interrupt Controller des AT91SAM7X256 ist ein Controller, der dem Mikrocontroller vorgeschalten ist und dessen Interrupt-Eingänge verwaltet. Dadurch wird bewerkstelligt, dass der Mikrocontroller eine Vielzahl von unterschiedlichen Interrupts abarbeiten kann.

API – Application Programming Interface

Der Begriff API steht wie der englische Name bereits sagt für eine Programmierschnittstelle, die von einem Softwaresystem anderen Programmen zur Anbindung an das System bereitgestellt wird. Die API findet ihre Verwendung nur auf der Quelltextebene.

AT91 ISP – AT91 In System Programmer

Der AT91 In System Programmer der Firma Atmel ist ein frei erhältliches Tool, mit dem man auf ARM basierende Controller der SAM3-, AT91SAM7- und AT91SAM9-Mikrocontrollerfamilien programmieren kann. Mit der enthaltenen SAM-BA Software kann man die Programmierung entweder über eine grafische Oberfläche oder über die DOS-Eingabeaufforderung programmieren. Der In System Programmer läuft nur unter Windows 2000 und XP.

bss-Segment

Das bss-Segment ist der Teil im Speicher, in dem die statischen Variablen enthalten sind, die beim Programmstart mit Werten von 0 initialisiert wurden. (14)

CAN – Controller Area Network

Bei CAN handelt es sich um ein serielles Bussystem, das zu den Feldbussen gehört. CAN wurde 1983 von Bosch für die Vernetzung von Steuergeräten in Automobilen entwickelt. Der CAN-Bus arbeitet nach dem CSMA/CR-Verfahren. (15)

Chunk

„Der Ausdruck Chunk bezeichnet ein Codestück, das in Lua ausgeführt wird. Ein Chunk ist eine einfache Folge von Befehlen.“ (2)

data-Segment

Das data-Segment ist der Teil im Speicher, in dem die vom Programmierer initialisierten globalen Variablen abgelegt sind. (14)

DBGU – Debug Unit

Durch die "Fehlersuch-Einheit" des AT91SAM7X256 wird gewährleistet, dass alle von Atmel bereitgestellten Debugmöglichkeiten genutzt werden können. (4)

Ethernet MAC

Ethernet ist eine kabelgebundene Datennetztechnik für LANs. Hierbei wird mit sogenannten Datenframes gearbeitet. Bei Ethernet sind derzeit Datenraten von 10 MBit/s bis zu 10 GBit/s möglich. Ethernet ist weitgehend in der IEEE 802.3 Norm standardisiert. (16) (17)

FLASH

Der Flash-Speicher ist ein EEPROM-Speicher(Electrically Erasable Programmable Read Only Memory). Beim Flash-Speicher werden die Daten als elektrische Ladungen auf dem Floating Gate eines MISFET's gespeichert. Die Daten gehen auch beim Abschalten der Betriebsspannung nicht verloren. Um den Speicher zu Löschen muss die Ladung gezielt entfernt werden. (18)

Heap

Der Heap ist ein dynamischer Speicher, bei dem zur Laufzeit zusammenhängende Speicherabschnitte angefordert und wieder freigegeben werden können. Im Gegensatz zum Stack arbeitet er nicht nach dem First-In/Last-Out-Prinzip. (14)

I^2C Bus

Ein von der Firma Philips entwickeltes Zwei-Draht-Bussystem, für die Kommunikation zwischen IC's.

OpenOCD – Open On Chip Debugger

Open OCD ist ein frei erhältlicher On Chip Debugger, der eine Fehlersuche direkt auf der jeweiligen Plattform ermöglicht. Dieser Debugger wurde von Dominic Rath im Zuge einer Diplomarbeit entwickelt.

PIO-Controller – Programmable Parallel Input Output Controller

Beim PIO-Controller des AT91SAM7X256 handelt es sich um einen Controller, der die parallelen Ein- bzw. Ausgangspins des Mikrocontrollers verwaltet. Durch ihn hat der Nutzer die Möglichkeit die Pins den integrierten Peripheriegeräten zuzuordnen, oder diese für andere Zwecke zu verwenden.

PWM – Pulse Width Modulation

„Bei der Pulsweitenmodulation wird die Einschalt – und Ausschaltzeit eines Rechtecksignals bei fester Periodendauer variiert. Das Verhältnis bezeichnet man auch als Tastverhältnis (= Zahl zwischen 0 und 1). Die Hauptanwendung für die PWM liegt in der Ansteuerung von Motoren." (19)

SCons

SCons ist ein auf einem Python-Skript basierendes Software-Konstruktions-Werkzeug. Durch seinen Aufbau hat es sehr starke Vorteile gegenüber anderen Software-Konstruktions-Werkzeugen. SCons stellt eine Alternative zu make dar. (20)

SPI – Serial Peripheral Interface

Die serielle Peripherie Schnittstelle ist eine serielle Datenverbindung, die es ermöglicht mit externen Baugruppen entweder im Master oder Slave Mode zu kommunizieren. Die SPI ist im Grunde genommen ein Schieberegister, das Datenbits mit anderen SPI's austauschen kann. (4)

SRAM – Static random-access memory

SRAM ist ein elektronisches Speichermedium, welches beim Abschalten der Betriebsspannung die gespeicherten Daten verliert. Man kann dabei auf beliebige Speicherinhalte zugreifen. (14)

SSC – Serial Synchronous Controller

Der SSC der Firma Atmel stellt eine synchrone Kommunikationsleitung für externe Geräte zur Verfügung. Es unterstützt viele synchrone Kommunikationsprotokolle, wie z.B. I2S. (4)

Stack

Unter dem Namen Stack verbirgt sich ein Stapelspeicher, der auf dem Prinzip Last-In/First-Out basiert. Die einzelnen Daten werden „übereinandergestapelt" und werden nacheinander wieder vom Stapel „heruntergenommen".

TC – Timer Counter

Bei dem Timer Counter handelt es sich um eine Schnittstelle mit der gewisse Zählvorgänge durchgeführt werden können. Mit ihrer Hilfe werden zum Beispiel Verzögerungsfunktionen umgesetzt.

TWI – Two Wire Interface

Die TWI von Atmel ist im Prinzip nichts anderes als ein I^2C Bus.

UDP – USB Device Port

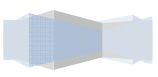
Der UDP ist der Anschlussport für USB Geräte. Der UDP des AT91SAM7X512 ist konform zur USB 2.0 full speed device specification. (4)

USART – Universal Synchronous Asynchronous Receiver Tranceiver

USART ist eine Schnittstelle zum Senden und Empfangen von Daten über eine Datenleitung. Die Schnittstelle kann sowohl synchron, als auch asynchron arbeiten. (4)

XMODEM-Protokoll

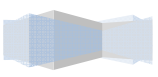
XMODEM ist die Bezeichnung für ein Sende-und Warte-ARQ-Protokoll, das eine gesicherte Datenübertragung regelt. Das Protokoll arbeitet blockorientiert, wobei jeder Block 132 Byte groß ist. (14)



Tabellenverzeichnis

Abbildungsverzeichnis

Listingverzeichnis



CD ROM

Stichwortverzeichnis

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X